Imperial College London

Department of Computing

Benchmarking Replication in NoSQL Data Stores

by

Gerard Haughian (gh413)

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science of Imperial College London

September 2014

Abstract

The proliferation in Web 2.0 applications has increased both the volume, velocity, and variety of data sources which have exceeded the limitations and expected use cases of traditional relational DBMSs. Cloud serving NoSQL data stores address these concerns and provide replication mechanisms to ensure fault tolerance, high availability, and improved scalability. However, no research or studies exist which emperically explore the impact of replication on data store performance.

As such this study paves the way for a new replication benchmarking tier for YCSB, a leading benchmark tool for cloud serving data stores, by benchmarking replication in four NoSQL data stores of variable categorization and replication strategy. These data stores include: Redis, a key-value store that uses a master-slave replication model; MongoDB, a document store with replica set replication; Cassandra, an extensible-record store with a multi-master replica architecture; and VoltDB, a distributed DBMS which utilizes a synchronous multi-master replication strategy. This study focuses on the impact of replication on performance and availability compared to non-replicated clusters on constant amounts of hardware. To increase the relevancy of this study to real-world use cases, experiments include different workloads, distributions, and tunable consistency levels, on clusters hosted on a private cloud environment.

This study presents an in-depth analysis of the overall throughput, read latencies, and write latencies of all experiments per data store. Each analysis section concludes with a presentation and brief analysis of the latency histogram and CDF curves of reads and writes to aid predictions on how each data store should behave. Subsequently, a comparative analysis is performed between data stores to identify the most performant data store and replication model.

This study should serve as a point of reference for companies attempting to choose the right data store and replication strategy for their use cases. It also highlights the need to investigate the effects of geo-replication in more detail due to its importance in ensuring high availability in the face of network partitions and data center failures. Additionally, the impact that replication has on the consistency of data is of importance and warrants further attention. By making the data collected in this study publicly available, we encourage further research in an area with considerable research gaps: performance modelling of NoSQL data stores.

Acknowledgements

A special thanks to my supervisor, Rasha Osman, for her unyielding support and enthusiasm for this work and her fantastic guidance throughout.

Dedication

To Paddy, Una, Luke, Eamon, and Ewa who, as always, have proven the rock that bonds us remains so pure and unbreakable. With their belief and understanding, much has been and will continue to be achieved. Thanks for never giving up on me.

Contents

List of Tables

List of Figures

1 Introduction

Traditional relational database systems of the 1970s were designed to suit the requirements of On-line Transactional Processing (OLTP) applications as "one-size-fits-all" solutions [69]. These systems are typically hosted on a single server, where database administrators respond to increases in data set sizes by increasing the CPU compute power, the amount of available memory, and the speed of hard disks on that single server, *i.e.*, by scaling vertically. Relational database systems still remain relevant in today's modern computing environment, however the limitations of vertical scalability is of greatest concern.

The volume of data consumed by many organizations in recent years has now considerably outgrown the capacity of a single server, due to the explosion of the web [32]. Therefore, new technologies and techniques had to be developed to address what has become known as the *Big Data* era. In 2010, it was claimed that Facebook hosted the world's largest data warehouse hosted on HDFS (Hadoop's distributed file system), comprising 2000 servers, consuming 21 Petabyte's (PB) of storage [7], which rapidly grew to 100 PB as early as 2012 [62].

This situation is further complicated by the fact that traditional OLTP applications remain only a subset of the use cases that these new technologies must facilitate, proliferated by a diverse range of modern industry application requirements [14]. Subsequently, Big Data technologies are required to scale in a new way to overcome the limitations of a single server's CPU compute power, memory capacity, and disk I/O speeds. Horizontal scalability has therefore become the new focus for data store systems; offering the ability to spread data across and serve data from multiple servers.

Elasticity is a property of horizontal scalability, which enables linear increases in throughput as more machines are added to a cluster[1]. The modern computing environment has also seen an increase in the number of cloud computing service providers. Many of which are inherently elastic, for example Amazon's AWS [64] and Rackspace [57], which provide horizontal scalability with ease and at low cost to the user[2].

Many new data stores have been designed with this change of landscape in mind; some of which have even been designed to work exclusively in the cloud, for example Yahoo's PNUTS [13], Google's BigTable [11], and Amazon's DynamoDB [66]. These new data store systems are commonly referred to as NoSQL data stores, which stands for "Not Only SQL"; since they do not use the Structured Query Language (SQL)[3], or have a relational model[4]. Hewitt [32] further suggests that this term also means that traditional systems should not be the only choice for data storage.

There exists a vast array of NoSQL data store solutions, and companies that turn to these solutions to contend with the challenges of data scalability, are faced with the initial challenge of choosing the best one for their particular use case. The data which companies gather and collect is highly valuable and access to this data often needs to be highly available [27]. The need for high availability of data becomes particularly evident in the Web 2.0[5] applications which we ourselves may have become accustomed to interacting with on a daily basis, for example social media platforms like Facebook and Twitter.

A feature of horizontal scalability is the lower grade of servers which form a cluster. These servers are typically commodity hardware which have inherently higher failure rates due to their cheaper components. A larger number of servers also increases the frequency of experiencing a node failure. The primary mechanism in which NoSQL data stores offer high availability in order to overcome a higher rate of failure and meet industry expectations is through replication.

[1] Throughout this report, the term "cluster" refers to a collection of servers or nodes connected together with a known topology, operating as a distributed data store system.

[2] On Amazon's S3 service, the cost of storing 1 GB of data is only $0.125. For more detail see: `http://docs.aws.amazon.com/gettingstarted/latest/wah/web-app-hosting-pricing-s3.html` Last Accessed: 2014.07.07

[3] SQL is a feature-rich and simple declarative language for defining and manipulating data in traditional relational database systems.

[4] The relational model is a database model based on first order logic, were all data is represented as tuples and grouped into relations. Databases organized in this way are referred to as relational databases.

[5] Web 2.0 as described by O'Reilly [50] : "Web 2.0 is the network as [a] platform [...]; Web 2.0 applications are those that make the most of the intrinsic advantages of that platform: delivering software as a continually-updated service that gets better the more people use it, consuming and remixing data from multiple sources, [...], creating network effects through an 'architecture of participation', and going beyond the page metaphor of Web 1.0 to deliver rich user experiences.".

Replication not only offers higher redundancy in the event of failure but can also help avoid data loss (by recovering lost data from a replica), and improve performance (by spreading load across multiple replicas) [14].

1.1 Aims and Objectives

The aims and objectives of this study are:

- Offer insights based on the quantitative analysis of benchmarking results on the impact replication has on four different NoSQL data stores of variable categorization and replication model.
- Illustrate the impact replication has on performance and availability of cloud serving NoSQL data stores on various cluster sizes relative to non-replicated clusters of equal sizes.
- Evalute, through the application of both read- and write-heavy workloads the impact of each data stores underlying optimizations and design decisions within replicated clusters.
- Explore the impact on performance within replicated clusters of three varying (in terms of strictness) levels of tunable consistency including ONE, QUORUM[6], and ALL.
- Construct a more comprehensive insight into each data store's suitability to different industry applications by experimenting with three different data distributions, each simulating a different real-world use case.
- Help alleviate the challenges faced by companies and individuals choosing an appropriate data store to meet their scalability and fault-tolerance needs.
- Provide histograms and CDF curves as a starting point for future research into performance modeling of NoSQL data stores.

1.2 Contributions

This study extends the work of Cooper *et al.* [14] who created a benchmarking tool (YCSB) specifically for cloud serving NoSQL data stores. Cooper *et al.* highlighted four important measures required for adding a fourth tier to their YCSB tool for benchmarking replication. Those four measures were: performance cost and benefit; availability cost or benefit; freshness (how consistent data is); and wide area performance (the effect of geo-replication[7]). As of yet none of these measures have been addressed. Subsequently, the first two measures form the basis of this study. These measures assess the performance and availability impact as the replication factor[8] is increased on a constant amount of hardware [14].

One of the biggest challenges faced by researchers who attempt performance modeling is attaining reliable data sources on the performance characteristics of the data stores they are attempting to model. This study aims to reduce the barrier to entry for future research into performance modeling of cloud serving NoSQL data stores by making the data collected in this study publicly available and providing elementary analysis based on latency histograms and CDF curves of all experiments conducted in this study.

Further research in this field unlocks the potential to enable service providers to improve system deployments and user satisfaction. By using performance prediction techniques, providers can estimate NoSQL data store response times considering variable consistency guarantees, workloads, and data access patterns.

This area of research could also be of additional benefit to cloud hosting service providers. By helping them gain more in-depth insight into how best to provision their systems based on the intricate details on the data store systems they support. They can use this information to guarantee response times, lower service level agreements, and hopefully reduce potential negative impacts on revenue caused by high latencies [42].

[6] Quorum consistency exists when the majority of replica nodes in a cluster respond to an operation.

[7] The term "geo-replication" refers to a replication strategy which replicates data to geographically separated data centers.

[8] The term "replication factor" indicates the number of distinct copies of data that exist within a cluster.

1.3 Report Outline

First an in-depth look into the existing work done within the NoSQL data store benchmarking sphere will be conducted. Additional literature and studies which provided inspiration and informative guidance for this study will also be presented. It is promising that the existing literature highlights a lack of studies that directly address benchmarking replication, and as such is one of this study's primary contributions to the subject area.

The next section will then be dedicated to describing the NoSQL data stores that have been chosen for this study, taking each data store in turn and describing their major design choices and tradeoffs, focusing particularly on how they handle replication.

Following a discussion of each data store, specific details on how each was configured for optimal performance for this study will be described. Additional extensions were made to the YCSB tool in order to support additional benchmarking features. Subsequently, these extensions will be described in greater detail also. Finally, a complete list of all the experiments conducted in this study are listed, and a description of the exact steps taken to perform each experiment will be described under the Methodology subsection.

The major focus of this report will be an evaluation of the results gathered from the experiments. This evaluation subsequently forms the primary focus of Section 5. Each data store will be considered separately initially, before a comparative analysis is conducted to investigate the relative performance of each data store and replication strategy. This analysis will focus on the impact replication has on the performance and availability of a cluster compared to non-replicated clusters of equal size by considering throughput, and read & write latency data. Points of interest that contradict trends or warrant comment will also be discussed in detail. Finally histograms of read & write latencies are plotted along with their corresponding Cumulative Distribution Function (CDF) curves to aid interpretation of response times and act as a starting point for future research into performance modelling of NoSQL data stores.

The major findings and insights that this work has provided are reiterated in the concluding section. This section also indicates key areas for future development while highlighting some of the current limitations of this study which will be addressed in line with future work.

2 Related Work

Brewer's CAP theorem [8], is one of the key distinguishing design choices of NoSQL data stores in comparison to traditional relational databases. These systems trade off ACID compliance[9] with BASE[10] semantics in order to maintain a robust distributed system [8]. Brewer argues that one must choose two of the three underlying components of his theorem, *i.e.* between; Consistency, Availability, and network Partition-tolerance.

Most NoSQL data stores have more relaxed consistency guarantees due to their BASE semantics and CAP theorem tradeoffs (in comparison to strictly consistent ACID implementations of traditional relational DBMSs). However, the majority of NoSQL data stores offer ways of tuning the desired consistency level to ensure a minumum number of replicas acknowledge each operation. Data consistency is an important consideration in data store systems since different levels of consistency play an important role in data integrity and can impact performance when data is replicated multiple times across a cluster.

There are various approaches to replication including synchronous and asynchronous replication. While synchronous replication ensures all copies are up to date, it potentially incurs high latencies on updates, and can impact availability if synchronously replicated updates cannot complete while some replicas are offline. Asynchronous replication on the other hand avoids high write latencies but does allow replicas to return stale data.

As such, each NoSQL data store varies in its choice and handling of these components, along with having their own distinct optimizations and design tradeoffs. This subsequently raises the challenge for companies choosing the perfect fit for their use case. Cattell's study [10] highlights that a users prioritization of features and scalability requirements differ depending on their use case, when choosing a scalable SQL or NoSQL data store. Subsequently concluding that not all NoSQL data stores are best for all users.

In order to gain an appreciation for the performance characteristics of various database systems, a number of benchmarking tools have been created to facilitate this. The following subsection describes a number of key benchmarks that are used for both traditional relational DBMSs and NoSQL data stores in turn.

2.1 Benchmarking Tools

There are a number of popular benchmarking tools that are designed predominantly for traditional database systems including the TPC suite of benchmarks [5] and the Wisconsin benchmark [21], both of which are described below. However, Binning *et al.* [6] argues that traditional benchmarks are not sufficient for analyzing cloud services, suggesting several ideas which better fit the scalability and fault-tolerance characteristics of cloud computing. Subsequently, several benchmarks are described that have been designed specifically for NoSQL data stores.

2.1.1 Benchmarking Traditional Systems

Each TPC benchmark is designed to model a particular real-world application including transaction processing (OLTP) applications with benchmarks TPC-C and TPC-E, decision support systems with benchmarks TPC-D and TPC-H, database systems hosted in virtualized environments with benchmark TPC-VMS (which consists of all four benchmarks just mentioned), and most recently a Big Data benchmark TPCx-HS for profiling different Hadoop layers [5].

The Wisconsin benchmark on the other hand benchmarks the underlying components of a relational database, as a way to compare different database systems. While not as popular as it once was, it still remains as a robust single-user evaluation of the basic operations that a relation system must provide, while highlighting key performance anomalies. The benchmark is now used to evaluate the sizeup, speedup and scaleup characteristics of parallel DBMSs [21].

[9] To be ACID compliant a data store must always ensure the following four characteristics hold for all transactions: Atomicity, Consistency, Isolation, and Durability.

[10] BASE stands for: Basically Available, Soft-state, and Eventual Consistency.

2.1.2 Benchmarking NoSQL Systems

In 2009, Pavlo *et al.* [52] created a benchmarking tool designed to benchmark two approaches to large scale data analysis: Map-Reduce and parallel database management systems (DBMS) on large computer clusters. Map-Reduce is a programming model and associated implementation which parallelizes large data set computations across large-scale clusters, handling failures and scheduling inter-machine communication to make efficient use of networks and disks [19]. Parallel DBMSs represent relational database systems which offer horizontal scalability in order to mange much larger data set sizes.

YCSB

The Yahoo Cloud Serving Benchmark (YCSB) was developed and open-sourced[11] by a group of Yahoo! engineers to support benchmarking clients for many NoSQL data stores. The YCSB tool implements a vector based approach highlighted by Seltzer *et al.* [63] as one way of improving benchmarks to better reflect application-specific performance [14]. Cooper *et al.* [14] further adds that the YCSB tool was designed for database systems deployed on the cloud, with an understanding that these systems don't typically have an SQL interface, they support only a subset of relational operations[12], and whose use cases are often very different to traditional relational database applications, and subsequently are ill suited to existing tools that are used to benchmark such systems.

The YCSB Core Package was designed to evaluate different aspects of a system's performance, consisting of a collection of workloads to evaluate a system's suitability to different workload characteristics at varying points in the performance space [14].

Central to the YCSB tool is the YCSB Client, which is a Java program that generates the data to be loaded into a data store and the operations that make up a workload. Cooper *et al.* [14] explain that the basic operation of the YCSB Client is for the workload executor to drive multiple client threads. Each thread executing a sequential series of operations by making calls to the database interface layer, both to load the database (the load phase) and to execute the workload (the run phase). Additionally, each thread measures the latency and achieved throughput of their operations, and report these measurements to the statistics module which aggregates all the results at the end of a given experiment.

When executed in load mode, the YCSB Client inserts a user specified number of randomly generated records, containing 10 fields each 100 bytes in size (totalling 1KB), into a specific data store with a specified distribution. YCSB supports many different types of distributions including uniform, zipfian, and latest which determine what the overall distribution of data will be when inserted into the underlying data store.

In run mode, the user specified number of records, and all columns of those records are read or only one record updated depending on the current operation. The chosen distribution again plays a role in determining the likelihood of certain records being read or updated.

Each distribution models the characteristics of a different real world use case, which makes them instructive to include in this study. A brief summary of each, as formalized by Cooper *et al.* [14] follows.

Uniform: Items are distributed uniformly at random. This form of distribution can be useful to model applications where the number of items associated with a particular event can have a variable number of items, for example blog posts.

Zipfian: Items are distributed according to popularity. Some items are extremely popular and will be at the head of the list while most other records are unpopular and will be placed at the tail of the list. This form of distribution models social media applications where certain users are very popular and have many connections, regardless of how long they have been a member of that social group.

[11] Available at `https://github.com/brianfrankcooper/YCSB`

[12] Most NoSQL data stores support only 'CRUD' operations i.e. Create, Read, Update, and Delete.

Latest: Similar to the zipfian distribution however, items are ordered according to insertion time. That is, the most recently inserted item will be at the head of the list. This form of distribution models applications where recency matters, for example news items are popular when they are first released but popularity quickly decays over time.

YCSB currently offers two tiers for evaluating the performance and scalability of NoSQL data stores. The first tier (Performance) focuses on the latency of requests when the data store is under load. Since there is an inherent tradeoff between latency and throughput the Performance tier aims to characterize this tradeoff. The metric used in this tier is similar to *sizeup* from [21]. The second tier (Scaling) focues on the ability of a data store to scale elastically in order to handle more load as data sets and application popularity increases. There are two metrics to this tier, including Scaleup, which measures how performance is affected as more nodes are added to a cluster, and Elastic Speedup, which assess performance of a system as nodes are added to a running cluster. These metrics are similar to *scaleup* and *speedup* from [21], respectively.

The YCSB benchmarking tool has become very popular for benchmarking and drawing comparisons between various NoSQL data stores, as illustrated by a number of companies that have made use of it [15, 22, 48, 59], and several academic benchmarks [20, 53, 56] including Yahoo's original benchmark [14], conducted prior to the release of the YCSB tool. These studies are discussed in greater detail below along with a few additional studies that extend the YCSB tool, and finally some auxiliary studies which focus on replicationin in other domains will also be presented.

2.2 Academic YCSB Benchmarking Studies

In their original YCSB paper, Cooper *et al.* [14] performed a benchmarking experiment on four data stores to illustrate the tradeoffs of each system and highlight the value of the YCSB tool for benchmarking. Two of the data stores used in their experiments shared similar data models but differed architecturally: HBase [31] and Cassandra [9]. The third; PNUTS, which differs entirely in its data model and architecture to all other systems was included, along with a *sharded*[13] MySQL database which acted as a control in their experiments. The sharded MySQL system represents a conventional relational database and contrasts with the cloud serving data stores which YCSB was designed specifically to benchmark. They report average throughput's on read-heavy, write-heavy, and short scan workloads. Replication was disabled on all systems in order to benchmark baseline performances only. The versions of each data store were very early versions, some of which have seen considerable improvements over the past few years. Nonetheless, the authors found that Cassandra and PNUTS scaled well as the number of servers and workload increased proportionally[14] and their hypothesized tradeoffs between read and write optimization were apparent in Cassandra and HBase *i.e.* they both had higher read latencies, but lower update latencies.

Pirzadeh *et al.* [53] evaluated range query dominant workloads with three different data stores using YCSB as their benchmarking tool. The three data stores they evaluated included: Cassandra, HBase, and Voldemort [74]. The focus of this study was on real-world applications of range queries beyond the scope of batch-oriented Map-Reduce jobs. As such, a number of extensions to the YCSB tool were implemented to perform extensive benchmarking on range queries that weren't (and still aren't) currently supported by the YCSB tool. However, experimenting with different data consistency levels and replication factors was beyond the scope of this study. Pirzadeh *et al.* state that their observation of no clear winner in their results implies the need for additional physical design tuning on their part [53].

The study conducted by Rabl *et al.* [56] looks at three out of the four categories of NoSQL data stores, as categorized by Stonebraker & Cattell [69]. This study choose to exclude Document stores due to the lack of available systems that matched their requirements at the time. They also excluded replication, tunable consistency and different data distributions in their ex-

[13] *Sharding* refers to the practice of splitting data horizontally and hosting these distinct portions of data on separate servers.

[14] The cluster size was gradually increased from two to twelve nodes.

periments. The lack of a stable release of Redis Cluster meant their experiments on a multi-node Redis [58] cluster was implemented on the client-side via a *sharded* Jedis[15] library. This led to sub-optimal throughput scalability since the Jedis library was unable to balance the workload efficiently. Rabl *et al.* [56] state that Cassandra is the clear winner in terms of scalability, achieving the highest throughput on the maximum number of nodes in all experiments, maintaining a linear increase from 1 to 12 nodes. They conclude also that Cassandra's performance is best suited to high insertion rates. Rabl *et al.* claim comparable performance between Redis and VoltDB [75], however they were only able to configure a single node VoltDB cluster.

Dede *et al.* [20] evaluated the use of Cassandra for Hadoop [29], discussing various features of Cassandra, such as replication and data partitioning which affect Hadoop's performance. Dede *et al.* concentrate their study on the Map-Reduce paradigm which is characterized by predominantly read intensive workloads and range scans. As such, only the provided C[16] workload from YCSB was utilized in this study and again does not take into account variable consistency levels or data distributions. Dede *et al.*'s approach to consistency was to use the default level of ONE, which does not account for potential inconsistencies between replicas and would result in better throughputs. The extent of benchmarking replication in this study was limited and did not explore the impact replication had on Cassandra clusters versus non-replicated clusters of equal size, regarding any of the four key properties highlighted by Cooper *et al.* [14] as important measures when benchmarking replication. However, the study does report encouraging findings, claiming that increasing the replication factor to eight only resulted in 1.1 times slower performance when Hadoop was coupled with Cassandra.

All of these studies do not address the impact replication had on performance and the availability of a cluster, by comparing clusters of variable replication factors to non-replicated clusters of equal size. These studies also do not experiment with various data distributions which model real-world use cases or the tradeoffs that variable consistency level settings have.

2.3 Industry YCSB Benchmarking Studies

Several companies that are active within the NoSQL community have performed their own in-house benchmarks on various systems. It is important to note that most of these companies have strategic and/or commercial relationships with various NoSQL data store providers. For example, Datastax is a leading enterprise Cassandra provider [18], a data store which they subsequently benchmark in [15]. Likewise, Altoros Systems have a proven track record serving technology leaders including Couchbase which they benchmark in [22]. Thumbtack Technologies state they have strategic and/or commercial relationships with Aerospike and 10gen (the makers of MongoDB). Aerospike also sponsored the changes to the YCSB tool, and rented the hardware needed for their benchmarks [47, 48] discussed below.

Three out of the four industry benchmarks previously highlighted [15, 22, 48], focus particularly on Document stores in contrast to the academic studies. Two of these benchmarks [22, 48] include the same two Document stores: Couchbase [16] and MongoDB [45], and the same Extensible-Record store: Cassandra. Aerospike [2], a proprietary NoSQL data store optimized for flash storage (*i.e.* Solid-state Drives (SSD)) was also included in [48]. These two benchmarks however are very narrow in the problem domain they seek to highlight by highly optimizing their studies for specific use cases on small clusters with limited variation in the type of experiments conducted.

In [22] the authors modelled an interactive web application looking at a single workload comprising a 5-60-33-2% CRUD decomposition of in-memory operations only. They had fixed replication factors of two set for MongoDB and Couchbase.

The authors of [48], focused on highly optimizing their chosen data stores for extremely high loads on two of YCSB's standard workloads (A[17] and B[18]). They intentionally made use of SSDs for disk-bound operations due to a total data set size of approximately 60GB which was too large to fit entirely in memory. They enabled synchronous replication on all data stores except

[15] Jedis is a popular Java driver for Redis. The YCSB tool uses this driver to interact with Redis.
[16] Workload C has a 100% composition of read operations only.
[17] Workload A has a 50/50% breakdown of read/write operations.
[18] Workload B has a 95/5% breakdown of read/write operations.

Couchbase, with a replication factor of two.

The third industry benchmark [15], used MongoDB and two Extensible-Record stores: Cassandra and HBase in their experiments. This was a much more extensive benchmark than the others considering it used seven different workloads (a mixture of both customized and YCSB provided ones), on a thirty-two node cluster, and a total data set size twice that of available RAM, requiring both disk-bound and memory-bound operations. Replication was not enabled or explored in the experiments and neither were different distributions and consistency levels. The results indicate that Cassandra consistently out performed MongoDB and HBase on all tests and cluster sizes.

Finally, the fourth and most recent industry benchmark [59] looks exclusively at benchmarking VoltDB, and was conducted by the in-house developers at VoltDB, Inc. This benchmark explores three different built-in YCSB workloads with a constant replication factor of two. Results indicated a linear increase in throughput as the cluster size increased to a total of twelve nodes. This study did not include experiments with different data distributions or variable replication factors.

All of these studies no not address the impact replication had on performance and the availability of a cluster, by comparing clusters of variable replication factors to non-replicated clusters of equal size. A constant replication factor was used in all studies, and no comparison was done to evaluate the impact this replication factor had compared to baseline performances. These studies also do not experiment with various data distributions or variable consistency settings either, being highly optimized for specific use cases.

2.4 Extended YCSB Benchmarking Studies

Cooper *et al.* [14] indicated two key areas for future work, only one of which, a third tier for benchmarking availability, has been actively pursued by Pokluda & Sun [55], and Nelubin & Engber at Thumbtack Technologies [47].

Pokluda & Sun's study [55] include benchmarking results for both standard YCSB benchmarking tier's (Performance and Scalability) and in addition, provide an analysis of the failover characteristics of two systems: Voldemort and Cassandra. Both systems were configured to make use of a fixed replication factor, however variations in replication, consistency levels, and data distributions were not explored further. The focus of the study was primarily on how each system handled node failure within a cluster operating under different percentages of max throughput (50% and 100%), with varying amounts of data (1 million and 50 million records), and six different YCSB built-in workloads.

Nelubin & Engber [47] performed similar experiments for benchmarking failover recovery using the same data stores as in their previous paper discussed above: [48]. In contrast to Pokluda & Sun's work, they included experiments which looked at the impact of different replication strategies *i.e.* between synchronous and asynchronous replication to see how node recovery was affected when replicating data to recovered nodes in various ways. A fixed replication factor of two was used and experiments into how replication affected read/write performance was not evaluated and subsequently neither were different data distributions. Both memory-bound and disk-bound tests were performed, with three different percentages of max throughput (50%, 75%, and 100%), for a single workload (50/50 read/write) on a four node cluster.

2.5 Additional Studies

Ford *et al.* [24] analyzed the availability of nodes in globally distributed storage systems, including an evaluation on how increasing replication factors affect the chance of node failure and therefore the overall availability of the cluster based on data collected at Google on the failures of tens of their clusters.

García-Recuero *et al.* [26] introduced a tunable consistency model for geo-replication in HBase. They used the YCSB tool to benchmark performance under different consistency set-

tings by performing selective replication. They were able to successfully maintain acceptable levels of throughput, reduce latency spikes and optimize bandwidth usage during replication. The authors however did not explore the effect of changing the replication factor or access distribution on their results.

Müller *et al.* [46] first proposed a benchmarking approach to determine the performance impact of security design decisions in arbitrary NoSQL systems deployed in the cloud, followed by performance benchmarking of two specific systems: Cassandra and DynamoDB. Variable replication factors were not explored however.

Osman & Piazzolla [49] demonstrate that a queuing Petri net [4] model can scale to represent the characteristics of read workloads for different replication strategies and cluster sizes for a Cassandra cluster hosted on Amazon's EC2 cloud.

Gandini *et al.* [25] benchmark the impact of different configurations settings with three NoSQL data stores and compare the behavior of these systems to high-level queueing network models. This study demonstrates the relative performance of three different replication factors, however offers limited insight.

2.6 Summary

All of the academic and industry studies presented fail to evalute the impact various replication factors have on the performance and availability of clusters compared to non-replicated clusters on constant amounts of hardware. However, the five additional studies indicate that replication is an important area of interest and research, each of which address many different aspects not directly related to benchmarking. Subsequently, it is reasonable to conclude that there does not currently exist any tangible metrics or benchmarks on the performance impact that replication has on cloud serving NoSQL data stores, which companies and individuals can call upon when making critical business decisions.

3 Systems Under Investigation

In order to encompass a representative NoSQL data store from each of the four categories defined by Stonebraker & Cattell [69], this study focuses on one data store selected from each of the following categories:

- Key-Value Stores - Have a simple data model in common: a map/dictionary, which allows clients to put and request values per key. Most modern key-value stores omit rich ad-hoc querying and analytics features due to a preference for high scalability over consistency. [70].
- Document Stores - The data model consists of objects with a variable number of attributes, some allowing nested objects. Collections of objects are searched via constraints on multiple attributes through a (non-SQL) query language or procedural mechanism [69].
- Extensible Record Stores - Provide variable width record sets that can be partitioned vertically and horizontally across multiple nodes [69].
- Distributed SQL DBMSs - Focus on simple-operation application scalability. They retain SQL and ACID transactions, but their implementations are often very different from those of traditional relational DBMSs [69].

Table 3.1 illustrates the NoSQL data stores that are included in this study based on their categorization and replication strategy. Additional information regarding the properties that they prioritize in terms of Brewer's CAP theorem [8] are also presented for completeness.

Database	Category	Replication Strategy	Properties
Redis	Key-Value	Master-Slave (Asynchronous)	CP
MongoDB	Document	Replica-sets (Asynchronous)	CP
Cassandra	Extensible Record	Asynchronous Multi-Master	AP
VoltDB	Distributed SQL DBMS	Synchronous Multi-Master	ACID[19]

Table 3.1: NoSQL Data Store Choices by Category, and Replication Strategy.

Redis, MongoDB, Cassandra, and VoltDB each fit into a different category of NoSQL data store, and each have distinguishable replication strategies, along with other distinctive features and optimizations. These data stores were chosen to be included in this study in order to cover as much of the spectrum of cloud serving data store solutions as possible. The following subsections illustrate in greater detail the underlying design decisions of each data store in turn.

3.1 Redis

Redis is an in-memory, key-value data store with optional data durability. The Redis data model supports many of the foundational data types including strings, hashes, lists, sets, and sorted sets [61]. Although Redis is designed for in-memory data, data can also be persisted to disk either by taking a snapshot of the data and dumping it onto disk periodically or by maintaining an append-only log (known as an AOF file) of all operations, which can be replayed upon system restart or during crash recovery. Without snapshotting or append-only logging enabled, all data stored in memory is purged when a node is shutdown. This is generally not a recommended setup if persistence and fault tolerance is deemed essential. Atomicity[20] is guaranteed in Redis as a consequence of its single-threaded design, which leads to reduced internal complexity also.

Data in Redis is replicated using a master-slave architecture, which is non-blocking (i.e. asynchronous) on both the master and slave. This enables the master to continue serving queries while one or more slaves are synchronizing data. It also enables slaves to continue servicing read-only queries using a stale version of the data during that synchronization process, resulting in a highly scalable architecture for read-heavy workloads. All write operations however must be administered via the master node. Figure 3.1 illustrates the master-slave architecture of Redis and the operations a client application can route to each component.

[19] VoltDB does not tradeoff CAP components, rather it is fully ACID compliant, maintaining Atomicity, Consistency, Isolation, and Durability for all transactions.

[20] An atomic operation means that all parts of a transaction are completed or rolled-back in an "all or nothing" fashion.

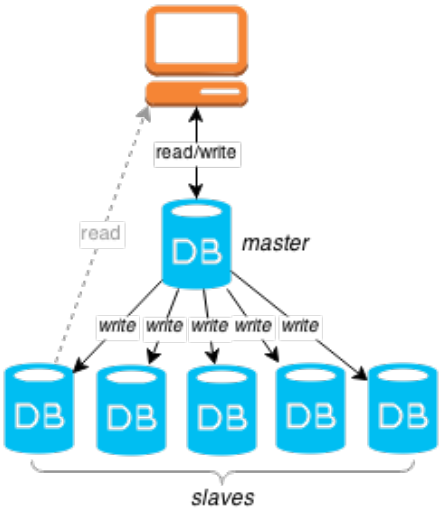

Figure 3.1: Redis Architecture.

Redis offers relaxed tunable consistency guarantees when the *min-slaves-to-write* configuration parameter is passed at server start-up time. This parameter enables an administrator to set the minimum number of slaves that should be available to accept each write operation within a replicated architecture. Redis provides no guarantees however that write operations will succeed on the specified number of slaves.

Redis Cluster is currently in development which, when released as production stable code, will enable automatic partitioning of data across multiple Redis nodes. This will enable much larger data sets to be managed within a Redis deployment and assist higher write throughputs also. Replication is an inherent component of Redis Cluster, having built-in support for node failover and high availability.

Redis is sponsored by Pivotal [54] and an important technology in use by Twitter [72], Github [28], and StackOverflow [68] among others.

3.2 MongoDB

MongoDB is a document-oriented NoSQL data store that stores data in BSON[21] format with no enforced schema's which offers simplicity and greater flexibility.

Automatic *sharding* is how MongoDB facilitates horizontal scalability by auto-partitioning data across multiple servers to support data growth and the demands of read and write operations.

Typically, each shard exists as a replica set providing redundancy and high availability. Replica sets consist of multiple Mongo Daemon (*mongod*) instances, including an arbiter node[22], a master node acting as the primary, and multiple slaves acting as secondaries which maintain the same data set. If the master node crashes, the arbiter node elects a new master from the set of remaining slaves. All write operations must be directed to a single primary instance. By default, clients send all read requests to the master; however, a *read preference* is configurable at the client level on a per-connection basis, which makes it possible to send read requests to slave nodes instead. Varying read preferences offer different levels of consistency guarantees and other tradeoffs, for example by reading only from slaves, the master node can be relieved of undue pressure for write-heavy workloads [36]. MongoDB's sharded architecture is represented in Figure 3.2.

Balancing is the process used to distribute data of a sharded collection evenly across a sharded cluster. When a shard has too many of a sharded collections chunks compared to other shards, MongoDB automatically balances the chunks across the shards. The balancing procedure for sharded clusters is entirely transparent to the user and application layer, and takes

[21] BSON: A JSON document in binary format.

[22] An arbiter node does not replicate data and only exist to break ties when electing a new primary if necessary.

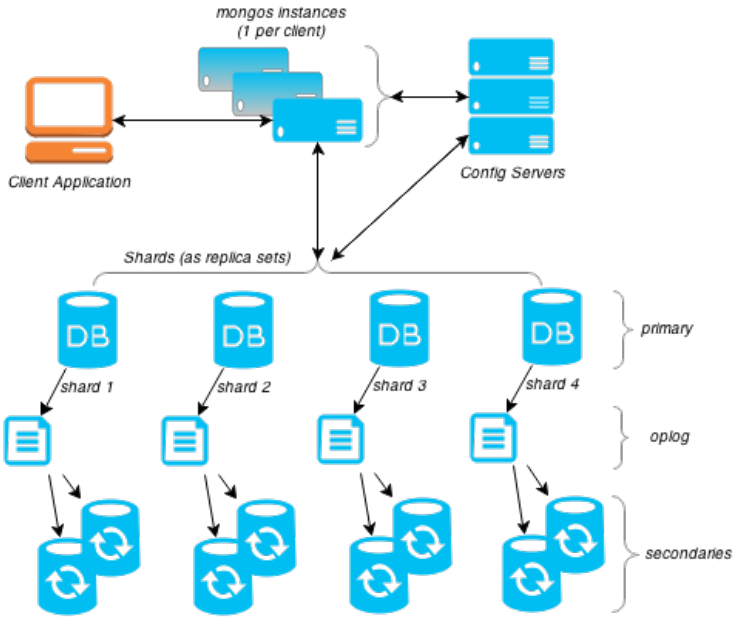

Figure 3.2: MongoDB Architecture.

place within the *mongos* App server (required in sharded clusters) [35].

Replication operates by way of an *oplog*, to which the master node logs all changes to its data sets. Slave nodes then replicate the master's oplog, applying those operations to their data sets. This replication process is asynchronous, so slave nodes may not always reflect the most up to date data. Varying *write concerns* can be issued per write operation to determine the number of nodes that should process a write operation before returning to the client successfully. This allows for fine grained tunable consistency settings, including quorum and fully consistent writes [33].

Journaling is a more recent feature that facilitates faster crash recovery. When a replica set runs with journaling enabled, mongod instances can safely restart without any administrator intervention. Journaling requires some resource overhead for write operations but has no effect on read performance, however.

MongoDB was created by 10gen, and has found multiple use cases including Big Data as used by AstraZeneca [3], Content Management as used by LinkedIn [43] and Customer Data as used by Eventribe [23], among many others.

3.3 Cassandra

Cassandra is a non-relational, distributed extensible record data store, developed at Facebook [41] for storing large amounts of unstructured data on commodity servers. Cassandra's architecture is a mixture of Google's BigTable data model and Amazon's DynamoDB peer-to-peer distribution model [32]. As in Amazon's DynamoDB, every node in the cluster has the same role, and therefore no single point of failure which supports high availability. This also enables a Cassandra cluster to scale horizontally with ease, since new servers simply need to be informed of the address of an existing cluster node in order to contact and retrieve start-up information.

Cassandra offers a column oriented data model in which a column is the smallest component: a tuple of name, value, and time stamp (to assist conflict resolution between replicas). Columns associated with a certain key can be depicted as a row; rows do not have a predetermined structure as each of them may contain several columns. A column family is a collection of rows, like a table in a relational database. The placement of rows on the nodes of a Cassandra cluster depends on the row key and the partitioning strategy. *Keyspaces* are containers for column families just as databases have tables in relational DBMSs [20].

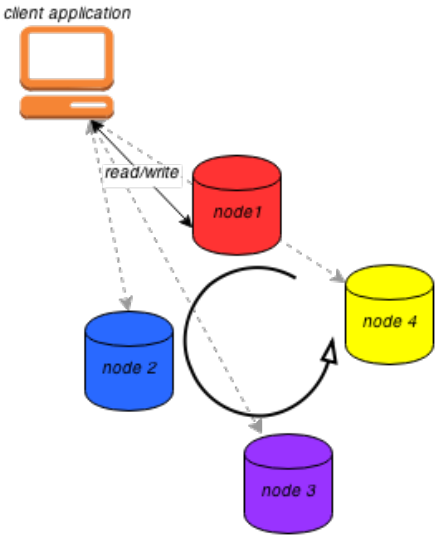

Figure 3.3: Cassandra Architecture.

Cassandra offers three main data partitioning strategies: Murmur3Partitioner, RandomPartitioner, and ByteOrderedPartitioner [32]. The Murmur3Partitioner is the default and recommended strategy. It uses consistent hashing to evenly distribute data across the cluster using an order preserving hash function. Each Cassandra node has a token value that specifies the range of keys for which they are responsible. Distributing the records evenly throughout the cluster balances the load by spreading out client requests.

Cassandra automatically replicates records throughout a cluster determined by a user specified replication-factor and replication strategy. The replication strategy is important for determining which nodes are responsible for which key ranges.

Client applications can contact any one node to process an operation. That node then acts as a coordinator which forwards client requests to the appropriate replica node(s) owning the data being claimed. This mechanism is illustrated in Figure 3.3. For each write request, first a commit log entry is created. Then, the mutated columns are written to an in-memory structure called MemTable. A MemTable, upon reaching its size limit, is committed to disk as a new SSTable by a background process. A write request is sent to all replica nodes, however the consistency level determines how many of them are required to respond for the transaction to be considered complete. For a read request, the coordinator contacts a number of replica nodes specified by the consistency level. If replicas are inconsistent the out-of-date replicas are auto-repaired in the background. In case of inconsistent replicas, a full data request is sent out and the most recent (by comparing timestamps) version is forwarded to the client.
Cassandra is optimized for large volumes of writes as each write request is treated like an in-memory operation, while all I/O is executed as a background process. For reads, first the versions of the record are collected from all MemTables and SSTables, then consistency checks and read repair calls are performed. Keeping the consistency level low makes read operations faster as fewer replicas are checked before returning the call. However, read repair calls to each replica still happen in the background. Thus, the higher the replication factor, the more read repair calls that are required.

Cassandra offers tunable consistency settings, which provide the flexibility for application developers to make tradeoffs between latency and consistency. For each read and write request, users choose one of the predefined consistency levels: ZERO, ONE, QUORUM, ALL or ANY [41].

Cassandra implements a feature called Hinted Handoff to ensure high availability of the cluster in the event of a network partition[23], hardware failure, or for some other reason. A

[23]A network partition is a break in the network that prevents one machine from interacting with another. A network partition can be caused by failed switches, routers, or network interfaces.

hint contains information about a particular write request. If the coordinator node is not the intended recipient, and the intended recipient node has failed, then the coordinator node will hold on to the hint and inform the intended node when it restarts. This feature means the cluster is always available for write operations, and increases the speed at which a failed node can be recovered and made consistent again. [32]

Cassandra is in use at a large number of companies including Accenture [1], Coursera [17], and Spotify [67] among many others for use cases including product recommendations, fraud detection, messaging, and product catalogues.

3.4 VoltDB

VoltDB is a fully ACID compliant relational in-memory data store derived from the research prototype H-Store [40], in use at companies like Sakura Internet [39], Shopzilla [65], and Social Game Universe [73], among others. It has a shared nothing architecture designed to run on a multi-node cluster by dividing the data set into distinct partitions and making each node an owner of a subset of these partitions, as illustrated in Figure 3.4.

While similar to the traditional relational DBMSs of the 1970s, VoltDB is designed to take full advantage of the modern computing environment [38]. It uses in-memory storage to maximize throughput by avoiding costly disk-bound operations. By enforcing serialized access to data partitions (as a result of its single threaded nature), VoltDB avoids many of the time-consuming operations associated with traditional relational databases such as locking, latching, and maintaining transaction logs. Removal of these features have been highlighted by Abadi *et al.* [30] as key ways to significantly improve DBMS performance [24].

The unit of transactions in VoltDB is a stored procedure written and compiled as Java code, which also support a subset of ANSI-standard SQL statements. Since all data is kept in-memory, if stored procedures are directed towards the correct partition, it can execute without any I/O or network access, providing very high throughput for transactional workloads. An additional benefit to stored procedures is that they ensure all transactions are fully consistent, either completing or rolling-back in their entirety.

To provide durability against node failures, *K-safety* is a feature which duplicates data partitions and distributes them throughout the cluster, so that if a partition is lost (due to hardware or software problems) the database can continue to function with the remaining duplicates. This mechanism is analogous to how other data stores replicate data based on a configured replication factor. Each cluster node is responsible for hosting a set number of data partitions, as determined by the *sitesperhost* configuration parameter. All duplicate partitions are fully functioning members of the cluster however, and include all read and write privileges which enables client applications to direct queries to any node in the cluster. Data is synchronously committed to replicated partitions within the cluster before each transaction commits, therefore ensuring consistency. Duplicates function as peers similar to Cassandra's multi-master model rather than as slaves in a master-slave relationship used in Redis and MongoDB, and hence the reason why the architectures in Figures 3.4 (VoltDB) and 3.3 (Cassandra) look very similar.

[24] Abadi *et al.* [30] indicate that each of the following operations account for a certain percentage of the total operations performed by traditional relational DBMSs: Logging(11.9%), Locking(16.3%), Latching(14.2%), and Buffer Management(34.6%).

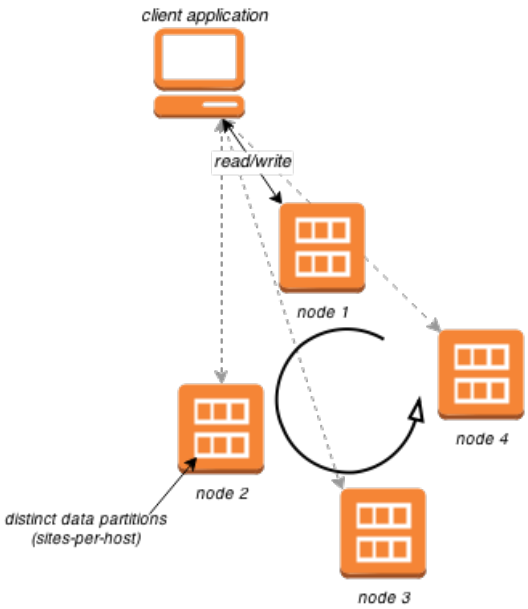

Figure 3.4: VoltDB Architecture.

VoltDB's replication model is similar to K-safety, however, rather than creating redundant partitions within a single cluster, replication creates and maintains a complete copy of the entire cluster in a separate geographic location (*i.e.* geo-replication). The replica cluster is intended to take over only when the primary cluster fails completely, and as such both clusters operate independently of each other. Replication is asynchronous and therefore does not impact the performance of the primary cluster.

VoltDB implements a concept called command logging for transaction-level durability. Unlike write-ahead logs found in traditional systems, VoltDB logs the instantiation of commands only, rather than all subsequent actions. This style of logging greatly reduces the I/O load of the system while providing transaction-level durability either synchronously or asynchronously [37].

4 Experimental Setup

All experiments conducted in this study where carried out on a cluster of Virtual Machines (VM) hosted on a private cloud infrastructure within the same data center. Each Virtual Machine had the same specifications and kernel settings as indicated in Table 4.1.

Setting	Value
OS	Ubuntu 12.04
Word Length	64-bit
RAM	6 GB
Hard Disk	20 GB
CPU Speed	2.90GHz
Cores	8
Ethernet	gigabit
Additional Kernel Settings	*atime* disabled[25]

Table 4.1: Virtual Machine Specifications and Settings.

To assist performance analysis and cluster monitoring, a third party system monitoring tool was installed on all VM's to collect important system metrics while experiments were being conducted.

Patil *et al.* [51] added extensions to YCSB to improve performance understanding and debugging of advanced NoSQL data store features by implementing a custom monitoring tool built on top of Ganglia [71]. This suggested Ganglia could be a good fit for this study also since it provides near real time monitoring and performance metrics data for large computer networks.

Ganglia's metric collection design mimics that of any well-designed parallel application. Each individual host in the cluster is an active participant, cooperating together, distributing the workload while avoiding serialization and single points of failure. Ganglia's protocol's are optimized at every opportunity to reduce overhead and achieve high performance [44].

Ganglia was attractive for a number of reasons including its simple implementation and configuration, its ability to scale easily to accommodate large cluster sizes, and its provision of a web based user interface for fine-grained time series analysis on different metrics in an interactive and easy way. Appendix C illustrates in more detail the precise Ganglia setup and configuration that was used for this study.

4.1 YCSB Configuration

In this study, one read-heavy and one write-heavy workload are included. The read-heavy workload is one provided in the YCSB Core Package; workload B comprising a 95/5% breakdown of read/write operations. The write-heavy workload was custom designed to consist of a 95/5% breakdown of write/read operations.

A VoltDB client driver is not currently supported by YCSB, however the VoltDB community of in-house developers built their own publicly available driver[26], which was utilized in this study. The VoltDB developers made several tweaks in order to effectively implement this driver comparatively to non-relational data store systems that don't have as strong an emphasis on server-side logic for enhanced performance. Details of the drivers implementation is given by a VoltDB developer in [60].

The benchmarking experiments carried out by Cooper *et al.* in their original YCSB paper [14] used a fixed metric of eight threads per CPU core when running experiments. This number was considered optimal for their experiments, and after several preliminary tests was found to be equally optimal for this study also. If the thread count was too large, there would be increased contention in the system, resulting in increased latencies and reduced throughputs. Since Redis and VoltDB have a single threaded architecture, only eight threads in total were used by the YCSB Client, to ensure the Redis and VoltDB servers were not overwhelmed by operation requests. This contrasts with a total of sixty-four threads which were used for both Cassandra and MongoDB experiments, which are not single threaded and can make use of all

[25] Disabling atime reduces the overhead of updating the last-access time of each file.

[26] Available at `https://github.com/VoltDB/voltdb/tree/master/tests/test_apps/ycsb`.

available CPU cores[27].

4.1.1 Warm-up Extension

Preliminary experiments indicated that each data store took variable lengths of time to reach a steady performance state. It was therefore deemed beneficial to implement an additional warm-up stage to the YCSB code base to improve results and comparative analysis. Nelubin & Engber extended YCSB to include a warm-up stage for both of their benchmarking studies mentioned previously: [47,48]. This extended code was open sourced[28] and so was subsequently implemented in this study also.

In order to determine the ideal warm-up time for each data store, a set of test experiments were repeated for each data store on various cluster sizes ranging from a single node to a twelve node cluster. Averages of the time it took for each data store to level out at or above the overall average throughput of a given test experiment are summarized in Table 4.2. These warm-up times where subsequently passed as an additional configuration parameter to the YCSB Client for run phases only.

Data Store	Warm-up Time (seconds)
Redis	1680
MongoDB	1500
Cassandra	1800
VoltDB	1200

Table 4.2: Optimal Warm-up Times For Each Data Store.

Unfortunately however, using the customized VoltDB client driver for YCSB introduced many nontrivial incompatibility issues with the warm-up extended YCSB code base. As such, all VoltDB experiments in this study do not include a warm-up stage.

Figure 4.1 illustrates the number of operations per second that were recorded for a sample experiment every 10 seconds for the full duration of the experiment (10 minutes). It highlights how long it would take for a VoltDB cluster to reach a point were it was consistently matching or exceeding the average throughput of that experiment. This indicates a warm up period of approximately two minutes would be optimal and result in an increased throughput of almost 1000 operations per second. The impact this had overall on VoltDB can be observed from the experimental results presented in Section 5.4, along with comparative results among all data stores in Section 5.5.

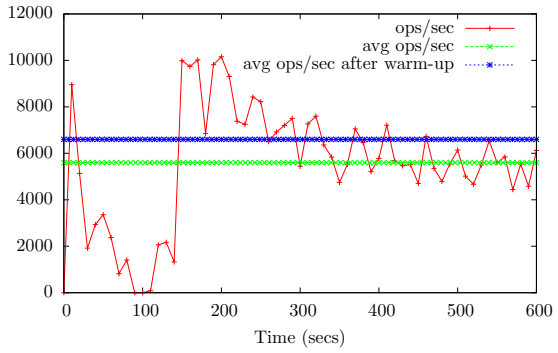

Figure 4.1: VoltDB Warm-up Illustration.

4.2 Data Store Configuration and Optimization

A total of fourteen Virtual Machine nodes where provisioned for this study. One node was designated for the YCSB Client, and one additional node was reserved for MongoDB configuration and App servers which are required in *sharded* architectures to run on separate servers to the rest of the cluster. The remaining twelve nodes operated as standard cluster nodes which had

[27]The total number of CPU cores available on each server was 8. Using 8 threads each gives a grand total of 64 threads.

[28]Available at `https://github.com/thumbtack-technology/ycsb`.

all four data stores installed but only one running at any given time. No other processes other than the Ganglia Daemon and Network Time Protocol (NTP) Daemon processes (to ensure all node clocks were synchronized) were running on cluster nodes. To ensure all nodes could interact effectively each node was bound to a set IP address, and known hosts of all others were maintained on each node. To assist administrative tasks, password-less ssh was configured on all nodes also. Each data store was configured and optimized for increased throughput, low latency, and where possible to avoid costly disk-bound operations. Each subsection below discusses in detail the configurations and optimizations used for each data store.

4.2.1 Redis

Version 2.8.9 of Redis was used in this benchmark. Prior to conducting this study, Redis Cluster was still beta quality code, and the extra processing required on the YCSB Client to interact with the cluster was deemed too invasive on the YCSB Client and so was excluded.

To allow slave nodes to service read requests, the YCSB Client was extended to randomly pick a node (master or a slave) to send read requests to. This was determined on a per read basis. A full code listing of this extension is illustrated in Appendix A.

Different consistency settings where enabled on each Redis node by passing the *min-slaves-to-write* configuration parameter to the start-up command. The value of which was determined based on the cluster size, and desired level of consistency. The following write consistency levels were explored: ONE (*min-slaves-to-write* = 0), QUORUM (*min-slaves-to-write* = (cluster_size/2)+1), and ALL (*min-slaves-to-write* = cluster_size−1).

A complete list of configurations and optimizations that were applied for Redis experiments are listed in Table 4.3.

Configuration Parameter	Description
--appendonly no	Disable AOF persistence
--activerehashing no	Disable active rehashing of keys. Estimated to occupy 1 ms every 100 ms of CPU time to rehash the main Redis hash table mapping top-level keys to values
--appendfsync no	Let the Operating System decide when to flush data to disk
--stop-writes-on-bgsave-error no	Continue accepting writes if there is an error saving data to disk
--aof-rewrite-incremental-fsync no	Disable incremental rewrites to the AOF file
disable snapshotting	Avoiding disk-bound background jobs from interfering.
Kernel Setting	**Description**
memory overcommitting set to 1	Recommended on the Admin section of the Redis website[29]

Table 4.3: Redis Configuration Settings.

4.2.2 MongoDB

Version 2.6.1 of MongoDB with the majority of all standard factory settings was used in this study. Journaling however was disabled since the overhead of maintaining logs to aid crash recovery was considered unnecessary as crash recovery was not a major consideration in this benchmark. If a node did crash, the experiment would simply be repeated. Additionally, the balancer process was configured not to wait for replicas to copy and delete data during chunk migration, in order to increase throughput when loading data onto a cluster.

MongoDB offers different write concerns for varying tunable consistency settings, of which NORMAL, QUORUM, and ALL write concerns where explored. The YCSB Client did not support write concerns or read preferences, therefore the YCSB Client was extended to facilitate them. A code listing of these extensions are given in Appendix B. For all experiments the primary preferred read preference was used to favor queries hitting the master preferably, but if for whatever reason the master was unavailable, requests would be routed to a replicated slave. Write concerns and read preferences where passed as additional command line parameters to the YCSB Client.

[29] `http://redis.io/topics/admin`

As suggested at the beginning of this subsection, additional configuration and App servers are required for *sharded* MongoDB clusters. The recommended production setup is to have three configuration servers all located on separate hosts, and at least one App server per client. For the purpose of this benchmark study, only one configuration server was used and resided on the same host as a single App server which the YCSB Client interacted with exclusively. This setup seemed appropriate for this study since having only one configuration server is adequate for development environments [34], and it was observed that having both reside on the same host did not prove to be a bottleneck.

The shard key used was the default index key ('*_id*') enforced by MongoDB and subsequently the key used by YCSB to insert, read and update records.

4.2.3 Cassandra

For this benchmark, version 1.2.16 (the latest 1.X release available before commencing this study) of Cassandra was used because the latest version supported by the YCSB Cassandra client driver was 1.X; even though the current release of Cassandra has progressed to version 2.0.9. This unfortunately meant that the performance enhancements available in verion 2.0 could not be assesed[30]. Nonetheless, most of the default configurations where used except those mentioned below.

A token representing the range of data each node would be responsible for in each independent cluster configuration was pre-calculated and saved in separate copies of a Cassandra configuration file. The configuration file specific to the cluster setup required for a given experiment was then passed to Cassandra at start-up time. This was necessary to overcome the problems associated with Cassandra attempting to define these distinct partitions itself as new nodes were added to a cluster, which become evident during preliminary testing. These tokens were calculated using a token generator tool[31], using the Murmur3 partitioner (which distributes data across a cluster evenly using the Murmur3_128 hash function[32]) and passing as a parameter the number of nodes in the cluster.

Hinted-handoff was disabled on all nodes within the cluster to avoid the situation were a node could fail and remain offline for the duration of an experiment, causing hints to build up rapidly as the YCSB Client attempted to saturate the cluster. Preliminary testing indicated that such a situation often led to detrimental impact and increased instability throughout the cluster. In keeping with the rule of rerunning experiments that are subjected to failures; it seemed reasonable to disable hinted-handoffs and therefore avoid the additional overhead associated with it.

Additional configuration optimizations included increasing the maximum number of concurrent reads and writes to match the same number of threads used by YCSB Client *i.e.* 64. Finally, the RPC server type was changed to 'hsha' to reduce to amount of memory used by each Cassandra node; ideal for scaling to large clusters.

Three Java JVM setting changes included: setting the JVM heap size to 4GB (to approximately match the amount of memory used by each node); setting the heap new size to 800MB (100MB per core); and finally disabling all assertions in order to increase performance.

4.2.4 VoltDB

In this study version 4.4 of the VoltDB Enterprise edition was used with its default configurations, including asynchronous command logging. For illustrative purposes, the VoltDB plots of Section 5.4 include comparative results for experiments conducted without command logging or replication. A constant number of distinct data partitions per host was used based on the recommendation of approximately 75% of the number of available cores on each server [38].

[30] For a complete description of the internal optimizations and improvements featured in Cassandra 2.0, see: `http://www.datastax.com/dev/blog/whats-under-the-hood-in-cassandra-2-0`, Last Accessed: 2014.08.27

[31] `http://www.geroba.com/cassandra/cassandra-token-calculator/`

[32] An informative description can be found here: `http://code.google.com/p/smhasher/wiki/MurmurHash3`, Last Accessed: 19/08/2014

This amounted to approximately six partitions per host.

VoltDB's implementation of replication is primarily for geo-replication if an entire data center cluster falls over. VoltDB's method of inter-data center partition replication is referred to as K-Safety and is analogous to how other data stores handle partition replication. Therefore, K-Safety was the feature used to evaluate the performance impact of replication within a VoltDB cluster in this study. Since VoltDB is fully ACID compliant, partition replicas remain consistent by nature regardless of the replication factor. Consequently, VoltDB does not offer tunable consistency settings and therefore, different consistency levels could not be explored further.

4.3 Methodology

The experiments carried out in this study include the three different data distributions mentioned in Section 2, each simulating a different industry application use case. Two different workloads (one read-heavy and one write-heavy) were used, and each cluster node hosted enough data to utilize a minimum of 80% RAM. Disk-bound operations were not considered since Redis and VoltDB are designed to have all data kept in memory.

MongoDB was configured to have a constant replication factor of two replicas per *shard*, meeting the minimum recommended production settings. The number of shards were incremented from 1 to 4, in order to directly explore the write-scalability of MongoDB.

An equivalent setup could have been established for Redis using the soon to be released Redis Cluster (still currently in beta testing). After experimenting with the beta version of Redis Cluster, it became apparent that reads rely too heavily on client side processing. In the interest of not laboring the YCSB Client with this task and subsequently skewing results, it was decided that Redis Cluster would not be explored further. Subsequently, a basic set of replication experiments were conducted which for a given cluster size, consisted of assigning one node to be the master and having the remaining nodes all act as replicas.

Cassandra and VoltDB offer greater flexibility in replicating data partitions due to their multi-master architectures and ability to have multiple partitions held on a single node. As such, less trivial experiments could be performed. Details of which are indicated in Table 4.4.

The precise set of experiments conducted for all data stores in this study are summarized in Table 4.4. Each permutation of the experiments listed were repeated a minimum of three times. Additional repeats of experiments, up to a maximum of ten, were scheduled if the results from three repeats showed considerable anomalies (for example due to instability of the cloud environment or because of node failure).

	Redis					MongoDB					Cassandra					VoltDB				
Workload																				
Read Heavy (G)			✓					✓					✓					✓		
Write Heavy (H)			✓					✓					✓					✓		
Consistency																				
ONE			✓					✓					✓							
QUORUM			✓					✓					✓							
ALL			✓					✓					✓					✓		
Distribution																				
Uniform			✓					✓					✓					✓		
Zipfian			✓					✓					✓					✓		
Latest			✓					✓					✓					✓		
Cluster Size	1	3	6	9	12	1	3	6	9	12	1	3	6	9	12	1	3	6	9	12
Replication Factor	0	2	5	8	11	0	2	2	2	2	0	2	4	6	8	0	2	4	6	8
Number Of Partitions[33]						1	1	2	3	4						6	18	36	54	72
Distinct Partitions						1	1	2	3	4						6	9	9	9	9
Base Line Experiment						✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 4.4: Complete List of Experiments.

Additional base line experiments were carried out in order to have a basis for comparison, a key determinant in evaluating the performance impact of replication per Cooper *et al.* [14].

[33]Applicable to MongoDB and VoltDB only. In MongoDB, a partition is another name for a *shard*. In VoltDB each node is partitioned into a number of 'sites per host' which applies to each node in the cluster. Therefore the total number of partitions shown is across the whole cluster, each node being responsible for 6 individual partitions/sites.

These base line experiments consisted of maintaining the same cluster sizes, with no replication, using the uniform distribution only, for all applicable consistency levels. Redis, unfortunately does not support multiple partitions and as such, additional base line experiments could not be included.

Prior to conducting experiments, a number of load tests were carried out to identify the amount of data required to utilize 80% RAM on all nodes within a given cluster for each data store, and the optimal number of threads the YCSB Client should use when loading this data onto the cluster.

The following steps outline the procedure that was carried out for each experiment. A number of bash scripts where written to automate these steps and all experiments highlighted in Table 4.4 for each data store. Once all experiments had been conducted three times, the results would be analyzed, anomalies identified and new automated scripts generated to rerun those experiments following the same steps highlighted below:

1. Start up a fresh cluster of the correct size and replication factor specific to the experiment in question.
2. Delay for a short period of time to ensure all nodes within the cluster have been fully configured and are ready to accept requests.
3. Load the data store cluster with sample data based on the factors indicated by load tests conducted previously.
4. Run the given experiment workload for a preconfigured warm-up time in order for the cluster to reach optimal throughput before starting the official test.
5. Run the experiment specifying a maximum execution time of 10 minutes.
6. Allow a short cool down period to ensure all actions related to the experiment have been completed.
7. Collect and backup results of the experiment conducted in step 5.
8. Tear down the entire cluster and wipe all data associated with the experiment on each node.
9. Repeat steps 1 to 8 until all experiments have been carried out a minimum number of times.

5 Experimental Results

YCSB reports the results of an experiment by printing them to the standard output terminal every 10 seconds. These were subsequently redirected to log files and an additional set of scripts were written to parse the output into a more amenable format for analysis. Where possible, the number of experiment reruns were kept below 7 additional runs. However due to varying levels of stability of each data store on the private cloud environment utilized in this study a number of rerun's were inevitable.

Redis was the most stable data store, which becomes clear when we consider the 95th% confidence interval figures presented in Table 5.2 later in this section. Cassandra was also very stable with minimal experiment failures and minimal supervision while experiments were running. However a number of repeated experiments were required, again obvious from the 95th% confidence interval figures presented in Table 5.8 later in this section.

MongoDB and VoltDB required considerable levels of supervision and intervention while experiments where conducted. For VoltDB specifically, if a sinlge or multi node crash was experienced, the YCSB Client was unable to continue conducting the experiment correctly even though VoltDB clusters are able to function when a node fails. Similarly, MongoDB failures often increased the instability of an experiment and required manual intervention to prevent the YCSB Client from hanging indefinitely and subsequently saturating disk space by spewing errors into the MongoDB logs.

In this section, the results of each data store will be discussed and analyzed in-depth followed by a presentation of several key features that are comparative among all four data stores.

5.1 Redis

Since baseline experiments could not be conducted for Redis, we are unable to evaluate the impact replication has on performance relative to non-replicated clusters of equal size. Nonetheless, some interesting observations have been made regarding the impact on availability that replication has for read and write operations.

Adding 2 additional nodes as replicas to a single node cluster resulted in an increase in performance on the read-heavy workload (G). On a single node cluster, the average throughput across all distributions and consistency levels is 23708.33 ops/sec which increases by 45.2% on average when an additional 2 nodes are added as slaves. This is expected since these additional nodes are able to service read requests *i.e.* reads have been scaled horizontally. However, as additional slaves are added to a cluster beyond size 3, the read performance remains fairly constant. On average, throughputs are equal to 28846.3, 43136.2, and 42605 ops/sec for ONE, QUORUM, and ALL consistency levels respectively for all cluster sizes with a replication factor greater than 0. These relate to minimal standard deviations of 440.1 (1.53%), 678.2 (1.57%), and 966.8 (2.27%) ops/sec respectively. Due to Redis' master-slave replication model, the YCSB Client is only able to pick one node at random to send a read request to. Therefore performance is unable to benefit from a further increase in the number of slaves. If more clients were interacting with the cluster, we would potentially observe an increase in throughput. This suggests that read operations remain highly and consistently available as the replication factor increases.

We observe in the read-heavy workload (G) that QUORUM and ALL consistency levels have fairly equal performances. Throughputs across all cluster sizes and distributions average 39034.7 ops/sec with a standard deviation of only 297.4 (0.76%) ops/sec. Read and write latencies average at 0.2107 ms and 0.2050 ms with standard deviations of 0.00108 (0.5%) ms and 0.00141 (0.7%) ms respectively. Different consistency settings are not expected to have an impact on read operations, due to a lack of support in Redis, therefore we expect the throughputs to be similar.

A noticeable contradiction is apparent when we consider the trends exhibited by workload G in Figure 5.1. That it, consistency level ONE is on average 39.1% less performant than QUORUM and ALL consistency levels on cluster sizes greater than 1. There is also an increase in read and write latencies *c.f.* Figures 5.2 and 5.3. The only difference between these experiments is that the *min-slaves-to-write* configuration setting is disabled for consistency level ONE. This appears to be an anomaly of some sort since there is nothing in the Redis source code that suggests a decline in performance would result from disabling this option. However, the Redis

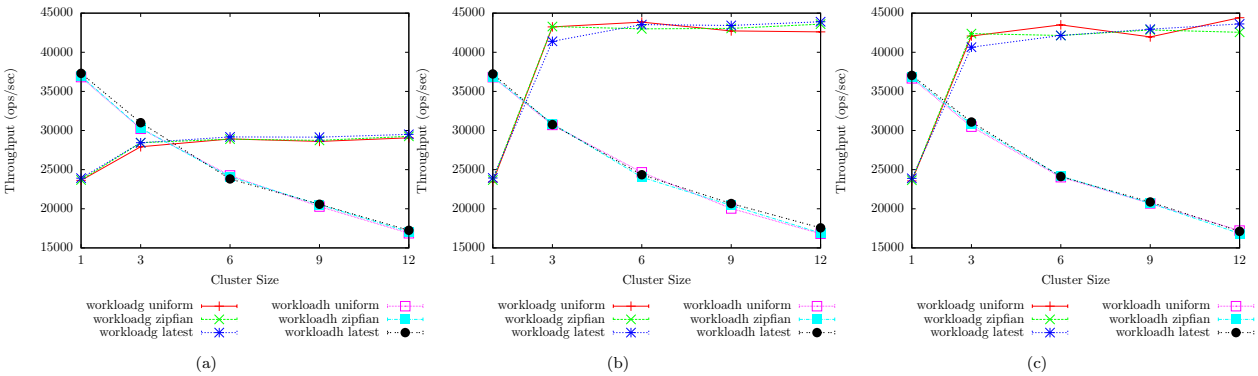

Figure 5.1: Redis: Overall Throughputs per Consistency Level for all Workloads and Distributions: (a) ONE (b) QUORUM (c) ALL.

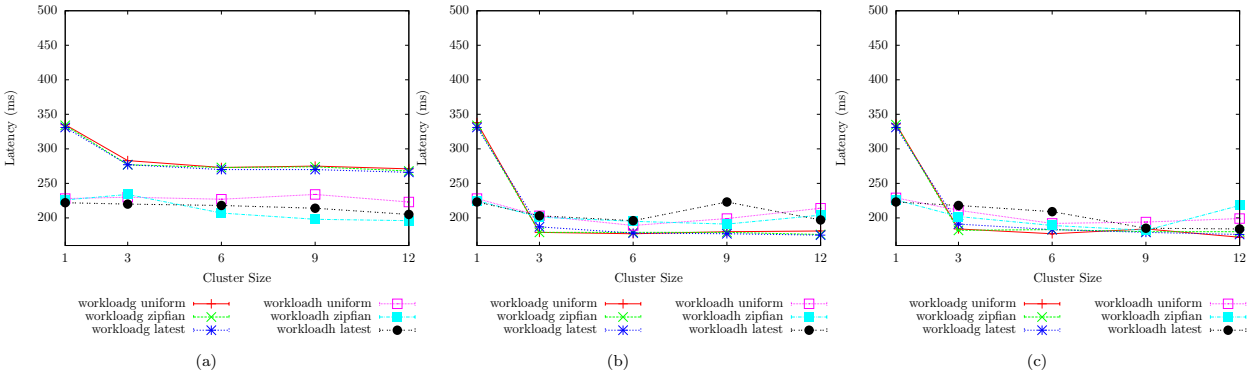

Figure 5.2: Redis: Read Latencies per Consistency Level for all Workloads and Distributions: (a) ONE (b) QUORUM (c) ALL.

release notes[34] indicate that a major issue had been discovered relating to the min-slaves-to-write option, present in the Redis version used in this study. A fresh set of results where gathered using the most recent Redis version (2.8.13) which had this issue resolved, however the results showed identical trends.

Overall throughput on the write-heavy workload (H) degrades as the cluster size increases. Considering the figures presented in Table 5.2, there is a 19.2% decrease in throughput on average each time the cluster size increases by 3 nodes, independent of the consistency level or distribution. This is likely a result of more slaves interacting with the master and the mechanism in which slaves process new data. When each slave receives new data from the master (asynchronously in the background), once it is ready to replace its stale version of the data, the slave blocks waiting on the replacement to take place. As such read throughput will decrease and subsequent writes will be delayed due to the single threaded architecture of Redis. The results suggest that the availability for write operations will continue to be affected as additional slaves are added to the cluster.

For the write-heavy workload (H) consistency levels ONE, QUORUM, and ALL are all pretty similar. They each have throughputs averaging 25882.7 ops/sec across all cluster sizes and distributions, with a standard deviation of only 54.1 (0.2%) ops/sec between them. This is because Redis only offers a relaxed form of consistency for writes only, with no guarantees. When processing a write operation the master simply checks an internal hash table to ensure a set number of slaves, determined by the consistency level, pinged the master within the previous 10 second interval. This would be a pretty constant time operation regardless of the cluster size and consistency requirements.

We observe minimal differences in overall throughput between distributions for both workloads. Observing an average difference of only 1.7%, and 1.3% between distributions across all cluster sizes and consistency levels for workload G and H respectively and standard deviations of 1.2% and 1% respectively for workload G and H. Table 5.1 gives the details of this compar-

[34] Available here: `http://download.redis.io/redis-stable/00-RELEASENOTES`. Last accessed: 12/08/2014

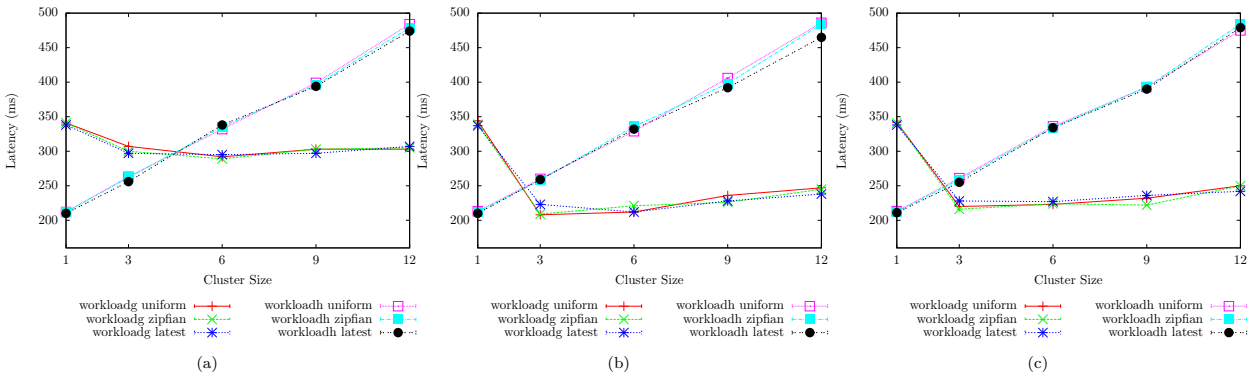

Figure 5.3: Redis: Write Latencies per Consistency Level for all Workloads and Distributions: (a) ONE (b) QUORUM (c) ALL.

ison. The likely reason for this is because Redis keeps all data in memory on a single server. Subsequently, requests to access or update records of varying distributions will result in the same action: performing a constant time operation on a single hash table. As with all hash tables, this constant time operation is characterized by an asymptotic run time of $O(1)$ in the average case.

As illustrated in Figure 5.1 and supported by the data in Table 5.3, on a single node cluster the write-heavy workload (H) outperforms the read-heavy workload (G) by 43.7% on average across all distributions and consistency levels, with a standard deviation of only 0.3%. This is as a direct result of how the YCSB Client interacts differently with a Redis cluster for read and write operations. Read operations will access all the fields a particular key maps (to 10 in this case), as opposed to write operations which only update 1 field. This suggests that for one node, there is a 4.85% decline in performance for each additional field that needs read since Redis must traverse the key list searching for a repeated key and its required associated field.

However, when replicas are added to a cluster, the read-heavy workload (G) starts to outperform the write-heavy workload (H), averaging 48.9% increases across all subsequent cluster sizes for all distributions and consistency levels. This correlates to a 24.4%, 61.8%, and 60.5% increase for ONE, QUORUM, and ALL consistency levels respectively, and 49.3%, 49.1%, and 48.2% increases for the uniform, zipfian, and latest distributions respectively. This increase in performance is due to the ability of additional slaves to service read requests. Whereas only one single-threaded master can process write requests which in workload H acts as a bottleneck due to the large percentage of write operations (95%).

The availability of write operations does not appear to be affected by the percentage of read operations in a given workload and vice versa. We observe a 37.1% (0.115 ms) decrease in write latencies for Workload G (5% writes) compared to Workload H (95% writes), and a 40.8% (0.108 ms) reduction in read latencies in workload H (5% reads) compared to workload G (95% reads). This is not a proportional decrease since the overhead of the underlying operation remains the same *i.e.* a hash table lookup. The additional overhead is a direct result of the single threaded nature of Redis which is forced to contend with an increase in the number of operations it must service. Writes are more affected since all writes must be directed to a single master instance.

Figures 5.4, 5.5, 5.6, and 5.7 illustrate the latency histograms for read and write operations on both workloads. As we can see from the CDF curves plotted on the same figures, 95% of operations can be answered in less than 1 ms for all cluster sizes, distributions, and workloads. The 95th percentile figures are subsequently not included in Table 5.2 for the sake of brevity.

				Workload G					Workload H				
Cluster Size				1	3	6	9	12	1	3	6	9	12
Replication Factor				0	2	5	8	11	0	2	5	8	11
Type	Metric	Distribution	Consistency										
Read	Latency	Uniform vs Zipfian	ONE	0.3	2.1	0.0	0.4	1.1	0.9	1.7	9.2	16.7	12.9
		Uniform vs Latest		1.2	2.1	1.1	1.8	1.9	2.7	4.4	4.0	8.9	8.4
		Uniform vs Zipfian	ALL	0.3	1.1	3.3	2.2	4.5	1.3	4.4	1.6	6.4	9.1
		Uniform vs Latest		1.5	3.7	3.3	2.8	2.3	2.7	3.3	8.5	4.7	7.8
		Uniform vs Zipfian	QUORUM	1.2	0.0	1.1	0.6	2.8	1.3	1.0	3.1	4.1	4.8
		Uniform vs Latest		2.1	4.4	0.6	1.7	3.4	2.2	0.0	3.6	11.4	8.3
Write	Latency	Uniform vs Zipfian	ONE	0.0	2.3	1.0	0.0	0.7	0.0	0.4	0.9	0.8	1.0
		Uniform vs Latest		0.9	3.3	1.0	2.0	1.3	0.9	2.7	1.8	1.3	2.1
		Uniform vs Zipfian	ALL	0.3	1.8	0.4	4.4	0.0	0.5	0.8	0.6	0.0	1.9
		Uniform vs Latest		1.2	3.6	1.8	1.7	3.3	0.9	2.3	0.6	0.8	0.8
		Uniform vs Zipfian	QUORUM	1.5	0.5	4.2	4.3	0.8	0.5	0.8	2.1	2.0	0.4
		Uniform vs Latest		2.1	7.0	0.0	3.4	3.7	1.4	0.4	0.9	3.5	4.4
Overall	Throughput	Uniform vs Zipfian	ONE	0.2	1.8	0.1	0.5	0.7	0.3	0.2	0.7	1.2	1.1
		Uniform vs Latest		1.2	1.8	0.9	1.9	1.5	1.3	2.5	1.9	1.5	2.2
		Uniform vs Zipfian	ALL	0.4	0.7	3.2	2.1	4.3	0.4	1.0	0.6	0.1	2.2
		Uniform vs Latest		1.4	3.5	3.2	2.3	1.8	1.0	1.9	0.4	0.9	0.6
		Uniform vs Zipfian	QUORUM	1.1	0.1	2.0	0.7	2.3	0.1	0.5	2.6	1.9	0.5
		Uniform vs Latest		2.1	4.3	0.7	1.6	3.0	1.1	0.2	1.2	3.1	4.4

Table 5.1: Redis: Percentage Differences In Read & Write Latencies and Overall Throughput Between Distributions for each Workload and Consistency Level.

				Workload G					Workload H				
Cluster Size				1	3	6	9	12	1	3	6	9	12
Replication Factor				0	2	5	8	11	0	2	5	8	11
Type	Metric	Distribution	Consistency										
Read	Latency (ms)	Uniform	ONE	0.335	0.283	0.273	0.275	0.271	0.228	0.23	0.227	0.234	0.223
			QUORUM	0.338	0.179	0.177	0.18	0.181	0.228	0.203	0.189	0.199	0.214
			ALL	0.336	0.184	0.177	0.184	0.172	0.229	0.211	0.192	0.194	0.199
		Zipfian	ONE	0.334	0.277	0.273	0.274	0.268	0.226	0.234	0.207	0.198	0.196
			QUORUM	0.334	0.179	0.179	0.179	0.176	0.225	0.201	0.195	0.191	0.204
			ALL	0.335	0.182	0.183	0.18	0.18	0.226	0.202	0.189	0.182	0.218
		Latest	ONE	0.331	0.277	0.27	0.27	0.266	0.222	0.22	0.218	0.214	0.205
			QUORUM	0.331	0.187	0.178	0.177	0.175	0.223	0.203	0.196	0.223	0.197
			ALL	0.331	0.191	0.183	0.179	0.176	0.223	0.218	0.209	0.185	0.184
Write	Latency (ms)	Uniform	ONE	0.341	0.307	0.292	0.303	0.303	0.212	0.263	0.332	0.399	0.484
			QUORUM	0.344	0.208	0.212	0.236	0.247	0.213	0.26	0.329	0.406	0.486
			ALL	0.342	0.22	0.223	0.232	0.25	0.213	0.261	0.336	0.393	0.475
		Zipfian	ONE	0.341	0.3	0.289	0.303	0.305	0.212	0.262	0.335	0.396	0.479
			QUORUM	0.339	0.209	0.221	0.226	0.245	0.212	0.258	0.336	0.398	0.484
			ALL	0.341	0.216	0.224	0.222	0.25	0.212	0.259	0.334	0.393	0.484
		Latest	ONE	0.338	0.297	0.295	0.297	0.307	0.21	0.256	0.338	0.394	0.474
			QUORUM	0.337	0.223	0.212	0.228	0.238	0.21	0.259	0.332	0.392	0.465
			ALL	0.338	0.228	0.227	0.236	0.242	0.211	0.255	0.334	0.39	0.479
Overall	Throughput	Uniform	ONE	23633	27922	28919	28610	29065	36829	30221	24262	20282	16869
			QUORUM	23429	43245	43853	42733	42607	36816	30717	24655	20043	16813
			ALL	23558	42079	43501	41961	44416	36670	30494	24036	20682	17220
		Zipfian	ONE	23684	28443	28902	28741	29260	36933	30296	24081	20523	17062
			QUORUM	23679	43286	42992	43053	43594	36867	30878	24034	20433	16905
			ALL	23655	42383	42144	42852	42553	36825	30786	24172	20705	16840
		Latest	ONE	23910	28441	29186	29152	29514	37309	30999	23814	20579	17245
			QUORUM	23933	41404	43536	43430	43901	37226	30767	24351	20672	17564
			ALL	23894	40649	42150	42953	43619	37047	31086	24124	20871	17117
	95% CI	Uniform	ONE	0	3.92	0	1.96	1.96	1.96	3.92	3.92	0	0
			QUORUM	0	1.96	0	1.96	1.96	1.96	3.92	9.8	0	1.96
			ALL	0	5.88	5.88	1.96	0	3.92	5.88	0	0	0
		Zipfian	ONE	0	1.96	0	1.96	1.96	0	5.88	0	1.96	0
			QUORUM	0	0	5.88	0	1.96	1.96	3.92	1.96	1.96	1.96
			ALL	0	7.84	3.92	0	5.88	1.96	5.88	0	0	0
		Latest	ONE	0	1.96	0	0	0	1.96	3.92	0	0	0
			QUORUM	0	5.88	5.88	1.96	0	1.96	5.88	3.92	1.96	0
			ALL	0	13.72	1.96	5.88	0	0	3.92	0	0	3.92

Table 5.2: Redis: Read & Write Latency and Overall Throughput & 95th% Confidence Interval Data per Workload, Broken Down by Distribution and Consistency Level.

Cluster Size				1	3	6	9	12
Replication Factor				0	2	5	8	11
Type	Metric	Distribution	Consistency					
Read	Latency	Uniform	ONE	38.0	20.7	18.4	16.1	19.4
			QUORUM	38.9	12.6	6.6	10.0	16.7
			ALL	37.9	13.7	8.1	5.3	14.6
		Zipfian	ONE	38.6	16.8	27.5	32.2	31.0
			QUORUM	39.0	11.6	8.6	6.5	14.7
			ALL	38.9	10.4	3.2	1.1	19.1
		Latest	ONE	39.4	22.9	21.3	23.1	25.9
			QUORUM	39.0	8.2	9.6	23.0	11.8
			ALL	39.0	13.2	13.3	3.3	4.4
Write	Latency	Uniform	ONE	46.7	15.4	12.8	27.4	46.0
			QUORUM	47.0	22.2	43.3	53.0	65.2
			ALL	46.5	17.0	40.4	51.5	62.1
		Zipfian	ONE	46.7	13.5	14.7	26.6	44.4
			QUORUM	46.1	21.0	41.3	55.1	65.6
			ALL	46.7	18.1	39.4	55.6	63.8
		Latest	ONE	46.7	14.8	13.6	28.1	42.8
			QUORUM	46.4	14.9	44.1	52.9	64.6
			ALL	46.3	11.2	38.1	49.2	65.7
Overall	Throughput	Uniform	ONE	43.7	7.9	17.5	34.1	53.1
			QUORUM	44.4	33.9	56.0	72.3	86.8
			ALL	43.5	31.9	57.6	67.9	88.2
		Zipfian	ONE	43.7	6.3	18.2	33.4	52.7
			QUORUM	43.6	33.5	56.6	71.3	88.2
			ALL	43.6	31.7	54.2	69.7	86.6
		Latest	ONE	43.8	8.6	20.3	34.5	52.5
			QUORUM	43.5	29.5	56.5	71.0	85.7
			ALL	43.2	26.7	54.4	69.2	87.3

Table 5.3: Redis: Percentage Differences In Read & Write Latencies and Overall Throughput Between Workloads for each Distribution and Consistency Level.

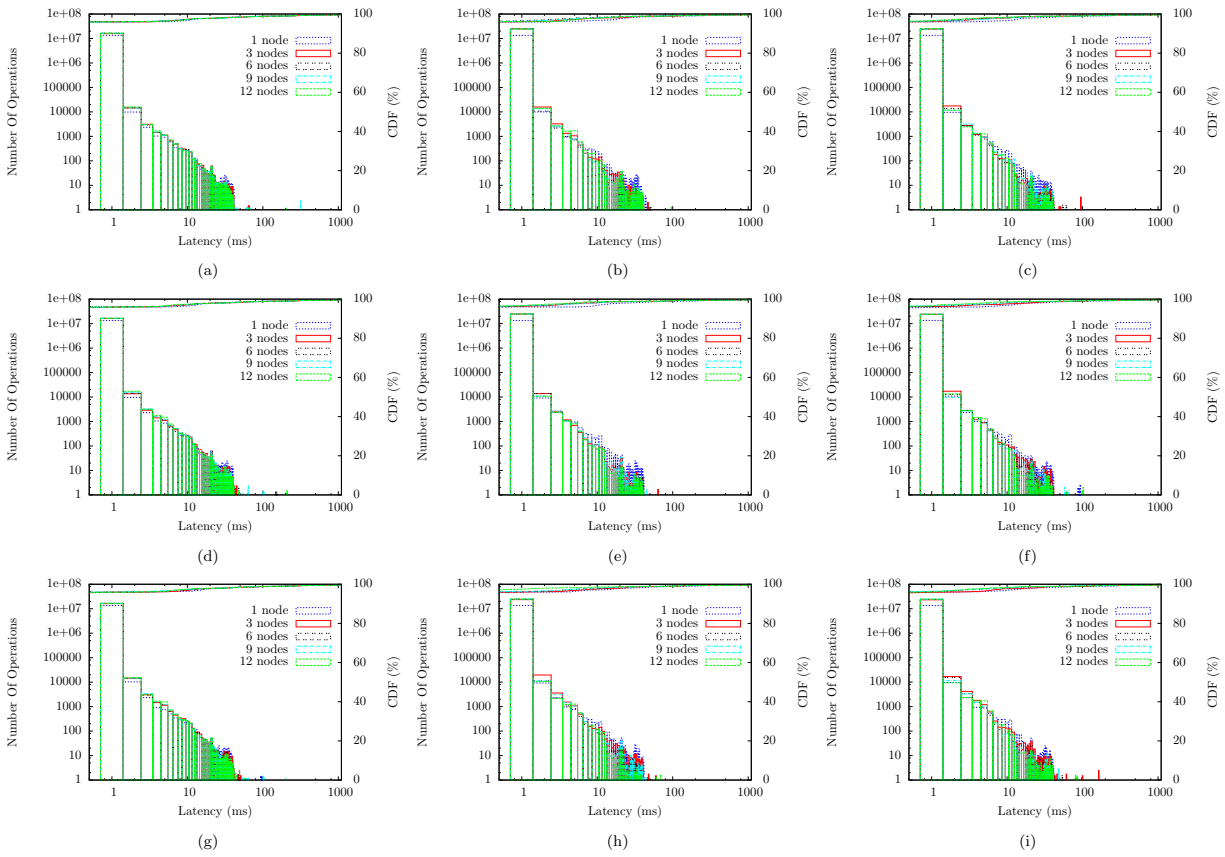

Figure 5.4: Redis Workload G Read Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

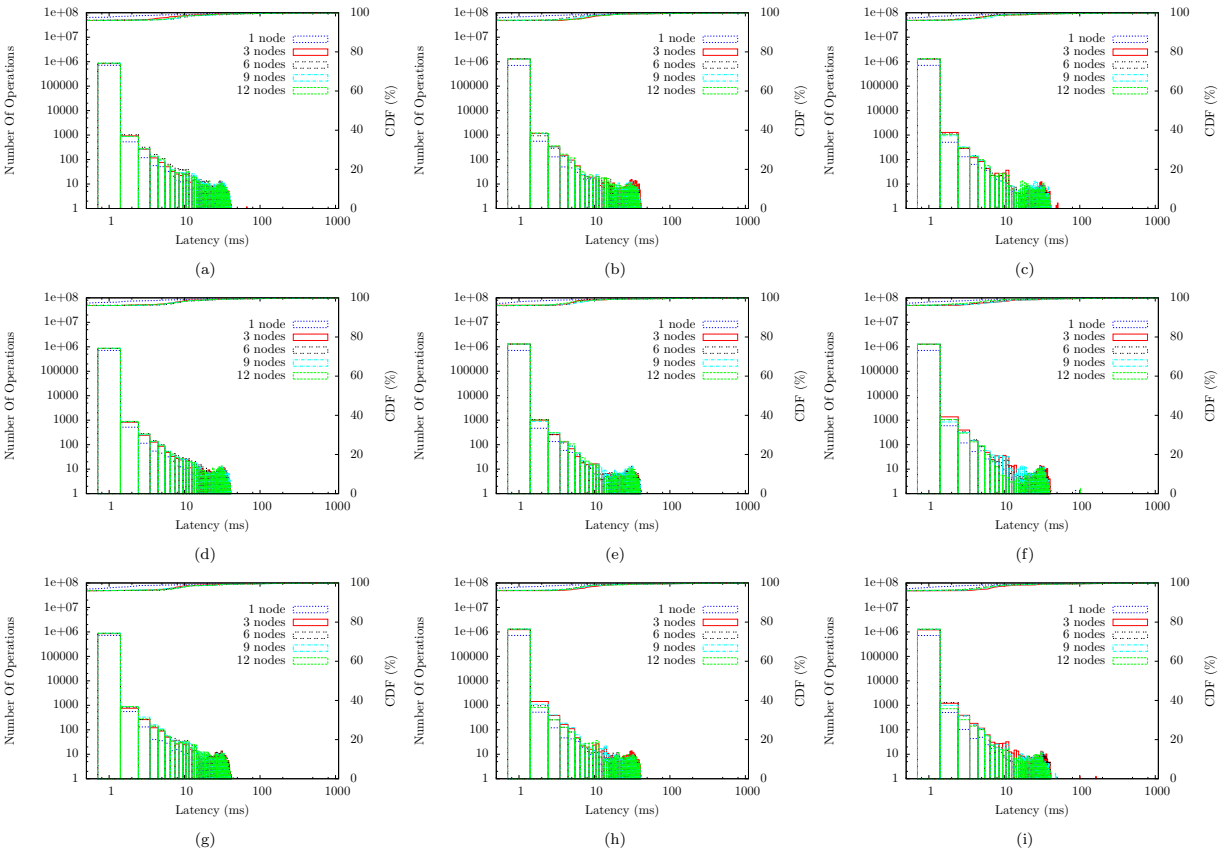

Figure 5.5: Redis Workload G Write Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

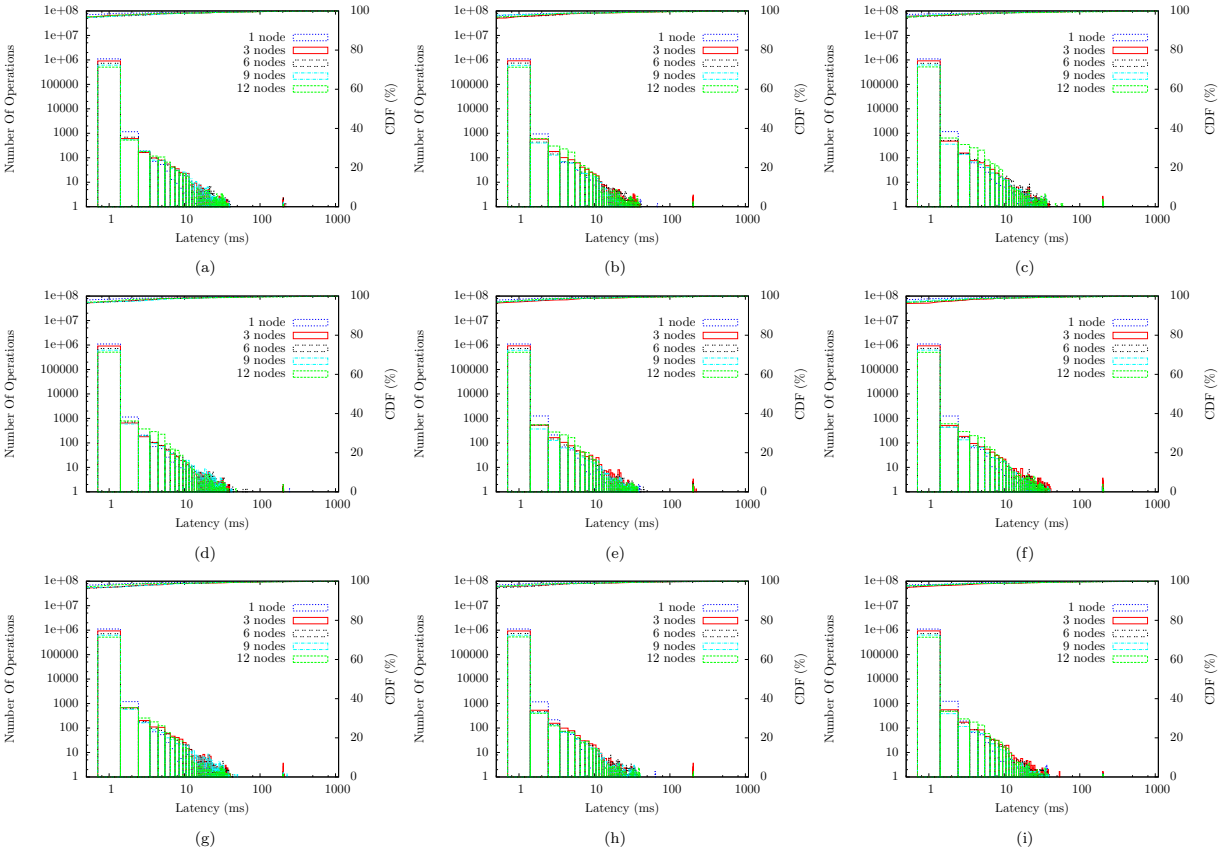

Figure 5.6: Redis Workload H Read Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

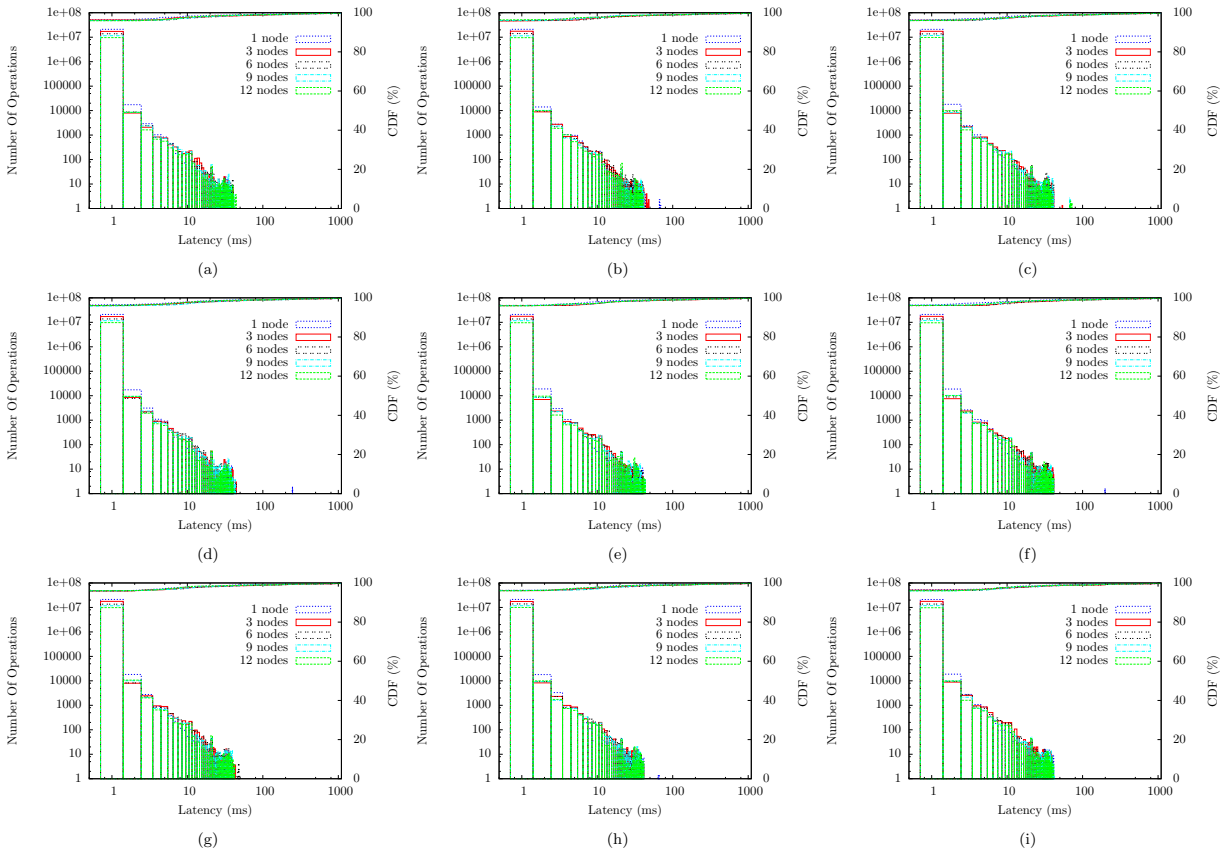

Figure 5.7: Redis Workload H Write Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

Cluster Size				1	3	6	9	12
Replication Factor				0	2	2	2	2
Type	Metric	Distribution	Consistency					
Read	Latency	Uniform	ONE	71.7	31.0	59.4	69.0	59.8
			QUORUM	60.0	32.3	66.9	78.5	53.9
			ALL	79.6	25.8	68.9	75.6	58.3
		Zipfian	ONE	65.6	39.9	5.3	65.1	68.7
			QUORUM	78.0	42.1	38.5	59.9	64.8
			ALL	78.8	28.1	30.0	46.4	60.8
		Latest	ONE	51.2	17.0	36.1	46.0	53.0
			QUORUM	66.7	90.5	25.9	59.3	58.1
			ALL	40.5	11.8	31.3	50.3	52.5
Write	Latency	Uniform	ONE	36.3	97.4	73.4	70.7	45.8
			QUORUM	33.2	73.0	81.5	69.4	41.0
			ALL	36.1	56.7	76.2	69.9	52.4
		Zipfian	ONE	19.8	54.3	42.4	53.7	53.8
			QUORUM	24.3	69.3	56.1	57.7	53.7
			ALL	23.2	71.9	61.4	52.5	54.7
		Latest	ONE	43.6	76.5	57.6	45.0	43.3
			QUORUM	40.8	31.1	55.0	55.3	50.3
			ALL	43.6	77.4	50.1	51.2	45.7
Overall	Throughput	Uniform	ONE	66.9	121.5	109.5	106.5	88.6
			QUORUM	66.5	99.0	115.2	103.9	81.3
			ALL	67.1	88.1	114.9	106.6	87.2
		Zipfian	ONE	54.8	80.3	79.8	94.4	93.6
			QUORUM	59.6	82.4	97.1	96.5	92.6
			ALL	58.7	90.5	93.4	92.1	94.2
		Latest	ONE	76.0	104.1	96.6	86.1	83.8
			QUORUM	73.8	61.9	96.2	92.9	87.7
			ALL	76.7	103.1	91.9	91.1	86.7

Table 5.4: MongoDB: Percentage Differences In Read & Write Latencies and Overall Throughput Between Workloads for each Distribution and Consistency Level.

5.2 MongoDB

The read-heavy workload (G) has on average an 88.7% higher level of throughput than the write-heavy workload (H). This corresponds to 94.8%, 84%, and 87.2% increases for uniform, zipfian, and latest distributions respectively, on average across all consistency levels and cluster sizes. When broken down by consistency level, we can observe a 89.5%, 87.1%, and 89.5% increase for ONE, QUORUM, and ALL consistency levels respectively. Figure 5.8 illustrates how this trend varies as the cluster size increases, while Table 5.4 gives more in-depth information per cluster size.

These trends are due to MongoDB's contrasting concurrency mechanism for reads and writes. MongoDB allows for concurrent reads on a collection, but enforces a single threaded locking mechanism on all write operations to ensure atomicity. In addition, all write operations need to be appended to an oplog on disk, which involves a greater overhead. The oplog is critical for replicas to maintain an accurate and complete copy of the primary shard's data. Furthermore, regardless of the requested *read concern* no additional consistency checks are performed between replicas on read operations.

For both workloads we observe a performance drop from cluster sizes 1 to 3. This is due to an additional replication factor of 2 being applied to a single shard (*i.e.* in the 3 node cluster). The original shard in the single node cluster now needs to save data to an oplog on disk and manage 2 additional servers. However, for cluster sizes greater than 3 nodes, more shards are able to distribute the load of reads and writes.

We subsequently see an increase in performance in line with non-replicated clusters of equal size when the cluster size is greater than 3. On average a 3 node cluster has a 91.8% and 87.7% decrease in throughput across all distributions and consistency levels for workload G and H respectively compared to non-replicated clusters of equal size. Meanwhile, on all subsequent cluster sizes (6+), the average decrease in throughput is only 13.6% and 40.3% for workload G and H respectively. This suggests that replication has a minimal affect on performance in read-heavy workloads once the overhead of maintaining a small number of shards have been overcome. This impact is more noticeable for write-heavy workloads however since consistency checks are performed on writes. Table 5.6 illustrates these observations at a more granular level.

Read and write latencies for the read-heavy (G) and write-heavy (H) workloads follow similar trends, as illustrated in Figures 5.9 and 5.10. However, on average workload H has 8.7% and 56.7% increases in read and write latencies respectively across all cluster sizes, distributions and

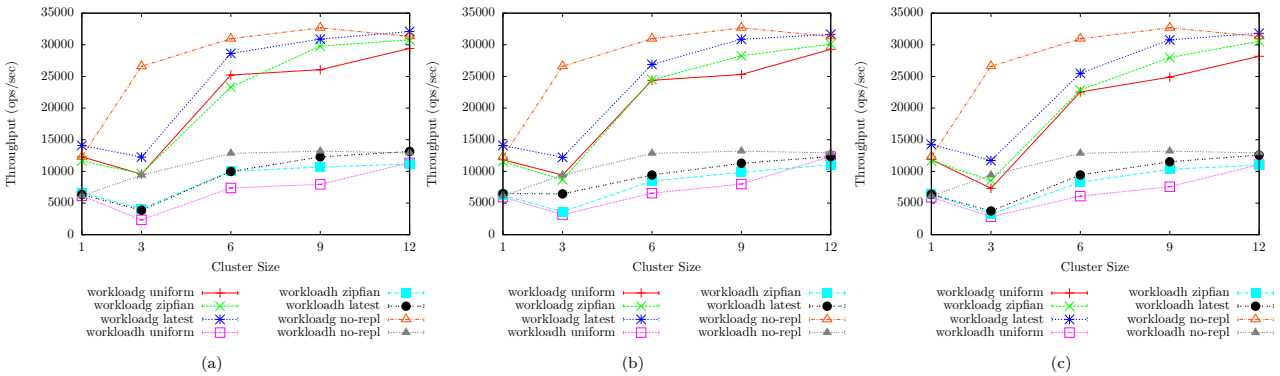

Figure 5.8: MongoDB: Overall Throughputs per Consistency Level for all Workloads and Distributions: (a) ONE (b) QUORUM (c) ALL.

consistency levels compared to workload G. On an individual distribution level, read latencies are increased by 18%, 3.8%, and 2.1% for uniform, zipfian, and latest distributions respectively across all cluster sizes. Similarly, write latencies have a 63.9%, 51.2%, and 53.6% increases for uniform, zipfian, and latest workloads respectively. When considered per consistency level, we observe 11.3%, 1.6%, and 12.7% increases in read latencies, and 59.2%, 54.1%, 56.7% increases in write latencies for ONE, QUORUM, and ALL consistency levels respectively across all cluster sizes. Further metrics are shown in Table 5.5.

Contrary to these clear differences however is the indication that read latencies tend to be more problematic on smaller cluster sizes for read-heavy workloads. We observe on average a 88.3% increase in read latencies with a standard deviation of 1.4% between different consistency levels on 1 & 3 node clusters compared to 6, 9 & 12 node clusters. For a single node cluster the poorer latencies would seem to be as a direct result of the sheer volume of reads hitting a single server, and the overhead of the App server (*i.e.* the mongos instance) and configuration servers maintaining a single shard. Little benefit is achieved for latencies on 3 node cluster because of the primary preferred read preference directing all reads to the same server.

It is a known limitation that due to the overhead of moving data, maintaining meta-data, and routing, small numbers of shards generally exhibit higher latencies and may even have lower throughputs than non-sharded systems [12].

Since the replication factor remains relatively small and constant as the cluster size increases (*i.e.* 2), we observe that read and write latencies also remain fairly constant as the cluster size increases beyond 3 nodes on workload G. We observe 11.1%, 10.3%, and 13.8% deviations from the average read latency (2.25 ms) for consistency levels ONE, QUORUM, and ALL respectively for cluster sizes greater than 3. Similarly we observe 11.1%, 11.7%, and 13.9% deviations from the average write latency (3.75 ms) for consistency levels ONE, QUORUM, and ALL respectively for cluster sizes greater than 3. This suggests reads remain highly available as the cluster size grows with a constant replication factor and more than one shard. The deviations in read and write latencies for workload H are just less than twice that for workload G which suggests writes are not as highly available as reads. The average standard deviation is 20% of the mean.

In general we observe higher throughputs for a consistency level of ONE on average across all distributions and cluster sizes, with slight degradation's for QUORUM and ALL consistency levels. For workload G a throughput of 21715.7 ops/sec is attainable for consistency level ONE, with 2.1% and 4.8% degradation's for QUORUM and ALL consistencies respectively. For workload H a throughput of 8226.1 ops/sec is attainable for consistency level ONE, with 1.7% and 5.9% degradation's for QUORUM and ALL consistencies respectively. This is expected behavior since increasingly more nodes are required to acknowledge each operation.

The latest distribution outperforms the zipfian and union distributions on both workloads. For workload G, the latest distribution has a 15% and 17.9% increase in throughput on average across all cluster sizes and consistency levels compared to zipfian and uniform distributions respectively. For workload H, the latest distribution has a 10.9% and 27.9% increase in throughput on average across all cluster sizes and consistency levels compared to zipfian and uniform distributions respectively. Table 5.7 illustrates these observations per cluster size and consistency

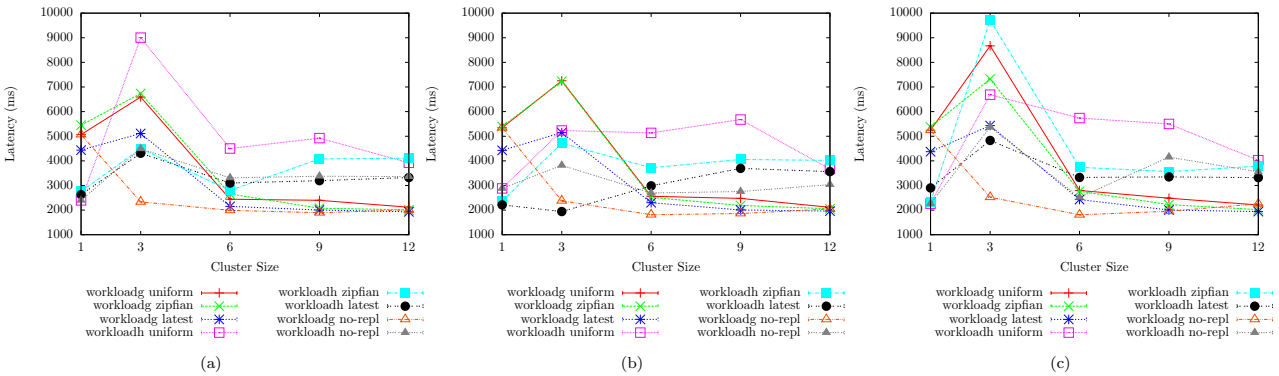

Figure 5.9: MongoDB: Read Latencies per Consistency Level for all Workloads and Distributions: (a) ONE (b) QUORUM (c) ALL.

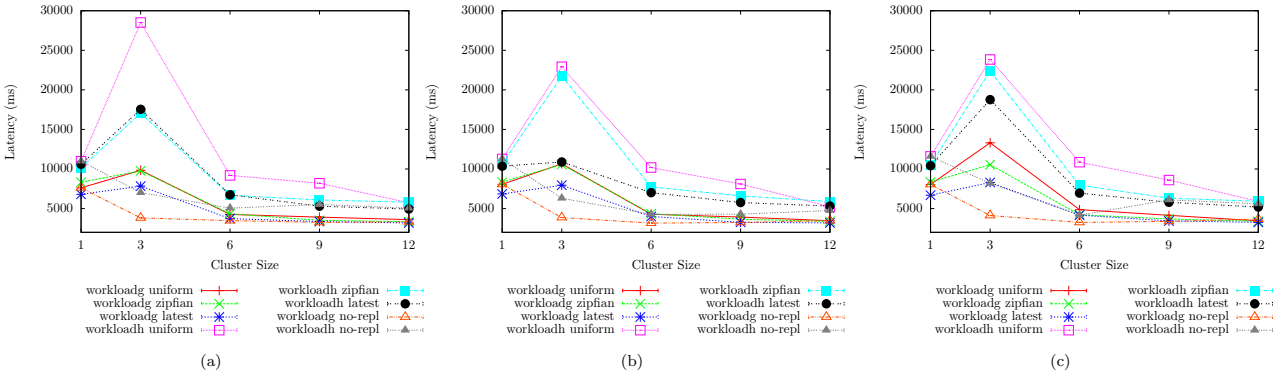

Figure 5.10: MongoDB: Write Latencies per Consistency Level for all Workloads and Distributions: (a) ONE (b) QUORUM (c) ALL.

level, including read and write latencies also.

Since MongoDB stores all data on disk and pulls data into RAM on a need to basis. We would expect the latest and zipfian distributions to outperform uniform because they would typically have everything they need in RAM after a short number of operations. However, the uniform distribution will keep randomly selecting documents which could require considerable RAM paging. The warm-up stage added to the YCSB Client would greatly assist the latest and zipfian distributions in this regard.

Figures 5.11, 5.12, 5.13, and 5.14 illustrate the latency histograms for read and write operations on both workloads. As we can see from the CDF curves plotted on the same figures, 95% of operations can be answered in less than 6.53 ms, 9.97 ms, 2.6 ms, and 13.1 ms for read and write operations on workloads G and H respectively, on average across all cluster sizes, distributions, and consistency levels. A more informative breakdown of these figures are presented in Table 5.5.

				Workload G					Workload H				
Cluster Size				1	3	6	9	12	1	3	6	9	12
Replication Factor				0	2	2	2	2	0	2	2	2	2
Type	Metric	Distribution	Consistency										
Read	Latency (ms)	Uniform	ONE	5.068	6.59	2.44	2.397	2.116	2.394	9.005	4.502	4.923	3.92
			QUORUM	5.345	7.252	2.56	2.476	2.111	2.879	5.234	5.135	5.678	3.668
			ALL	5.246	8.67	2.797	2.482	2.207	2.26	6.688	5.737	5.498	4.025
		Zipfian	ONE	5.451	6.735	2.655	2.075	2.009	2.759	4.496	2.8	4.076	4.11
			QUORUM	5.389	7.23	2.517	2.187	2.053	2.365	4.717	3.718	4.056	4.019
			ALL	5.374	7.324	2.762	2.222	2.02	2.335	9.717	3.738	3.564	3.785
		Latest	ONE	4.435	5.111	2.151	2.001	1.93	2.626	4.31	3.099	3.195	3.321
			QUORUM	4.426	5.148	2.3	2.005	1.959	2.213	1.941	2.984	3.694	3.563
			ALL	4.373	5.432	2.427	2.002	1.939	2.9	4.828	3.327	3.348	3.318
	95th Percentile	Uniform	ONE	11	9	5	4	4	2	2	2	3	4
			QUORUM	10	9	5	4	4	2	2	3	3	4
			ALL	11	10	5	4	4	2	2	3	3	4
		Zipfian	ONE	11	7	6	4	4	2	1	2	4	4
			QUORUM	10	9	5	4	4	2	1	3	3	4
			ALL	11	8	5	4	4	2	1	2	3	4
		Latest	ONE	10	11	5	4	4	2	2	3	3	4
			QUORUM	10	9	5	4	4	2	1	3	5	4
			ALL	10	10	5	4	4	2	1	2	3	4
Write	Latency (ms)	Uniform	ONE	7.614	9.844	4.26	3.918	3.598	10.993	28.516	9.195	8.201	5.737
			QUORUM	8.063	10.659	4.285	3.93	3.462	11.278	22.92	10.183	8.105	5.246
			ALL	8.047	13.312	4.87	4.146	3.45	11.593	23.838	10.864	8.598	5.902
		Zipfian	ONE	8.335	9.768	4.344	3.512	3.346	10.166	17.051	6.678	6.089	5.809
			QUORUM	8.335	10.565	4.349	3.65	3.391	10.638	21.757	7.738	6.608	5.878
			ALL	8.34	10.543	4.215	3.688	3.379	10.526	22.377	7.952	6.316	5.92
		Latest	ONE	6.803	7.838	3.727	3.356	3.182	10.594	17.544	6.743	5.303	4.94
			QUORUM	6.849	7.958	3.992	3.267	3.182	10.362	10.889	7.018	5.764	5.318
			ALL	6.691	8.297	4.167	3.43	3.262	10.421	18.765	6.953	5.791	5.192
	95th Percentile	Uniform	ONE	15	13	9	7	7	17	23	13	9	7
			QUORUM	14	13	8	7	7	15	25	14	9	7
			ALL	15	14	9	7	7	19	21	15	10	7
		Zipfian	ONE	14	10	9	7	7	15	19	13	8	7
			QUORUM	14	14	9	7	7	17	23	12	9	7
			ALL	14	12	8	7	7	16	16	13	9	7
		Latest	ONE	14	15	8	7	7	16	21	11	8	7
			QUORUM	14	13	8	7	7	18	20	13	8	7
			ALL	14	14	9	7	7	13	20	11	8	7
Overall	Throughput	Uniform	ONE	12303	9572	25227	26065	29421	6134	2339	7376	7954	11361
			QUORUM	11846	9418	24391	25310	29289	5933	3180	6559	8007	12360
			ALL	11870	7326	22545	24896	28186	5908	2845	6089	7587	11065
		Zipfian	ONE	11633	9636	23306	29786	30785	6630	4114	10007	10688	11161
			QUORUM	11561	8687	24472	28263	30105	6251	3618	8479	9860	11044
			ALL	11662	8570	22849	27985	30559	6367	3231	8299	10338	10987
		Latest	ONE	14105	12262	28631	30894	32110	6339	3867	9984	12294	13144
			QUORUM	14097	12226	26897	30878	31648	6496	6449	9425	11285	12350
			ALL	14271	11722	25486	30780	31827	6362	3747	9440	11518	12575
	95% CI	Uniform	ONE	1.96	9.8	1.96	13.72	13.72	9.8	3.92	11.76	0	5.88
			QUORUM	13.72	27.44	11.76	17.64	0	7.84	15.68	9.8	0	0
			ALL	1.96	9.8	19.6	0	7.84	11.76	7.84	7.84	3.92	7.84
		Zipfian	ONE	15.68	15.68	0	7.84	7.84	13.72	13.72	15.68	5.88	0
			QUORUM	3.92	5.88	1.96	3.92	0	0	25.48	0	0	1.96
			ALL	7.84	3.92	15.68	17.64	0	5.88	23.52	7.84	0	0
		Latest	ONE	9.8	9.8	3.92	9.8	9.8	7.84	9.8	21.56	1.96	0
			QUORUM	5.88	13.72	11.76	7.84	9.8	9.8	17.64	7.84	0	11.76
			ALL	5.88	13.72	9.8	0	1.96	0	11.76	1.96	13.72	7.84
Replication Factor				0	0	0	0	0	0	0	0	0	0
Type	Metric	Distribution	Consistency										
Read	Latency	Uniform	ONE	5.068	2.333	1.991	1.893	2.009	2.394	4.489	3.308	3.374	3.361
	95th Percentile	Uniform	ONE	11	4	4	4	4	2	3	4	4	4
Write	Latency	Uniform	ONE	7.614	3.823	3.475	3.259	3.321	10.993	7.011	5.056	5.521	5.052
	95th Percentile	Uniform	ONE	15	7	7	7	7	17	9	7	9	7
Overall	Throughput	Uniform	ONE	12303	26598	30960	32684	31319	6134	9417	12848	13200	12922
	95% CI	Uniform	ONE	1.96	9.8	7.84	11.76	31.36	9.8	9.8	0	50.96	7.84

Table 5.5: MongoDB: Read & Write Latency & 95th Percentiles and Overall Throughput & 95th% Confidence Interval Data per Workload, Broken Down by Distribution and Consistency Level.

				Workload G					Workload H				
Cluster Size				1	3	6	9	12	1	3	6	9	12
Replication Factor				0	2	2	2	2	0	2	2	2	2
Type	Metric	Distribution	Consistency										
Read	Latency	Uniform	ONE	0	95.4	20.3	23.5	5.2	0	66.9	30.6	37.3	15.4
			QUORUM	0	102.6	25.0	26.7	5.0	0	15.3	43.3	50.9	8.7
			ALL	0	115.2	33.7	26.9	9.4	0	39.3	53.7	47.9	18.0
		Zipfian	ONE	0	97.1	28.6	9.2	0.0	0	0.2	16.6	18.8	20.1
			QUORUM	0	102.4	23.3	14.4	2.2	0	5.0	11.7	18.4	17.8
			ALL	0	103.4	32.4	16.0	0.5	0	73.6	12.2	5.5	11.9
		Latest	ONE	0	74.6	7.7	5.5	4.0	0	4.1	6.5	5.4	1.2
			QUORUM	0	75.3	14.4	5.7	2.5	0	79.3	10.3	9.1	5.8
			ALL	0	79.8	19.7	5.6	3.5	0	7.3	0.6	0.8	1.3
Write	Latency	Uniform	ONE	0	88.1	20.3	18.4	8.0	0	121.1	58.1	39.1	12.7
			QUORUM	0	94.4	20.9	18.7	4.2	0	106.3	67.3	37.9	3.8
			ALL	0	110.8	33.4	24.0	3.8	0	109.1	73.0	43.6	15.5
		Zipfian	ONE	0	87.5	22.2	7.5	0.7	0	83.5	27.6	9.8	13.9
			QUORUM	0	93.7	22.3	11.3	2.1	0	102.5	41.9	17.9	15.1
			ALL	0	93.6	19.2	12.4	1.7	0	104.6	44.5	13.4	15.8
		Latest	ONE	0	68.9	7.0	2.9	4.3	0	85.8	28.6	4.0	2.2
			QUORUM	0	70.2	13.8	0.2	4.3	0	43.3	32.5	4.3	5.1
			ALL	0	73.8	18.1	5.1	1.8	0	91.2	31.6	4.8	2.7
Overall	Throughput	Uniform	ONE	0	94.1	20.4	22.5	6.2	0	120.4	54.1	49.6	12.9
			QUORUM	0	95.4	23.7	25.4	6.7	0	99.0	64.8	49.0	15.3
			ALL	0	113.6	31.5	27.1	10.5	0	107.2	71.4	54.0	15.5
		Zipfian	ONE	0	93.6	28.2	9.3	1.7	0	78.4	24.9	21.0	14.6
			QUORUM	0	101.5	23.4	14.5	4.0	0	89.0	41.0	29.0	15.7
			ALL	0	102.5	30.1	15.5	2.5	0	97.8	43.0	24.3	16.2
		Latest	ONE	0	73.8	7.8	5.6	2.5	0	83.6	25.1	7.1	1.7
			QUORUM	0	74.0	14.0	5.7	1.0	0	37.4	30.7	15.6	4.5
			ALL	0	77.6	19.4	6.0	1.6	0	76.7	127.0	137.4	122.2

Table 5.6: MongoDB: Percentage Differences in Read & Write Latencies and Overall Throughput From Baseline Experiments per Workload, Broken Down by Distribution and Consistency Level.

				Workload G					Workload H				
Cluster Size				1	3	6	9	12	1	3	6	9	12
Replication Factor				0	2	2	2	2	0	2	2	2	2
Type	Metric	Distribution	Consistency										
Read	Latency	Uniform vs Zipfian	ONE	7.3	2.2	8.4	14.4	5.2	14.2	66.8	46.6	18.8	4.7
		Uniform vs Latest		13.3	25.3	12.6	18.0	9.2	9.2	70.5	36.9	42.6	16.5
		Uniform vs Zipfian	ALL	2.4	16.8	1.3	11.1	8.8	3.3	36.9	42.2	42.7	6.1
		Uniform vs Latest		18.2	45.9	14.2	21.4	12.9	24.8	32.3	53.2	48.6	19.3
		Uniform vs Zipfian	QUORUM	0.8	0.3	1.7	12.4	2.8	19.6	10.4	32.0	33.3	9.1
		Uniform vs Latest		18.8	33.9	10.7	21.0	7.5	26.2	91.8	53.0	42.3	2.9
Write	Latency	Uniform vs Zipfian	ONE	9.0	0.8	2.0	10.9	7.3	7.8	50.3	31.7	29.6	1.2
		Uniform vs Latest		11.3	22.7	13.3	15.5	12.3	3.7	47.6	30.8	42.9	14.9
		Uniform vs Zipfian	ALL	3.6	23.2	14.4	11.7	2.1	9.6	6.3	31.0	30.6	0.3
		Uniform vs Latest		18.4	46.4	15.6	18.9	5.6	10.6	23.8	43.9	39.0	12.8
		Uniform vs Zipfian	QUORUM	3.3	0.9	1.5	7.4	2.1	5.8	5.2	27.3	20.3	11.4
		Uniform vs Latest		16.3	29.0	7.1	18.4	8.4	8.5	71.2	36.8	33.8	1.4
Overall	Throughput	Uniform vs Zipfian	ONE	5.6	0.7	7.9	13.3	4.5	7.8	55.0	30.3	29.3	1.8
		Uniform vs Latest		13.6	24.6	12.6	17.0	8.7	3.3	49.2	30.0	42.9	14.6
		Uniform vs Zipfian	ALL	1.8	15.7	1.3	11.7	8.1	7.5	12.7	30.7	30.7	0.7
		Uniform vs Latest		18.4	46.2	12.2	21.1	12.1	7.4	27.4	43.2	41.2	12.8
		Uniform vs Zipfian	QUORUM	2.4	8.1	0.3	11.0	2.7	5.2	12.9	25.5	20.7	11.2
		Uniform vs Latest		17.4	25.9	9.8	19.8	7.7	9.1	67.9	35.9	34.0	0.1

Table 5.7: MongoDB: Percentage Differences In Read & Write Latencies and Overall Throughput Between Distributions for each Workload and Consistency Level.

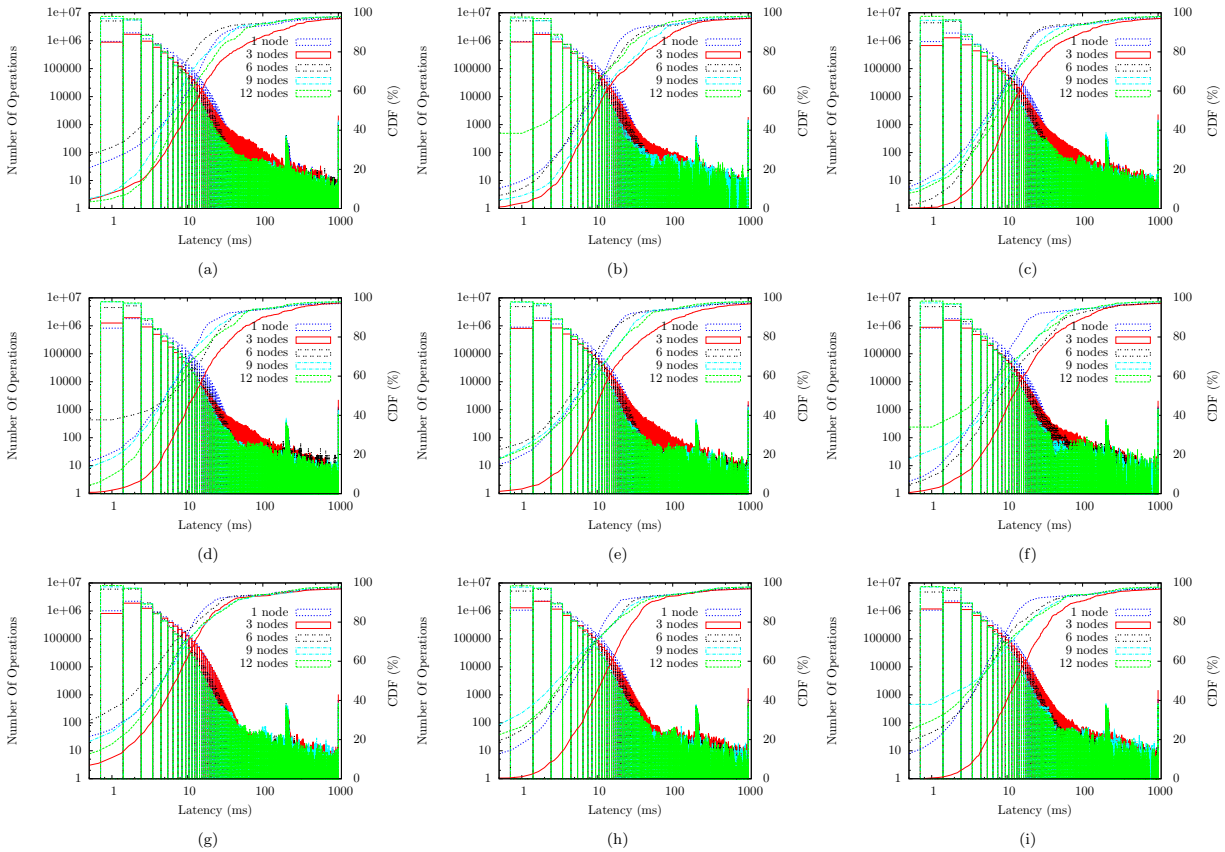

Figure 5.11: MongoDB Workload G Read Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

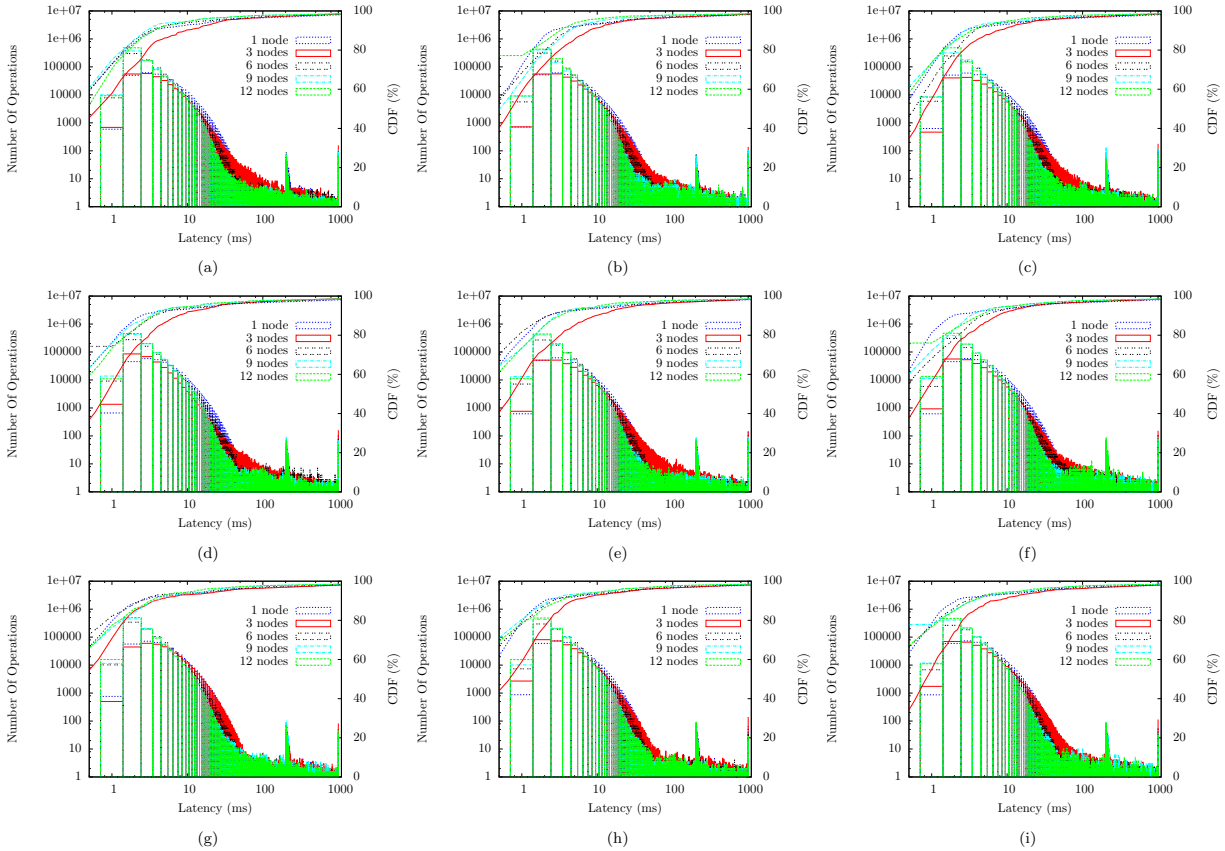

Figure 5.12: MongoDB Workload G Write Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

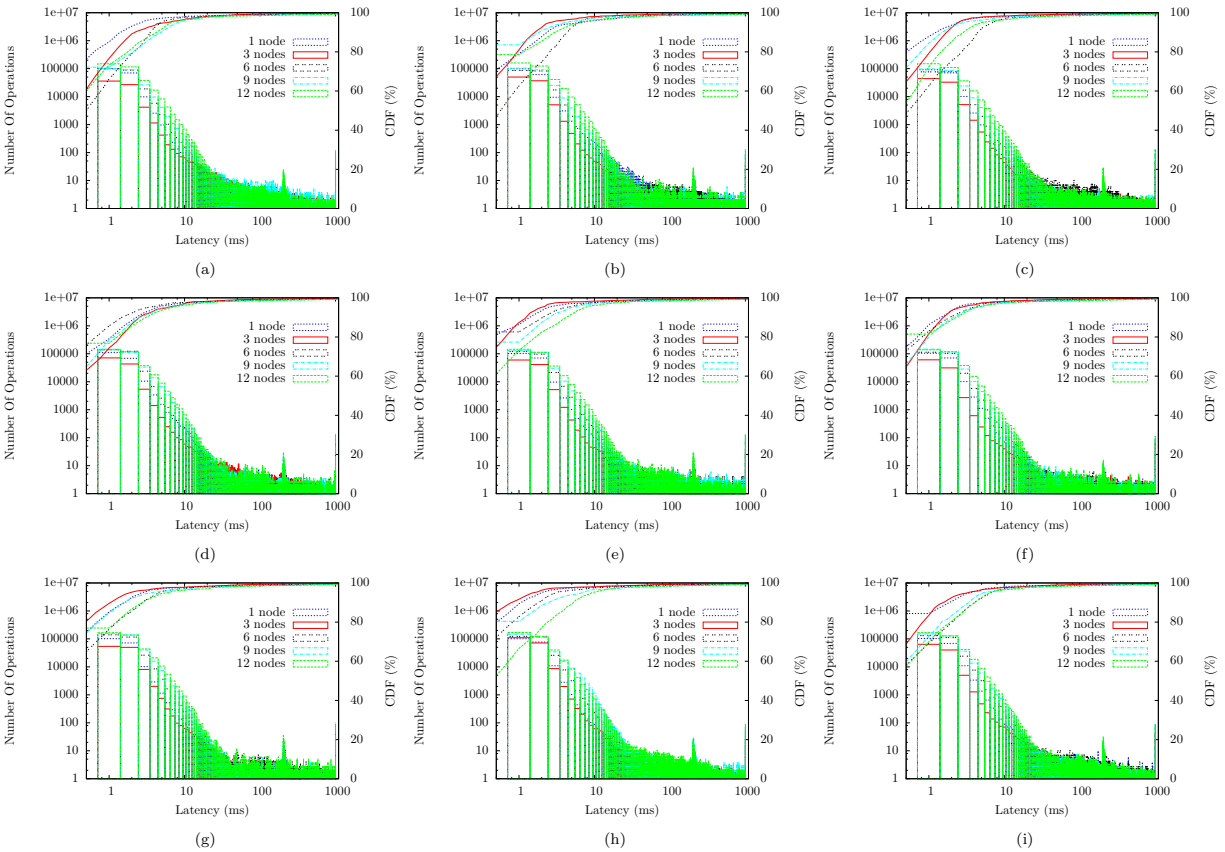

Figure 5.13: MongoDB Workload H Read Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

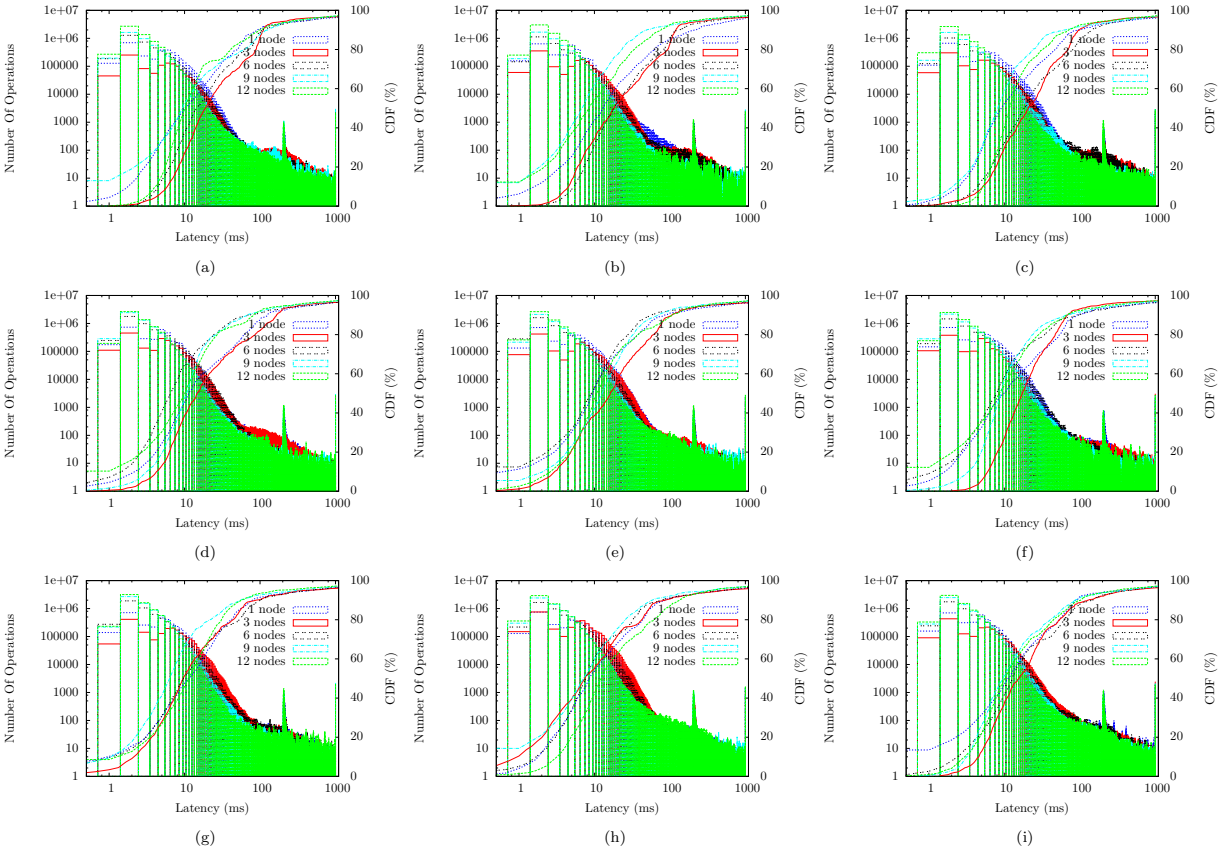

Figure 5.14: MongoDB Workload H Write Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

5.3 Cassandra

On a single non-replicated node, throughputs are 45.7% higher for the write-dominated workload (H) than the read-dominated workload (G). This is expected due to Cassandra's write optimized architecture which is supported by an observable average reduction in write latencies of 38.1% and 54% across all non-replicated cluster sizes for workloads G and H respectively compared to read latencies. Regarding throughputs, on non-replicated clusters workload H consistently outperforms workload G by an average of 33.1% per cluster size.

However, on cluster sizes greater than 1 and a replication factor greater than 0, we observe a 39.6%, 37.9%, and 30.3% decrease in throughput for the write-heavy workload (H) compared to workload G, on average across all cluster sizes and consistency levels for uniform, zipfian, and latest distributions respectively. This corresponds to a 19.5%, 38.6%, and 49.7% decrease on average across all cluster sizes and distributions for ONE, QUORUM, and ALL consistency levels respectively. This trend is illustrated in Figure 5.15.

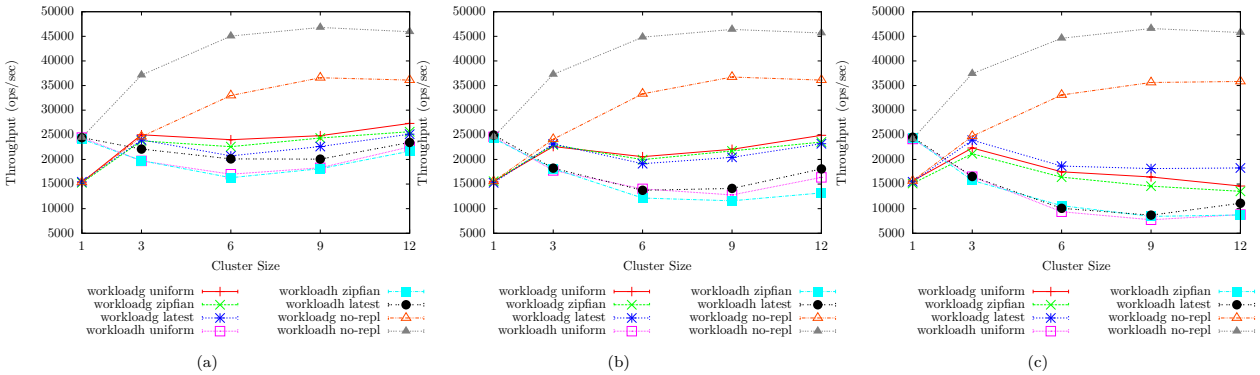

Figure 5.15: Cassandra: Overall Throughputs per Consistency Level for all Workloads and Distributions: (a) ONE (b) QUORUM (c) ALL.

Read latencies are considerably affected in write-heavy workloads when the replication factor of a cluster is greater than zero. We can observe an average percentage difference of 150% between workload G which is composed of 95% read operations and workload H which has only 5% read operations, across all consistency levels and distributions. This corresponds to 170.2%, 147.4%, and 132.5% increases for ONE, QUORUM, and ALL consistency levels respectively from workload G to H. As Table 5.11 illustrates, the average increase in read latency from 1 to 3 nodes (with a replication factor of 2) is 118.3%, and across subsequent cluster sizes and replication factors is 150%. Since the average increase in read latencies on the read-heavy workload (G) is only 40.5% compared to non-replicated clusters of equal size, the additional 110% average increase appears to be attributed to the significant increase in the number of write operations since all other factors remain constant.

This is likely due to a large number of write operations forcing MemTables to be constantly updated, which are saved to SSTables on disk more frequently and routinely compacted. Because commit logs and SSTables both reside on the same disk there is much greater contention and therefore increased latencies. The large amount of write operations also increases the potential for out-of-date data being returned on reads. Since the replication factor becomes quite large, there subsequently becomes much more read-repairs that are needed for read operations. This indicates that the availability of the cluster for read operations is considerably adversely affected by the number of write operations that are present in a workload.

The availability of write operations does not appear to be as adversely affected by the number of read operations in a given workload. For workload G, the average increase in write latencies is 72.9% compared to non-replicated clusters, however the average increase is similarly 71.2% for workload H across all distributions and consistency levels on cluster sizes greater than 3. This breaks down to 70.7%, 76.6%, and 71.3% increases for uniform, zipfian, and latest distributions and 36.1%, 50%, and 132.5% increases for ONE, QUORUM, and ALL consistency levels on workload G. Likewise, for workload H we observe a 73.1%, 75.5%, and 65.3% increase for uniform, zipfian, and latest distributions and 16.8%, 84.6%, and 112.4% increases for ONE, QUORUM, and ALL consistency levels. A further breakdown of these figures are presented in Table 5.10.

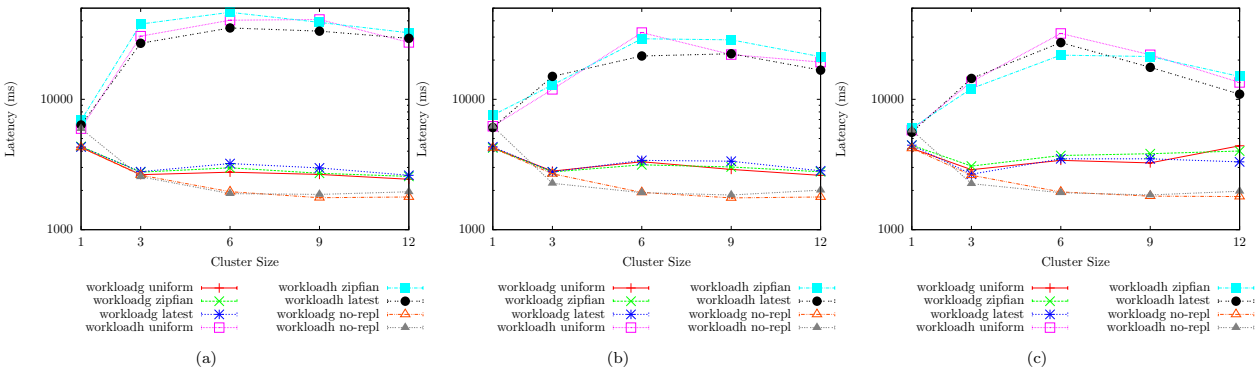

Figure 5.16: Cassandra: Read Latencies per Consistency Level for all Workloads and Distributions: (a) ONE (b) QUORUM (c) ALL.

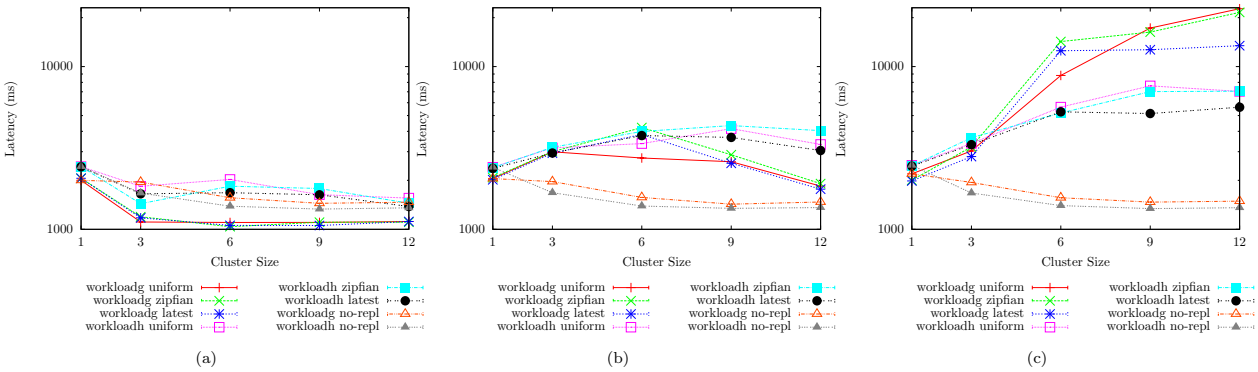

Figure 5.17: Cassandra: Write Latencies per Consistency Level for all Workloads and Distributions: (a) ONE (b) QUORUM (c) ALL.

When comparing write latencies between workloads we observe an average increase of 44.5% from workload G to H on replicated clusters. This corresponds to 45.6%, 46.6%, and 41.6% increases for uniform zipfian, and latest distributions respectively and 39.2%, 28.7%, and 65.8% increases for ONE, QUORUM, and ALL consistency levels respectively. These metrics are further broken down in Table 5.11.

Write latencies drop when multiple nodes are added to a cluster with and without replication for a consistency level of ONE only, on both workloads. This is because more nodes are able to service write operations resulting in less load on a single host. Separate hard disks are also able to process commit log entries leading to less contention.

Overall, write latencies tend to be lower than read latencies on both workloads, applicable to all distributions and consistency levels as illustrated in Figures 5.16 and 5.17. This is supported by the existing benchmarks highlighted in Section 2 and Cassandra's architecture design.

To ensure the correct record is returned when serving read requests; first all versions of the record are collected from all MemTables and SSTables. Cassandra must perform these costly disk seeks and wait for responses synchronously. Then it must perform read repairs as necessary (in the background) if records are detected as out of date.

Write operations however are sequential append only operations to commit logs which act like in-memory operations. All other tasks are backgrounded including saving to commit logs and SSTables, and compacting the layout of data on disk for better performance, therefore the speed of the hard disk is the only possible limiting factor on performance.

The impact of varying consistency levels and replication factors can be observed by viewing the average standard deviations from mean throughputs across all replicated cluster sizes. For workload G and consistency levels ONE, QUORUM, and ALL we observe 44.9%, 38.8%, and 37.3% increases in throughput from 1 to 3 nodes which then maintain an average of 24161.8, 22015.5, and 17961.2 operations per second with standard deviations of 6.9%, 7.7%, and 17.8% across all cluster sizes greater than 1 for ONE, QUORUM, and ALL consistency levels respec-

tively. For workload H we see 11.2%, 16.8%, and 29.9% deviations from their respective mean throughputs on cluster sizes greater than 1. As expected, performance is most affected by the strictest consistency level ALL.

This suggests that Cassandra is scalable at the cost of maintaining a lower level of consistency. However, stronger consistency levels tend to reduce scalability as the cluster size and replication factor increase. The QUORUM consistency level demonstrates a promising *sweet spot* in the consistency versus throughput tradeoff battle. However, stricter consistencies have a much greater impact on write-heavy workloads than on read-heavy workloads.

Considering the impact that varying levels of consistency have relative to non-replicated clusters of equal size, we observe a consistent ordering of performance metrics for both workloads. For workload G, we see a 28.8% decrease in throughput for consistency level ONE on average for all distributions and cluster sizes with a replication factor greater than zero compared to non-replicated clusters of equal size. An additional 26.3%, and 65.6% decrease in throughput is observed for QUORUM and ALL consistency levels respectively. Regarding workload H, consistency level ONE has a decreased throughput of 74.6%, with an additional 29.4%, and 46.1% decreases in throughput for consistency levels QUORUM and ALL respectively compared to non-replicated clusters of equal size.

These results are expected as increasingly more nodes are required to confirm each operation resulting in additional overhead and reduced performance. This trend is a reflection of Cassandra's CAP theorem properties favoring availability and network partition intolerance over consistency.

We expect and subsequently observe that all three consistency levels have approximately equal throughputs and latencies on a single node cluster since they all only have one node to confirm operations with.

Throughputs for fully consistent transactions tend to continue to decline on larger cluster sizes, likely because the number of nodes that need to write to commit logs increase. The replication factor also increases so there is more contention for partitions within a single server contending for the same hard disk. Additionally, more replicas mean more read repairs are required from MemTables and SSTables resulting in both in-memory and costly on-disk operations. The fact that the strictest consistency level is more downwards sloping in contrast to the others and we notice this degradation more predominantly on the ready heavy workload is a direct result of Cassandra delaying consistency checks to read time.

For consistency level ONE on a read-heavy workload (G), the throughput on a 3 node cluster is pretty similar to a non-replicated cluster of size 3. This is most likely due to the ability of any node in the cluster to service read requests. Since all 3 nodes in the cluster will have a full copy of the data due to a replication factor of 2, any node will be able to process read requests without considerable overhead of routing to the correct node and acting as a coordinator to the YCSB Client.

For workload G we observe that the uniform distribution on average outperforms the zipfian and latest distributions by 4.2% and 0.8% respectively. However, for workload H the latest distribution on average outperforms the uniform and zipfian distributions by 7.1% and 9.7% respectively. See Table 5.9 for a more in-depth break down.

Zipfian's poorer performance could be related to compaction. Since one key is frequently updated in the zipfian distribution, it is very likely that the mutations all end up in flushed SSTables on disk. Therefore, a greater amount of disk access is required particularly for reading. Compaction is supposed to prevent the database from having to perform costly seeks to pull data from each SSTable. However, the cost of compaction is designed to be amortized over the lifetime of the server. Since the lifetime of experiments in this study is only 10 minutes, the effect of compaction might not be noticeable and hence the observable impact on the zipfian distribution.

YCSB's method of picking a node to forward requests to is done so by selecting a node uniformly at random. The chosen Cassandra node will then act as a coordinator in the transaction. This is likely to impact relative performance between distributions, favoring the uniform distribution due to a stronger correlation in their random number generators. The uniform distribution will also spread the requested data more evenly throughout the cluster, enabling individual nodes to better catch up with background read-repairs, compactions, commit log appends and SSTable flushes.

Figures 5.18, 5.19, 5.20, and 5.21 illustrate the latency histograms for read and write opera-

tions on both workloads. As we can see from the CDF curves plotted on the same figures, 95% of operations can be answered in less than 4.5 ms, 4.8 ms, 90.7 ms, and 3.1 ms for read and write operations on workloads G and H respectively, on average across all cluster sizes, distributions, and consistency levels. A more informative breakdown of these figures are presented in Table 5.8.

				Workload G					Workload H				
Cluster Size				1	3	6	9	12	1	3	6	9	12
Replication Factor				0	2	4	6	8	0	2	4	6	8
Type	Metric	Distribution	Consistency										
Read	Latency (ms)	Uniform	ONE	4.278	2.643	2.759	2.654	2.437	5.961	30.49	40.408	40.918	27.307
			QUORUM	4.242	2.814	3.321	2.897	2.61	6.207	11.921	32.489	22.044	19.274
			ALL	4.203	2.885	3.402	3.253	4.399	5.8	13.634	31.97	21.947	13.466
		Zipfian	ONE	4.348	2.764	2.982	2.707	2.563	6.894	37.737	46.478	38.903	32.358
			QUORUM	4.235	2.771	3.144	3.027	2.78	7.552	12.876	29.124	28.571	21.204
			ALL	4.444	3.076	3.707	3.824	4.012	6.012	12.084	21.804	21.282	14.961
		Latest	ONE	4.308	2.787	3.216	2.977	2.612	6.349	26.889	35.251	33.335	29.342
			QUORUM	4.32	2.781	3.401	3.351	2.827	6.072	14.981	21.482	22.343	16.731
			ALL	4.449	2.664	3.495	3.498	3.313	5.582	14.447	27.219	17.583	10.943
	95th Percentile	Uniform	ONE	9	5	3	3	3	7	143	182	198	90
			QUORUM	10	5	3	3	3	9	37	170	78	76
			ALL	10	5	3	3	2	7	43	184	106	48
		Zipfian	ONE	9	5	4	3	3	8	227	247	171	141
			QUORUM	9	5	3	3	3	11	50	127	126	78
			ALL	8	5	3	2	2	8	31	104	95	57
		Latest	ONE	9	5	3	3	3	8	131	168	137	136
			QUORUM	9	5	3	3	3	8	76	85	86	59
			ALL	8	5	3	2	2	8	60	138	86	38
Write	Latency (ms)	Uniform	ONE	2	1.109	1.102	1.103	1.117	2.43	1.838	2.025	1.632	1.556
			QUORUM	2.05	2.993	2.744	2.603	1.843	2.407	3.165	3.363	4.157	3.324
			ALL	2.193	3.032	8.825	17.259	22.698	2.473	3.371	5.639	7.617	7.036
		Zipfian	ONE	2.058	1.194	1.038	1.105	1.106	2.428	1.437	1.84	1.782	1.447
			QUORUM	2.078	2.999	4.221	2.872	1.909	2.387	3.194	3.993	4.339	4.041
			ALL	2.003	3.211	14.259	16.304	21.545	2.453	3.657	5.189	7.02	7.061
		Latest	ONE	2.058	1.176	1.057	1.055	1.117	2.423	1.652	1.679	1.63	1.381
			QUORUM	2.018	2.94	3.827	2.545	1.766	2.369	2.945	3.769	3.67	3.05
			ALL	1.984	2.801	12.518	12.702	13.44	2.445	3.312	5.271	5.15	5.631
	95th Percentile	Uniform	ONE	7	2	2	2	2	4	2	2	1	2
			QUORUM	7	6	3	3	2	4	3	2	3	3
			ALL	7	6	5	6	22	4	4	3	5	5
		Zipfian	ONE	6	2	1	2	1	4	2	1	1	1
			QUORUM	6	6	3	2	2	4	3	3	3	4
			ALL	6	5	5	5	20	4	4	4	5	5
		Latest	ONE	6	2	1	1	1	4	2	1	2	1
			QUORUM	6	6	3	2	2	4	3	3	3	3
			ALL	6	6	6	6	7	4	3	4	3	4
Overall	Throughput	Uniform	ONE	15351	25009	24002	24821	27313	24446	19707	16992	18303	22551
			QUORUM	15546	22622	20554	22062	24894	24577	17766	14027	12786	16349
			ALL	15641	22323	17490	16432	14597	24152	16491	9371	7749	8789
		Zipfian	ONE	15258	23775	22597	24346	25626	24142	19756	16293	18117	21671
			QUORUM	15565	22949	19965	21731	23528	24257	18129	12151	11596	13196
			ALL	15113	21153	16405	14570	13519	24358	15727	10609	8428	8715
		Latest	ONE	15465	23971	20732	22617	25133	24439	22103	20101	20055	23445
			QUORUM	15252	23146	19125	20422	23188	24957	18232	13733	14112	18065
			ALL	15460	23994	18654	18138	18259	24509	16535	10066	8687	11098
	95% CI	Uniform	ONE	1.96	7.84	11.76	15.68	15.68	3.92	9.8	15.68	13.72	9.8
			QUORUM	5.88	3.92	27.44	7.84	9.8	3.92	5.88	19.6	11.76	23.52
			ALL	5.88	13.72	9.8	15.68	33.32	3.92	7.84	11.76	5.88	5.88
		Zipfian	ONE	7.84	3.92	19.6	7.84	3.92	5.88	11.76	15.68	13.72	13.72
			QUORUM	5.88	3.92	9.8	17.64	11.76	9.8	19.6	0	7.84	9.8
			ALL	13.72	13.72	23.52	9.8	13.72	9.8	9.8	1.96	9.8	7.84
		Latest	ONE	11.76	13.72	15.68	19.6	7.84	5.88	11.76	27.44	11.76	17.64
			QUORUM	5.88	11.76	15.68	23.52	13.72	3.92	11.76	3.92	9.8	25.48
			ALL	19.6	1.96	31.36	29.4	31.36	3.92	5.88	5.88	31.36	13.72
Replication Factor				0	0	0	0	0	0	0	0	0	0
Type	Metric	Distribution	Consistency										
Read	Latency (ms)	Uniform	ONE	4.278	2.622	1.955	1.76	1.782	5.961	2.548	1.89	1.865	1.953
	95th Percentile	Uniform	ONE	9	6	3	2	2	7	4	3	2	3
Write	Latency (ms)	Uniform	ONE	2	1.951	1.564	1.445	1.468	2.43	1.671	1.387	1.332	1.355
	95th Percentile	Uniform	ONE	7	5	3	2	2	4	3	2	2	2
Overall	Throughput	Uniform	ONE	15351	24643	32983	36595	36119	24446	37128	45054	46803	45937
	95% CI	Uniform	ONE	1.96	1.96	3.92	3.92	3.92	3.92	3.92	5.88	3.92	1.96

Table 5.8: Cassandra: Read & Write Latency & 95th Percentiles and Overall Throughput & 95th% Confidence Interval Data per Workload, Broken Down by Distribution and Consistency Level.

				Workload G					Workload H				
Cluster Size				1	3	6	9	12	1	3	6	9	12
Replication Factor				0	2	4	6	8	0	2	4	6	8
Type	Metric	Distribution	Consistency										
Read	Latency	Uniform vs Zipfian	ONE	1.6	4.5	7.8	2.0	5.0	14.5	21.2	14.0	5.0	16.9
		Uniform vs Latest		0.7	5.3	15.3	11.5	6.9	6.3	12.6	13.6	20.4	7.2
		Uniform vs Zipfian	ALL	5.6	6.4	8.6	16.1	9.2	3.6	12.1	37.8	3.1	10.5
		Uniform vs Latest		5.7	8.2	2.7	7.3	28.2	3.8	5.8	16.1	22.1	20.7
		Uniform vs Zipfian	QUORUM	0.2	1.5	5.5	4.4	6.3	19.6	7.7	10.9	25.8	9.5
		Uniform vs Latest		1.8	1.2	2.4	14.5	8.0	2.2	22.7	40.8	1.3	14.1
Write	Latency	Uniform vs Zipfian	ONE	2.9	7.4	6.0	0.2	1.0	0.1	24.5	9.6	8.8	7.3
		Uniform vs Latest		2.9	5.9	4.2	4.4	0.0	0.3	10.7	18.7	0.1	11.9
		Uniform vs Zipfian	ALL	9.1	5.7	47.1	5.7	5.2	0.8	8.1	8.3	8.2	0.4
		Uniform vs Latest		10.0	7.8	34.6	30.4	51.2	1.1	1.8	6.7	38.6	22.2
		Uniform vs Zipfian	QUORUM	1.4	0.2	42.4	9.8	3.5	0.8	0.9	17.1	4.3	19.5
		Uniform vs Latest		1.6	1.8	33.0	2.3	4.3	1.6	7.2	11.4	12.4	8.6
Overall	Throughput	Uniform vs Zipfian	ONE	0.6	5.1	6.0	1.9	6.4	1.3	0.2	4.2	1.0	4.0
		Uniform vs Latest		0.7	4.2	14.6	9.3	8.3	0.0	11.5	16.8	9.1	3.9
		Uniform vs Zipfian	ALL	3.4	5.4	6.4	12.0	7.7	0.8	4.7	12.4	8.4	0.8
		Uniform vs Latest		1.2	7.2	6.4	9.9	22.3	1.5	0.3	7.2	11.4	23.2
		Uniform vs Zipfian	QUORUM	0.1	1.4	2.9	1.5	5.6	1.3	2.0	14.3	9.8	21.3
		Uniform vs Latest		1.9	2.3	7.2	7.7	7.1	1.5	2.6	2.1	9.9	10.0

Table 5.9: Cassandra: Percentage Differences In Read & Write Latencies and Overall Throughput Between Distributions for each Workload and Consistency Level.

				Workload G					Workload H				
Cluster Size				1	3	6	9	12	1	3	6	9	12
Replication Factor				0	2	4	6	8	0	2	4	6	8
Type	Metric	Distribution	Consistency										
Read	Latency	Uniform	ONE	0	0.8	34.1	40.5	31.1	0	169.2	182.1	182.6	173.3
			QUORUM	0	7.1	51.8	48.8	37.7	0	129.6	178.0	168.8	163.2
			ALL	0	9.6	54.0	59.6	84.7	0	137.0	177.7	168.7	149.3
		Zipfian	ONE	0	5.3	41.6	42.4	35.9	0	174.7	184.4	181.7	177.2
			QUORUM	0	5.5	46.6	52.9	43.8	0	133.9	175.6	175.5	166.3
			ALL	0	15.9	61.9	73.9	77.0	0	130.3	168.1	167.8	153.8
		Latest	ONE	0	6.1	48.8	51.4	37.8	0	165.4	179.6	178.8	175.0
			QUORUM	0	5.9	54.0	62.3	45.3	0	141.9	167.7	169.2	158.2
			ALL	0	1.3	56.5	66.1	60.1	0	140.0	174.0	161.6	139.4
Write	Latency	Uniform	ONE	0	55.0	34.7	26.8	27.2	0	9.5	37.4	20.2	13.8
			QUORUM	0	42.2	54.8	57.2	22.7	0	61.8	83.2	102.9	84.2
			ALL	0	43.4	139.8	169.1	175.7	0	67.4	121.0	140.5	135.4
		Zipfian	ONE	0	48.1	40.4	26.7	28.1	0	15.1	28.1	28.9	6.6
			QUORUM	0	42.3	91.9	66.1	26.1	0	62.6	96.9	106.0	99.6
			ALL	0	48.8	160.5	167.4	174.5	0	74.5	115.6	136.2	135.6
		Latest	ONE	0	49.6	38.7	31.2	27.2	0	1.1	19.0	20.1	1.9
			QUORUM	0	40.4	84.0	55.1	18.4	0	55.2	92.4	93.5	77.0
			ALL	0	35.9	155.6	159.1	160.6	0	65.9	116.7	117.8	122.4
Overall	Throughput	Uniform	ONE	0	1.5	31.5	38.3	27.8	0	61.3	90.5	87.5	68.3
			QUORUM	0	8.6	46.4	49.6	36.8	0	70.5	105.0	114.2	129.6
			ALL	0	9.9	61.4	76.0	84.9	0	77.0	131.1	143.2	135.8
		Zipfian	ONE	0	3.6	37.4	40.2	34.0	0	61.1	93.8	88.4	71.8
			QUORUM	0	7.1	49.2	51.0	42.2	0	68.8	115.0	120.6	110.7
			ALL	0	15.2	67.1	86.1	91.1	0	81.0	123.8	139.0	136.2
		Latest	ONE	0	2.8	45.6	47.2	35.9	0	50.7	76.6	80.0	64.8
			QUORUM	0	6.3	53.2	56.7	43.6	0	68.3	106.6	107.3	87.1
			ALL	0	2.7	55.5	67.4	65.7	0	76.7	127.0	137.4	122.2

Table 5.10: Cassandra: Percentage Differences in Read & Write Latencies and Overall Throughput From Baseline Experiments per Workload, Broken Down by Distribution and Consistency Level.

Cluster Size				1	3	6	9	12
Replication Factor				0	2	4	6	8
Type	Metric	Distribution	Consistency					
Read	Latency	Uniform	ONE	32.9	168.1	174.4	175.6	167.2
			QUORUM	37.6	123.6	162.9	153.5	152.3
			ALL	31.9	130.1	161.5	148.4	101.5
		Zipfian	ONE	45.3	172.7	175.9	174.0	170.6
			QUORUM	56.3	129.2	161.0	161.7	153.6
			ALL	30.0	118.8	141.9	139.1	115.4
		Latest	ONE	38.3	162.4	166.6	167.2	167.3
			QUORUM	33.7	137.4	145.3	147.8	142.2
			ALL	22.6	137.9	154.5	133.6	107.0
Write	Latency	Uniform	ONE	19.4	49.5	59.0	38.7	32.8
			QUORUM	16.0	5.6	20.3	46.0	57.3
			ALL	12.0	10.6	44.1	77.5	105.3
		Zipfian	ONE	16.5	18.5	55.7	46.9	26.7
			QUORUM	13.8	6.3	5.6	40.7	71.7
			ALL	20.2	13.0	93.3	79.6	101.3
		Latest	ONE	16.3	33.7	45.5	42.8	21.1
			QUORUM	16.0	0.2	1.5	36.2	53.3
			ALL	20.8	16.6	81.5	84.6	81.9
Overall	Throughput	Uniform	ONE	45.7	23.7	34.2	30.2	19.1
			QUORUM	45.0	24.0	37.7	53.2	41.4
			ALL	42.8	30.1	60.5	71.8	49.7
		Zipfian	ONE	45.1	18.5	32.4	29.3	16.7
			QUORUM	43.7	23.5	48.7	60.8	56.3
			ALL	46.8	29.4	42.9	53.4	43.2
		Latest	ONE	45.0	8.1	3.1	12.0	6.9
			QUORUM	48.3	23.8	32.8	36.5	24.8
			ALL	45.3	36.8	59.8	70.5	48.8

Table 5.11: Cassandra: Percentage Differences In Read & Write Latencies and Overall Throughput Between Workloads for each Distribution and Consistency Level.

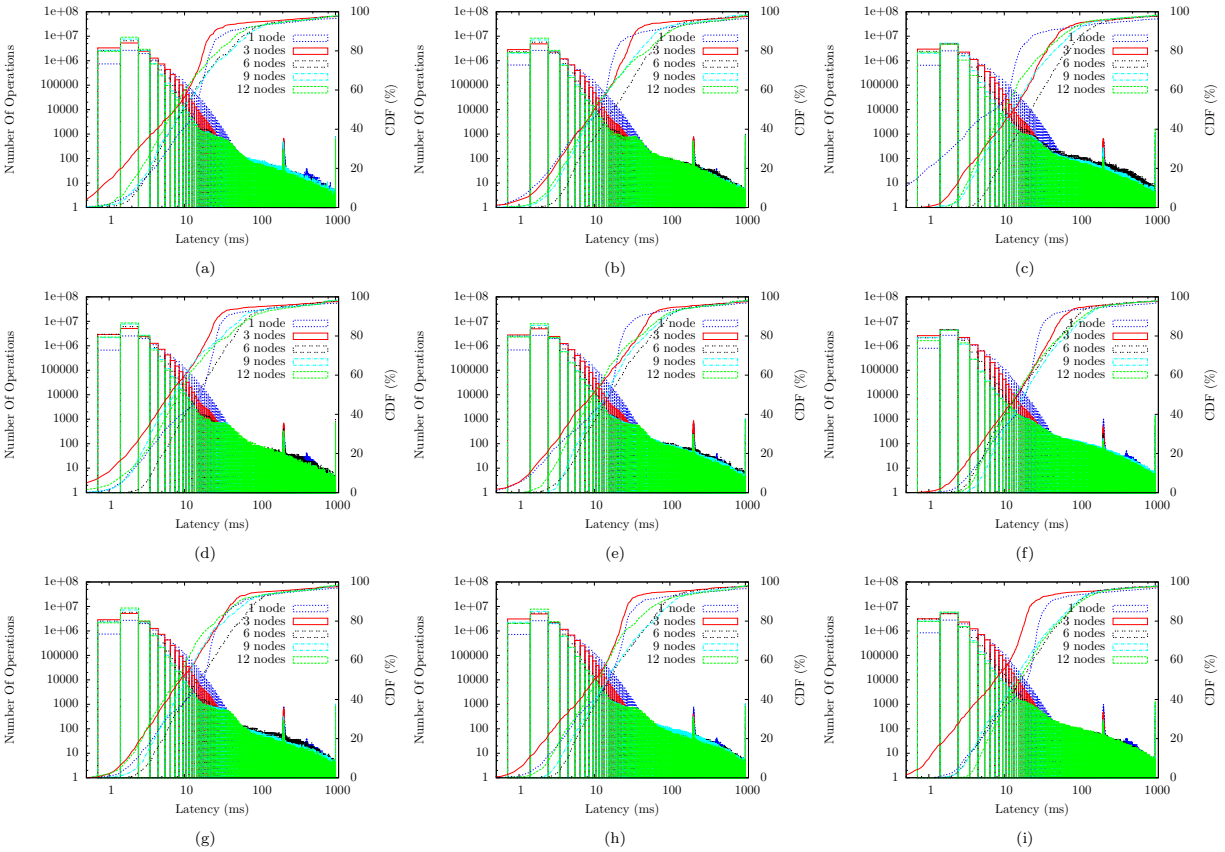

Figure 5.18: Cassandra Workload G Read Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

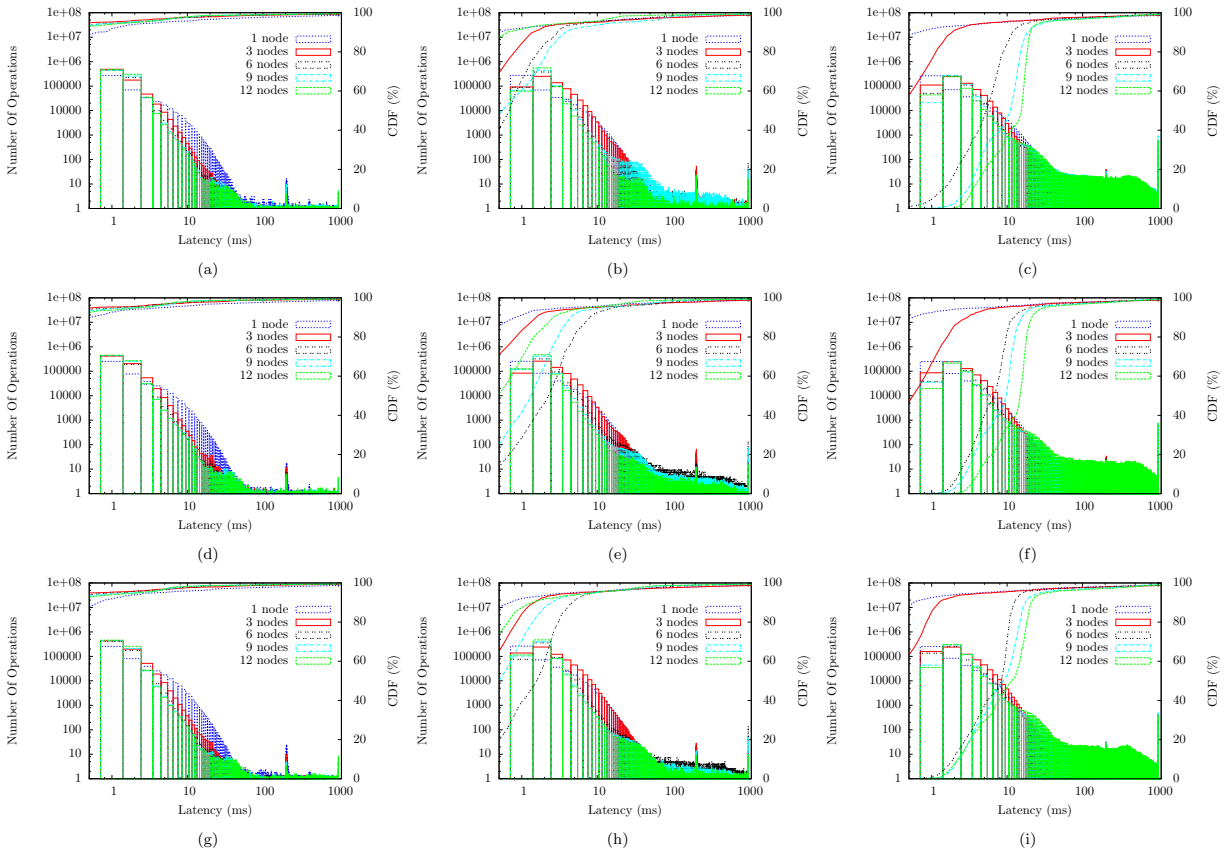

Figure 5.19: Cassandra Workload G Write Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

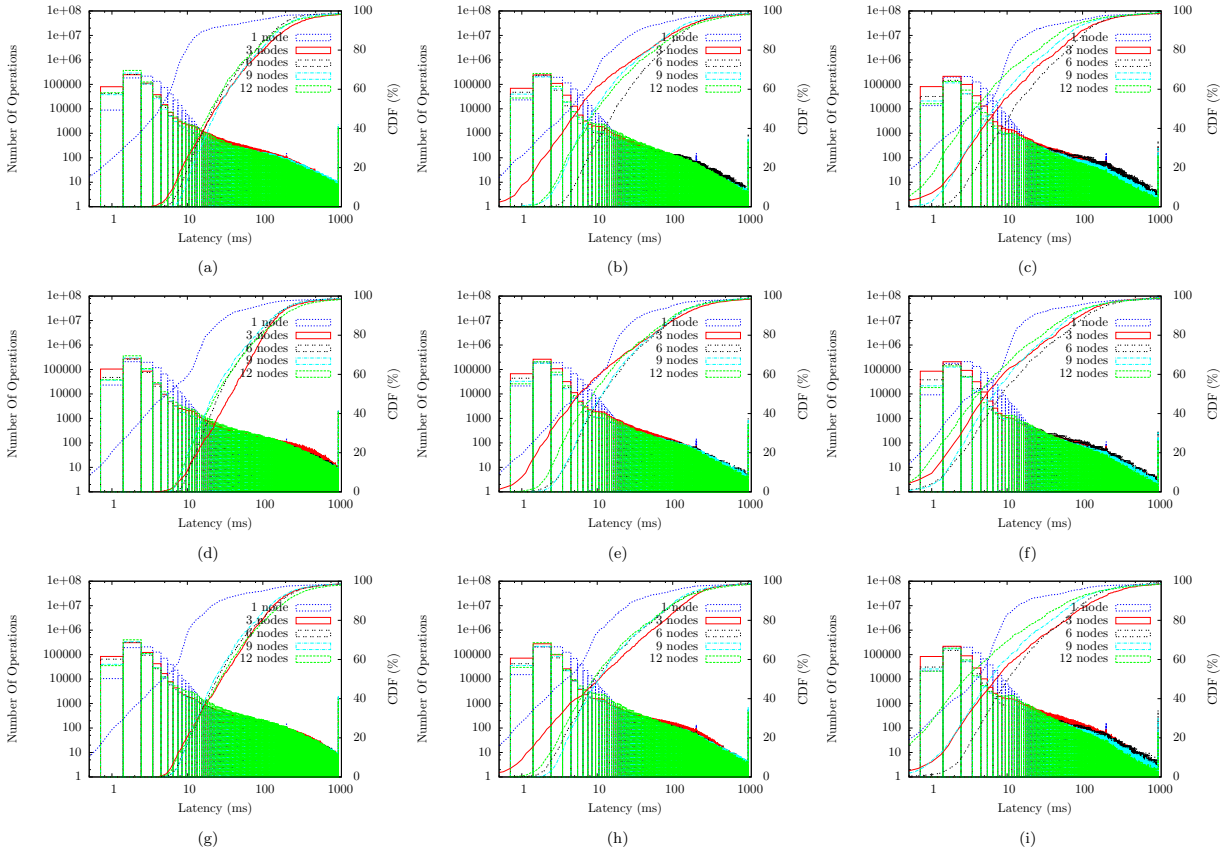

Figure 5.20: Cassandra Workload H Read Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

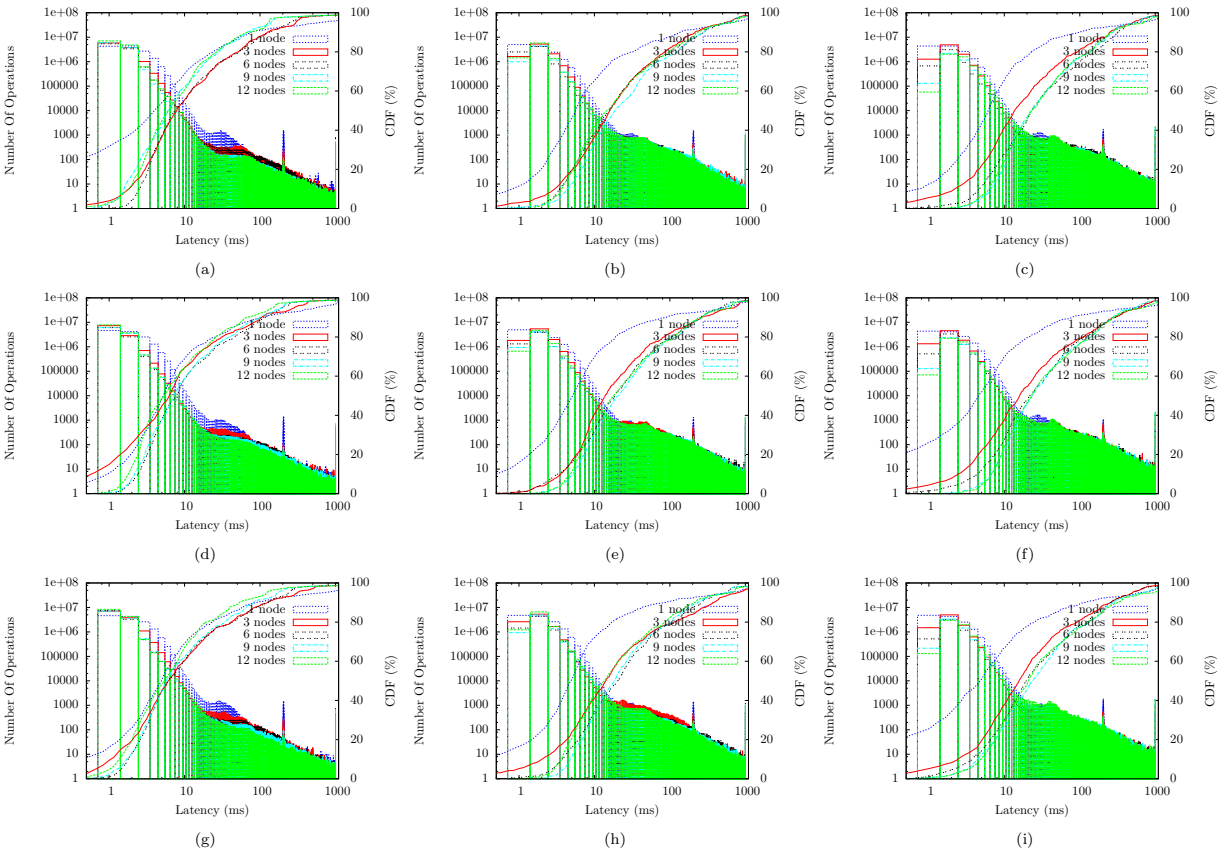

Figure 5.21: Cassandra Workload H Write Latency Histograms: (a) Uniform ONE (b) Uniform QUORUM (c) Uniform ALL (d) Zipfian ONE (e) Zipfian QUORUM (f) Zipfian ALL (g) Latest ONE (h) Latest QUORUM (i) Latest ALL.

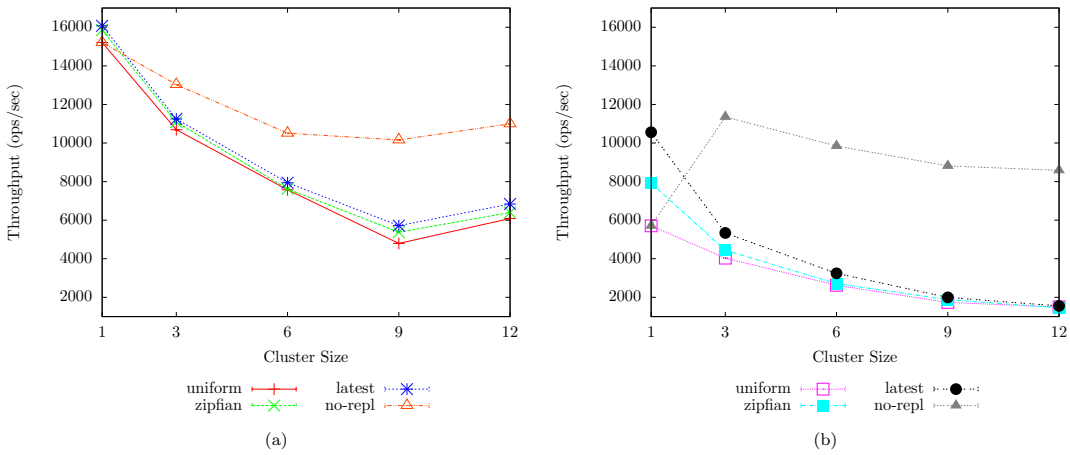

Figure 5.22: VoltDB: Overall Throughput per Distribution: (a) Workload G (b) Workload H.

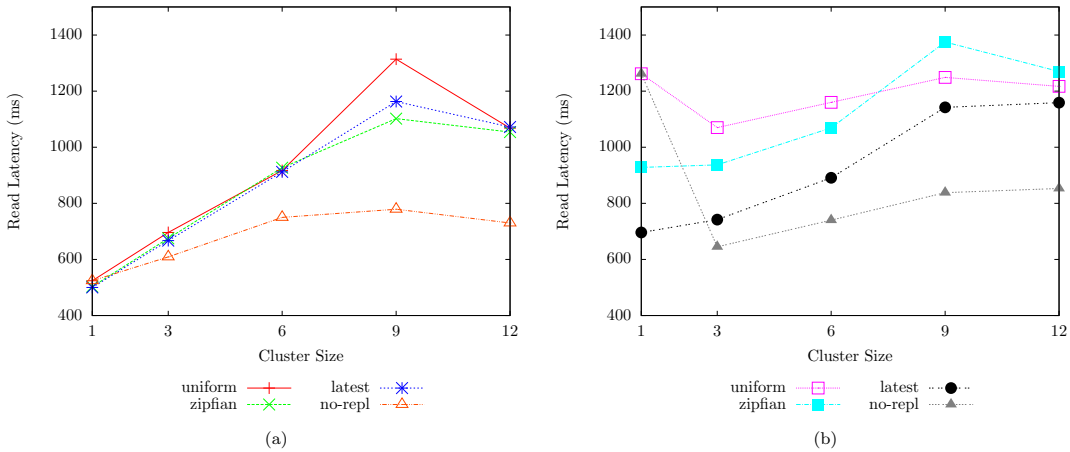

Figure 5.23: VoltDB: Read Latencies per Distribution: (a) Workload G (b) Workload H.

5.4 VoltDB

As Figure 5.22 illustrates, both workloads report a steady decline in throughput as the cluster size increases before starting to level off, which is an indication of VoltDB's ability to scale at larger cluster sizes after a certain overhead of maintaining a small number of nodes. This trend is applicable to both replicated and non-replicated clusters.

On average we see a 45.3%, 40.6%, and 36.3% decrease in throughput for uniform, zipfian, and latest distributions respectively across all cluster sizes with a replication factor greater than zero compared to non-replicated clusters of equal size for workload G. However, we observe greater decreases in throughput for workload H with 121.4%, 118.3%, and 109.5% decreases in throughput on average across all cluster sizes with a replication factor greater than zero for uniform, zipfian, and latest distributions respectively. Details of these metrics are included in Table 5.12. This considerable difference is due to the synchronous replication model which VoltDB employs for write operations.

For cluster sizes larger than 3, it's expected that the throughput would remain constant since the number of distinct partitions (*i.e.* 9) also remain constant. However, the slight decline in performance is most likely a side effect of the YCSB Client routing all operations to the same node, requiring additional work on that node's part to distribute read and write operations appropriately.

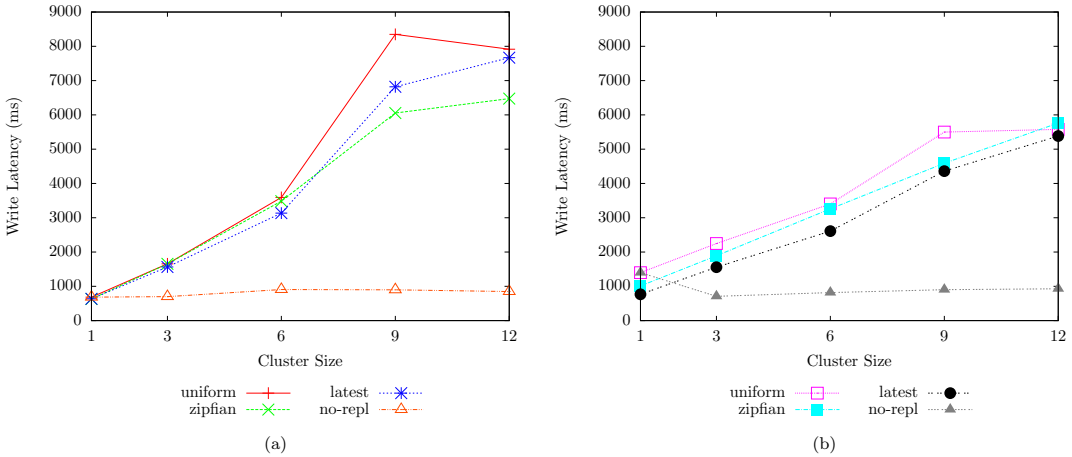

Figure 5.24: VoltDB: Write Latencies per Distribution: (a) Workload G (b) Workload H.

			Workload G					Workload H				
Cluster Size			1	3	6	9	12	1	3	6	9	12
Replication Factor			0	2	4	6	8	0	2	4	6	8
Type	Metric	Distribution										
Read	Latency	Uniform	0	13.3	20.0	51.1	37.6	0	49.6	44.2	39.4	35.2
		Zipfian	0	10.3	21.1	34.3	36.3	0	36.9	36.4	48.5	39.3
		Latest	0	9.1	19.5	39.5	38.0	0	14.0	18.5	30.7	30.4
Write	Latency	Uniform	0	81.2	119.4	161.2	161.3	0	104.2	122.7	143.6	142.9
		Zipfian	0	81.3	117.4	148.4	153.7	0	91.0	119.9	134.3	144.5
		Latest	0	76.4	110.2	153.5	160.2	0	75.0	104.7	131.4	141.1
Overall	Throughput	Uniform	0	19.7	32.4	71.8	57.5	0	95.2	115.9	134.1	140.5
		Zipfian	0	16.1	31.8	61.6	53	0	87.6	113.6	129.7	142.1
		Latest	0	14.7	27.8	55.9	46.6	0	72.1	100.9	126.2	138.8

Table 5.12: VoltDB: Percentage Differences in Read & Write Latencies and Overall Throughput From Baseline Experiments per Workload, Broken Down by Distribution.

The latency for reads across all cluster sizes and distributions remain fairly similar between workloads and in comparison to non-replicated clusters of equal size, as illustrated in Figure 5.23. We observe a 22%, and 28.2% increase in read latencies on average across all cluster sizes and distributions on workload G, and H respectively compared to non-replicated clusters of equal size. This suggests that reads remain consistently available as the cluster size and replication factor increases.

However, we observe stark contrasts in read latencies in workload H which consists of only 5% read operations compared to workload G on smaller cluster sizes. In order to gain insight into this observation, additional experiments where conducted to illustrate the impact of workload composition on read latencies on a single node cluster. The results of these experiments are illustrated in Table 5.13. We can clearly see that as the percentage of read operations in a workload increases the latency decreases and subsequently throughput increases almost proportionally. For example, a 15% increase in the number of read operations in a workload from 5 to 15% corresponds to a 12% decrease in latency and 18% increase in throughput.

The cause of this phenomenon is likely that the large number of (more expensive) write operations in the workload impact the availability of a single node cluster for subsequent read operations. Since all data is located on a single node with 6 partitions, each of which is single threaded, the latency for reads is therefore bound by the latency of preceding writes which are more frequent and take longer to process due to synchronous replication. The reason latency and throughput measurements start to improve as the cluster size increases is likely due to data being spread across more servers, with less contention for single-threaded access.

The latency for write operations across all cluster sizes and distributions is considerably higher in comparison to non-replicated clusters of equal size, as illustrated in Figure 5.24. We observe a 133.1%, and 110.3% increase in write latencies on average across all cluster sizes and distributions on workload G and H respectively compared to non-replicated clusters of equal size. It's interesting however that the percentage difference as the cluster increases from 9 to 12 nodes, increases only by 3.7% and 6% for workload G and H respectively. This contrasts to an average increase in percentage difference to non-replicated clusters of 51.7% and 45.6% for workload G and H respectively each time the cluster grows by 3 additional nodes (until a total

% Reads In Workload	Latency (ms)	Throughput (ops/sec)
5	1.262	5706
20	1.122	6865
40	0.859	8875
80	0.580	12489
95	0.525	15216
100	0.416	19076

Table 5.13: VoltDB: Percentage of Workload Reads Impact on Read Latency and Overall Throughput.

cluster size of 9). This suggests that the availability for write operations is increasingly affected on smaller cluster sizes, but potentially stable on larger clusters. A possible explanation for this is related to the synchronous replication model affecting performance as more replicas are added, which appears to be amortized on larger clusters due to horizontal scalability.

We also notice the impact synchronous replication has on workload G which consists of only 5% write operations. An average increase in write latencies of 127% is observed compared to non-replicated clusters across all cluster sizes (greater than 1) and distributions. This compares similarly to an average increase in write latencies of 121.3% in workload H. This suggests the availability of the cluster for write operations is affected consistently and independently of the underlying workload composition within a cluster with a replication factor greater than zero.

Since the total number of partitions available in each cluster is not an integral multiple of the cluster size, a single host is likely to store more than 1 copy of the same data partition. This would result in larger latencies due to increased contention when writing to the same command log and snapshotting to the same disk, a known performance bottleneck [38]. Throughputs would also be lower since all client invocations are synchronous.

Figures 5.27, 5.28, 5.29, and 5.30 illustrate the latency histograms for read and write operations on both workloads. As we can see from the CDF curves plotted on the same figures, 95% of operations can be answered in less than 0.5 ms, 1.9 ms, 0.9 ms, and 2.1 ms for read and write operations on workloads G and H respectively, on average across all cluster sizes, and distributions. A more informative breakdown of these figures is presented in Tables 5.17.

Comparatively, workload G performs consistently better than workload H, reporting a 14.8% increase in throughput on average for non-replicated clusters and 98.6%, 94%, and 83.9% average increases in performance for uniform, zipfian, and latest distributions respectively on replicated clusters. Figure 5.25 demonstrates the comparative performance of both workloads in terms of throughput and latencies.

This contrast in performance can be attributed to VoltDB's handling of reads versus writes. When writing, VoltDB synchronously replicates data to additional nodes therefore incurring larger impacts on performance. Additionally, command logging and snapshotting is happening on the same disk which increases disk contention resulting in larger latencies. When reading however, since all data is in-memory and no consistency checks are performed between replicas, we observe much higher throughputs at lower latencies.

Read and write latencies show much smaller differences, averaging 13.1% and 27.1% decreases on average across all replicated cluster sizes and distributions respectively for workload G compared to workload H. On non-replicated clusters however, the average differences are as low as 6.8% and 0.1% respectively. These figures are summarized in Table 5.14.

Across both workloads the performance rankings between distributions remain almost perfectly consistent, regarding throughputs, read latencies, and write latencies. Uniform performs worse that zipfian and latest, with 5% and 8.9% decreases in throughput performance respectively for workload G. Larger decreases of 11.3% and 25.2% are observed for workload H. Table 5.16 illustrates these findings along with read and write latencies, which follow similar trends. An interesting phenomenon appears in workload H. As the cluster size increases, we observe the performance difference between distributions start to become quite small. This suggests at larger cluster sizes the difference between each distribution could become negligible.

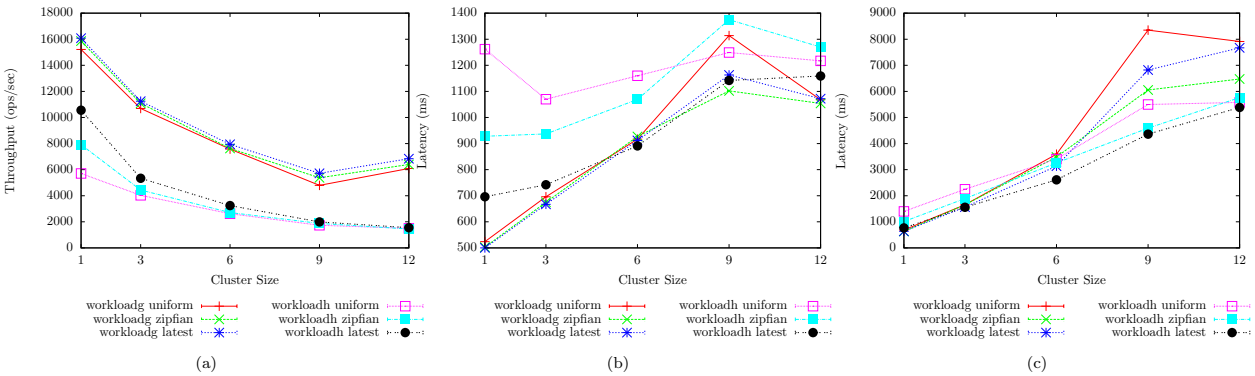

Figure 5.25: VoltDB: Combined Performance Metrics for each Workload and Distribution: (a) Throughputs (b) Read Latencies (c) Write Latencies.

Cluster Size			1	3	6	9	12
Replication Factor			0	2	4	6	8
Type	Metric	Distribution					
Read	Latency	Uniform	82.6	42.4	23.4	5.1	13.0
		Zipfian	59.4	32.5	14.2	22.0	18.6
		Latest	32.8	10.6	2.3	1.8	7.8
Write	Latency	Uniform	68.5	30.4	5.4	41.2	34.6
		Zipfian	46.3	13.3	6.6	27.5	11.7
		Latest	18.5	0.4	18.3	44.0	35.0
Overall	Throughput	Uniform	90.9	90.5	97.2	93.6	120.9
		Zipfian	66.8	85.7	95.1	96.3	126.0
		Latest	41.5	71.3	84.0	96.6	126.0
Replication Factor			0	0	0	0	0
Type	Metric	Distribution					
Read	Latency	Uniform	82.6	5.7	1.3	7.3	15.5
Write	Latency	Uniform	68.5	1.3	10.6	0.7	9.1
Overall	Throughput	Uniform	90.9	13.7	6.5	14.2	24.7

Table 5.14: VoltDB: Percentage Differences In Read & Write Latencies and Overall Throughput Between Workloads for each Distribution.

Command logging is enabled by default on the Enterprise edition of VoltDB, as such an additional set of experiments where conducted without command logging or replication to see what effect this had on performance and to verify if it followed a similar trend. Figure 5.26 indicates that it does follow a similar trend (*i.e.* a slight downwards trend) and on average results in a 72% increase in throughput for workload G, and a 76.7% increase in throughput for workload H compared to non-replicated clusters of equal size greater than 1 with the default settings enabled. The percentage difference figures per cluster size is indicated in Table 5.15.

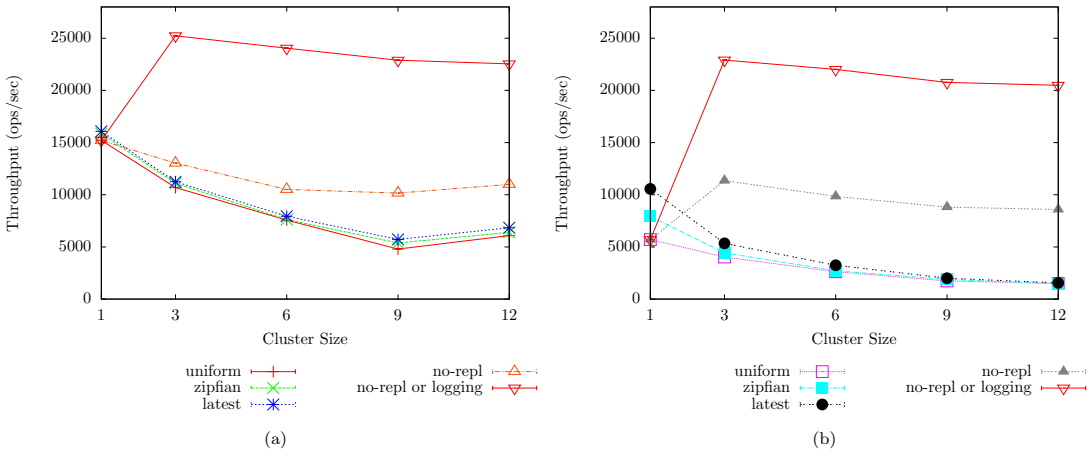

Figure 5.26: VoltDB Overall Throughputs per Distribution and Baseline Experiments with no Command Logging: (a) Workload G (b) Workload H.

	Throughput % Difference	
Cluster Size	Workload G	Workload H
1	-	-
3	63.806	67.4605
6	78.3771	76.3929
9	77.0599	80.8154
12	68.8411	81.8932
avg:	*72.02102*	*76.6405*

Table 5.15: VoltDB: Overall Throughput Differences Between No-Replication and No-Replication or Command Logging Experiments.

			Workload G						Workload H					
Cluster Size			1	3	6	9	12		1	3	6	9	12	
Replication Factor			0	2	4	6	8		0	2	4	6	8	
Type	Metric	Distribution						*avg*						*avg*
Read	Latency	Uniform vs Zipfian	4.1	3.1	1.1	17.5	1.3	5.4	30.5	13.3	8.2	9.6	4.3	13.2
		Uniform vs Latest	4.7	4.3	0.5	12.2	0.4	4.4	57.8	36.2	26.2	9.0	4.9	26.8
Write	Latency	Uniform vs Zipfian	8.4	0.1	3.1	31.8	20.0	12.7	32.3	17.3	4.4	18.0	3.3	15.0
		Uniform vs Latest	7.4	5.7	13.6	20.1	3.1	10.0	58.5	36.3	26.4	23.1	3.5	29.6
Overall	Throughput	Uniform vs Zipfian	4.2	3.7	0.6	11.4	5.0	5.0	32.4	9.6	3.3	7.9	3.3	11.3
		Uniform vs Latest	5.5	5.1	4.7	17.6	11.6	8.9	59.7	27.9	21.2	13.8	3.3	25.2

Table 5.16: VoltDB: Percentage Differences In Read & Write Latencies and Overall Throughput Between Distributions for each Workload.

			Workload G					Workload H				
Cluster Size			1	3	6	9	12	1	3	6	9	12
Replication Factor			0	2	4	6	8	0	2	4	6	8
Type	Metric	Distribution										
Read	Latency (ms)	Uniform	0.524	0.696	0.917	1.314	1.068	1.262	1.07	1.16	1.249	1.217
		Zipfian	0.503	0.675	0.927	1.102	1.054	0.928	0.937	1.069	1.375	1.27
		Latest	0.5	0.667	0.912	1.163	1.072	0.696	0.742	0.891	1.142	1.159
	95th Percentile	Uniform	0	0	1	1	0	1	1	1	1	1
		Zipfian	0	0	1	1	1	1	1	1	1	1
		Latest	0	1	1	1	0	0	0	1	1	1
Write	Latency (ms)	Uniform	0.685	1.655	3.592	8.348	7.914	1.399	2.249	3.404	5.498	5.577
		Zipfian	0.63	1.656	3.483	6.055	6.476	1.01	1.891	3.259	4.591	5.763
		Latest	0.636	1.564	3.134	6.82	7.669	0.766	1.558	2.609	4.36	5.384
	95th Percentile	Uniform	0	1	2	3	4	1	2	2	3	3
		Zipfian	0	1	2	2	4	1	1	2	3	4
		Latest	0	1	2	3	3	0	1	2	3	3
Overall	Throughput	Uniform	15216	10690	7577	4794	6087	5706	4031	2622	1737	1501
		Zipfian	15862	11089	7625	5375	6398	7916	4438	2711	1880	1452
		Latest	16080	11249	7942	5719	6838	10558	5338	3244	1995	1552
	95% CI	Uniform	9.80	3.92	1.96	1.96	13.72	0	15.68	11.76	11.76	1.96
		Zipfian	9.80	5.88	5.88	9.80	13.72	5.88	7.84	11.76	5.88	1.96
		Latest	13.72	9.80	9.80	11.76	21.56	9.80	5.88	7.84	5.88	1.96
Replication Factor			0	0	0	0	0	0	0	0	0	0
Type	Metric	Distribution										
Read	Latency (ms)	Uniform	0.524	0.609	0.75	0.779	0.73	1.262	0.645	0.74	0.838	0.853
	95th Percentile	Uniform	0	0	1	1	1	1	0	1	1	1
Write	Latency (ms)	Uniform	0.685	0.699	0.907	0.897	0.848	1.399	0.708	0.816	0.903	0.929
	95th Percentile	Uniform	0	1	1	1	1	1	0	1	1	1
Overall	Throughput	Uniform	15216	13029	10507	10161	10998	5706	11353	9845	8814	8584
	95% CI	Uniform	9.8	9.80	5.88	5.88	11.76	0	7.84	5.88	1.96	1.96

Table 5.17: VoltDB: Read & Write Latency & 95th Percentiles and Overall Throughput & 95th% Confidence Interval Data per Workload, Broken Down by Distribution.

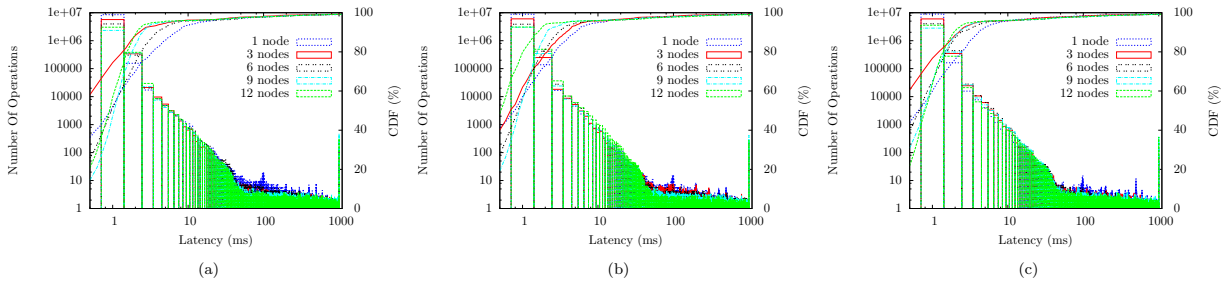

Figure 5.27: VoltDB Workload G Read Latency Histograms: (a) Uniform (b) Zipfian (c) Latest.

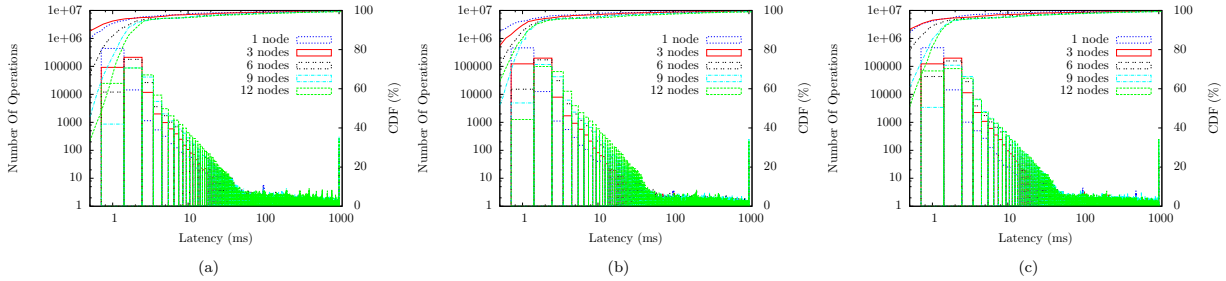

Figure 5.28: VoltDB Workload G Write Latency Histograms: (a) Uniform (b) Zipfian (c) Latest.

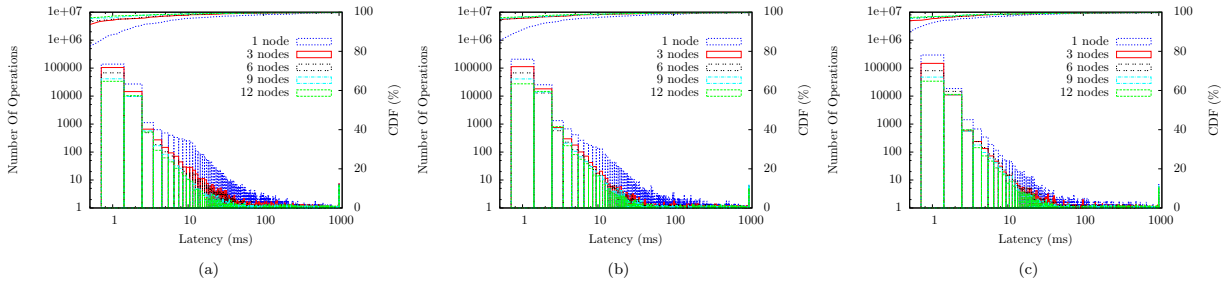

Figure 5.29: VoltDB Workload H Read Latency Histograms: (a) Uniform (b) Zipfian (c) Latest.

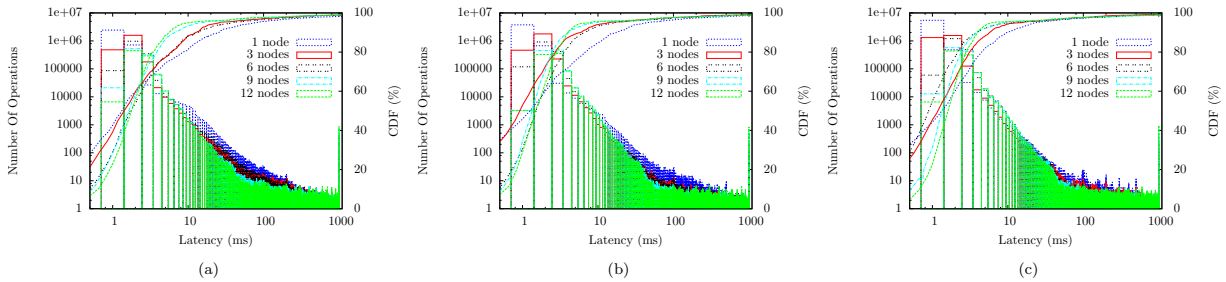

Figure 5.30: VoltDB Workload H Write Latency Histograms: (a) Uniform (b) Zipfian (c) Latest.

5.5 Comparative Analysis

While not the primary focus of this study, there are a number of comparisons that can be drawn between each data store. Based on the data and material presented so far, this section looks at several key things including: which data store offers the highest throughput and lowest latencies; what impact did each distribution have relative to each data store; and finally, which replication model had the least impact on baseline performances.

Redis, which averages 35298 ops/sec across all cluster sizes, consistency levels, and distributions outperforms MongoDB (averaging 21230 ops/sec) by 49.8%. MongoDB is only marginally better than Cassandra (which averages 20184 ops/sec) by 5.1%. Finally Cassandra outperforms VoltDB which averages 9236 ops/sec by 74.4% on the read-heavy workload (G).

A similar trend is observable for workload H which is write dominated, however the greatest difference is that Cassandra outperforms MongoDB by 72.5%. This stark contrast is a clear indication of Cassandra's write optimized architecture. Figures 5.31a and 5.32a illustrate how each data store performs relatively as the cluster size increases, for workload G and H respectively.

Overall, Redis outperforms all other data stores due its simplistic nature. Additional replicas do not have an impact on performance, and in-memory updates to a single hash table are quick, constant time operations. Additionally, varying consistency levels (due to a lack of supportability) and distributions have no impact on performance. VoltDB is the worst performer, most likely as a result of its strictly consistent architecture.

If we consider latencies, Redis again outperforms the other data stores by a large margin. Read latencies are 115.1% and 135% lower for workload G and H respectively compared to the next performant data store: VoltDB, with 174.9% and 163% lower write latencies also. A similar trend between MongoDB and Cassandra is observable for read and writes latencies as we saw for throughputs. MongoDB's read latencies are lower that Cassandra's (5.9% and 136.2% for workload G and H respectively). However, Cassandra's write latencies are much lower than MongoDB's (19.3% and 102% for workload G and H respectively).

The fact that Redis and VoltDB have lower latencies is expected due to the fact they are both in-memory data stores. Again, Redis's hash table data structure results in lower latencies since they are constant time operations. Figures 5.31b and 5.32b illustrate how each data store's read latencies differ as the cluster size increases, for workload G and H respectively. Similarly, Figures 5.31c and 5.32c illustrate how each data store's write latencies differ as the cluster size increases, for workload G and H respectively.

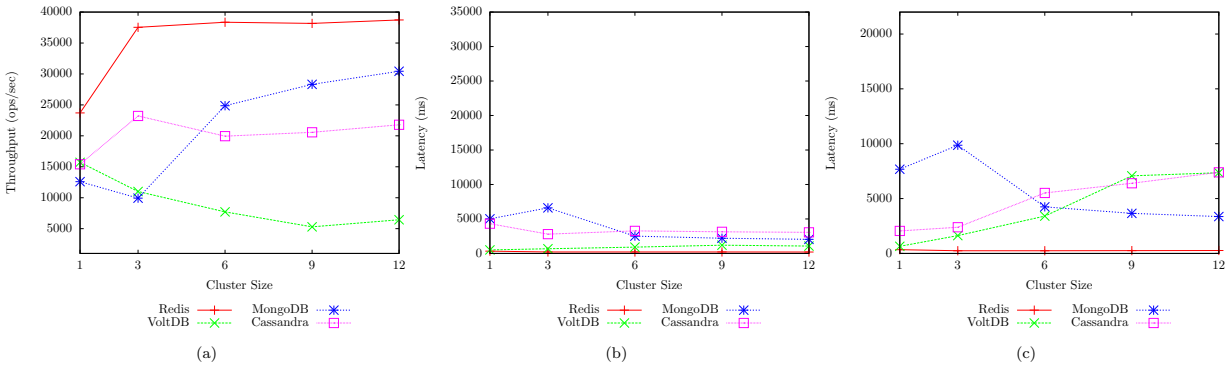

Figure 5.31: Comparison of Data Stores averaged across all distributions and consistency levels for Workload G: (a) Overall Throughput (b) Read Latency (c) Write Latency.

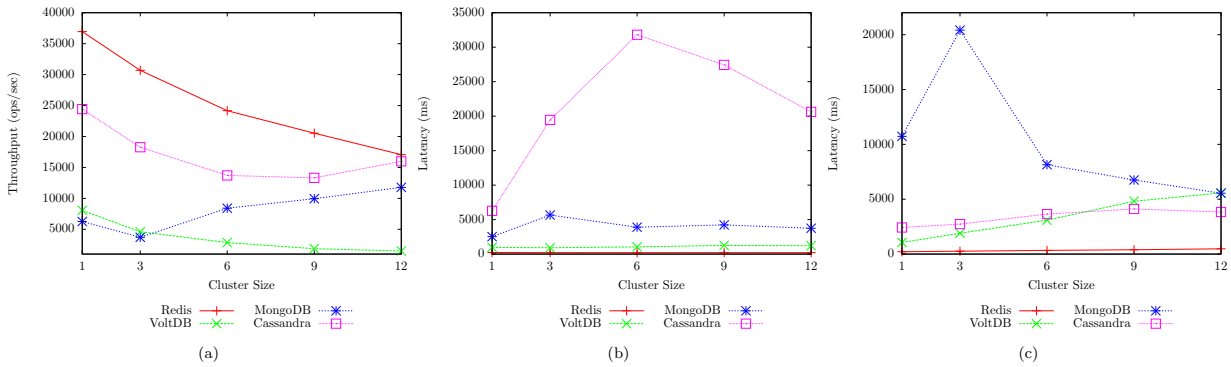

Figure 5.32: Comparison of Data Stores averaged across all distributions and consistency levels for Workload H: (a) Overall Throughput (b) Read Latency (c) Write Latency.

In workload G, all data stores demonstrate better performances with the latest distribution, except Cassandra which performs best with the uniform distribution. Redis consistently outperforms MongoDB on all distributions followed by Cassandra then VoltDB. Again, the only exception to this observation is that Cassandra has better throughputs than MongoDB with the uniform distribution. Cassandra's better performance on read-heavy workloads with a uniform distribution is likely a result of a strong correlation between how the YCSB Client selects a node to route requests to uniformly at random. This would spread the requests more evenly across the cluster. Whereas the latest distribution would force the same set of nodes to constantly handle operations, causing a backlog of read-repairs to build up. When accessed with the latest distribution, Redis is 1.5 times more performant than MongoDB, which is only 1.1 times more performant than Cassandra, which is 2.1 times more performant than VoltDB.

The latest distribution once again in workload H outperforms all other distributions on average across all cluster sizes and consistency levels, followed by zipfian, except for Cassandra which performs second best with the uniform distribution. When all data stores are accessed with the latest distribution, Redis is 1.4 times more performant than Cassandra which is 2 times better than MongoDB which is again 2 time better than VoltDB. The reason we observe larger contrasts in relative performance compared to workload G, is because Cassandra is write optimized, MongoDB performs consistency checks at write time, and VoltDB synchronously replicates data to replicas therefore incurring the largest impact on performance.

If we consider the impact replication has on clusters of equal size and compare two different replication strategies *i.e.* the multi-master model used by Cassandra, and the replica set model used by MongoDB. We can observe that apart from the exception of consistency level ONE on workload G, MongoDB's replica set replication model has less of an impact on throughput performance than Cassandra's multi-master replication model compared to non-replicated clusters. Cassandra's replication model accounts for a 41.1%, and 98% throughput performance degradation on all consistency levels and distributions, averaged across all replicated clusters sizes for workload G and H respectively. Whereas MongoDB's replication model only accounts for 33% and 52% degradation's in throughput performance for workload G and H respectively. Figures 5.33a and 5.34a illustrate how each replication strategy performs relatively and in relation to base line performances as the cluster size increases, for workload G and H respectively.

Both read and write latencies are more adversely affected with Cassandra's multi-slave replication model compared to non-replicated clusters of equal size. We observe 40.5% and 72.9% increases in read and write latencies for workload G. More prominently however, we observe 164.4% and 71.3% increases in read and write latencies on workload H. More modest increases in latencies are achieved for MongoDB's replica set replication model compared to non-replicated clusters of equal size. We observe 33.8% and 30.3% increases in read and write latencies for workload G and similarly 21.4% and 42.1% increases in read and write latencies respectively for workload H. This suggests that a master-slave replication model has less of an effect on clusters than a multi-master model. Figures 5.33b and 5.34b illustrate how each replication strategy's read latencies differ as the cluster size increases, for workload G and H respectively and compared to base line performances. Similarly, Figures 5.33c and 5.34c illustrate how each replication strategy's write latencies differ as the cluster size increases, for workload G and H respectively and compared to base line performances.

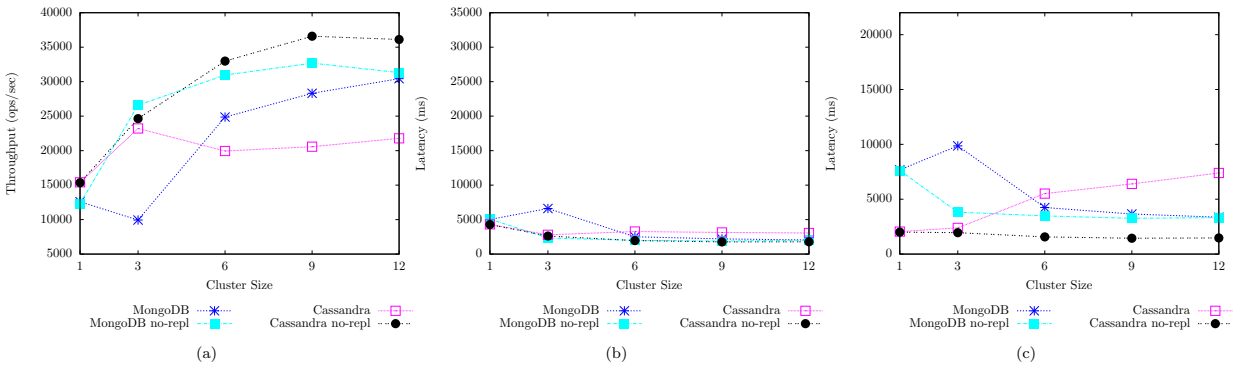

Figure 5.33: Comparison of Replication strategies: Multi-Master (Cassandra) and Replica Sets (MongoDB), averaged across all distributions and consistency levels for Workload G: (a) Overall Throughput (b) Read Latency (c) Write Latency.

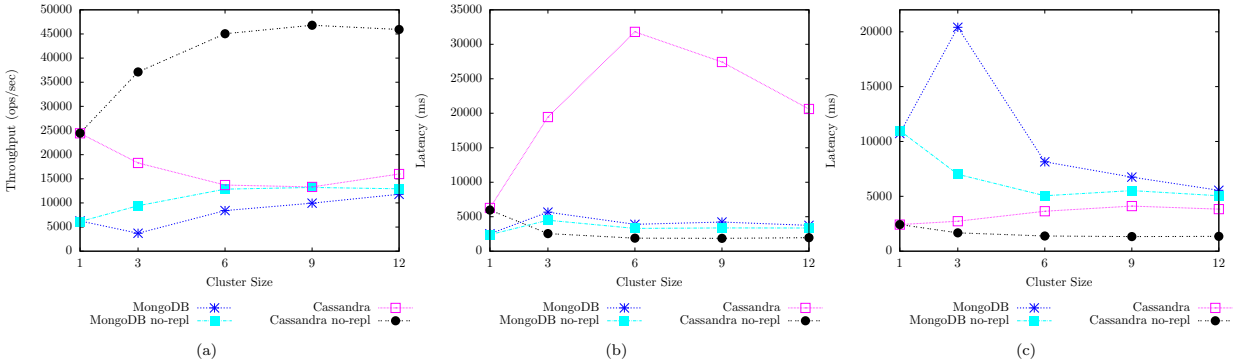

Figure 5.34: Comparison of Replication strategies: Multi-Master (Cassandra) and Replica Sets (MongoDB), averaged across all distributions and consistency levels for Workload H: (a) Overall Throughput (b) Read Latency (c) Write Latency.

6 Conclusion

This study looked at performance benchmarking replication in four cloud serving NoSQL data stores, focusing specifically on how performance and availability was affected compared to non-replicated clusters of equal size. The YCSB benchmarking tool was used to evaluate data stores of different categorization and replication models. These include key-value, document, extensible-record and distributed DBMS data stores and different variations of master-slave, and multi-master replication models.

To increase the applicability of this study to real-world use cases, a range of different data distributions (uniform, zipfian, and latest) were explored along with three levels of tunable consistency: ONE, QUORUM, and ALL, and two different workloads: one read-heavy and one write-heavy.

6.1 Benchmarking Results

Considering the extensive number and breadth of experiments conducted in this study, there are a number of key observations that have been made, the most significant of which are summarized below.

Redis:

- Read operations remain highly and consistently available in replicated clusters. Increasing the replication factor greater than 2 results in consistent performance metrics and no additional benefit for read operations.
- The availability for write operations are affected on average by 19.2% each time the cluster size is increased with 3 additional replicas.
- There are minimal differences between distributions since Redis keeps all data in memory on a single master.
- Varying levels of tunable consistency have no impact on performance due to Redis' lack of consistency guarantees and its efficient implementation.

MongoDB:

- Read heavy workloads outperform write-heavy workloads on average by a considerable margin: 88.7% across all cluster sizes, distributions and consistency levels.
- Replication has minimal impact on performance relative to non-replicated clusters of equal size once the overhead of maintaining a small number of shards have been overcome.
- Reads remain highly available (with 10% variance) as the cluster size grows with a constant replication factor on more than one shard.
- Writes tend to be half as available as reads (20% variance) when the replication factor is kept constant as the cluster size grows.
- Increasing the level of consistency reduces overall performance by increasingly small amounts. Between consistency levels ONE and ALL, a 4.8% and 5.9% drop in throughput for workload G and H respectively is observed.
- Performance is best when the entire or majority of a working data set can be kept in RAM, as it would be for latest and zipfian distributions.

Cassandra:

- Write latencies are lower than read latencies, however throughputs follow an opposite trend (35.9% decrease on average for writes) on replicated clusters.
- Cassandra is scalable at the cost of maintaining a lower level of consistency. 65.6% and 74.6% degradations are observed between consistency levels ONE and ALL for workload G and H respectively, on average across all replicated cluster sizes and distributions.
- Stricter consistency levels have a greater impact (9%) on write-heavy workloads than on read-heavy workloads.
- The availability of a replicated cluster for read operations is considerably impacted by the number of write operations than are present in a workload. Increases in latencies of 150% are observed on average across all cluster sizes, consistency levels, and distributions when increasing the write portion of a workload by 90%.

- Uniform distributions are 4.2% more performant on read-heavy workloads. However, the latest distribution is 7.1% more performant in write-heavy workloads.

VoltDB:

- VoltDB appears to scale well at cluster sizes larger than 9 and a replication factor of 6 due to an observable levelling off of performance degradation.
- With replication, the availability of the cluster for write operations is affected consistently and independently of the underlying workload composition.
- Read operations remain consistently available as the cluster size and replication factor increases, suffering only minor degradations compared to non-replicated cluster of equal size when the read portion of a workload forms the majority.
- When the percentage of write operations in a workload increases, the availability of the cluster for read operations is proportionally affected on single node clusters.
- Reads are more performant than writes which demonstrate greater degradation's in performance compared to non-replicated clusters of equal size.
- Variable distributions demonstrate minimal differences and on larger cluster sizes the differences are almost negligible.
- Disabling command logging results in approximately a 75% increase in performance.

The variety of data stores included in this study enabled basic comparisons to be drawn between each data store type and replication model.

In-memory data stores (Redis & VoltDB) have much lower latencies than data stores that rely on disk-bound operations (MongoDB & Cassandra) for both read and write-heavy workloads. For example, Redis had 172% and 187.4% lower read and write latencies compared to MongoDB for workload G and H respectively.

Redis, a key-value store, recorded the highest throughputs seemingly due to its simpler underlying implementation *i.e.* hash tables. However, due to lack of supportability in Redis; varying replication factors, consistency levels, and data distributions didn't have the same impact as on other data stores. Redis was 49.8% and 40.6% more performant than the second best data stores (MongoDB & Cassandra) on workload G and H respectively. Distributed DBMSs are least performant, likely as a result of being fully ACID compliant. VoltDB was 117% and 149% less performant than Redis for workload G and H respectively.

Master-slave type replication models, as exhibited by Redis and MongoDB tend to reduce the impact replication has on a non-replicated cluster of equal size compared to multi-master replication models exhibited by Cassandra and VoltDB. It was observed that MongoDB's replication model accounted for 8.1% and 46% less impact on performance compared to Cassandra on non-replicated clusters. This is a result of each master and slave being responsible for a single data partition leading to reduced access contention compared to the multi-master models used by Cassandra and VoltDB which contain more than one unique partition on a single server.

6.2 Limitations

Access to limited resources meant that the cluster and replication configurations which could be set up were not as exhaustive as they could have been. It would be informative to be able to evaluate the effect larger data sets and cluster sizes had on each system. With recent advances in flash memory, it would be instructive to observe the impact SSDs have on performance due to their lower access times and reduced latency compared to traditional hard disk drives. Flash memory is quickly becoming an industry standard for persistent data storage.

The instability of the private cloud environment meant that a lot of repeats of experiments had to be conducted which consumed a large amount of time. Surveying the 95th percentile figures and observing the performance metrics achieved in some of the benchmarks highlighted in the literature, suggests that there is room for improvement, which a more stable cloud environment could have greatly helped with.

Due to the poor client support for Redis Cluster, a feature that remained in beta testing throughout this project, which offers much more sophisticated cluster configurations comparable to the other data stores under test in this study, unfortunately was not implemented. Should

the client-side support for Redis Cluster improve over the coming software releases, it would be highly informative to run similar nontrivial replication experiments against it.

It was unfortunate that a warm-up stage could not be implemented for the customized VoltDB YCSB client driver within the time constraints of this study. As a result, the performance metrics for VoltDB are slightly skewed compared to the others. Analysis of the results however, suggest that adding a warm-up stage would not affect the relative performance between data stores. However, the poor results observable in this study contradict the VoltDB literature and benchmark conducted in [59]. As such, futher optimizations and experimentation would be beneficial.

This study does not account for the performance optimizations and additional features available in the most recent version of Cassandra due to an incompatible YCSB client driver. Subsequently, the results gathered in this study are only applicable to a portion of current industry implementations and offers limited guidance for future implementations.

6.3 Future Work

Initially, the current limitations would be addressed as summarized below:

- Update the Cassandra YCSB client driver to support Cassandra 2.0, subsequently releasing this code back into the community. This will more realistically aid future industry implementations of Cassandra.
- Add a warm-up stage to the VoltDB YCSB client driver and identify further optimizations to increase performance in line with existing literature and VoltDB benchmarks and increase the comparable strength with other data stores.
- Deploy and benchmark a stable release of Redis Cluster with similar replication experiments to MongoDB.
- Conduct a similar benchmark on Amazon's EC2 cloud, extending experiments to include larger data set and cluster sizes while making use of SSDs to better reflect industry standard deployments.
- Consider resource usage in analysis in order to support the conclusions derived about the underlying activity of each data store.
- Perform more in-depth analysis on the comparable nature of each data store.

Only two out of the four important measures highlighted in Cooper *et al.*'s original YCSB paper [14] for adding a new benchmarking tier for replication were explored in this study. In order to address all of these measures and work towards implementing a complete replication tier in YCSB, it would be essential to continue pursuing these measures in future iterations of this work. These measures are:

- Geo-replication: An industry recommended solution to ensure fault tolerance in the face of network partitions and entire data center failures. There are important implications regarding geo-replication which are likely to affect performance due to the increase in round trip time for communicating and transferring data between geographically dispersed cluster nodes. In addition, some data stores optimize replication for within a single data center or between nearby data centers, while others are optimized for globally distributed data centers [14].
- Data consistency: This measure would evaluate the impact replication had on the consistency of data, including metrics on what proportion of the data is stale and by how much.

To aid and encourage future performance modelling and additional studies investigating replication in NoSQL data stores, all raw YCSB log data collected in this study is publicly available[35]. The most immediate future studies could involve performance modelling Redis, MongoDB, and VoltDB.

[35] `http://www.doc.ic.ac.uk/~gh413/YCSB/`

References

[1] Accenture. http://www.accenture.com/.

[2] Aerospike. http://www.aerospike.com/.

[3] AstraZeneca. http://www.astrazeneca.co.uk/home.

[4] Falko Bause. Queueing petri nets-a formalism for the combined qualitative and quantitative analysis of systems. In *Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on*, pages 14–23. IEEE, 1993.

[5] TCP Benchmarks. http://www.tpc.org/information/benchmarks.asp. Last Accessed: 2014.07.19.

[6] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the weather tomorrow?: towards a benchmark for the cloud. In *Proceedings of the Second International Workshop on Testing Database Systems*, page 9. ACM, 2009.

[7] Dhruba Borthakur. Facebook has the world's largest hadoop cluster! http://hadoopblog.blogspot.co.uk/2010/05/facebook-has-worlds-largest-hadoop.html, May 2010. Last Accessed: 2014.07.19.

[8] Eric A Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[9] Cassandra. http://cassandra.apache.org/.

[10] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[12] Kristina Chodorow. *MongoDB: the definitive guide.* " O'Reilly Media, Inc.", 2013.

[13] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[15] Datastax Coperation. Benchmarking top nosql databases. a performance comparison for architects and it managers. 2013.

[16] Couchbase. http://www.couchbase.com/.

[17] Coursera. https://www.coursera.org/.

[18] Datastax. About datastax. http://www.datastax.com/company.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Elif Dede, Bedri Sendir, Pinar Kuzlu, Jessica Hartog, and Madhusudhan Govindaraju. An evaluation of cassandra for hadoop. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 494–501. IEEE, 2013.

[21] David J DeWitt. The wisconsin benchmark: Past, present, and future., 1993.

[22] Diomin and Grigorchuk: Altoros Systems Inc. Benchmarking couchbase server for interactive applications. http://www.altoros.com/, 2013.

[23] Eventribe. https://www.eventbrite.co.uk.

[24] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *OSDI*, pages 61–74, 2010.

[25] A. Gandini, M. Gribaudo, W. J. Knottenbelt, R. Osman, and P. Piazzolla. Performance analysis of nosql databases. In *11th European Performance Engineering Workshop (EPEW 2014)*, 2014.

[26] Álvaro García-Recuero, Sergio Esteves, and Luís Veiga. Quality-of-data for consistency levels in geo-replicated cloud data stores. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 164–170. IEEE, 2013.

[27] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[28] Github. `https://github.com/`.

[29] Apache Hadoop. `http://hadoop.apache.org`.

[30] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992. ACM, 2008.

[31] HBase. `http://hbase.apache.org/`.

[32] Eben Hewitt. *Cassandra: the definitive guide*. O'Reilly Media Inc., 2010.

[33] MongoDB Inc. Mongodb manual: Replication. `http://docs.mongodb.org/manual/replication/`.

[34] MongoDB Inc. Mongodb manual: Sharded cluster config servers. `http://docs.mongodb.org/manual/core/sharded-cluster-config-servers/`.

[35] MongoDB Inc. Mongodb manual: Sharded collection balancer. `http://docs.mongodb.org/manual/core/sharding-balancing/`.

[36] MongoDB Inc. Mongodb manual: Sharding. `http://docs.mongodb.org/manual/sharding/`.

[37] VoltDB Inc. Voltdb technical overview. `http://voltdb.com/downloads/datasheets_collateral/technical_overview.pdf`, 2010. Retrived 2014.07.16.

[38] VoltDB Inc. Using voltdb. `https://voltdb.com/downloads/documentation/UsingVoltDB.pdf`, 2014.

[39] Sakura Internet. `http://www.sakura.ne.jp`.

[40] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[41] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[42] Greg Linden. Make data useful. *Presentation, Amazon, November*, 2006.

[43] LinkedIn. `https://www.linkedin.com/`.

[44] Matt Massie, Bernard Li, Brad Nicholes, Vladimir Vuksan, Robert Alexander, Jeff Buchbinder, Frederiko Costa, Alex Dean, Dave Josephsen, Peter Phaal, et al. *Monitoring with Ganglia*. O'Reilly Media, Inc, 2012.

[45] MongoDB. `http://www.mongodb.org/`.

[46] Steffen Müller, David Bermbach, Stefan Tai, and Frank Pallas. Benchmarking the performance impact of transport layer security in cloud database systems.

[47] Nelubin and Engber: Thumbtack Technology Inc. Nosql failover characteristics: Aerospike, cassandra, couchbase, mongodb. `http://www.thumbtack.net/`, 2013.

[48] Nelubin and Engber: Thumbtack Technology Inc. Ultra-high performance nosql benchmarking. `http://www.thumbtack.net/`, 2013.

[49] R. Osman and P. Piazzolla. Modelling replication in nosql datastores. In *11th International Conference on Quantitative Evaluation of Systems (QEST)*, 2014.

[50] Tim Oreilly. Web 2.0: compact definition. *Message posted to http://radar. oreilly. com/archives/2005/10/web_20_compact_definition. html*, 2005. Retrived 2014.07.07.

[51] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 9. ACM, 2011.

[52] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[53] Pouria Pirzadeh, Junichi Tatemura, and Hakan Hacigumus. Performance evaluation of range queries in key value stores. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1092–1101. IEEE, 2011.

[54] Pivotal. `http://www.pivotal.io/`.

[55] Alexander Pokluda and Wei Sun. Benchmarking failover characteristics of large-scale data storage applications: Cassandra and voldemort.

[56] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, 2012.

[57] Rackspace. `http://www.rackspace.co.uk/`.

[58] Redis. `http://redis.io/`.

[59] Alex Rogers. Voltdb in-memory database achieves best-in-class results, running in the cloud, on the ycsb benchmark. `http://tinyurl.com/VoltDB-YCSB`, May 2014. Last Accessed: 2014.07.06.

[60] Alex Rogers. Ycsb for voltdb. `http://stackoverflow.com/a/23528926/2298888`, 2014. Last Accessed: 2014.08.25.

[61] Michael Russo. Redis, from the ground up. `http://blog.mjrusso.com/2010/10/17/redis-from-the-ground-up.html`, 2010. Last Accessed: 2014.08.25.

[62] Andrew Ryan. Under the hood: Hadoop distributed filesystem reliability with namenode and avatarnode. `http://tinyurl.com/Facebook-HDFS`, June 2012. Last Accessed: 2014.07.19.

[63] Margo Seltzer, David Krinsky, Keith Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, pages 102–107. IEEE, 1999.

[64] Amazon Web Services. `http://aws.amazon.com/`.

[65] Shopzilla. `http://www.shopzilla.co.uk/`.

[66] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730. ACM, 2012.

[67] Spotify. `https://www.spotify.com/`.

[68] StackOverflow. `http://stackoverflow.com/`.

[69] Michael Stonebraker and Rick Cattell. 10 rules for scalable performance in'simple operation'datastores. *Communications of the ACM*, 54(6):72–80, 2011.

[70] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 2011.

[71] Ganglia Monitoring System. `http://ganglia.sourceforge.net/`.

[72] Twitter. `https://twitter.com/`.

[73] Social Game Universe. `http://www.socialgameuniverse.com/`.

[74] Voldemort. `http://www.project-voldemort.com/voldemort/`.

[75] VoltDB. `http://voltdb.com/`.

A Extended Redis Client YCSB code

```
1  public class RedisClient extends DB {
2
3      static Random random = new Random();
4      private Jedis m_jedis, s_jedis;
5
6      public static final String HOST_PROPERTY = "redis.host";
7      public static final String PORT_PROPERTY = "redis.port";
8      public static final String PASSWORD_PROPERTY = "redis.password";
9      public static final String SLAVES_PROPERTY = "redis.slaves";
10     public static final String INDEX_KEY = "_indices";
11     public static final int TIMEOUT = 10000;
12
13     public void init() throws DBException {
14         Properties props = getProperties();
15         int port;
16
17         String portString = props.getProperty(PORT_PROPERTY);
18         if (portString != null) {
19             port = Integer.parseInt(portString);
20         }
21         else {
22             port = Protocol.DEFAULT_PORT;
23         }
24         String host = props.getProperty(HOST_PROPERTY);
25
26         String slaves = props.getProperty(SLAVES_PROPERTY);
27         if (slaves == null) {
28              throw new DBException("Required property \"slaves\" missing
                    for RedisClient");
29         }
30         String[] allslaves = slaves.split(",");
31         String myslave = allslaves[random.nextInt(allslaves.length)];
32
33         m_jedis = new Jedis(host, port, TIMEOUT);
34         m_jedis.connect();
35         s_jedis = new Jedis(myslave, port, TIMEOUT);
36         s_jedis.connect();
37
38         String password = props.getProperty(PASSWORD_PROPERTY);
39         if (password != null) {
40             m_jedis.auth(password);
41             s_jedis.auth(password);
42         }
43     }
44
45     public void cleanup() throws DBException {
46         m_jedis.disconnect();
47         s_jedis.disconnect();
48     }
49
50     private double hash(String key) {
51         return key.hashCode();
52     }
53
54     public int read(String table, String key, Set<String> fields,
55             HashMap<String, ByteIterator> result) {
56         if (fields == null) {
57             StringByteIterator.putAllAsByteIterators(result, s_jedis.
                    hgetAll(key));
58         }
59         else {
60             String[] fieldArray = (String[]) fields.toArray(new String[
                    fields.size()]);
61             List<String> values = s_jedis.hmget(key, fieldArray);
```

```
62
63              Iterator<String> fieldIterator = fields.iterator();
64              Iterator<String> valueIterator = values.iterator();
65
66              while (fieldIterator.hasNext() && valueIterator.hasNext()) {
67                  result.put(fieldIterator.next(),
68           new StringByteIterator(valueIterator.next()));
69              }
70              assert !fieldIterator.hasNext() && !valueIterator.hasNext();
71          }
72          return result.isEmpty() ? 1 : 0;
73      }
74
75      public int insert(String table, String key, HashMap<String,
            ByteIterator> values) {
76          if (m_jedis.hmset(key, StringByteIterator.getStringMap(values)).
                equals("OK")) {
77              m_jedis.zadd(INDEX_KEY, hash(key), key);
78              return 0;
79          }
80          return 1;
81      }
82
83      public int delete(String table, String key) {
84          return m_jedis.del(key) == 0
85              && m_jedis.zrem(INDEX_KEY, key) == 0
86                 ? 1 : 0;
87      }
88
89      public int update(String table, String key, HashMap<String,
            ByteIterator> values) {
90          return m_jedis.hmset(key, StringByteIterator.getStringMap(values))
                .equals("OK") ? 0 : 1;
91      }
92
93      public int scan(String table, String startkey, int recordcount,
94              Set<String> fields, Vector<HashMap<String, ByteIterator>>
                    result) {
95          Set<String> keys = s_jedis.zrangeByScore(INDEX_KEY, hash(startkey)
                ,
96                                  Double.POSITIVE_INFINITY, 0, recordcount);
97
98          HashMap<String, ByteIterator> values;
99          for (String key : keys) {
100             values = new HashMap<String, ByteIterator>();
101             read(table, key, fields, values);
102             result.add(values);
103         }
104         return 0;
105     }
106 }
```

B Extended MongoDB Client YCSB code

```
1  public class MongoDbClient extends DB {
2
3    Mongo mongo;
4    WriteConcern writeConcern;
5    ReadPreference readPreference;
6    String database;
7
8    public void init() throws DBException {
9      Properties props=getProperties();
10     String url=props.getProperty("mongodb.url","mongodb://localhost:27017"
         );
11     database=props.getProperty("mongodb.database","ycsb");
12     String writeConcernType=props.getProperty("mongodb.writeConcern","safe
         ").toLowerCase();
13     String readPreferenceType=props.getProperty("mongodb.readPreference","
         primary").toLowerCase();
14
15     if ("none".equals(writeConcernType)){writeConcern = WriteConcern.NONE;
16     } else if ("safe".equals(writeConcernType)){writeConcern =
         WriteConcern.SAFE;
17     } else if ("normal".equals(writeConcernType)){writeConcern =
         WriteConcern.NORMAL;
18     } else if ("fsync_safe".equals(writeConcernType)){writeConcern =
         WriteConcern.FSYNC_SAFE;
19     } else if ("replicas_safe".equals(writeConcernType)){writeConcern =
         WriteConcern.REPLICAS_SAFE;
20     } else if ("quorum".equals(writeConcernType)){writeConcern =
         WriteConcern.MAJORITY;
21     } else {System.err.println("ERROR: Invalid writeConcern: '" +
         writeConcernType); System.exit(1);}
22
23     String writeConcernWValue = props.getProperty("mongodb.writeConcern.w"
         , String.valueOf(writeConcern.getW()));
24     String writeConcernWtimeoutValue = props.getProperty("mongodb.
         writeConcern.wtimeout", String.valueOf(writeConcern.getWtimeout()))
         ;
25     String writeConcernFsyncValue = props.getProperty("mongodb.
         writeConcern.fsync", String.valueOf(writeConcern.getFsync()));
26     String writeConcernJValue = props.getProperty("mongodb.writeConcern.j"
         , String.valueOf(writeConcern.getJ()));
27     String writeConcernContinueValue = props.getProperty("mongodb.
         writeConcern.continueOnErrorForInsert", String.valueOf(writeConcern
         .getContinueOnErrorForInsert()));
28
29     writeConcern = new WriteConcern(writeConcern.getW(), writeConcern.
         getWtimeout(), writeConcern.getFsync(), writeConcern.getJ(),
         Boolean.parseBoolean(writeConcernContinueValue));
30
31     if ("primary".equals(readPreferenceType)){readPreference =
         ReadPreference.primary();
32     } else if ("primarypreferred".equals(readPreferenceType)){
         readPreference = ReadPreference.primaryPreferred();
33     } else if ("secondary".equals(readPreferenceType)){readPreference =
         ReadPreference.secondary();
34     } else if ("secondarypreferred".equals(readPreferenceType)){
         readPreference = ReadPreference.secondaryPreferred();
35     } else if ("nearest".equals(readPreferenceType)){readPreference =
         ReadPreference.nearest();
36     } else {System.err.println("ERROR: Invalid readPreference: '" +
         readPreferenceType); System.exit(1);}
37
38     try{if(url.startsWith("mongodb://")){url = url.substring(10);}
39       url += "/"+database;
40       mongo = new Mongo(new DBAddress(url));
```

```
41      } catch (Exception e1){System.err.println("Could not initialize
            MongoDB connection"); e1.printStackTrace(); return;}
42    }
43    public int insert(String table, String key, HashMap<String, ByteIterator
          > values) {
44      com.mongodb.DB db = null;
45      try{db = mongo.getDB(database);
46        db.requestStart();
47        DBCollection collection = db.getCollection(table);
48        DBObject r = new BasicDBObject().append("_id", key);
49        for(String k: values.keySet()){r.put(k, values.get(k).toArray());}
50        WriteResult res = collection.insert(r, writeConcern);
51        String error = res.getError();
52        if (error == null) { return 0;
53        } else {System.err.println(error); return 1;}
54      } catch (Exception e){ e.printStackTrace(); return 1;
55      } finally {if (db!=null){db.requestDone();}}
56    }
57    public int read(String table, String key, Set<String> fields, HashMap<
          String, ByteIterator> result) {
58      com.mongodb.DB db = null;
59      try{db = mongo.getDB(database);
60        db.requestStart();
61        DBCollection collection = db.getCollection(table);
62        DBObject q = new BasicDBObject().append("_id", key);
63        DBObject fieldsToReturn = new BasicDBObject();
64        boolean returnAllFields = fields == null;
65        DBObject queryResult = null;
66        if (!returnAllFields) {
67          Iterator<String> iter = fields.iterator();
68          while (iter.hasNext()){fieldsToReturn.put(iter.next(),1);}
69          queryResult = collection.findOne(q, fieldsToReturn, readPreference
                );
70        } else {queryResult = collection.findOne(q, null, readPreference);}
71        if (queryResult != null) { result.putAll(queryResult.toMap());}
72        return queryResult != null ? 0 : 1;
73      } catch (Exception e) { System.err.println(e.toString()); return 1;
74      } finally {if (db!=null){db.requestDone();}}
75    }
76    public int update(String table, String key, HashMap<String, ByteIterator
          > values) {
77      com.mongodb.DB db = null;
78      try{db = mongo.getDB(database);
79        db.requestStart();
80        DBCollection collection = db.getCollection(table);
81        DBObject q = new BasicDBObject().append("_id", key);
82        DBObject u = new BasicDBObject();
83        DBObject fieldsToSet = new BasicDBObject();
84        Iterator<String> keys = values.keySet().iterator();
85        while (keys.hasNext()) {
86          String tmpKey = keys.next();
87          fieldsToSet.put(tmpKey, values.get(tmpKey).toArray());}
88          u.put("$set", fieldsToSet);
89          WriteResult res = collection.update(q, u, false, false,
                writeConcern);
90          String error = res.getError();
91          if (error != null) { System.err.println(error);}
92          return res.getN() == 1 ? 0 : 1;
93      } catch (Exception e) { System.err.println(e.toString()); return 1;
94      } finally {if (db!=null){db.requestDone();}}
95    }
96  }
```

C System Monitoring: Ganglia Configuration and Setup

Ganglia is architecturally composed of three daemons: *gmond*, *gmetad*, and *gweb*. Operationally each daemon is self-contained, needing only its own configuration file to operate. However, architecturally the three daemons are cooperative. Figure C.1 illustrates how all three daemons interact in a simplified architecture [44].

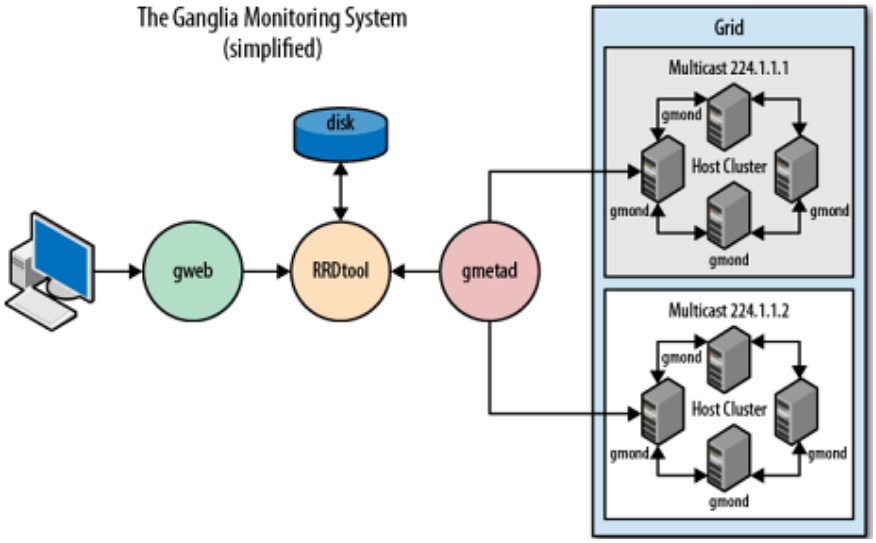

Figure C.1: Ganglia Architecture.

Ganglia monitor daemons (gmond) are deployed on each node within the cluster and are responsible for interacting with the host operating system to acquire system measurements for example: CPU load and disk capacity. On Linux systems, gmond daemons will interact with the proc filesystem. The memory footprint is modest and has negligible overhead compared to traditional monitoring agents.

All gmond daemons poll according to its own schedule indicated in its configuration file. Measurements are then subsequently shared with cluster peers via a multicast listen/announce protocol via XDR (External Data Representation). Since all metrics are multicasted to all nodes in the cluster, a request for all metric data can be directed to a single node only.

A single gmetad daemon was installed on the same host as the YCSB Client. Its primary job is to poll the cluster for measurement data, subsequently saving that data to a local RRDTool database for subsequent querying.

The gweb daemon, was also installed on the same host as the YCSB Client, offering a web based interactive data visualization user interface. gweb supports click-dragging on graphs to change time periods, and the ability to extract data in various formats. It does not need to be configured, it automatically establishes the hosts that exist in the cluster and the metrics to display. gweb is a PHP program which runs under an Apache web server which was also running locally. Figure C.2 shows an example of the web interface provided by gweb and an Apache web server.

The principle purpose behind having the gweb and gmetad daemons run on the same node was to ensure gweb had access to the RDDTool database containing all of the measurement data. Both processes where run on the YCSB Client node because of the limited number of available cluster nodes in the cloud environment in which this study took place. Since the YCSB Client node did not operate any where near capacity, it seemed a relatively safe option to have these additional processes running on that node.

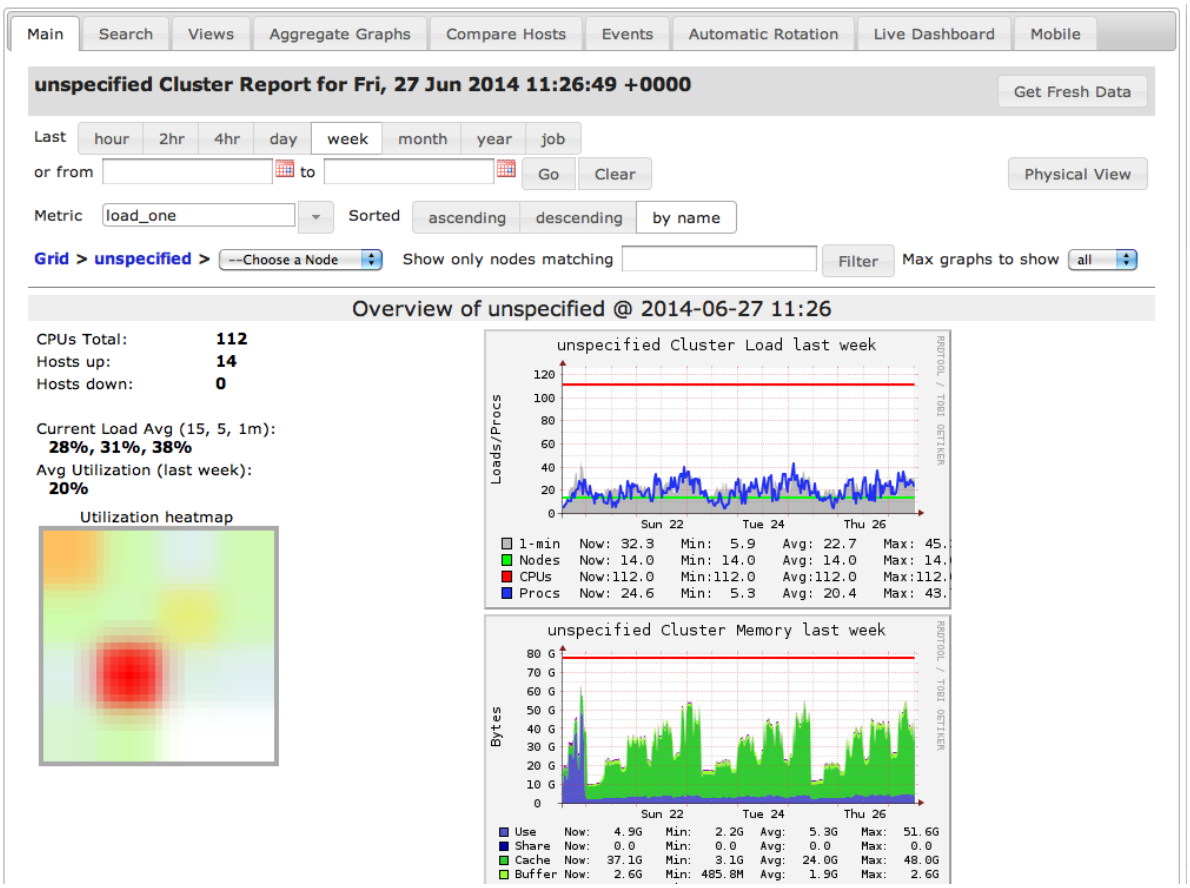

Figure C.2: Example Ganglia Web Interface.