Benchmarking Failover Characteristics of Large-Scale Data Storage Applications: Cassandra and Voldemort

Alexander Pokluda
Cheriton School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo ON N2L 3G1
Canada
apokluda@uwaterloo.ca

Wei Sun
Cheriton School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo ON N2L 3G1
Canada
w56sun@uwaterloo.ca

ABSTRACT

There has been a recent explosion in new Web-scale services that have large-scale data storage requirements. Such systems include social networking, business intelligence and Web 2.0 services. There has been a similar explosion in the number of systems designed to fulfil these requirements. These newer systems support online transaction processing, but generally do not support ACID semantics–often providing only create, read, update and delete (CRUD) operations on a set of records indexed by key–in order to provide increased availability and performance. These systems have become known as NoSQL systems, and examples include Cassandra, Voldemort, MongoDB, CouchDB, Redis, Riak, Membase, Neo4j, HBase and others. Choosing a particular system for a particular application is challenging and requires understanding the design tradeoffs made by each system as well as how they handle real-world workloads. In this report, we address both of these issues by providing a qualitative and quantitative analysis of two representative systems: Cassandra and Voldemort. We explore the design and implementation of each system and provide benchmark results using the industry standard benchmarking tool YCSB. Most prior benchmarks focus exclusively on the performance aspect of these systems. One of our main contributions is benchmarking availability in addition to performance and providing an analysis of the failover characteristics of each. Our results shown that Cassandra outperforms Voldemort, providing overall lower latency and higher throughput in the same configuration with the same resources. We conclude with a summary of our experiences and lessons learned.

1. INTRODUCTION

In the past several years, a new generation of applications like Social networking, Web 2.0 and Business Intelligence developed rapidly. These applications have to serve millions of users who expect the service to be always reliable and available, and process terabytes and even petabytes of data. This is achieved by distributed processing and is a big challenge for relational database management systems (RDBMSs). This new application environment requires:

- High concurrency for read and write operations
- An ability to efficiently store and access mass amounts of data
- High scalability and high availability

The traditional relational database design has been found to be inadequate for handling the huge amounts of unstructured data generated and analyzed by these applications. A growing number of users have turned to a new class of large scale data storage systems that relax the ACID guarantees of RDBMSs and instead provide only the basic functions of persistent storage: Create, Read, Update and Delete (CRUD). These systems have come to be known as NoSQL systems. Compared to RDBMSs, NoSQL systems are designed to handle huge amounts of data with high availability, performance and scalability. NoSQL data stores have the following key features:

- Horizontal scaling with commodity hardware
- Data distribution over many servers
- Concurrency model weaker than ACID
- Fault tolerance for node failure
- Schema free

Over 100 well-known NoSQL systems exist including Cassandra, MongoDB, CouchDB, Redis, Riak, Membase, Neo4j and HBase [13]. All of these systems offer overlapping features but differ in their design and implementation. System benchmarks are essential for evaluating different designs and implementations and ensuring that these systems meet their stated goals. System benchmarks also enable engineers and system architects to select the most appropriate storage application for a given project.

Benchmarking large scale distributed databases is challenging and an active area of research. The open-source Yahoo! Cloud Serving Benchmark (YCSB) has become the industry standard benchmarking tool for NoSQL systems. YCSB is an extensible and generic framework for the evaluation of key-value stores. It allows synthetic workloads to be generated that consist of a configurable distribution of CRUD operations on a set of records. Despite the fact that NoSQL systems have all been designed for high availability, existing benchmark comparisons focus almost exclusively on latency and throughput under blue-sky scenarios and do not evaluate metrics for availability directly. In this project, we will benchmark several NoSQL systems in the presence of simulated network and node failures.

Our main objective is to provide a measure of the level of fault tolerance provided by NoSQL systems and quantitative comparison between them. To the best of our knowledge, a thorough evaluation and comparison of the availability of each of these systems has not been done. Evaluating the availability of each system is challenging and not straightforward. To begin with, Brewer's Theorem [6], also known as the CAP Theorem, states that any distributed system must make tradeoffs between consistency, availability and performance. Only two of these metrics can be achieved at any one time and at the expense of the third. Most of the systems mentioned previously emphasize performance and either consistency or availability at the expense of the other. In order to ensure that we are making an apples-to-apples comparison, we will evaluate two representative NoSQL systems, Cassandra and Voldemort. We chose to evaluate these two systems because both are AP systems in the CAP parlance. In the first part of this report, we investigate how our chosen NoSQL systems attempt to ensure availability of data in the presence of node failures by surveying their design and implementation. In the second part of this report, we perform an experimental investigation into the failover characteristics of each in order to understand how these systems will perform in the real world.

2. DATABASE DESIGN AND IMPLEMENTATION

We now investigate how each of Cassandra and Voldemort have been designed for scalability, performance and high availability.

2.1 Cassandra

Apache Cassandra was first developed at Facebook by Prashant Malik and Avinash Lakshman, who is also one of the authors of Amazon's Dynamo [5]. Later in 2008 it was released as an open source project. Cassandra meets the strict operational requirements of performance, scalability and reliability on Facebook's platform and manages large amounts of structured data storing within tens of thousands of commodity servers across different data centres. The architecture of Cassandra is a mixture of BigTable [2] and Dynamo. It is a hybrid between a key-value and a column-oriented database. Cassandra is designed as a completely decentralized system similar to Dynamo's design so there is no single point of failure as in the case of the name node in HDFS [12]. On the other hand, the data model provides structured key-value storage where columns are added only to specified keys, so different keys can have different number of columns in any column family which is similar to Google's BigTable.

2.1.1 Partitioning

One of the key concerns of the design of Cassandra is to be able to scale incrementally. Cassandra uses consistent hashing [8] for data partitioning across the cluster with an order preserving hash function. The idea of order preserving is that if the keys are given in a certain order, the output of the hash function of the keys will be in the same order. Consistent hashing maps each node to a point on the edge of a fixed circular space or "ring". Each node is randomly assigned a value to indicate its position on the ring. The data item is assigned to the node by hashing its key to yield its position on the ring and then moving clockwise to find the first node with larger position than its position. As described above, each node is storing the data items with keys mapping to the position between it and its predecessor node on the ring. Using consistent hashing enables that only immediate neighbours are affected while the majority of the system remains unaffected when adding or removing a node in the system. Cassandra addresses the issue of non-uniform load distribution by analyzing the load information on the ring and moving lightly loaded nodes around to alleviate heavily loaded ones.

2.1.2 Replication

Cassandra provides various options for data replication to clients, such as Rack Unaware, Rack Aware and Datacenter Aware. The client configures the replication factor N for each instance, which means that each data item is replicated at N nodes across the storage system. Rack Unaware policy places the replicas on the $N-1$ subsequent nodes of the coordinator around the ring and simply ignores the network topology. Rack Aware and Datacenter Aware strategies involve using Zookeeper [7]. Rack Aware across datacenters guarantees one replica will be stored in the datacenter different from the coordinator and the rest stored together in a different datacenter but in the same racks. Datacenter Aware policy will place M of the replicas in another datacenter and the remaining on nodes in other racks within the same datacenter of the coordinator's. To guarantee system durability in the presence of datacenter failures, Cassandra replicates each row across multiple data centers.

2.1.3 Fault tolerance

Running on tens of thousands of commodity servers across datacenters around the world means failures are the norm rather than exceptions. Disk, node or datacenter failures happen due to hardware or network failure, power outages or natural disasters. To detect node failure Cassandra uses a Gossip based Accrual Failure Detector. Different from traditional failure detectors using a binary model to present the availability of a node, accrual failure detectors assign to each node a numeric value representing a suspicion level. In this way different actions can be triggered on each node depending on its suspicion level. Each node's metadata is cached in memory and stored inside Zookeeper in a fault-tolerant manner. A recovered node can get to know the ranges it was responsible for on the ring from the records in Zookeeper. When node failure occurs the dead node can be removed by the user. Its responsible range is assigned to other nodes and data is replicated there. To replace a dead node, an administrator can simply remove it from the ring and bootstrap a new node back into the cluster after the removing process is complete. A new node can stream the data from the nearest source node such that the new node is not necessary to be in the same datacenter as the primary replica for the range it is bootstrapping into, as long as another replica is in the datacenter with the new one. Before the process of bootstrapping finishes the new node will not accept any reading request from the clients.

2.2 Voldemort

Voldemort was developed by LinkedIn and first released as an Apache open source project in 2009. LinkedIn uses it as a read-write store for LinkedIn Signal, News and Push Notifications and a read-only store supporting the features of

People You May Know and Jobs For You. Resembling Dynamo, Voldemort is designed as a decentralized key-value store. The architecture of Voldemort consists of 6 layers while each layer takes responsibility for performing one function such as TCP/IP network communication, serialization, version reconciliation, inter-node routing, etc. The separation of layers enables different storage engines for various purposes, customized conflict resolutions and serialization support. The layers can be mixed and matched at runtime to meet different needs.

2.2.1 Partitioning

Similar to Cassandra, the data partitioning in Voldemort is also done via consistent hashing mechanism. The key space is divided into partitions, and these partitions are randomly mapped to nodes. Nodes are located from keys of the data item to store.

2.2.2 Replication

Instead of supporting Rack Aware or Datacenter Aware replication Voldemort intends to achieve topology awareness capability for data replication by forming zones. A zone is defined as a group of nodes close to each other in a network, which can be, for example, all nodes in a rack or all nodes in a datacenter. Users can define a per zone replication factor which determines the number of replicas in each zone.

Each node in Voldemort receives hundreds of requests such as PUT, GET, DELETE, and serves them with similar performance. Voldemort does not use heartbeats or regular pings–it simply sets an SLA for requests in order to detect node failures. A node is considered down and gets banned for a short period of time if it cannot meet the SLA. The load handled by the unavailable node is evenly dispersed across the remaining available nodes. The redundancy of storage makes the system resilient enough to tolerate as many as $N-1$ failures without data loss. However, this causes another problem, a crashed node may miss the updates during its downtime. Voldemort uses Vector Clocks data versioning to solve this problem. A vector clock is a list of server-version pairs which defines a partial order over values to resolve the update conflict and allows nodes to detect stale data. Voldemort uses hinted handoff to handle the update for data items on unavailable nodes. When a client tries to update a data item and finds the server storing the data is down, it will pick up another node and write both the updated value and the hint to the new node. When the failed node recovers it can get the update according to the hint.

3. BENCHMARK AND EXPERIMENTAL SETUP

In the first part of this report, we provided a qualitative comparison of two representative large-scale data storage applications: Cassandra and Voldemort. One of the main reasons why enterprises are rapidly adopting NoSQL systems such as these are for their availability characteristics. Enterprises want scalable data storage solutions that will continue to serve their customers' needs even when nodes are failing, part of the network goes down, or datacenters are being destroyed by tornadoes. Yet few benchmarks have been done to see how NoSQL systems live up to these promises and expectations. In this part of the report, we provide a quantitative comparison of these systems with a focus on understanding their performance and behaviour when nodes are failing. As we saw in the first part of this report, each system must make tradeoffs when striving to provide consistency, availability and performance. Different database systems are designed and optimized for different workloads and take different approaches to meet these goals. For example, Cassandra has be designed and optimized for writes using on-disk data structures that can be maintained and updated using sequential I/O. Other systems have been optimized for random reads by using traditional buffer pool architectures. Decisions about data partitioning, data placement, replication and consistency all affect performance. NoSQL systems generally sacrifice the complex query capabilities of traditional RDBMSs in order to achieve higher performance and scalability. Weaker transactional consistency guarantees mean that these types of systems can scale both vertically and horizontally. It is typical for NoSQL instances to span large geographic areas, encompassing multiple physical datacenters in order to provide the greatest level of availability possible.

Both Cassandra and Voldemort attack the challenge of ensuring high availability using similar means. The both use a form of synchronous replication by default, use hinted-handoff when a node is down, and use eventual consistency to balance replication and availability. Synchronous replication ensures that all copies of a data item are up to date, but incurs high latency on update. Also, when a failed node is restored and rejoins the cluster, the process of updating any stale data at that node introduces significant overhead into the system. Quantifying what happens when a node fails and rejoins a cluster is complex and challenging; however, it is precisely this process that we are trying to measure. In the rest of this report we will examine the *Yahoo! Cloud Serving Benchmark* (YCSB) benchmarking tool that we selected to use for benchmarking the availability of two representative NoSQL systems; we will describe the benchmarks and workloads that we used to establish baseline performance metrics for each system, including the parameters that we used for the benchmark tool and to tune each database to our specific cluster configuration; and we will examine the failover characteristics of each database and present our experiences working with YCSB, Cassandra and Voldemort.

3.1 Yahoo! Cloud Serving Benchmark Framework

The Yahoo! Cloud Serving Benchmark framework was designed to allow accurate and repeatable evaluations of emerging cloud serving systems. The YCSB authors define "cloud serving systems" as a new class of applications that address cloud-based OLTP applications. Such *serving* systems provide online read and write access to data. Typically, an impatient human is waiting for a Web page to load and reads and writes to the database are carried out as part of the page creation process. Large-scale data storage applications such as NoSQL systems fall squarely into this category.

One of the most important aspects of YCSB is that it facilitates apples-to-apples comparisons between different systems and it has quickly become an accepted industry standard for this purpose. Prior to YCSB, organizations that were looking to deploy a NoSQL system into their infrastruc-

ture would have to rely on disparate benchmarks published by the database developers using workloads that were tailored to their particular systems or painstakingly develop an in-house benchmark load generator that represents their application's requirements. Since YCSB was released as open source, however, many benchmarks of NoSQL systems have been published all using this framework, enabling accurate and reliable comparisons between them. We also use YCSB in our benchmark so that our results can be easily interpreted by the academic and industry communities.

As a benchmarking tool, YCSB enables us to gain insight into the real-world performance of the tradeoffs between different systems and the workloads for which they are suited. In order to fulfil its goal of becoming a standard benchmarking framework, YCSB comes with a core set of standard workloads that represent a variety of real-world applications. These include read and write-heavy workloads, among others. It should be noted that YCSB was designed primarily to test the raw performance of database systems. In its current form, it provides no direct support for benchmarking availability. Ideally, we would have liked to extend the framework to include support, however, due to time constraints, we had to work with only the features already implemented. We will mention some of the limitations that we encountered in the sections that follow.

3.2 Benchmarks

The results of our experimental evaluation of Cassandra and Voldemort are presented in Section 5. In that section, we first try to get an understanding of the baseline performance characteristics of each system by running the YCSB core workloads and generating latency versus throughput plots– this is the primary use case that YCSB was designed for. We then look at how failure and recovery events affect the system as a whole. During these tests, we examine how a node failure affects the latency of individual requests as well as the overall throughput of the system. Latency is an important metric that deserves our attention because, as previously mentioned, these systems are typically used in situations where there is an impatient human waiting for the result on the other end. On any given hardware setup, there is a tradeoff between latency and throughput: as throughput increases, there is more contention for resources such as the disk, network and CPU, causing the latency of individual requests increase as well. A system with better performance will achieve the desired latency and throughput with fewer servers. We generate latency versus throughput plots by loading data into the database and then executing operations against that dataset while measuring their performance at increasing rates of throughput until the database is saturated and throughput stops increasing. The YCSB client is used for both generating the workload and measuring the performance of the workload operations. The parameters for each workload are defined in properties files and the target throughput is specified as a command line parameter.

After establishing baseline performance characteristics of each system, we quantify what happens when a node fails. To do this, we bring each system to a steady state of transactional load and then after 1 minute kill one of the cluster nodes without giving it any opportunity to clean up. After 4 minutes of downtime, we bring the killed node back online and allow it to rejoin the cluster and monitor the system for another 5 minutes, during which time the killed node becomes fully operational again. Throughout the experiment, we monitor the effects on latency and throughput as well as the number of errors reported by the YCSB client, which indicates if an end-user would perceive a service outage. Ideally there should be no impact on the latency and throughput of the system and no errors should be reported.

The load generated for these tests was based on the YCSB write-heavy workload and consisted of 50% read operations and 50% write operations. We repeated each test using a small dataset containing 1 million records and a large dataset containing 50 million records at 50% and 100% of each systems maximum throughput as measured during the baseline performance measurements. At 50% of maximum throughput, there is plenty of spare capacity on the remaining nodes to take on the load of the failed node, while this is not the case at 100% of the maximum throughput. The latter case illustrates what would happen during a node failure at an enterprise that has not provisioned enough spare capacity to handle a node failure.

To simulate node failures, we killed the database process using the `kill -9` system command. This mimics a node crashing to due a software bug or hardware fault, such as a power failure. Although this is one of the most common failure scenarios, faults in real systems are often much more complex. In the real world, disks, servers, racks, networks and datacenters can all fail in unexpected ways. We had hoped to test additional failure scenarios, such as network partitioning by using `tcpkill` and bringing up firewalls between nodes, but we were unable to do so in the time available. Nevertheless, we were able to gain some valuable insight into the failover characteristics of these representative NoSQL systems which we present in the following sections.

4. EXPERIMENTAL SETUP

We performed the experimental evaluation of Cassandra and Voldemort on identical clusters using six Amazon Web Services instances. Our cluster consisted of four database servers each running on an m1.xlarge EC2 spot instance with 4 virtual CPUs with a total of 8 Elastic Compute Units (ECU)[1]and 8 GiB of RAM. Each database server also had four 420 GB instance storage disks which are virtual disks that reside on physical disks local to the server where the EC2 instance is running. These were setup in a RAID 1+0 configuration and mounted as the storage directory for each database. We used one m3.2xlarge spot instance with 8 virtual CPUs with a total of 26 ECUs and 30 GiB of RAM to host YCSB. EC2 spot instances have no persistent storage, so we setup an additional m1.small on-demand instance as an NFS server. The NFS server exported a persistent EBS volume that was mounted at `/home` on all servers.

We had to experiment with many different versions of both the databases and the YCSB benchmark tool in order to ensure everything was working as intended and establish the

[1]The amount of CPU time that is allotted to a particular virtual machine instance is expressed in terms of ECUs. ECUs are not defined in and of themselves, but rather provide a relative measure of computing power between different instance types.

parameters that we would use for our tests. (Some of the problems that we encountered can be found in Section 7). This involved setting up and tearing down the cluster many times. For this reason, we automated the cluster setup and configuration as much as possible. We Created Amazon Machine Images (AMIs) for both the YCSB and database servers so that we could deploy and configure a new cluster in minutes[2]. These images contained startup scripts that performed all necessary configuration, such as creating the RAID array and mounting the NFS storage.

4.1 Software Versions, Modifications and Tuning

The official YCSB github repository[3] is out of date and the documentation provided there is incomplete. Only Cassandra 0.5, 0.6 and 0.7 are officially supported. A patch for Cassandra 1.0.6 was submitted but not documented. In order to benchmark the Cassandra 2.0.2, the latest version available at the time of our benchmark, we used the version of YCSB from Chrisjan Master's github repository that includes a database interface layer using the Cassandra Query Language[4], the primary means for interacting with Cassandra as of version 0.8.

The official YCSB github repository also supports only Voldemort 0.81. In order to benchmark the latest version of Voldemort, we used a version of YCSB from Carlos Tasada's, one of the Voldemort developers, github repository[5]. This version of YCSB contained support for Voldemort up to version 0.96, and we found that when running it against Voldemort 1.3.0, the latest version at the time of our benchmark, YCSB would crash and report `ObsoleteVersionExcetpion`s when run with more than one client thread and a database replication factor of 2. We contacted the Voldemort developers about this issue and they said that some `ObsoleteVersionException`s are to be expected during normal operation. It means that two threads tried to update the same value at the same time and that it is the client application's responsibility to deal with the conflict. We modified the Voldemort database interface layer in this version of YCSB to retry any conflicting update operations in a tight loop until they succeed. Real world applications employ more sophisticated conflict resolution logic, such as intelligent merging of the conflicting values. Voldemort's design is similar to that of Amazon's Dynamo, which is discussed at length in [5].

We tuned each database to enable it to scale to match the resources provided by the AWS m1.xlarge instances. The primary means of tuning Cassandra for a particular hardware configuration is the `cassandra-env.sh` script in the `conf` directory of the Cassandra distribution. The script provides two variables that it recommends adjusting for a production system: `MAX_HEAP_SIZE` and `HEAP_NEW_SIZE`, which control the overall maximum Java heap size and maximum size of the young generation heap, that is, the maximum size for all objects that have not yet been through a round of the garbage collection process. The larger the young generation is, the longer application pauses due to garbage collection will be, but if it is set too small, then garbage collection will become more expensive overall. We set `MAX_HEAP_SIZE` and `HEAP_NEW_SIZE` to "12G" and "800M" respectively. The virtual machine has 15 GiB of RAM, so this configuration leaves some room for operating system buffers.

The primary means of tuning Voldemort for a particular hardware configuration is through the `VOLD_OPTS` environment variable. If not set, the startup script for Voldemort, `bin/voldemort-server.sh` simply sets a maximum heap size of 2 GiB. We set `VOLD_OPTS` to the following value based on an example provided to us by the Voldemort developers:

```
VOLD_OPTS="VOLD_OPTS="-Xms4G -Xmx10G \
    -XX:NewSize=1024M -XX:MaxNewSize=1024M \
    -XX:+UseConcMarkSweepGC -XX:+UseParNewGC \
    -XX:CMSInitiatingOccupancyFraction=70 \
    -XX:SurvivorRatio=2 \
    -XX:+AlwaysPreTouch
    -XX:+UseCompressedOops \
    -server -Dcom.sun.management.jmxremote"
```

The first two lines are the most important–they set the minimum heap size to 4 GiB, the maximum heap size to 10 GiB and the maximum size for the young generation heap to 1024 MiB. Again, this leaves some room for operating system buffers. The remainder of the options tune specific aspects of the Java garbage collection process.

At the end of every run, YCSB aggregates measurements and reports the average throughput, 95th and 99th percentile latencies and either a histogram or timeseries of latencies in a log format. We wrote python scripts to parse the log and generate plots using gnuplot. The `-s` command line flag to YCSB causes it to print status messages to standard error with status messages that include the average throughput for all operations over the last 10 seconds. We use these status messages to plot the throughput values in our failover experiments.

5. EXPERIMENTAL RESULTS

We now present the results of our experimental evaluation of Cassandra and Voldemort. As mentioned previously, YCSB comes with six predefined core workloads representing a variety of application types that are meant to facilitate apples-to-apples comparisons between different systems. We ran the core workloads against each database in order to perform a baseline performance comparison at 19 different throughput values for Cassandra and 14 different throughput values for Voldemort while measuring the latency of individual requests. We then ran our node failure test as described in Section 3.2 at both 50% of each database's maximum throughput as measured in the baseline comparison, and at 100% of each database's maximum throughput while measuring the overall throughput and latency of individual requests. We repeated each of the core workload and node failure tests on two datasets–a "small" dataset containing 1

[2]This does not include the database load time, which was on the order of one hour for our 50 million record dataset.
[3]`https://github.com/brianfrankcooper/YCSB`
[4]`https://github.com/cmatser/YCSB/commit/7c41e0e732660a40634873c9edfa20b5d14326c4`
[5]`https://github.com/ctasada/YCSB/commit/5b4a408481d116ceb86b3ef28128961792df7145`

million records and a "large" dataset containing 50 million records. The small dataset had a raw size of 190 MiB for the values only at a replication factor of two–this does not include the size of the keys nor any extra bookkeeping information that the database must maintain. The large dataset has a raw size of 9.3 GiB plus the keys and bookkeeping information. Each database has more than enough memory to keep the small dataset in memory while the large dataset is much more likely to have required hard disk access.

The YCSB core workloads were developed by examining a variety of systems and applications to determine what workloads Web applications place on cloud data systems and to explore different tradeoffs. Each workload consists of a mix of insert, read, update and scan operations and used a variety of data sizes and request distributions. They operate on records that are identified by a primary key and contain a number of fields. Each key has the form "user1234232" and field values are random sequences of ASCII characters. Insert operations insert a new record into the database; read operations read either one randomly chosen field or all fields in a record; update operations replace the value in one filed of a record; and scan operations read a random number of records in order starting at a randomly chosen key. The core workloads are:

Workload A: Update heavy workload This workload consists of 50% read operations and 50% update operations. Example: session store recording recent user actions.

Workload B: Read mostly workload This workload consists of 95% reads and 5% updates. Example: photo tagging.

Workload C: Read only workload This workload consists of 100% read operations. Example: user profile cache.

Workload D: Read latest workload This workload consists of 95% read and 5% insert operations. The most recently inserted records are the most popular. Example: user status updates.

Workload E: Short ranges workload This workload consist of 5% insert and 95% scan operations–new records are inserted and short ranges of records are queried using scan operations. Example: retrieving threaded conversations.

Workload F: Read-modify-write workload This workload consists of 50% read and 50% read-modify-write operations implemented as sequential read and update operations on the same key. Example: user database.

Each of the workloads uses a Zipfian request distribution, except for Workload D, which uses a read-latest distribution. In the Zipfian distribution some records are extremely popular while most records are unpopular. Items retain their popularity even as new records are inserted. This distribution closely models many Web workloads, where, for example, a particular user might be extremely popular and have many profile page views even though they joined the network long ago.

We ran the workloads in the order recommended by the YCSB wiki to keep the database size consistent. This consisted of loading the database, running Workloads A, B, C, F, D in that order, each for a variety of throughputs, deleting and reloading the database, and then running Workload E for a variety of throughputs. Loading the small dataset into the four node cluster of Cassandra and Voldemort each took on the order of 1 - 2 minutes, while loading the large dataset took about one hour. For Cassandra, we ran each workload for 100,000 operations and for Voldemort, for 20,000 operations due to its lower maximum throughput. Each workload took a few minutes to run at low throughput values and on the order of 10s of seconds at high throughput values.

We defined each record to have 10 fields of 10 bytes each and ran each test with 64 YCSB load generator threads. The server hosting the YCSB load generator was monitored using `htop` to ensure that the CPU and memory on the server were not a bottleneck. Each database was configured with a replication factor of 2. The keyspace for Cassandra was created with the CQL statement

```
CREATE KEYSPACE ycsb WITH REPLICATION =
{'class': 'SimpleStrategy',
'replication_factor': 2};
```

The data store for Voldemort was configured using the `stores.xml` file provided with `test_config1` in the Voldemort distribution. In particular, this file sets a replication factor of 2 and requires 2 nodes to participate in the reads and writes of each data item.

The on-disk size of the 1 million record dataset for Cassandra was between 271 MiB and 306 MiB and between 6.4 GiB and 6.9 GiB for the 50 million record dataset. For Voldemort, the on-disk size of the 1 million record dataset was 243 MiB for each node and 9.8 GiB for the 50 million record dataset. It is interesting to note that the on-disk size for the dataset in Cassandra increased by a factor of 23 for the 50 million record dataset and by a factor of 41 for Voldemort. One explanation for this could be that Cassandra has a larger constant overhead for each dataset; however, the on disk size of the 1 million and 50 million record datasets for Voldemort are smaller and larger than the on-disk sizes for Cassandra respectively, which makes this explanation unlikely.

5.1 Baseline Performance Comparison

The latency versus throughput results for Workloads A - F for Cassandra using the 1 million record dataset are shown in Figure 1. The results for the 50 million record dataset have been omitted due to space constraints but are very similar. The shape of the latency versus throughput curves for the 50 million record datasets mirror that of the 1 million record dataset, except they have a slightly lower maximum throughput, approximately 8,000 operations per second versus 9,000 operations per second, and approximately 500 microseconds higher latency per operation at the maximum throughput. From these plots, it is clear that the latency for individual operations increases as throughput increases. This phenomenon is due to resource contention within the database server as throughput increases. For Workloads A, C and D, the latency of update, insert and read operations

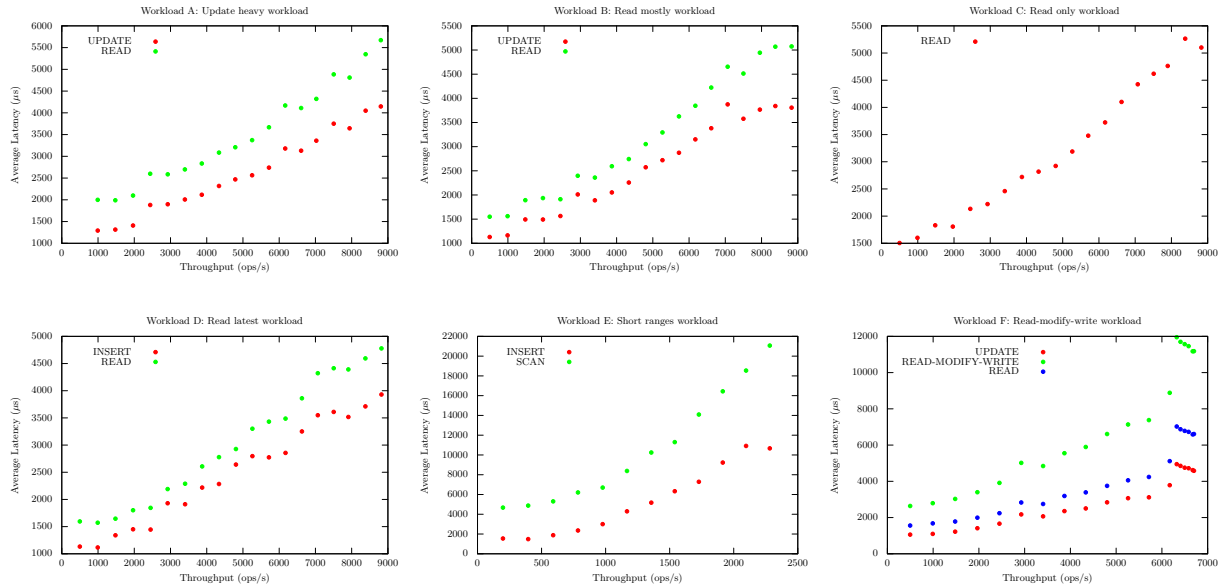

Figure 1: Workload A-E results for Cassandra 2.0.2 using 1 million records with replication factor 2 in a four node cluster.

increases almost linearly with increasing throughput with latency for all operations increasing at the same rate. The latencies for Workload B, the read mostly workload, increase steeply near the middle of the throughput range while it increases less steeply at low and high throughput values. The latency of scan operations in Workload E appear to increase almost exponentially as throughput increases. This is due to the large amounts of contention caused by the scan operations, which are more resource intensive than simple reads and writes. This contention also significantly decreases the maximum throughput possible for this workload. Latencies for Workload F also increase linearly with increasing throughput, but the maximum throughput is less that Workloads A - D.

The latency versus throughput results for Voldemort for Workloads A - D and F for the 1 million record dataset are shown in Figure 2. Workload E was not run against Voldemort because Voldemort does not support scan operations. Again, the results for the 50 million record are omitted due to space constraints. The maximum throughput for each workload in the 50 million record dataset was approximately 700 operations per second less and again the latency values vary widely. The most striking difference between these results and the results for Cassandra is the lack of a clear relationship between the latency of individual operations and throughput. Workloads A, D, and F show that the latencies of update operations vary greatly and are much higher than the latency for read operations. This phenomenon is likely due to the modifications that we made to the Voldemort database layer in YCSB in order to get the benchmark to run: it simply retries conflicting update operations in a tight loop until they succeed. Since these workloads use a Zipfian distribution, there are likely a lot of conflicting updates to popular keys which causes some update operations to livelock for short periods. More sophisticated conflict resolution strategies would likely reduce the variation in latency for update operations and produce more clear relationships between latency and throughput. Nevertheless, these results highlight a potential issue that application developers need to be aware of. It is interesting to note that the latency values for insert operations in Workload D also exhibit a large variation although our modified update operation is not used in this case. Workload C does show a clear relationship between latency and throughput, however. Latency increases slowly with increasing throughput until about 1750 operations per second, after which point it increases sharply. This likely represents a tipping point at which resource contention within the database server causes latency to go up.

We also ran each workload against Voldemort using a replication factor of 1 and found that the problem of update operation conflicts to be much less of an issue. The latency versus throughput plot for Workload A is shown in Figure 3 and shows a clear relationship between latency and throughput. The latencies of both update and read operations show the same pattern as read operations in Workload C above. Workloads B - F show similar patterns in latency values but have been omitted due to space constraints.

The maximum throughput for each workload for Cassandra and Voldemort is shown in Figure 4. This figure shows that the performance of each database was approximately the same when loading each database; however, Cassandra was able to achieve significantly greater throughput for all workloads under identical conditions as Voldemort. Cassandra also had the lowest average latency for all workloads. Interestingly, the throughput for both Cassandra and Voldemort increased when loading the 50 million record dataset and the throughput for Voldemort for Workload C increased slightly for the 50 million record dataset while the throughput dropped for Cassandra for all workloads. Nevertheless, Cassandra is the overall winner in terms of latency and throughput. Cassandra was able to achieve higher through-

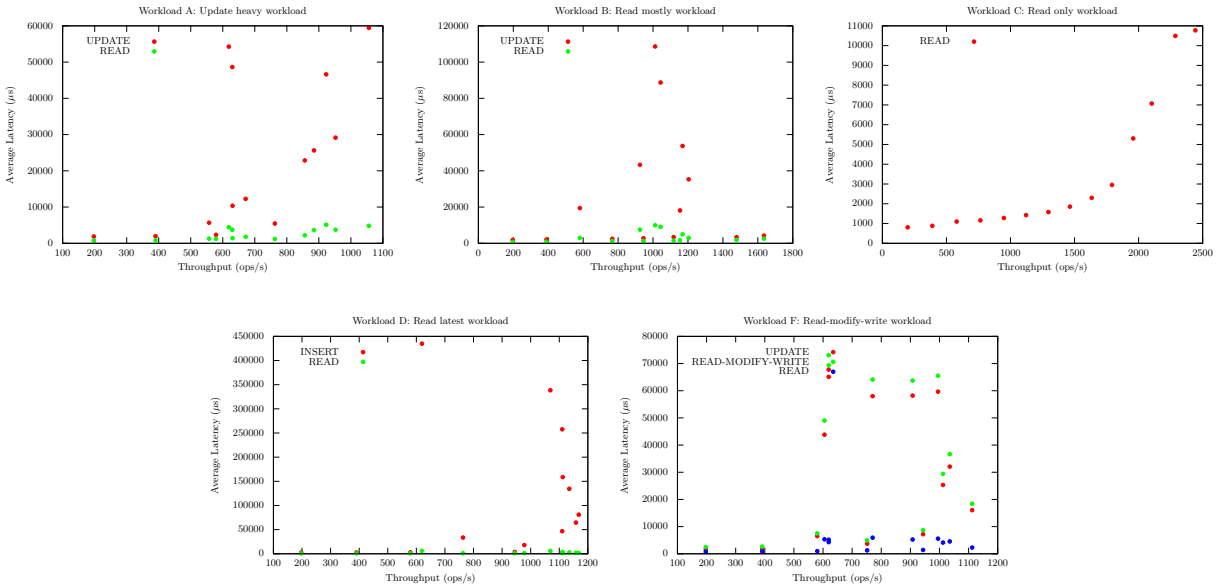

Figure 2: Workload A-D and F results for Voldemort 1.3.0 using 1 million records with replication factor 2 in a four node cluster.

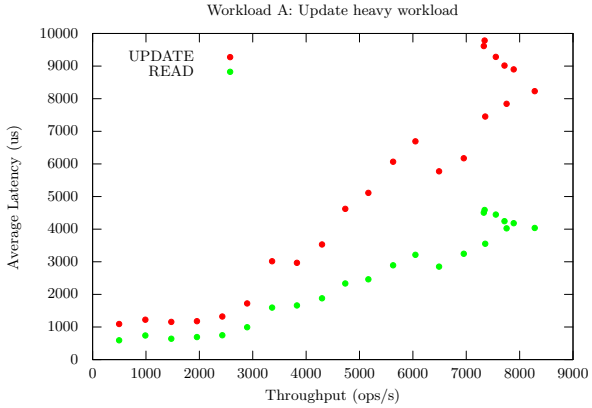

Figure 3: Latency versus throughput for Workload A run against Voldemort 1.3.0 with a dataset size of 1 million records. Unlike for the 50 million dataset, there is a clear relationship between latency and throughput.

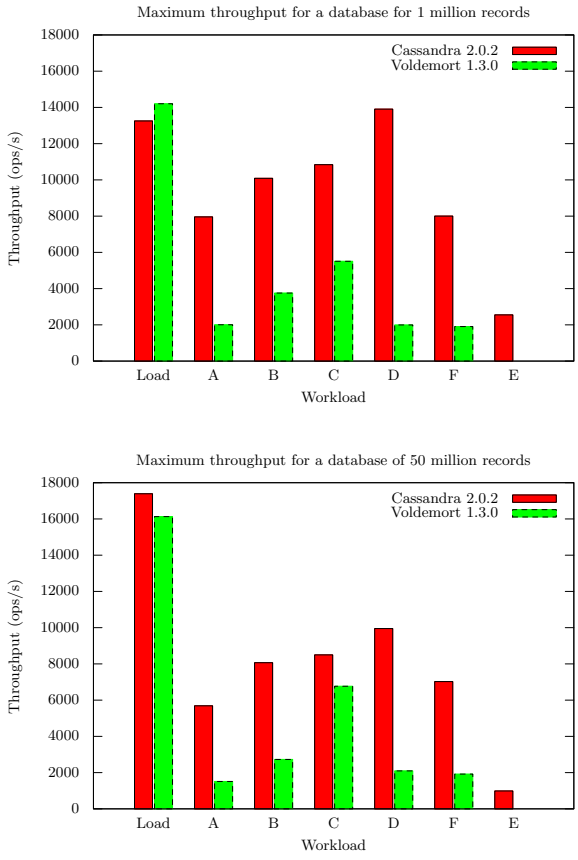

Figure 4: Maximum throughput for Cassandra and Voldemort for each workload with a database of 1 million and 50 million records.

put and lower latency with the same resources as Voldemort.

5.2 Failover Performance and Characteristics

Now that we have benchmarked the baseline performance of each database, we examine their failover performance and characteristics. As described in Section 3.2, each four node database cluster is brought to a steady state of transactional load and then one database server is killed using the `kill -9` system command while running a workload consisting of 50% read operations and 50% write operations. After four minutes of down time, we bring the node back up and allow it to rejoin the cluster while continuing to measure the latency and throughput for another five minutes. The results of these experiments for both Cassandra and Voldemort are shown in Figure 5 for the 1 million record dataset and Figure 6 for the 50 million record dataset at both 50% and 100% of each databases maximum throughput. The plots for each dataset exhibit similar characteristics; however, we will focus our discussion on the results for the 1 million record dataset. In the plots, the vertical lines at 60 seconds and 300 seconds mark the points at which the database server was killed and then brought back up respectively.

Trying to quantify what happens during failover is complex; however, we can make a number of observations from these plots. For both databases there is a large spike in latency values at the instant the node goes down. This could be due to the fact that the client must restart any transactions that were in progress at the instant that the node went down. While the node remains down, Cassandra shows increased latency for all operations. It is very evident that rejoining a node to a Cassandra cluster is an expensive operation–much more expensive than when the node goes down. This is due to the fact that the node must update any stale data that it has, which requires resources from the remaining nodes, before the downed node can rejoin the cluster. The throughput values also exhibit some interesting trends: when the downed Cassandra node starts to update its stale data at about 325 seconds in each test, there is a sharp drop in the throughput. It appears as if all nodes momentarily stop serving requests when the downed node comes back online. As can be clearly seen in the test at 50% of maximum throughput, it takes almost 200 seconds for the system to recover from one node being down for 240 seconds. Except for the drop in throughput when the downed node rejoins the cluster, the overall throughput remains unaffected when the node goes down and comes back up in the 50% of maximum throughput test. However, in the 100% of maximum throughput test, the throughput decreases both when the node goes down and comes back up, and does not fully recover in either case. According to [9], this is due to the reconnect settings in the Cassandra YCSB database layer, which has not been designed to handle node failures, and is not representative of the database system performance. The measured latencies also do not return to their pre-node-failure values and the database layer is likely the cause here well. During the 1 million record dataset test at 50% of maximum throughput, 4,311 operations out of 2.4 million, representing 0.2% of requests, failed. For the remaining tests, between 0.05% and 0.1% of requests failed.

It appears that Voldemort has very different failover characteristics from Cassandra; however, these results may be due to the nature of our test. We discovered during our analysis that unlike the Cassandra YCSB database interface layer which reported failed requests to standard output, the Voldemort database interface layer silently ignored failed requests. The configuration for the Voldemort datastore that we used (based on the sample configuration provided with Voldemort) required reads and writes to succeed at two locations for a request to be successful. When a read or write was required at the down node, the request would fail, which means approximately 23.5% of requests failed without us knowing. Since these failed requests did not take as long as long to execute as successful requests, we observe a decrease in overall latency when a node goes down. Also, since these requests fail, the data at the down node never becomes stale, and there is almost no recovery to be done when the node comes back up.

6. RELATED WORK

Benchmarking large scale distributed databases is challenging and an active area of research. YCSB was introduced in a paper from Yahoo! in which they presented benchmark results for four widely-used systems: Apache Cassandra, Apache HBase, Yahoo!'s PNUTS and a sharded MySQL implementation [3]. Many other industry and academic groups have used YCSB to compare the performance of NoSQL systems including Aerospike, Cassandra, Couchbase, HBase, MongoDB and Riak [1, 4, 9]. An influential paper presented at the 2012 VLDB conference by researchers from the University of Toronto provided a relatively recent comparison of the throughput of Cassandra, HBase, Voldemort, Redis, VoltDB, and a MySQL cluster, which were chosen as a representative set covering a broad area of modern storage architectures [11]. This study found that Voldemort had the lowest latency while Cassandra had the highest throughput. We also found Cassandra to have the highest throughput, but found Voldemort to have the highest latency. This discrepancy could be due to our modifications to the Voldemort database interface layer in YCSB, or our configuration which required a replication factor of 2. Despite the fact that these systems have all been designed for high availability, existing benchmark comparisons focus almost exclusively on latency and throughput under blue-sky scenarios and do not evaluate metrics for availability directly. However, one recent industry white paper [10] does investigate the failover characteristics of Aerospike, Cassandra, Couchbase and MongoDB using the same methodology as us. That is, they kill one node in a cluster using `kill -9` and bring it up again while continuing to monitor throughput and latency, and observed the same effects that we saw for Cassandra. Unlike our work, they do not give a qualitative comparison of the systems they evaluate and they do not consider Voldemort.

7. EXPERIENCES AND LESSONS LEARNED

We encountered a number of challenges when striving to meet our goal of providing a meaningful benchmark of failover characteristics of large scale data storage applications. First of all, the documentation for YCSB was incomplete and out of date. This led us to spend a significant amount of time trying different versions of Cassandra with different versions of the Cassandra database interface layer to see which combinations worked and which did not. Before discovering the

Figure 5: Throughput and latency for the update heavy workload for Cassandra and Voldemort with a single node failure in a four node cluster and a database size of 1 million records. The vertical lines at 30 seconds and 300 seconds mark the point at which the node was killed and brought back online respectively.

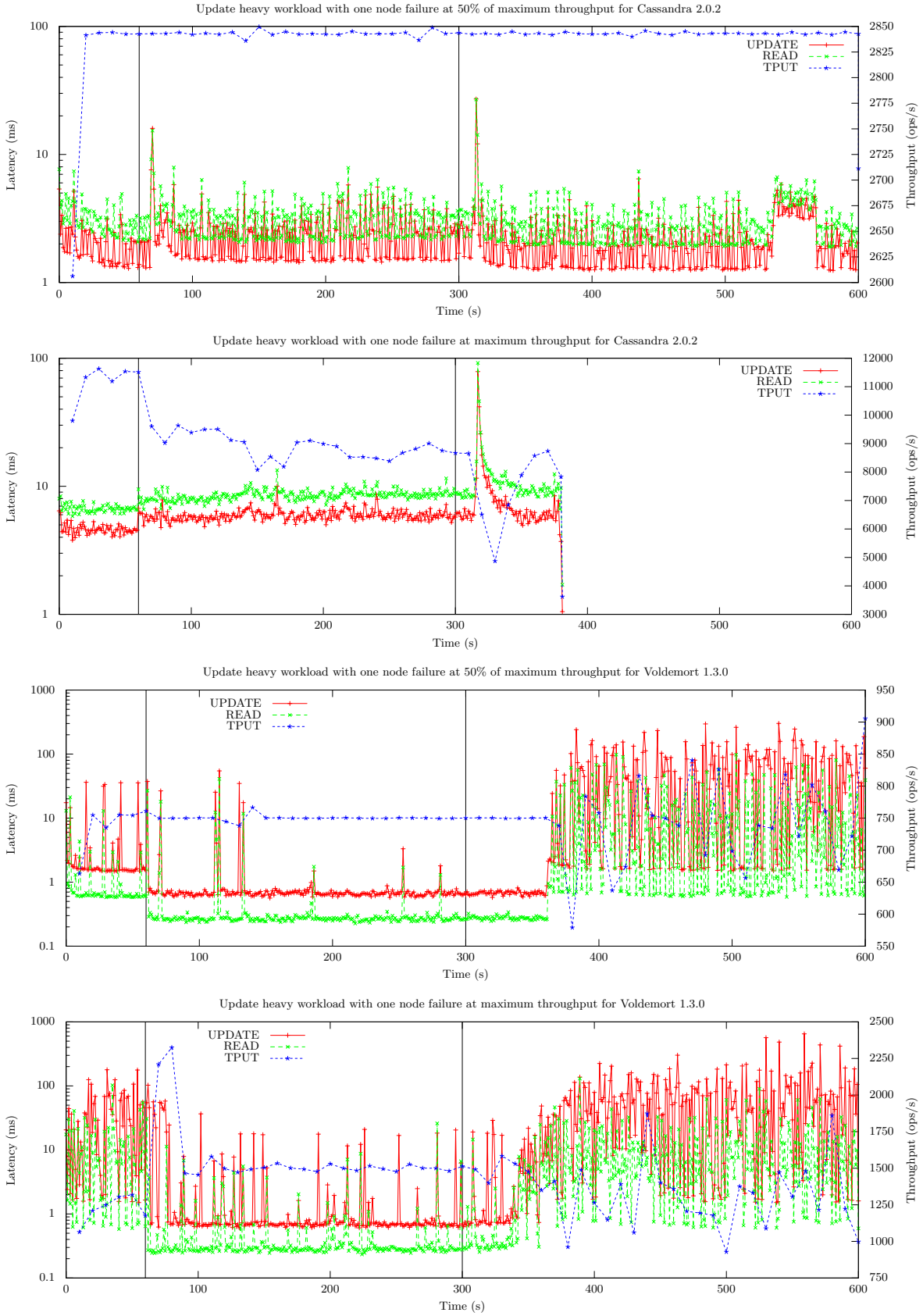

Figure 6: Throughput and latency for the update heavy workload for Cassandra and Voldemort with a single node failure in a four node cluster and a database size of 50 million records. The vertical lines at 30 seconds and 300 seconds mark the point at which the node was killed and brought back online respectively. The data in the second plot ends at approximately 375 seconds because the test finished early for the reason discussed in Section 7.

version of YCSB that was used in this report from a post in the YCSB github issue tracker, we experimented with Cassandra versions 1.2.1, 1.1.12, 1.1.7, 1.0.6 and 0.7.9. We had a similar problem with Voldemort; however, it did not take us long to find the version of YCSB updated for Voldemort 0.96 (we found it through a post in the issue tracker for Voldemort). This version of YCSB more-or-less worked with Voldemort 1.3.0 in our setup, but we still had to patch it to handle conflicting update operations.

The fact that YCSB is not designed to benchmark availability caused a number of problems. For example, YCSB does not allow a time duration for a test to be specified– the duration of a test may be specified only in terms of the number of operations to be performed. Our goal was to run each benchmark for 10 minutes; however, we found that when running the database at maximum throughput, the throughput values could be unpredictable, resulting in a test that ran significantly longer or shorter than intended. If the test ran shorter, we would have to run the test again. If the test ran longer, there was no way to cancel it early or else YCSB would not print latency statistics. Similarly, the throughput values were output only as status messages indicating the average throughput for all operations over the last 10 seconds. On the other hand, latency values would be recorded at specified intervals as small as 1 millisecond. The Cassandra and Voldemort database interface layers were also not designed to benchmark availability and this caused the problems that were discussed in Section 5.2.

We had no prior experience using AWS, therefore it took us longer than expected to get our cluster up and running. Many features of AWS are not intuitive for a first-time user, so it took us a while to understand how to work with spot instances that have no persistent storage and how to use snapshots and AMIs effectively. However, after learning how to use AWS effectively, we saw that its design to support scalability and availability has a lot in common with the design of the storage applications that we were studying.

8. CONCLUSION

We surveyed two representative NoSQL systems, Cassandra and Voldemort, providing a qualitative and quantitative analysis of each. In the qualitative analysis, we explored the design tradeoffs and implementation of each system. In the quantitative analysis, we first provided a baseline performance comparison of each system using a set of standard workloads provided with YCSB. These workloads are representative of many Web application workloads and are designed to facilitate comparison across different systems and even different benchmarks. We found Cassandra to be the clearly superior, providing the lowest latency and highest throughput across all workloads. After providing a baseline performance comparison, we explored the failover characteristics of each system by monitoring the throughput and latency of individual requests while a node failed and came back online. Although we encountered some difficulties, we were able to show that Cassandra was able to continue processing transactions through node failure and recovery, and characterize the process. We found that transaction latency increased slightly while the node was down and that recovery was an expensive process–more expensive that node failure– and latency increased significantly while the recovering node updated stale data. We believe that our work provides a solid basis for further investigation into failover characteristics of NoSQL systems.

9. REFERENCES

[1] S. Bushik. A vendor-independent comparison of nosql databases: Cassandra, hbase, mongodb, riak. *Network World*, October 2012.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[4] D. Corporation. Benchmarking top nosql databases: A performance comparison for architects and it managers. White Paper, February 2013.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007.

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

[7] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[8] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[9] D. Nelubin and B. Engber. Nosql failover characteristics: Aerospike, cassandra, couchbase, mongodb. *Thumbtack Technology*, March 2013.

[10] D. Nelubin and B. Engber. Ultra-high performance nosql benchmarking: Analyzing durability and performance tradeoffs. *Thumbtack Technology*, January 2013.

[11] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, 2012.

[12] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[13] Wikipedia. Nosql — wikipedia, the free encyclopedia, 2013. [Online; accessed 4-December-2013].