CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

A STUDY AND COMPARISON OF NOSQL DATABASES

A thesis submitted in partial fulfillment of the requirements

For the degree of Master of Science in Computer Science

By

Christopher Jay Choi

May 2014

The thesis of Christopher Jay Choi is approved:

_____	_____
Dr. Adam Kaplan	Date
_____	_____
Dr. Ani Nahapetian	Date
_____	_____
Dr. George Wang, Chair	Date

California State University, Northridge

Dedication

This thesis is dedicated to all the professors who I have encountered over the course of my academic career who have helped to broaden my views and academic interests. I am indebted to several in particular who have inspired me to pursue the field of computer science and accomplish more than I, initially, thought possible of myself.

A thanks to the open source community for their willingness to develop, at cost to them, frameworks that push the future of technology and, with it, various other fields. Also, their willingness to share that knowledge and bring others up to speed.

Lastly, I'd like to thank my family and friends who have supported me throughout this endeavor. Without you all, I would not have been able to succeed.

Table of Contents

List of Figures

List of Tables

Abstract

A STUDY AND COMPARISON OF NOSQL DATABASES

By

Christopher Jay Choi

Master of Science in Computer Science

The advent of big data and the need for large scale data analysis has led to a rapid increase in the number of systems designed to serve data on the cloud. These systems address the need for OLTP, often adhering to BASE principles in order to maintain a high degree of availability and scalability. The wide range of systems, now available, make it difficult to determine which system will best meet a user's needs without testing each system individually.

In 2010, Yahoo! filled the need for an apples-to-apples performance comparison by open sourcing their Yahoo! Cloud Serving Benchmark (YCSB) framework. The framework was designed with the goal of facilitating performance comparisons of the new generation of cloud data serving systems. YCSB provides a set of core workloads that reflect common real world use cases.

This thesis discusses the architectures of several widely used NoSQL database systems as well as the results of applying YCSB workloads to the most readily available versions of those systems.

1. Introduction

Though NoSQL databases began development prior to the emergence of big data, significant industry adoption did not occur until the need was created by big data [1]. Recent years have seen a dramatic increase in the amount of data generated, as nearly all devices are now connected to the internet. The explosion in data generation resulted in a need for effective storage and manipulation of large scale data sets in order to effectively derive insight. Wide spread adoption of cloud computing, and access to server class resources at commodity hardware prices, laid the groundwork for NoSQL's success.

The purpose of this thesis is two-fold. The first is to provide a breadth analysis of NoSQL database technologies and their advantages over relational databases in the context of big data. The second is to analyze the performance data of several databases, under varying use cases, in order to provide an understanding of how the architecture of these NoSQL databases impacts performance.

The first purpose is met by providing an overview of NoSQL technologies and their qualities that allow them to effectively manage big data. The second purpose is met by deploying three of the most widely used NoSQL databases to the cloud using Amazon Web Services and benchmarking their performance using the YCSB framework.

This thesis is structured as follows. Section 2 provides background on big data and the challenges it poses along with a discussion of cloud computing. Section 3 focuses on NoSQL and the architectures of the NoSQL databases used in this thesis. Section 4 provides an overview of the testing framework and the database configurations

used during the benchmark process. Sections 5 and 6 provide benchmark results and the conclusion, respectively. Finally, section 7 discusses the opportunities for future work.

2. Background

2.1 Big Data

Big data is defined as a collection of data sets so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications [2]. Conventionally characterized by its 3V (high volume, high velocity, high variety) nature [3], big data presents a number of challenges.

The explosion of data generation coupled with the dropping costs of storage has created immense volumes of data which are continuously growing in size. The problem which was once managed by simply applying newer and better hardware has reached a point where the growth of data volume outpaces computer resources [4].

The velocity, or the rate at which data flows in and out of the system, also plays a large factor. The constant influx of data makes it difficult to distinguish relevant data from noise. Data's usefulness, particularly at the rate which things change today, is extremely short lived making it incredibly difficult to make value judgments in an acceptable amount of time.

Big data now comes in all forms: structured, semi-structured, and unstructured. The task of transforming and effectively integrating the large variety of data with each other and into existing systems has proven to be a difficult and time consuming task [5].

These characteristics have driven data to become tall or wide. Tall data, with its vast number of records, suffers from long running queries. Wide data's combinatorial set of relationships among its many variables complicates data quality assessment and model design.

These challenges have ultimately required new forms of processing to enable enhanced decision making, insight discovery and process optimization [6].

2.2 Cloud Computing

Cloud computing is simply computing that involves a large number of computers connected through a communication network such as the internet [7]. Cloud computing has been pushed as a rentable IT infrastructure, providing the benefits of elasticity, low upfront cost, and low time to market [8]. These qualities have enabled the deployment of applications that would not have been feasible in a traditional enterprise infrastructure setting. Its ease of use has made it the standard for distributed management systems.

Without cloud computing, big data would not have been of much interest as it would have simply represented an insurmountable wall of data. However, big data, in conjunction with cloud computing, has presented incredible opportunity

3. Databases

Databases, like other distributed computing systems, remain subject to the CAP theorem which states that distributed computing systems must make trade-offs between consistency, availability, and partition tolerance [9]. Consistency is the ability for all nodes in the cluster to be able to see the same data at the same time. Availability is the guarantee that every request received by a non-failing node provides a response. Partition tolerance is the system's ability to continue to run even in the case of partial failure.

Relational databases have been designed with consistency and availability in mind, supporting transactions that adhere to ACID (Atomicity, Consistency, Isolation, Durability) principles [10]. This design had been successful for relational databases as the majority of the data, at the time, integrated well with the transaction model. However, the shift towards large quantities of unstructured data, and relational database's strict adherence to ACID principles, has led to a need for a new database paradigm.

In the realm of big data, NoSQL databases have become the primary choice to fill the gap left by relational databases. The quantity and variety of data as well as the need for speed are better suited by NoSQL databases. NoSQL's usage of flexible data models allows it to handle most any data type whether it is structured, semi-structured, or unstructured. The databases are designed, from the ground up, to scale out allowing the database to be run across numerous commodity servers. The databases are also capable of adding additional capacity at any time through the addition of more servers, something that relational databases are unable to do. High availability and redundancy can be attained by implementing replica sets. However, this is not without cost as NoSQL

sacrifices ACID principles choosing to adopt BASE (Basically Available, Soft state, Eventual consistency) principles [10] instead.

The various approaches to NoSQL results in a wide taxonomy of database types. NoSQL databases are commonly categorized into one of four types: column, document, key-value, or graph.

Key-value stores are considered the simplest NoSQL storage mechanism as it is essentially a large hash table. Each data value is associated with a unique key generated by using an appropriate hashing function which is then used to determine an even distribution of hashed keys across data storage. Write and update operations insert or overwrite values for an associated key. Read operations retrieve the value associated with the desired key.

Graph databases store data in nodes and edges. Nodes contain the data about an entity while edges contain data about the relationships between entities. Graph databases excel at analyzing relationships between entities. Writes store data in new nodes. Reads require a starting node to be provided from where it traverses relationships until the desired node is found or no more nodes remain.

The NoSQL databases benchmarked in this thesis represent the column and document families and a more in-depth look into these types are provided in the following sections.

3.1 Cassandra

Cassandra was originally developed by Facebook for the purpose of handling their inbox search feature enabling users to search through their Facebook inbox [11]. It

was later released as an open source project in 2008 [12] before becoming a top-level Apache project in 2010 [13].

Cassandra is designed to be a scalable and robust solution for data persistence, utilizing the core distributed system techniques of partitioning, replication, membership, failure handling, and scaling [11].

Cassandra, a distributed column store, uses a data model inspired by Google's BigTable. As such, a table in Cassandra is a distributed multi-dimensional map indexed by a key with the data in the table being grouped together in columns. The column oriented data is stored contiguously on disk, with the underlying assumption that in most cases not all columns will be needed for data access, allowing for more records in a disk block potentially reducing disk I/O [26]. This layout is more effective at storing sparse data where many cells have NULL value or storing multi-value cells. The column approach differs from relational databases which are designed to store rows contiguously on disk.

Relational databases are required to have a fixed set of columns pre-defined by a table schema. The column model is able to get around this limitation by using column families which is a grouping of columns. Each row in a column oriented database has a fixed number of column families but each column family can consist of a variable number of columns.

The Big Table model is designed to perform three major operations: writes, reads, and compaction.

Write operations were designed to be highly optimized with sequential write, utilizing a log-structured merge tree data structure. Writes are performed by first appending the write to a commit log on disk and then storing the data to an in-memory memtable. All updates are stored in the memtable until it reaches a sufficient size where it is then flushed and stored to disk as an SSTable file using sequential I/O. This architecture allows for concurrent I/O to be avoided, improving insert times.

Reads operations utilize the merged read process in order to retrieve requested data. The merged read process consists of first performing a lookup on the memtable to determine if it contains the requested data, if the data is not found in the memtable it searches through the SSTables on disk in order to retrieve the requested data. This process can result in high I/O as row fragments from multiple SSTables may need to be merged in order to complete the request [14]. The performance impact of a merged read is partially mitigated, using a bloom filter to determine the absence of the requested row key. This allows the database to avoid performing detailed lookups on all SSTables and only perform the process on those that return a positive bloom filter result.

Once a memtable is flushed to disk as an SSTable, the SSTable file is immutable allowing no further writes. Compaction is performed to bound the number of SSTable files that must be consulted on reads and to reclaim space taken by unused data [15]. The compaction process takes two SSTables of the same size and merges them into a single SSTable that at most will be double the size of the two previous SSTables [16].

Cassandra uses a decentralized architecture where each node in the cluster has the same role with all nodes being equally capable of serving the same request. As such, it is capable of being distributed across many commodity servers with no single point of

failure [14]. It achieves this by using a peer-to-peer architecture inspired by Amazon's Dynamo DB [17].

Nodes use the gossip protocol in order to periodically exchange state information about themselves and about other nodes that they know about. By gossiping all nodes can quickly learn about all other nodes in the cluster [14].

Data partitioning is done using consistent hashing. Consistent hashing sequences hashed keys and places them in a ring structure. Each node is responsible for the range of keys up until the token assigned to the next node in the cluster.

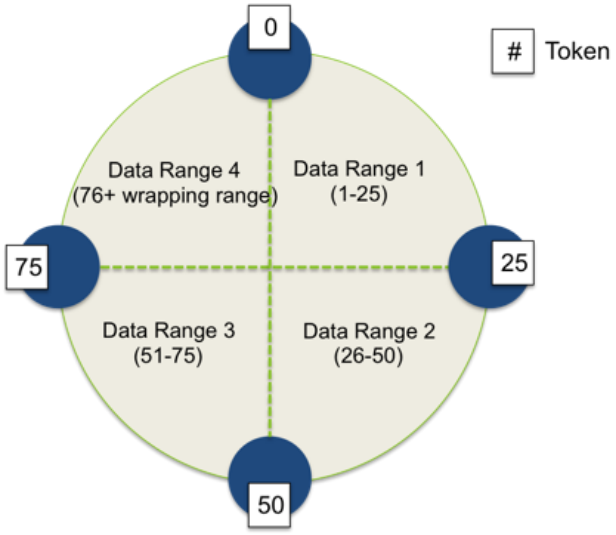

Figure 3.1 Cassandra Consistent Hashing

3.2 HBase

HBase is an open source implementation of Google's BigTable. Built on top of the Hadoop Distributed File System (HDFS), it is designed to scale efficiently with structured data of immense size by making use of HDFS's strengths. As an open source implementation of BigTable, HBase shares its data model with Cassandra.

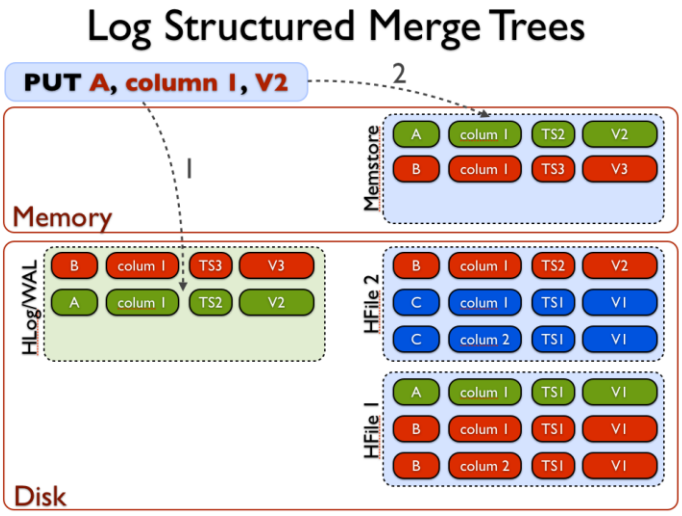

Figure 3.2 HBase Log Structured Merge Tree

HBase's differences from Cassandra lie in its architecture. Having been built on top of HDFS, HBase makes effective use of the existing HDFS, leaving it to handle the tasks for which HDFS is unsuited.

Running on top of HDFS requires HBase to utilize a master-slave architecture consisting of HMaster nodes and HRegionServer nodes. The HMaster server is responsible for monitoring all HRegionServer instances and acts as the interface for all metadata changes. The HMaster server is responsible for assigning regions to the region servers and load balancing. HRegionServers are responsible for storing data in HFiles, splitting regions when necessary, and handling client requests.

HDFS though effective at storing large files is slow to provide individual record lookups. HBase overcomes this by utilizing HFiles, an indexed map file, allowing quick lookups by key. Sufficiently full HFiles can then perform a bulk insert of the data into HDFS taking advantage of HDFS's sequential write speeds.

ZooKeeper, a high-performance coordination service, provides inter-node communication. It is used to coordinate, communicate, and share states between HMaster and HRegionServer nodes [18].

Partitioning is done by either using pre-split regions set by the user or automated dynamic partitioning which automatically splits a region and redistributes the data to the HRegionServers [19].

Building on top of HDFS gives HBase tight integration with the MapReduce framework, making it an efficient means of analyzing the big data stored in the database. MapReduce functionality, however, is outside the scope of this thesis.

3.3 MongoDB

MongoDB was developed in 2007 by 10gen as a PaaS product but later shifted to an open source development model in 2009 with 10gen offering commercial support [20].

MongoDB is designed to be a quick, schema-free, document store scalable up to medium sized storage systems with an emphasis on consistency and partition tolerance.

MongoDB uses a document data model to store data in BSON (a binary variant of JSON) documents composed of key-value pairs. Relationships between data can be stored as references or embedded data. References link one document to another based on a shared key-value while embedded data stores all related data in a single document allowing all related data to be retrieved and manipulated in a single database operation. Documents can be grouped into collections which is the rough equivalent of a traditional table but without the schema constraints [21].

MongoDB's write and read operations are much simpler to understand thanks to its memory-mapped file architecture. The memory-map maps its locations to RAM which in turn is mapped to disk [22].

Writes are performed by first writing the change to an on-disk journal [23]. The journal passes the change along and a write attempt is made to memory. If the desired page is already in RAM, values are quickly swapped in memory and the page is marked to be flushed back to disk. If the page is not yet in RAM, the page is fetched from disk before the changes are applied. Reads, similarly, will attempt to access the memory page where the document is located. If the page is not yet in RAM it will be fetched from disk before being read.

Write changes are persisted to disk every sixty seconds, by default, using the FSYNC process which flushes the memory-map to disk.

This design makes smaller data sets very quick but causes performance to falter if operations exceed the amount of memory available to the system.

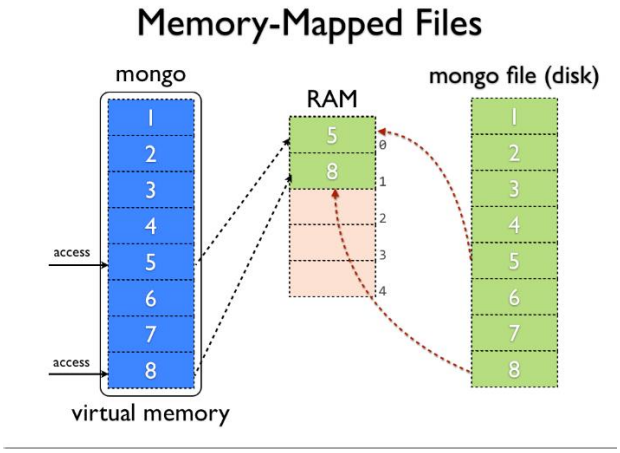

Figure 3.3 MongoDB Memory-Mapped Files

A MongoDB cluster is composed of three components: config, mongos, and mongod servers. The config server is a specially configured mongod server which stores all metadata for a sharded cluster. The failure of a config server prevents access to the rest of the cluster making it a single point of failure. The mongos server is used to process queries from client servers and determines the location of data in a sharded cluster. Mongod servers utilize memory-maps and are responsible for data retrieval and storage.

MongoDB's partitioning scheme is dependent upon a manually set shard key. The shard key is used to determine the distribution of the collection's documents among the cluster's shards.

3.4 MySQL

MySQL was included as a baseline to represent conventional RDBMS performance. MySQL, the second most widely used RDBMS in the world, has been covered heavily in prior literature and will be left to the reader to pursue.

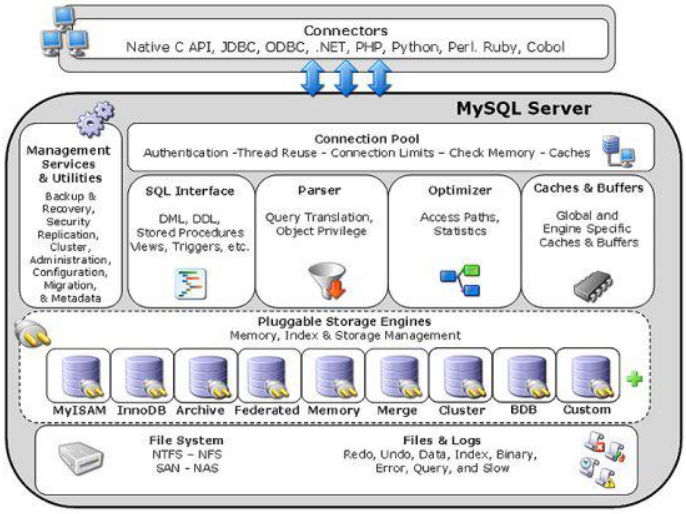

Figure 3.4 MySQL Architecture

4. Testing Framework

The benchmarks to collect performance results were run using the YCSB framework against databases deployed to Amazon's EC2 instances.

4.1 Amazon Web Services

Amazon Web Services (AWS) offers resizable computing capacity via their Elastic Compute Cloud (EC2) which was utilized when deploying both the databases and benchmark client. The tests were run using EC2's m1.xlarge instances. M1.xlarge instances provide 8 Effective Compute Units, 15 GB RAM, and 4 x 420 GB of ephemeral storage. An Amazon Elastic Block Storage (EBS) root was used to host the server OS while the data was stored on ephemeral drives attached directly to the machine in order to eliminate the latency associated with using an EBS volume.

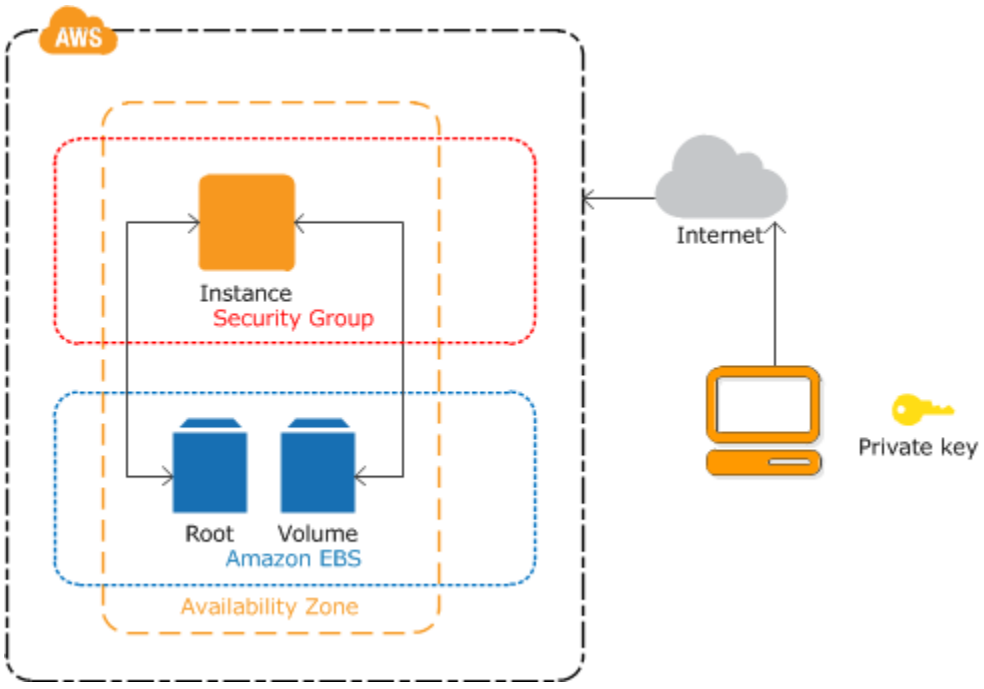

Figure 4.1 AWS EC2 Instance

4.2 Yahoo! Cloud Serving Benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) was open sourced in 2010 in order to provide an apples-to-apples comparison of cloud serving systems. The benchmark is designed to perform CRUD (Create, Read, Update, Delete) operations in varying proportions to simulate real world use cases. It has since become the de-facto standard when benchmarking cloud serving systems. The framework provides an extensible workload generator along with a set of workload scenarios to be executed by the generator.

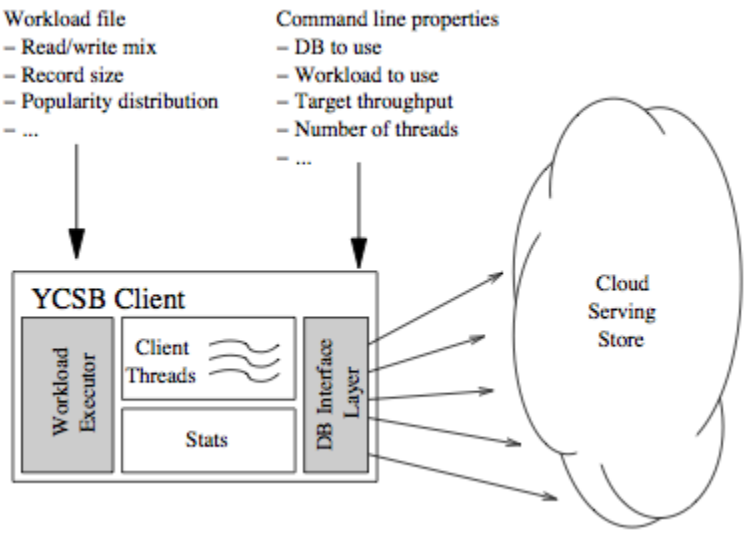

Figure 4.2 YCSB Client Architecture

The version of YCSB currently available on GitHub is no longer up to date with the most recent versions of many databases. The database bindings for Cassandra and MongoDB were updated using code provided by the community [24][25].

In this thesis, YCSB is used to benchmark the performance of the cloud serving systems by evaluating the tradeoff between latency and throughput under both light and heavy data loads.

4.3 Database Configurations

The most readily available version of each database was selected for benchmarking and configured according to vendor recommendations. Each of the databases were installed on top of the Ubuntu 12.04 OS, unless otherwise noted. Data was stored on ephemeral drives, provided by the m1.xlarge EC2 instances, striped in RAID 0. Each database was configured to work as a four node cluster. Replication was not considered when setting up the cluster and is outside the scope of this paper.

4.3.1 Cassandra

The Cassandra cluster was setup using the auto-clustering DataStax AMI provided on the AWS marketplace. The AMI configures the Cassandra cluster using the RandomPartitioner for consistent hashing and sets seed nodes cluster wide. The AMI is bundled with Cassandra 2.0.6.

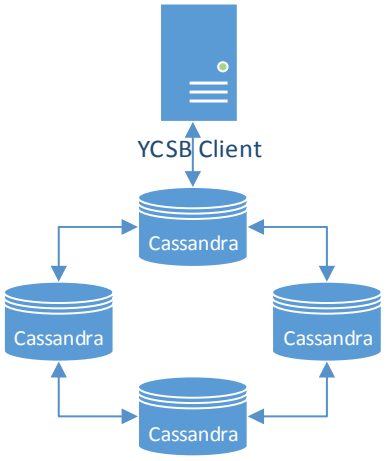

Figure 4.3 Cassandra Cluster

4.3.2 HBase

The HBase cluster was setup using Amazon's Elastic MapReduce auto-deployment. The configuration is optimized by Amazon to run HBase 0.92.0 on Debian Linux. The HBase regions were presplit following the recommended split strategy provided on the HBASE site [27].

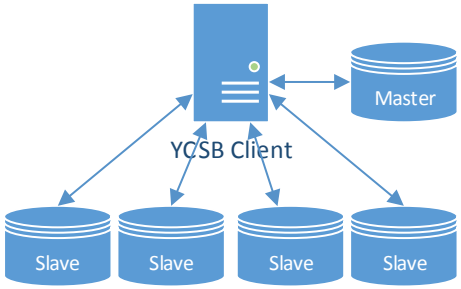

Figure 4.4 HBase Cluster

4.3.3 MongoDB

MongoDB 2.4.9 was obtained from the 10gen personal package archive. The ulimit and OS read ahead settings were configured following the recommended settings on 10gen's site [28]. The shards were split using a hashed _id field.

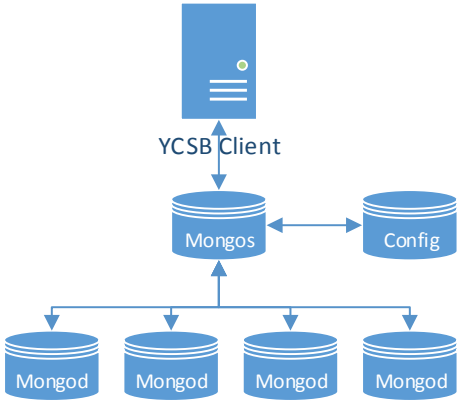

Figure 4.5 MongoDB Cluster

4.3.4 MySQL

MySQL 5.5.35 was installed from Ubuntu's public repositories. MySQL was configured to use the MyISAM engine with the key buffer size increased to 6 GB. Sharding MySQL is done client side on the primary key and distributes records evenly to each machine in the cluster.

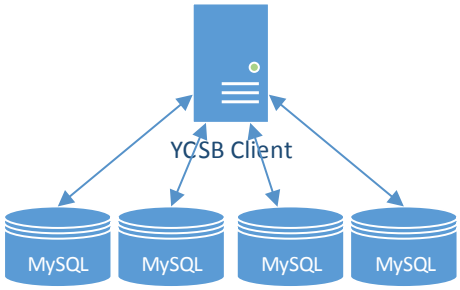

Figure 4.6 Sharded MySQL

5. Benchmark Results

Benchmarks were run on each database using the YCSB framework and a bash script to automate the process. Two sets of workloads were run, one to test a smaller data set with higher throughput and another to test a larger data set with lower throughput. The choice to test the larger data set with lower throughputs was made, to maintain a level playing field for MongoDB and MySQL which were unable to reach higher throughput values.

The smaller data set consists of 1,000,000 records with target throughputs set to 2,000, 4,000, 6,000, 8,000, and 10,000. The intent of the smaller workload is to demonstrate database performance when the entire data set is capable of fitting in memory.

The larger data set consists of 100,000,000 records with target throughputs set to 500, 1,000, 1,500, and 2,000. The larger workload is used to demonstrate database performance with the additional constraint of disk I/O.

Each inserted record consists of ten, 100 byte fields resulting in a total record size of 1 KB. Reads fetch an entire record while updates alter a single field.

The workloads were run with 100 threads and performed 1,000,000 operations with a five minute recovery period provided between each workload in order to normalize network usage between runs.

5.1 Load Records

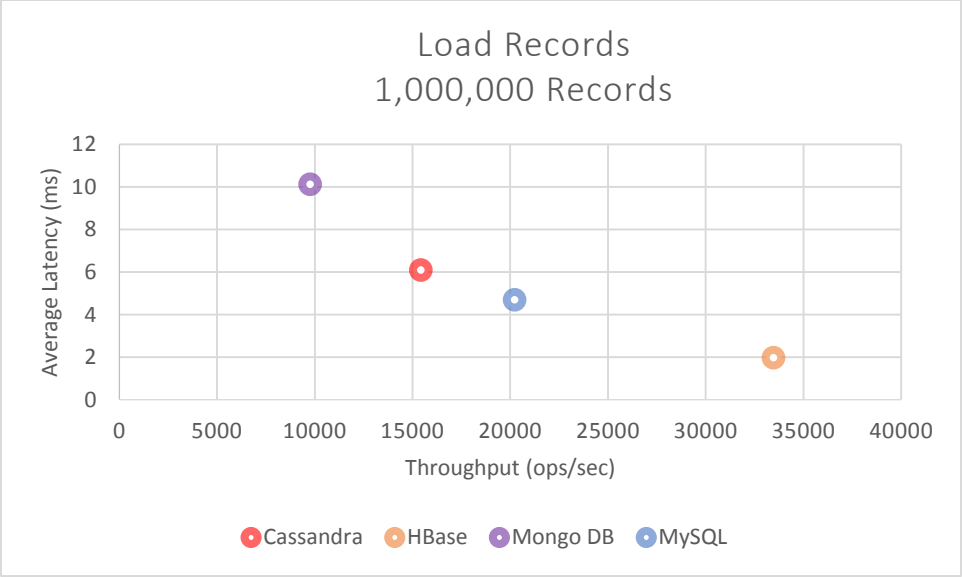

Figure 5.1 Load 1,000,000 Records

Cassandra's high performance can be attributed to its process of writing to its memtable. HBase demonstrated the highest performance, taking advantage of the memtable, pre-split regions, and asynchronous inserts. MongoDB performed respectably writing to its memory map.

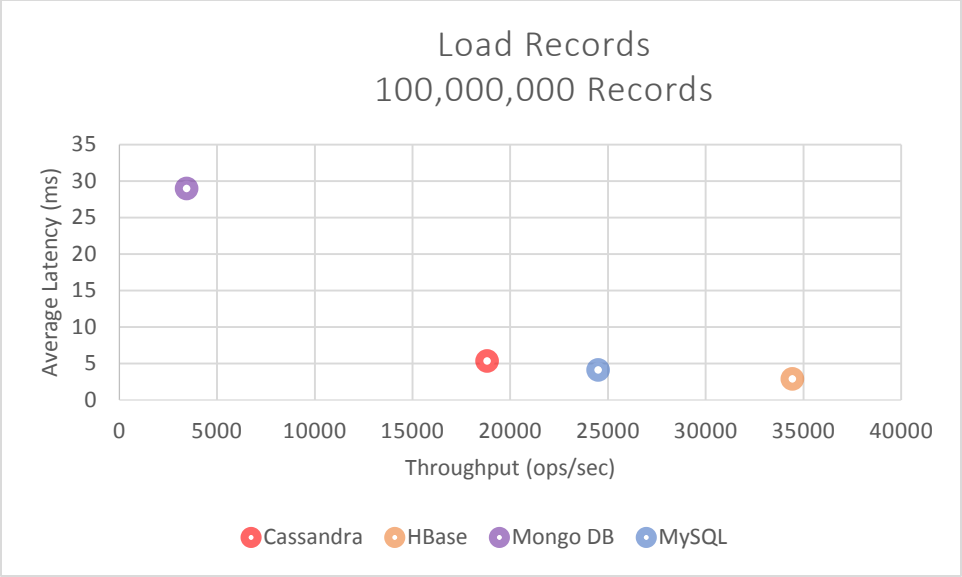

Figure 5.2 Load 100,000,000 Records

The results appear similar to the 1,000,000 record load with a few differences. Cassandra and MySQL see their throughput increase as the systems stabilize under the larger load. MongoDB performs well initially, but performance begins to deteriorate rapidly as the data set quickly outgrows the size of memory forcing FSYNC calls to flush the memory map.

5.2 Workload A: Update Heavy

Workload A is an update heavy scenario with a mix of 50/50 reads and writes. It simulates a session store recording the recent actions of a user.

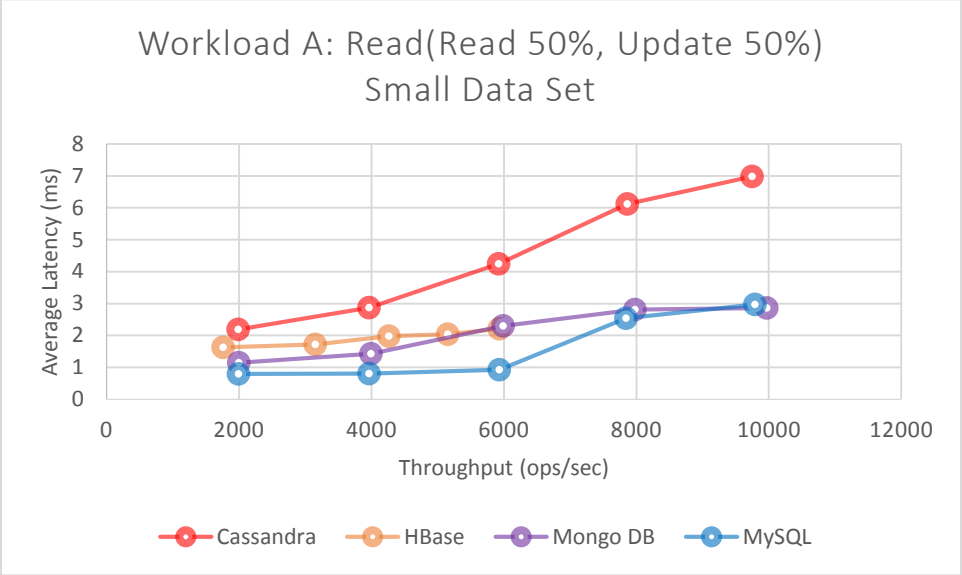

Figure 5.3 Workload A Read Small Data Set

Cassandra's latency increases relatively linearly with the increase in throughput. HBase maintains a more stable latency but suffers in throughput due to the overhead generated by updates. MongoDB performs well as it is able to successfully hit the memory map when performing read operations. MySQL makes good use of its key cache, resulting in the best overall performance.

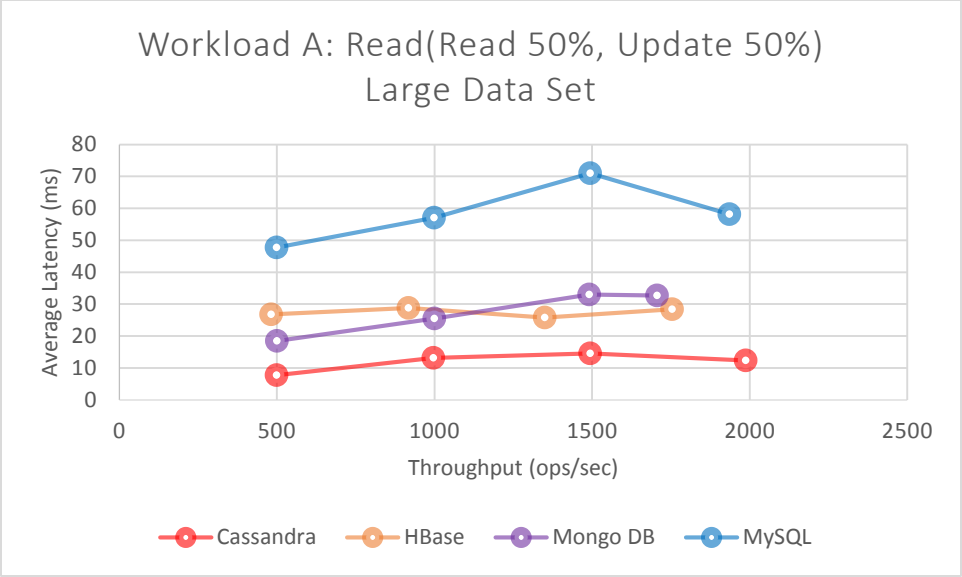

Figure 5.4 Workload A Read Large Data Set

Cassandra's performance is the most consistent between the two runs showing little change in latency despite the drastic increase in data size. MySQL's read performance drops significantly under update heavy conditions with a large data set.

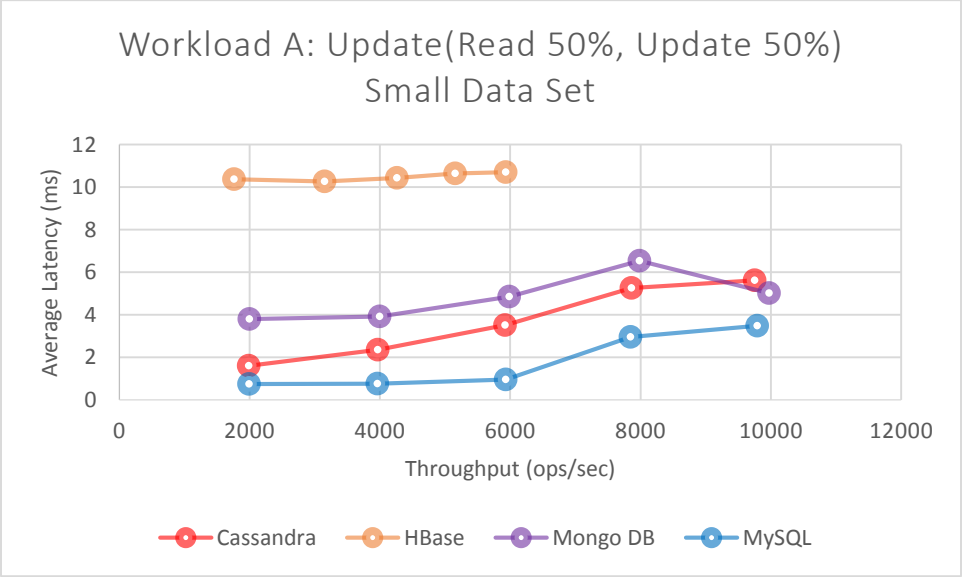

Figure 5.5 Workload A Update Small Data Set

Cassandra lives up to its reputation and is able to perform updates with significantly less latency than its reads. HBase's updates suffer from the flushCommit process that occurs at the end of the benchmark forcing a synchronous flush of the write-buffer. MongoDB and MySQL remain adept at updating smaller data sets as they do reading from them.

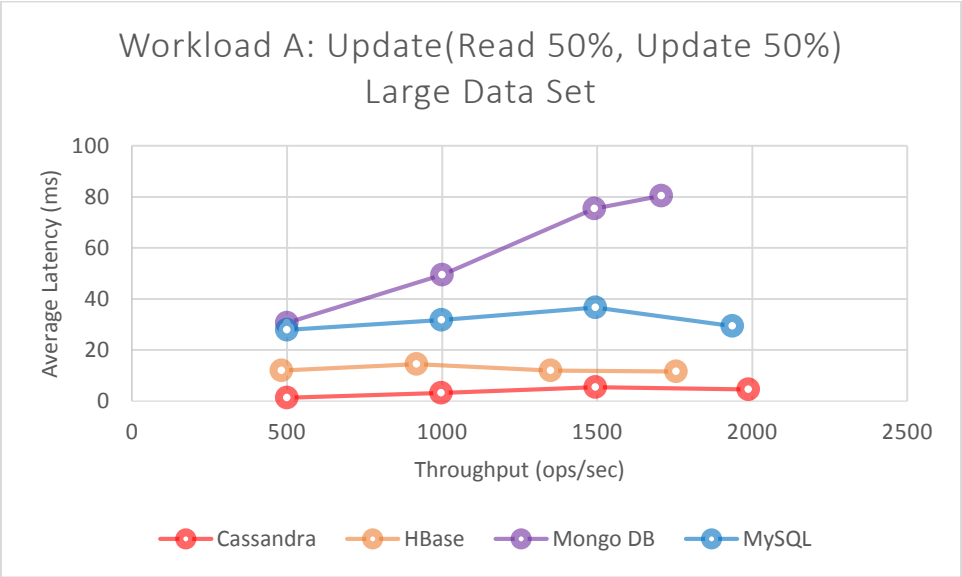

Figure 5.6 Workload A Update Large Data Set

Heavy updates on a larger data set play to Cassandra and HBase's strengths as they pull ahead of the other databases. MongoDB update performance suffers due to a full memory map triggering constant FSYNC calls.

5.3 Workload B: Read Mostly

Workload B is a read mostly workload designed to simulate content-tagging.

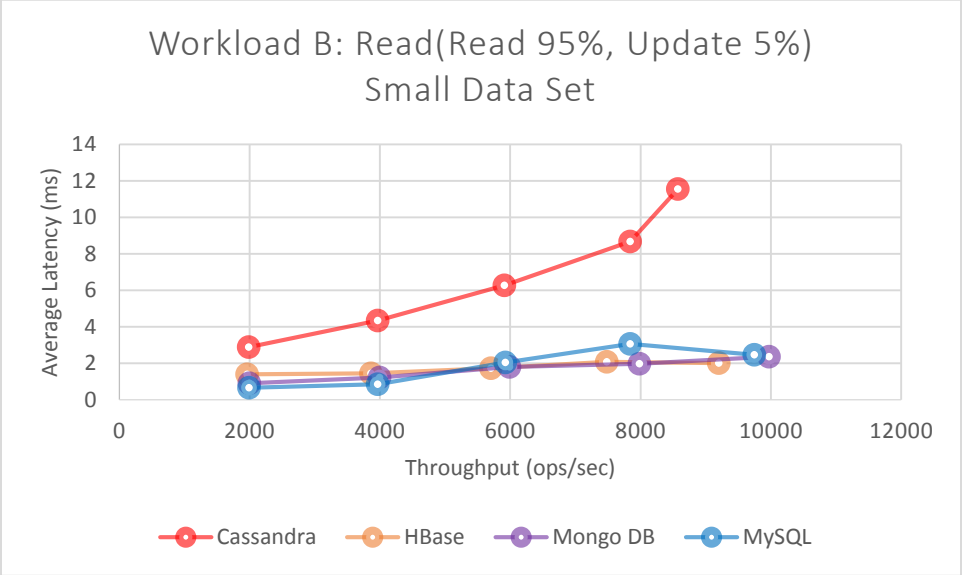

Figure 5.7 Workload B Read Small Data Set

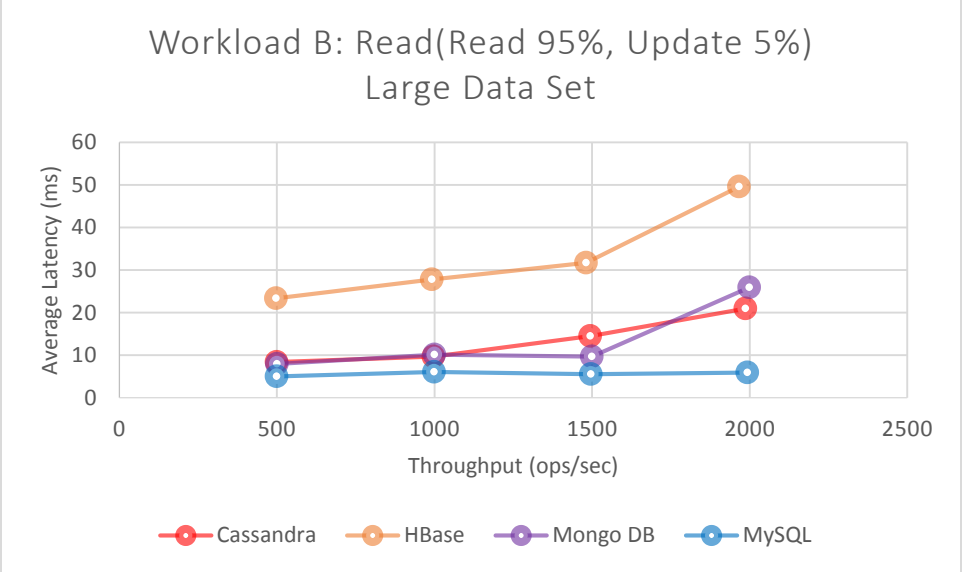

Figure 5.8 Workload B Read Large Data Set

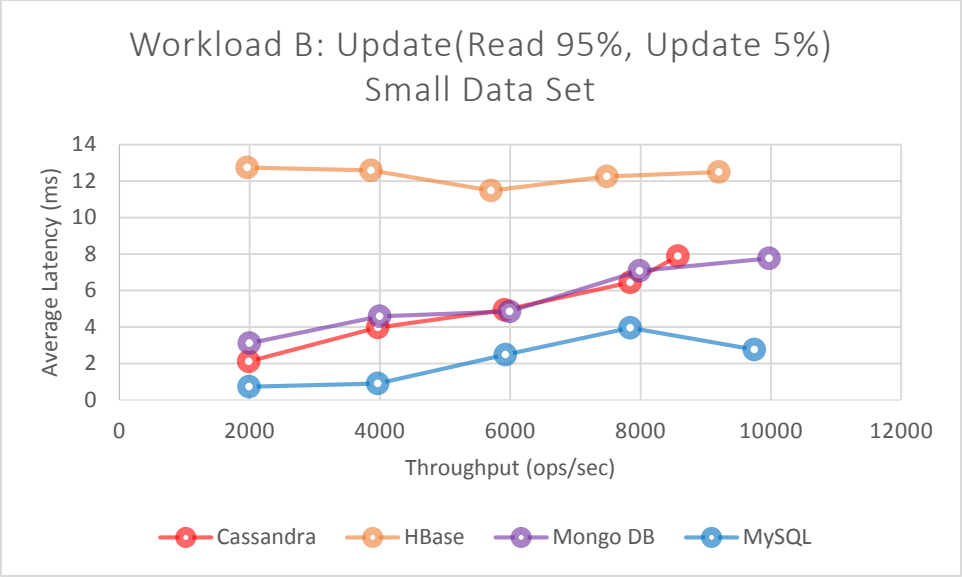

Figure 5.9 Workload B Update Small Data Set

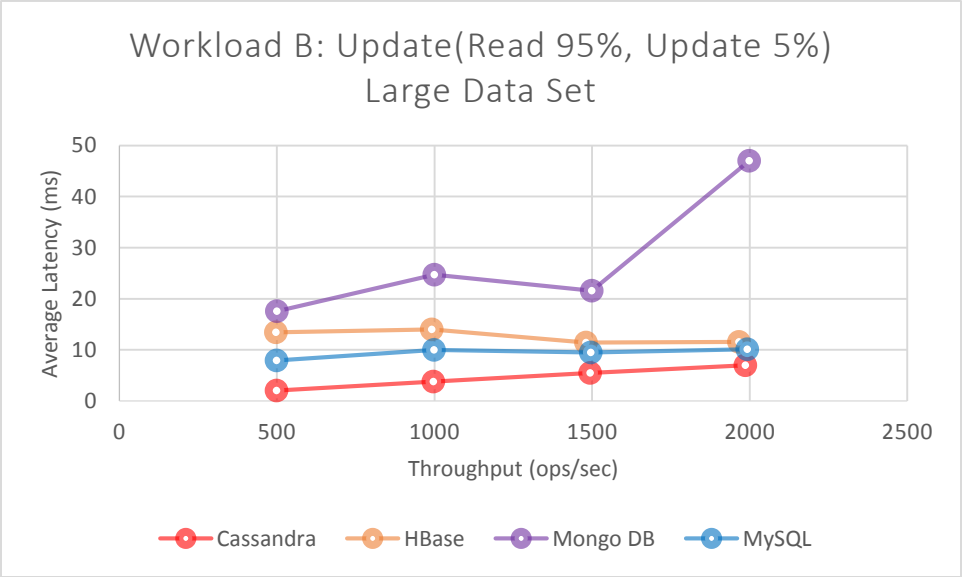

Figure 5.10 Workload B Update Large Data Set

Workload B results appear consistent with workload A's results with a few exceptions. HBase's performance on the smaller data set improves as the load shifts from updates to writes as it is able to successfully read from the memtable. The larger

data set, however, shows how much performance falters when requests become limited by disk I/O.

5.4 Workload C: Read Only

The read only workload is intended to simulate a data caching system where the data is constructed elsewhere.

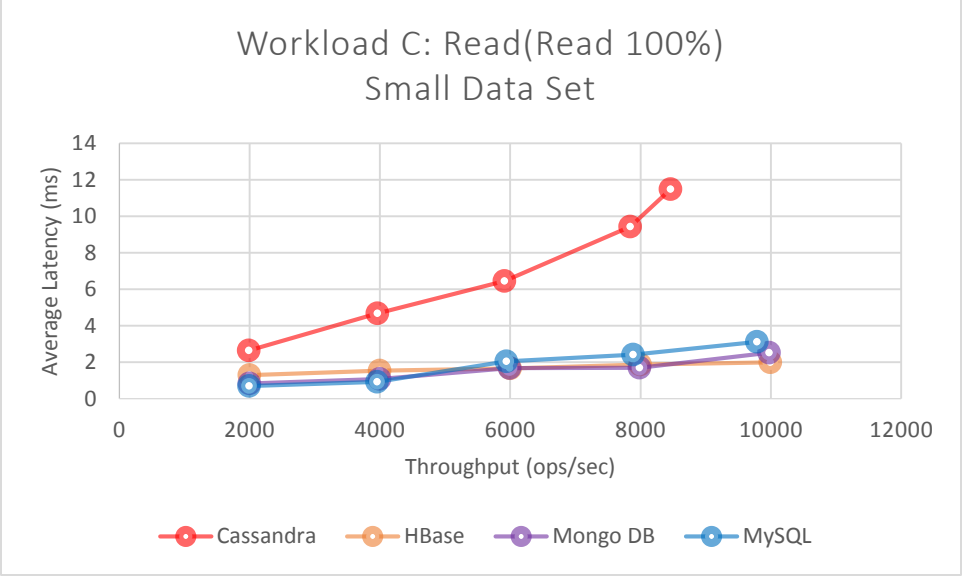

Figure 5.11 Workload C Read Small Data Set

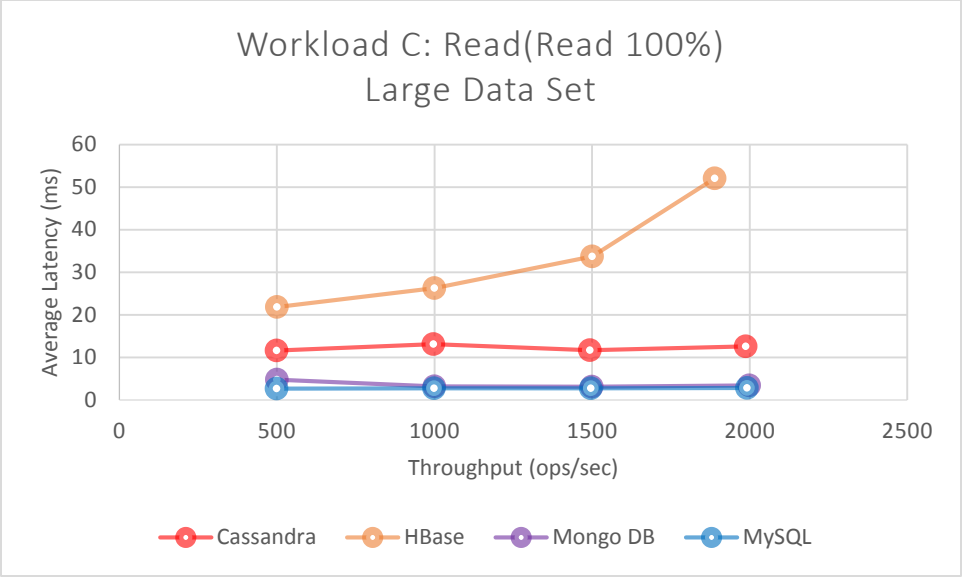

Figure 5.12 Workload C Read Large Data Set

The read only workload continues the trend from workload B's read heavy results. The exception being MongoDB whose read performance increases unconstrained by updates.

5.5 Workload D: Read Latest

The read latest workload inserts new records and the most recently inserted records are the most popular. This workload simulates user status updates or users who want to read the latest post.

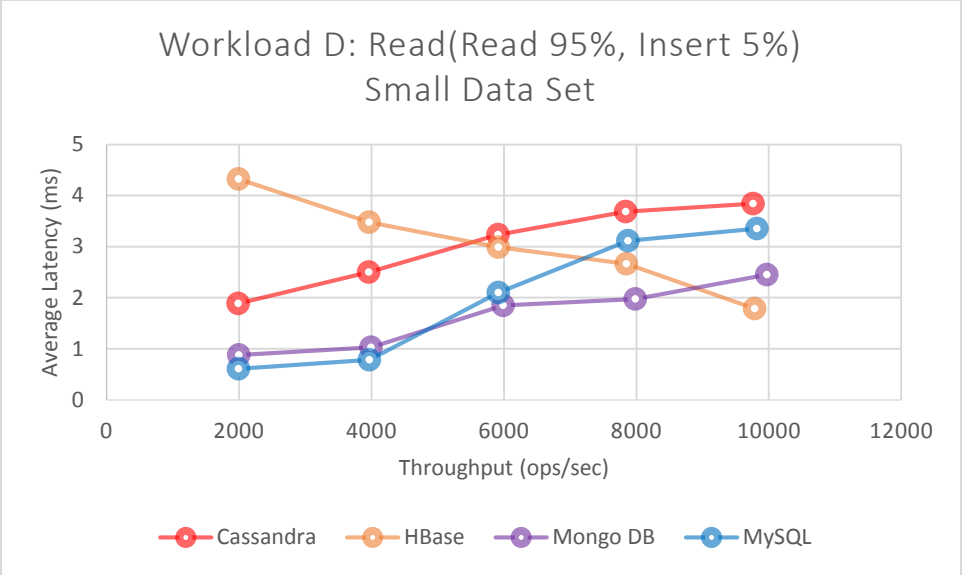

Figure 5.13 Workload D Read Small Data Set

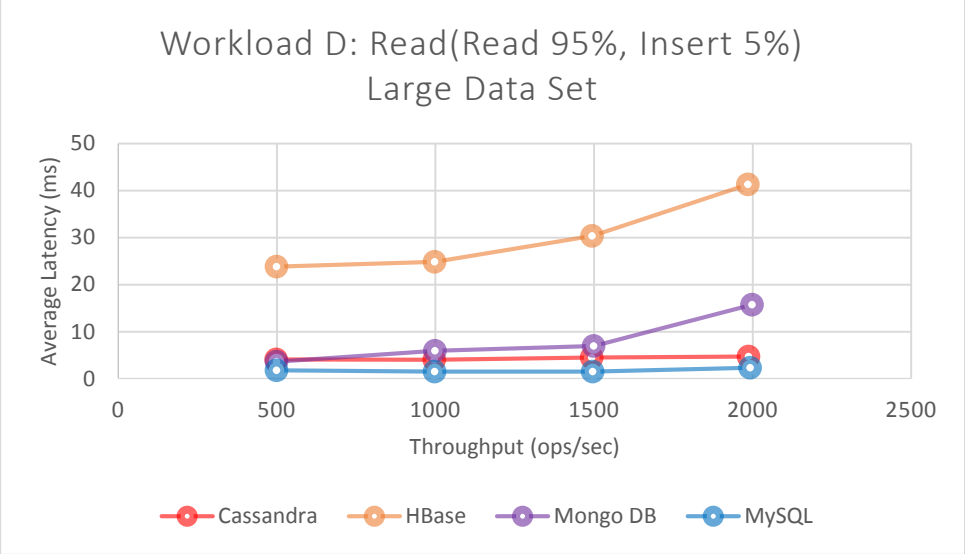

Figure 5.14 Workload D Read Large Data Set

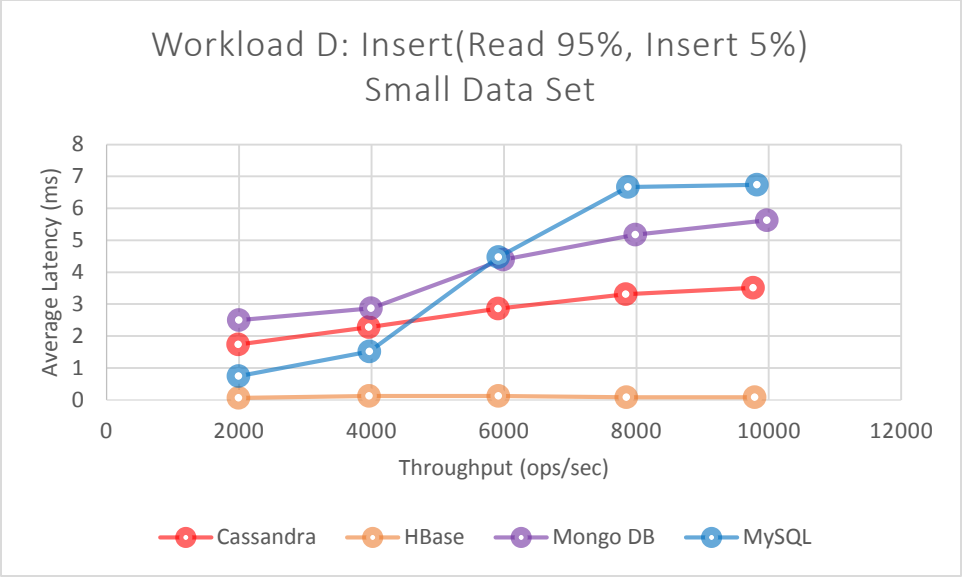

Figure 5.15 Workload D Insert Small Data Set

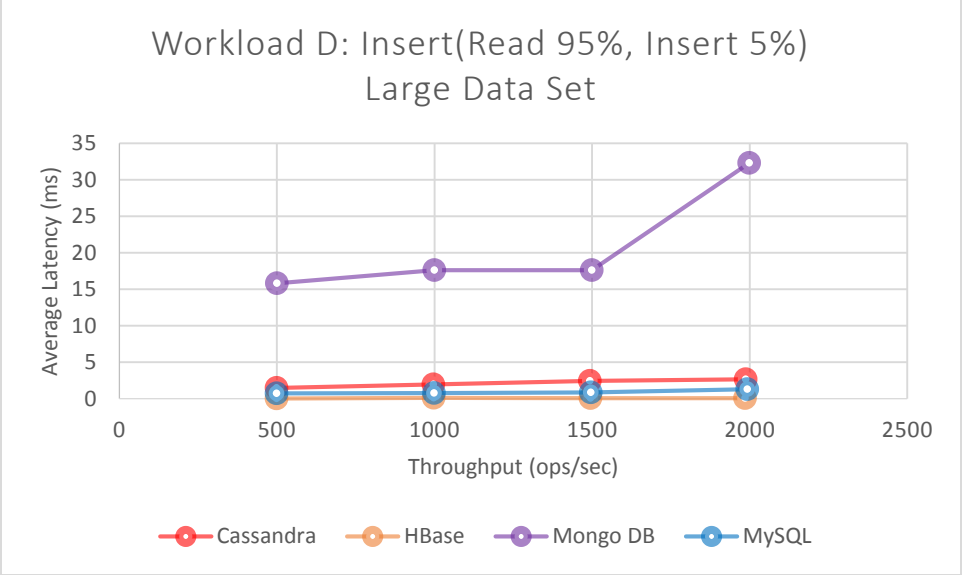

Figure 5.16 Workload D Insert Large Data Set

The inserts demonstrate the efficiency of NoSQL write performance at higher throughputs as the performance of all the NoSQL databases exceed MySQL. MySQL is

unfazed at lower throughputs despite the larger data set while MongoDB continues to suffer from being unable to fit the entire data set in memory.

5.6 Workload F: Read-Modify-Write

This workload has the client read a record, modify it, and write back the changes. The workload simulates a user database where user records are read and modified by the user.

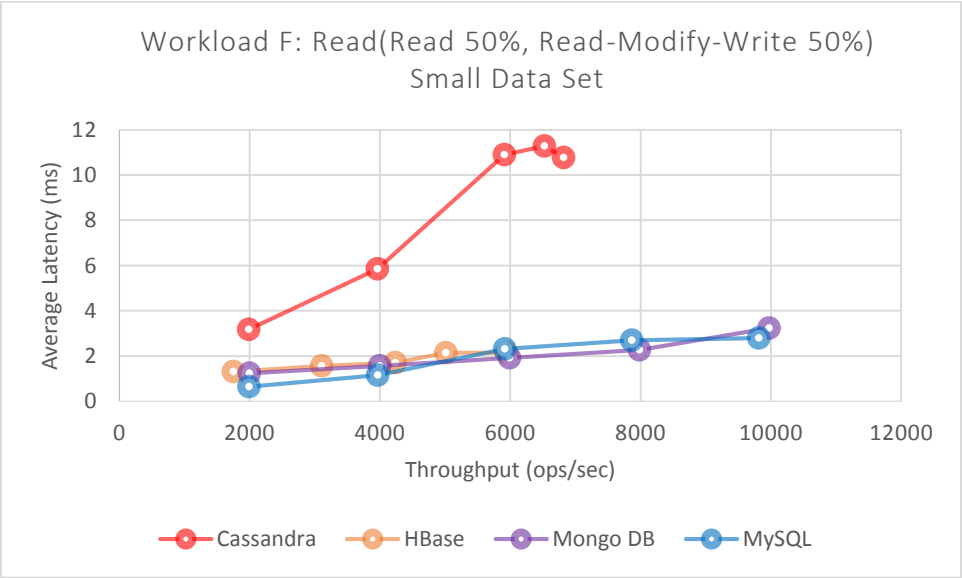

Figure 5.17 Workload F Read Small Data Set

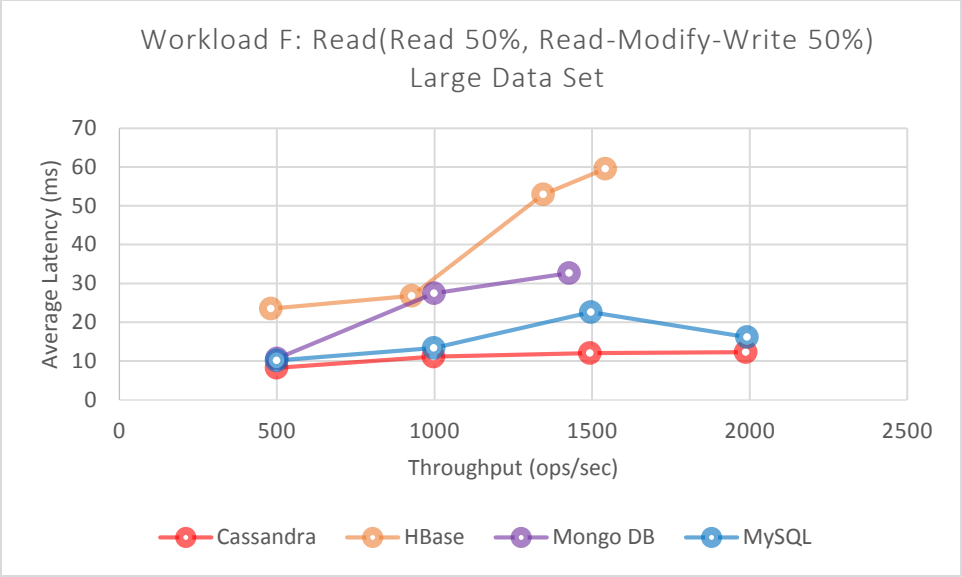

Figure 5.18 Workload F Read Large Data Set

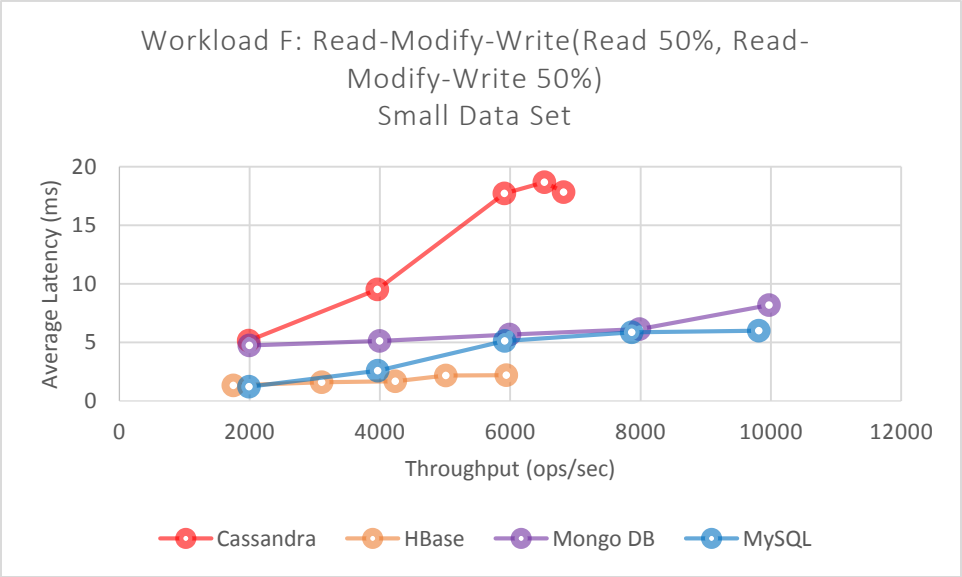

Figure 5.19 Workload F Read-Modify-Write Small Data Set

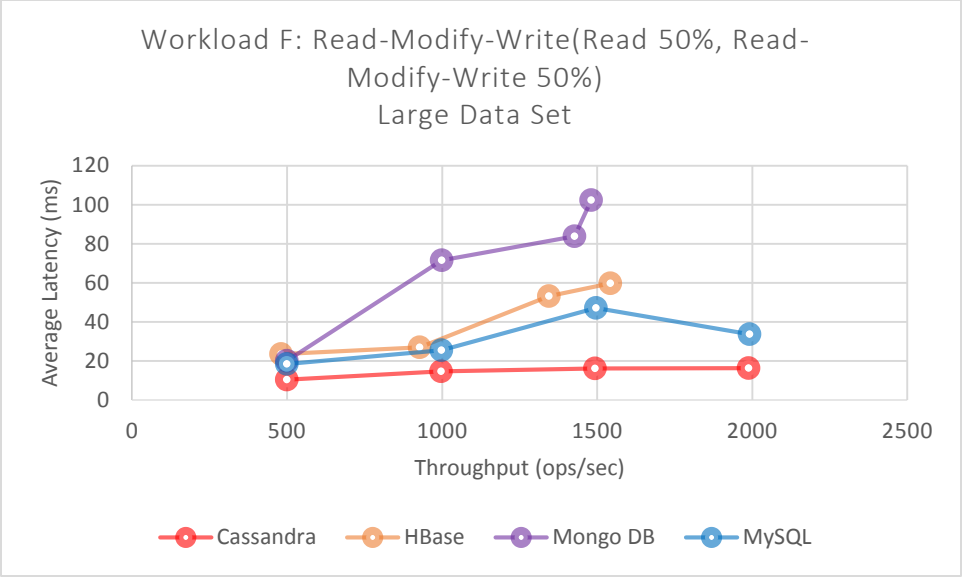

Figure 5.20 Workload F Read-Modify-Write Large Data Set

Cassandra again demonstrates the most consistent performance across small and large data sets as latency does not increase much across the two runs causing it to go from last under the small data set to first on the large data set.

6. Conclusion

It is safe to say that in the realm of big data there is no single database that is optimal for all use cases. There are no good or bad databases but rather those that fit the needs of a particular set of use cases. It is up to the end user to determine whether a specific database will meet their requirements in terms of data sets and data model, transaction support, data consistency, and performance.

Cassandra lived up to its expectations in terms of writes while maintaining acceptable read performance. Unfortunately, Cassandra's greatest strength was outside the scope of this benchmark as Cassandra is capable of achieving far higher throughput at the expense of latency.

HBase performed well overall but suffered during updates. The low throughput and high latency is attributable to the design of the YCSB framework. The YCSB framework forces HBase to flush the commit log during the cleanup portion of the test, forcing what would normally be an asynchronous process to become synchronous. Had the operation count been larger, we would expect HBase's results to not be as heavily impacted by the flushing of the commit log as much of the work would've been done asynchronously.

It's clear that when working with smaller data sets MongoDB is the highest performing NoSQL database, overall. Being able to fit the entire data set into the memory-map allows MongoDB to work with cache the entire time. Despite having a lower input throughput than Cassandra and HBase, read and update operations were

substantially better. However, when working with larger data sets MongoDB performance degrades rapidly as a large number of operations resort to disk I/O.

Sharded MySQL demonstrated that relational databases are not outclassed by NoSQL databases when it comes to raw performance in terms of CRUD operations. MySQL was capable of both keeping up with and surpassing the NoSQL databases under a variety of scenarios.

7. Future Work

Time and financial constraints limited the amount of testing that could be performed. Though the tests performed provided points of interest, much more work can be done to further the study.

Statistical integrity can be achieved by running the same set of tests at different times on different sets of machines to mitigate the effects of a lame machine, noisy neighbor, or high network traffic on shared tenant machines.

Tests could be run that better demonstrate the strengths of NoSQL databases that were outside the scope of the tests performed in this thesis. Partition tolerance could be tested by having nodes fail and monitoring performance during the failover period. Availability scaling could be tested by including replica sets and observing how performance scales with an increase in replica sets. Scalability tests that test performance increases with an increasingly large number of nodes could be done to demonstrate NoSQL's superiority over relational databases when a high number of shards are involved. Tests could also be performed to cover maximum throughput, in order to determine which database is most capable of serving high loads.

References

[1] "RDBMS dominate the database market, but NoSQL systems are catching up". DB-Engines.com. Retrieved March 2014.

[2] “Big Data”. Wikipedia.org. Retrieved April 2014.

[3] "Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data". Gartner.com. Retrieved March 2014.

[4] “Challenges and Opportunities with Big Data”. CRA.org. Retrieved April 2014.

[5] Jeffrey Heer and Sean Kandel. 2012. Interactive analysis of big data. XRDS 19, 1 (September 2012), 50-54.

[6] “The Importance of ‘Big Data’: A Definition”. Gartner.com. Retrieved April 2014.

[7] Mariana Carroll, Paula Kotzé, Alta van der Merwe. 2012. Securing Virtual and Cloud Environments. Cloud Computing and Services Science, Service Science: Research and Innovations in the Service Economy, 73-90.

[8] Surajit Chaudhuri. 2012. What next?: a half-dozen data management research goals for big data and the cloud. In Proceedings of the 31st symposium on Principles of Database Systems (PODS '12), Markus Krötzsch (Ed.). ACM, New York, NY, USA, 1-4.

[9] Gilbert, S.; Lynch, Nancy A., "Perspectives on the CAP Theorem," Computer , vol.45, no.2, pp.30,36, Feb. 2012

[10] Dan Pritchett. 2008. BASE: An Acid Alternative. Queue 6, 3 (May 2008), 48-55.

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

[12] "Facebook Releases Cassandra as Open Source". Perspective.mvdirona.com. Retrieved March 2014.

[13] "Cassandra is an Apache top level project". Mail-archive.com. Retrieved March 2014.

[14] "Apache Cassandra 2.0 Documentation – Understanding the Architecture”. DataStax.com. Retrieved March 2014

[15] “MemtableSSTable”. Apache.org. Retrieved April 2014.

[16] “BigTable Model with Cassandra and HBase”. Dzone.com. Retrieved April 2014.

[17] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35-40.

[18] “What are HBase znodes?”. Cloudera.com. Retrieved April 2014.

[19] “Apache HBase Region Splitting and Merging”. Hortonworks.com. Retrieved April 2014.

[20] “10gen embraces what it created, becomes MongoDB Inc.”. Gigaom.com. Retrieved April 2014.

[21] “FAQ: MongoDB Fundamentals”. Mongodb.org. Retrieved April 2014.

[22] “MongoDB Architecture Guide”. Mongodb.com. Retrieved March 2014.

[23] “How MongoDB’s Journal Works”. Mongodb.org. Retrieved April 2014.

[24] “YCSB MongoDB Binding”. https://github.com/achille/YCSB. Retrieved March 2013.

[25] “YCSB Cassandra-CQL Binding”. https://github.com/cmatser/YCSB. Retrieved March 2013.

[26] Hemanth Gokavarapu. Exploring Cassandra and HBase with BigTable Model.

[27] “Create Split Strategy for YCSB Benchmark”. Apache.org Retrieved March 2013.

[28] “Amazon EC2”. Mongodb.org Retrieved March 2013.

Appendix A: Benchmark Results Data

Load 1,000,000 Records		
Database	**Throughput**	**Average Latency**
Cassandra	15432.09877	6.077716486
HBase	33477.28566	1.965811263
MongoDB	9762.764815	10.11995494
MySQL	20236.77021	4.686284544

Table A.1 Load 1,000,000 Records

Load 100,000,000 Records		
Database	**Throughput**	**Average Latency**
Cassandra	18818.46038	5.300832428
HBase	34438.26201	2.865327013
MongoDB	3448.059942	28.93257253
MySQL	24499.30765	4.061760399

Table A.2 Load 100,000,000 Records

Workload A Read Small Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	2000	1990.489441	2.185008452
	4000	3964.934123	2.866947365
	6000	5923.012681	4.238572756
	8000	7862.376954	6.120254213
	10000	9751.150636	6.978600453
HBase	2000	1763.270373	1.627156244
	4000	3153.489809	1.717391287
	6000	4262.501918	1.97501614
	8000	5153.178223	2.039249214
	10000	5930.248418	2.208115217
MongoDB	2000	1998.812705	1.14995955
	4000	3995.65273	1.421723409
	6000	5987.557855	2.303176089
	8000	7981.928913	2.810917576
	10000	9973.371099	2.863461333
MySQL	2000	1992.480379	0.79761932
	4000	3964.32111	0.80599535
	6000	5932.6056	0.925049035
	8000	7846.522029	2.539194012
	10000	9791.154671	2.969590609

Table A.3 Workload A Read Small Data Set

Workload A Read Large Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	500	499.2266978	7.714154648
	1000	996.9791532	13.08787645
	1500	1493.997864	14.49436775
	2000	1987.083954	12.28926422
HBase	500	481.9442015	26.80996767
	1000	917.8892952	28.73030007
	1500	1349.975835	25.75944868
	2000	1754.195158	28.40644699
MongoDB	500	499.8822777	18.47138876
	1000	999.6221428	25.49955832
	1500	1490.783973	32.96070977
	2000	1706.507939	32.68357965
MySQL	500	499.6030654	47.72360012
	1000	998.4055463	57.07430099
	1500	1494.632029	70.9419413
	2000	1936.014714	58.12886486

Table A.4 Workload A Read Large Data Set

Workload A Update Small Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	2000	1990.489441	1.610023321
	4000	3964.934123	2.366101581
	6000	5923.012681	3.523959684
	8000	7862.376954	5.256343647
	10000	9751.150636	5.623357847
HBase	2000	1763.270373	10.38111022
	4000	3153.489809	10.26960981
	6000	4262.501918	10.435173
	8000	5153.178223	10.64733838
	10000	5930.248418	10.69837866
MongoDB	2000	1998.812705	3.796783102
	4000	3995.65273	3.926953404
	6000	5987.557855	4.862835965
	8000	7981.928913	6.546119075
	10000	9973.371099	5.022346623
MySQL	2000	1992.480379	0.751661616
	4000	3964.32111	0.765881901
	6000	5932.6056	0.957378566
	8000	7846.522029	2.955856511
	10000	9791.154671	3.489971154

Table A.5 Workload A Update Small Data Set

Workload A Update Large Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	500	499.2266978	1.38129229
	1000	996.9791532	3.231030122
	1500	1493.997864	5.464871106
	2000	1987.083954	4.606863745
HBase	500	481.9442015	11.96443577
	1000	917.8892952	14.50925272
	1500	1349.975835	11.98100984
	2000	1754.195158	11.57680457
MongoDB	500	499.8822777	30.6709065
	1000	999.6221428	49.45035379
	1500	1490.783973	75.35083252
	2000	1706.507939	80.39744475
MySQL	500	499.6030654	27.91982538
	1000	998.4055463	31.76520969
	1500	1494.632029	36.64634545
	2000	1936.014714	29.35865894

Table A.6 Workload A Update Large Data Set

Workload B Read Small Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	2000	1989.91511	2.878020642
	4000	3964.934123	4.329898557
	6000	5910.479872	6.272721671
	8000	7842.829693	8.673749879
	10000	8571.257146	11.5492409
HBase	2000	1964.960819	1.383469028
	4000	3863.047249	1.436271029
	6000	5706.622535	1.717600271
	8000	7484.637781	2.077407689
	10000	9202.510445	2.012750228
MongoDB	2000	1998.836677	0.896818373
	4000	3995.668695	1.210121923
	6000	5989.673802	1.790928613
	8000	7983.330805	1.969606381
	10000	9973.470568	2.364984851
MySQL	2000	1993.076054	0.654767946
	4000	3967.907564	0.855896173
	6000	5925.750348	2.032624627
	8000	7842.091643	3.057090266
	10000	9748.964173	2.458383026

Table A.7 Workload B Read Small Data Set

Workload B Read Large Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	500	499.2147352	8.399780168
	1000	996.9612621	9.667240027
	1500	1493.997864	14.49436775
	2000	1986.673395	20.94930012
HBase	500	497.7367908	23.36689041
	1000	991.1834234	27.81358645
	1500	1481.018526	31.69541695
	2000	1966.66116	49.57836273
MongoDB	500	499.8820278	7.921595419
	1000	999.6221428	10.11896448
	1500	1499.218907	9.658546537
	2000	1998.668887	25.8998346
MySQL	500	499.5598877	5.041458226
	1000	998.3567049	6.044793257
	1500	1496.291442	5.563976744
	2000	1992.52802	5.906995626

Table A.8 Workload B Read Large Data Set

Workload B Update Small Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	2000	1989.91511	2.108351185
	4000	3964.934123	3.962436106
	6000	5910.479872	4.937410679
	8000	7842.829693	6.435794449
	10000	8571.257146	7.8756817
HBase	2000	1964.960819	12.74374624
	4000	3863.047249	12.57934633
	6000	5706.622535	11.47891446
	8000	7484.637781	12.24609189
	10000	9202.510445	12.48589537
MongoDB	2000	1998.836677	3.107857976
	4000	3995.668695	4.578905711
	6000	5989.673802	4.842404727
	8000	7983.330805	7.074582068
	10000	9973.470568	7.746506084
MySQL	2000	1993.076054	0.714718602
	4000	3967.907564	0.887325597
	6000	5925.750348	2.482307964
	8000	7842.091643	3.951064676
	10000	9748.964173	2.761863155

Table A.9 Workload B Update Small Data Set

Workload B Update Large Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	500	499.2147352	2.020297043
	1000	996.9612621	3.801916179
	1500	1493.997864	5.464871106
	2000	1986.673395	6.986234237
HBase	500	497.7367908	13.44841088
	1000	991.1834234	13.97884059
	1500	1481.018526	11.42957043
	2000	1966.66116	11.53692145
MongoDB	500	499.8820278	17.59444297
	1000	999.6221428	24.71666296
	1500	1499.218907	21.55527335
	2000	1998.668887	47.01589669
MySQL	500	499.5598877	7.901268859
	1000	998.3567049	9.998819772
	1500	1496.291442	9.486139583
	2000	1992.52802	10.12688723

Table A.10 Workload B Update Large Data Set

Workload C Read Small Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	2000	1989.867594	2.641525092
	4000	3960.45878	4.674104617
	6000	5912.926248	6.452302763
	8000	7843.19877	9.437403416
	10000	8461.95505	11.48955562
HBase	2000	1999.604078	1.298509424
	4000	3998.864323	1.54234544
	6000	5997.900735	1.658990318
	8000	7995.970031	1.881611892
	10000	9994.203362	1.995758487
MongoDB	2000	1998.832682	0.838974581
	4000	3995.65273	1.081969586
	6000	5990.570841	1.685077662
	8000	7983.330805	1.695440513
	10000	9973.967944	2.530349713
MySQL	2000	1993.469394	0.696643997
	4000	3956.713554	0.936366376
	6000	5941.558827	2.042874655
	8000	7887.990534	2.4121164
	10000	9788.183705	3.117784325

Table A.11 Workload C Read Small Data Set

Workload C Read Large Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	500	499.1479544	11.58397699
	1000	996.9065988	13.09911356
	1500	1493.127879	11.67334378
	2000	1987.794939	12.54442516
HBase	500	499.9430065	21.80916427
	1000	999.8600196	26.2130077
	1500	1499.758539	33.69365782
	2000	1888.941569	52.03901166
MongoDB	500	499.8830274	4.713477014
	1000	999.6261398	3.15859433
	1500	1499.232393	3.082024657
	2000	1998.700844	3.37513365
MySQL	500	499.5558948	2.598851054
	1000	998.2979021	2.73651663
	1500	1496.186221	2.687125287
	2000	1991.928705	2.813363184

Table A.12 Workload C Read Large Data Set

Workload D Read Small Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	2000	1990.160646	1.887665234
	4000	3959.674673	2.497031478
	6000	5911.807653	3.233684858
	8000	7841.907152	3.68583578
	10000	9764.004023	3.838531689
HBase	2000	1991.539938	4.322946989
	4000	3962.797259	3.479929904
	6000	5914.989767	2.985695575
	8000	7852.067057	2.660882989
	10000	9789.621044	1.78476611
MongoDB	2000	1998.816701	0.875489203
	4000	3995.65273	1.026637884
	6000	5990.534955	1.8461936
	8000	7983.139609	1.972995619
	10000	9973.072704	2.448856975
MySQL	2000	1993.338264	0.606463712
	4000	3970.396722	0.783518039
	6000	5915.864574	2.095995319
	8000	7872.404075	3.115466886
	10000	9821.928437	3.353920182

Table A.13 Workload D Read Small Data Set

Workload D Read Large Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	500	499.2157321	4.11243213
	1000	996.9364144	4.006628318
	1500	1493.208143	4.514510932
	2000	1987.873969	4.706247507
HBase	500	499.4605826	23.82681082
	1000	998.001004	24.81260416
	1500	1494.895679	30.38005388
	2000	1985.864616	41.33761511
MongoDB	500	499.8732821	3.600394737
	1000	999.6081536	5.929854227
	1500	1499.200926	6.96756978
	2000	1998.632935	15.71402412
MySQL	500	499.5578913	1.74779998
	1000	998.3048783	1.49414396
	1500	1496.230994	1.486528993
	2000	1992.941003	2.287316681

Table A.14 Workload D Read Large Data Set

Workload D Insert Small Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	2000	1990.160646	1.731307976
	4000	3959.674673	2.276379835
	6000	5911.807653	2.854859602
	8000	7841.907152	3.311254343
	10000	9764.004023	3.505938084
HBase	2000	1991.539938	0.06506284
	4000	3962.797259	0.11980961
	6000	5914.989767	0.120301927
	8000	7852.067057	0.084852466
	10000	9789.621044	0.083823417
MongoDB	2000	1998.816701	2.494711103
	4000	3995.65273	2.863713719
	6000	5990.534955	4.392926748
	8000	7983.139609	5.171543749
	10000	9973.072704	5.626626161
MySQL	2000	1993.338264	0.740793669
	4000	3970.396722	1.518273457
	6000	5915.864574	4.47150901
	8000	7872.404075	6.666985515
	10000	9821.928437	6.733826505

Table A.15 Workload D Insert Small Data Set

Workload D Insert Large Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	500	499.2157321	1.500279849
	1000	996.9364144	1.976141772
	1500	1493.208143	2.445220929
	2000	1987.873969	2.670950093
HBase	500	499.4605826	0.051622488
	1000	998.001004	0.109283563
	1500	1494.895679	0.104853274
	2000	1985.864616	0.072152467
MongoDB	500	499.8732821	15.8180264
	1000	999.6081536	17.63389021
	1500	1499.200926	17.62606058
	2000	1998.632935	32.314847
MySQL	500	499.5578913	0.728548133
	1000	998.3048783	0.770934682
	1500	1496.230994	0.884571973
	2000	1992.941003	1.293872856

Table A.16 Insert Large Data Set

Workload F Read Small Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	2000	1989.970548	3.176714954
	4000	3960.615638	5.847154081
	6000	5912.716479	10.89665297
	8000	6525.966822	11.28007187
	10000	6820.212381	10.77355913
HBase	2000	1752.980505	1.314803204
	4000	3106.796117	1.573023129
	6000	4239.102328	1.714590648
	8000	5014.190158	2.147514645
	10000	5938.947618	2.149405094
MongoDB	2000	1998.812705	1.245491188
	4000	3995.604835	1.573906915
	6000	5990.678504	1.915844379
	8000	7982.948422	2.272626638
	10000	9974.166908	3.236985907
MySQL	2000	1992.758316	0.646828943
	4000	3967.8131	1.150763845
	6000	5917.75504	2.312037907
	8000	7865.097842	2.702517444
	10000	9817.685579	2.794231833

Table A.17 Workload F Read Small Data Set

Workload F Read Large Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	500	499.2017764	8.233920493
	1000	996.931445	11.11200919
	1500	1493.132338	12.03140047
	2000	1988.004382	12.27791636
HBase	500	480.863554	23.46949988
	1000	927.6558323	26.79564598
	1500	1344.941071	52.91743689
	2000	1542.215053	59.57658954
MongoDB	500	499.8812782	10.60350339
	1000	999.1746817	27.48471952
	1500	1426.926386	32.64052776
	2000	1480.325731	35.93146708
MySQL	500	499.5361807	10.15568306
	1000	998.2650154	13.38312674
	1500	1496.412351	22.59208354
	2000	1992.012032	16.14211225

Table A.18 Workload F Read Large Data Set

Workload F Read-Modify-Write Small Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	2000	1989.970548	5.135042276
	4000	3960.615638	9.508330247
	6000	5912.716479	17.70557296
	8000	6525.966822	18.67221296
	10000	6820.212381	17.83963338
HBase	2000	1752.980505	1.317495429
	4000	3106.796117	1.592910613
	6000	4239.102328	1.674520122
	8000	5014.190158	2.165678557
	10000	5938.947618	2.200675277
MongoDB	2000	1998.812705	4.741707221
	4000	3995.604835	5.118855844
	6000	5990.678504	5.663026287
	8000	7982.948422	6.132298852
	10000	9974.166908	8.166800531
MySQL	2000	1992.758316	1.201471255
	4000	3967.8131	2.583506378
	6000	5917.75504	5.111744704
	8000	7865.097842	5.86811079
	10000	9817.685579	5.982929808

Table A.19 Read-Modify-Write Small Data Set

Workload F Read-Modify-Write Large Data Set			
Database	**Target Throughput**	**Throughput**	**Average Latency**
Cassandra	500	499.2017764	10.3764697
	1000	996.931445	14.65446572
	1500	1493.132338	16.08305704
	2000	1988.004382	16.24818503
HBase	500	480.863554	23.48434334
	1000	927.6558323	26.81769759
	1500	1344.941071	53.1235534
	2000	1542.215053	59.71581907
MongoDB	500	499.8812782	20.07441485
	1000	999.1746817	71.47608397
	1500	1426.926386	83.84459524
	2000	1480.325731	102.2553256
MySQL	500	499.5361807	18.39277771
	1000	998.2650154	25.42677039
	1500	1496.412351	47.03104067
	2000	1992.012032	33.64840722

Table A.20 Read-Modify-Write Large Data Set

Appendix B: Sample Test Script

The bash script used to automate the benchmark for Cassandra large data set is provided.

```
#!/bin/bash

./bin/ycsb load cassandra-cql -P workloads/workloada -p threadcount=100 -p hosts=172.31.31.239 -p port=9042 -s > load.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloada -p threadcount=100 -p target=500 -p hosts=172.31.31.239 -p port=9042 -s > runA-500.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloada -p threadcount=100 -p target=1000 -p hosts=172.31.31.239 -p port=9042 -s > runA-1000.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloada -p threadcount=100 -p target=1500 -p hosts=172.31.31.239 -p port=9042 -s > runA-1500.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloada -p threadcount=100 -p target=2000 -p hosts=172.31.31.239 -p port=9042 -s > runA-2000.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadb -p threadcount=100 -p target=500 -p hosts=172.31.31.239 -p port=9042 -s > runB-500.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadb -p threadcount=100 -p target=1000 -p hosts=172.31.31.239 -p port=9042 -s > runB-1000.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadb -p threadcount=100 -p target=1500 -p hosts=172.31.31.239 -p port=9042 -s > runB-1500.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadb -p threadcount=100 -p target=2000 -p hosts=172.31.31.239 -p port=9042 -s > runB-2000.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadc -p threadcount=100 -p target=500 -p hosts=172.31.31.239 -p port=9042 -s > runC-500.txt
```

```
sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadc -p threadcount=100 -p target=1000 -p hosts=172.31.31.239 -p port=9042 -s > runC-1000.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadc -p threadcount=100 -p target=1500 -p hosts=172.31.31.239 -p port=9042 -s > runC-1500.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadc -p threadcount=100 -p target=2000 -p hosts=172.31.31.239 -p port=9042 -s > runC-2000.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadf -p threadcount=100 -p target=500 -p hosts=172.31.31.239 -p port=9042 -s > runF-500.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadf -p threadcount=100 -p target=1000 -p hosts=172.31.31.239 -p port=9042 -s > runF-1000.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadf -p threadcount=100 -p target=1500 -p hosts=172.31.31.239 -p port=9042 -s > runF-1500.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadf -p threadcount=100 -p target=2000 -p hosts=172.31.31.239 -p port=9042 -s > runF-2000.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadd -p threadcount=100 -p target=500 -p hosts=172.31.31.239 -p port=9042 -s > runD-500.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadd -p threadcount=100 -p target=1000 -p hosts=172.31.31.239 -p port=9042 -s > runD-1000.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadd -p threadcount=100 -p target=1500 -p hosts=172.31.31.239 -p port=9042 -s > runD-1500.txt

sleep 300

./bin/ycsb run cassandra-cql -P workloads/workloadd -p threadcount=100 -p target=2000 -p hosts=172.31.31.239 -p port=9042 -s > runD-2000.txt
```

Appendix C: Sample Output File

A shortened output file from Cassandra's workload A benchmark is provided as a sample.

YCSB Client 0.1

Command line: -db com.yahoo.ycsb.db.CassandraCQLClient -P workloads4/workloada -p threadcount=100 -p target=500 -p hosts=172.31.31.239 -p port=9042 -s -t

Connected to cluster: cassandra4

Datacenter: us-west-2; Host: /172.31.31.241; Rack: 2a

Datacenter: us-west-2; Host: /172.31.31.240; Rack: 2a

Datacenter: us-west-2; Host: /172.31.31.238; Rack: 2a

Datacenter: us-west-2; Host: /172.31.31.239; Rack: 2a

[OVERALL], RunTime(ms), 2003098.0

[OVERALL], Throughput(ops/sec), 499.226697845038

[UPDATE], Operations, 500400

[UPDATE], AverageLatency(us), 1381.2922901678658

[UPDATE], MinLatency(us), 591

[UPDATE], MaxLatency(us), 226928

[UPDATE], 95thPercentileLatency(ms), 1

[UPDATE], 99thPercentileLatency(ms), 11

[UPDATE], Return=0, 500400

[UPDATE], 0, 119779

[UPDATE], 1, 366554

[UPDATE], 2, 2088

[UPDATE], 3, 1347

[UPDATE], 4, 1060

[UPDATE], 5, 835

…

[READ], Operations, 499600

[READ], AverageLatency(us), 7714.154647718175

[READ], MinLatency(us), 780

[READ], MaxLatency(us), 591205

[READ], 95thPercentileLatency(ms), 25

[READ], 99thPercentileLatency(ms), 52

[READ], Return=0, 499600

[READ], 0, 17

[READ], 1, 230840

[READ], 2, 74911

[READ], 3, 2650

[READ], 4, 5975

[READ], 5, 8817

…

[CLEANUP], Operations, 100

[CLEANUP], AverageLatency(us), 140.66

[CLEANUP], MinLatency(us), 2

[CLEANUP], MaxLatency(us), 13827

[CLEANUP], 95thPercentileLatency(ms), 0

[CLEANUP], 99thPercentileLatency(ms), 0

[CLEANUP], 0, 99

[CLEANUP], 1, 0

[CLEANUP], 2, 0

[CLEANUP], 3, 0

[CLEANUP], 4, 0

[CLEANUP], 5, 0

…

java -cp /home/ubuntu/YCSB/voldemort-binding/lib/checkstyle-5.7-all.jar:/home/ubuntu/YCSB/voldemort-binding/lib/commons-codec-1.9.jar:/home/ubuntu/YCSB/voldemort-binding/lib/guava-16.0.1.jar:/home/ubuntu/YCSB/voldemort-binding/conf:/home/ubuntu/YCSB/hbase-binding/conf:/home/ubuntu/YCSB/jdbc-binding/conf:/home/ubuntu/YCSB/cassandra-

binding/lib/cassandra-binding-0.1.4.jar:/home/ubuntu/YCSB/core/lib/core-0.1.4.jar com.yahoo.ycsb.Client -db com.yahoo.ycsb.db.CassandraCQLClient -P workloads4/workloada -p threadcount=100 -p target=500 -p hosts=172.31.31.239 -p port=9042 -s –t

Appendix D: Computing Cost

At the time of this thesis, the price of an m1.xlarge on demand instance was $0.480 per hour and spot instances varied from $0.0875 - $0.250 per hour.

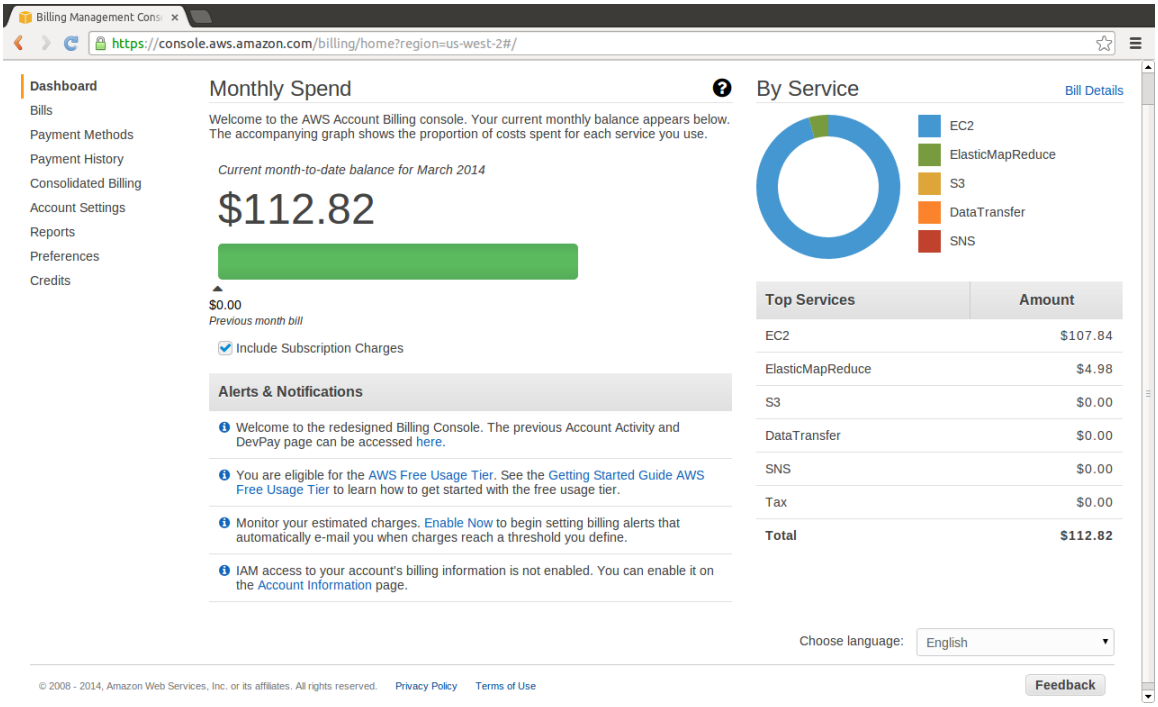

Figure D.1 Computing Cost