## Abstract

The remarkable growth of web applications such as Facebook and Twitter piqued my interest and led me to question that with such growth, there has to be a way to efficiently store so much data. This research paper helped me gain an insight into the highly available nature of the storage systems behind such web applications. It leads me to my research question,

**How do large scale systems efficiently store and manage application data in a highly available and consistent manner?**

After doing some brief research, I found out that such web applications have tended away from the common relational database management systems, and have created their own database management systems to handle their data. Apache Cassandra is a distributed database management system, created by Facebook developers, that manages huge amounts of data across many servers, while being highly available. On further reading, I found that web applications all over the world are building their system ground up on Cassandra, in the hope of servicing large amounts of data. This widespread use of Cassandra has been made possible by easy access to the "cloud", which lead me to setup Cassandra on Amazon's Elastic Compute Cloud.

To investigate further, I set up a small network of computers on the cloud, with Cassandra installed on each of them for testing purposes. I ran a series of experiments, each testing a different aspect of Apache Cassandra. Listed below are the outlines of the experiments:

- The highly available nature
- The efficiency of the handling of data
- The feature of there being no single point of failure

The results that I obtained from the experiments proved the power of this new database management system. It showed that Cassandra can support a very high throughput, while maintaining low latencies, which is vital to web applications these days.

# 1. Introduction

We live in a world that revolves around web applications, where to us the application is only a user interface but behind it lays an immense amount of data that is processed by thousands of computers to serve our every interaction with this interface. It is really fascinating how websites such as Facebook, Twitter and Google are rarely ever unavailable to users across the world. The question really is how such websites are able to respond to millions of users across the globe within milliseconds all the time.

Clearly, there is a highly engineered system at the core of every large-scale web application. These systems tend to run in the “cloud,” which is a network of thousands of computing nodes managed in the form of data centers around the world. These nodes communicate with each other through protocols that make it possible to store, process and interpret data that is presented to the user in the form of abstractions such as a search query on Google, a status update on Facebook, or a “tweet” on Twitter, to list some examples. This leads to my research question.

**How do large scale systems efficiently store and manage application data in a highly available and consistent manner?**

Listed below is the approach I will take to answer this question.

- Attempt to understand the key concepts involved in storing data on a large scale.
- Attempt to understand what the cloud means and how the cloud is leveraged to service these large scale systems.
- Experiment in the cloud to display the highly available nature of the data in such large scale systems.
- Experiment by mimicking failures commonly encountered and thereby, understand and display handle these challenges.

## 2. Background

Traditionally, web applications store their data in relational databases. For applications that store a large amount of data, that has to be highly available, a relational database is a solution that is far from ideal. Most of these applications only store and retrieve data by a primary key and do not require the complex querying and management functionality offered by a Relational Database Management System (RDBMS). This excess functionality requires expensive hardware and highly skilled personnel for its operation, making it a very inefficient solution. In addition, the available replication technologies are limited and typically choose consistency over availability. Although many advances have been made in the recent years, it is still not easy to scale-out databases or use smart partitioning techniques for load balancing.[1]

This is where Cassandra is introduced. “Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tunable consistent, column-oriented database that bases its distribution design on Amazon’s Dynamo and its data model on Google’s Bigtable. Created at Facebook, it is now used at some of the most popular sites on the Web.” [2]

[1]Vogels, Werner (2007) Amazon’s Dynamo. All Things Distributed, Retrieved: May 2010 from http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html
[2]Hewitt, Eben (2010) Cassandra: The Definitive Guide, O’Reilly Media, Inc. Retrieved: December 2010

## 2.1 Distributed and Decentralized

Cassandra is distributed, which means that it is capable of running on multiple machines while appearing to users combined.

Once you start to scale other data stores, such as MySQL, you have to set a master node in order to organize other nodes, which are set up as slaves. Cassandra, however, is decentralized, meaning that every node is identical; no Cassandra node performs operations different from any other node. As Cassandra is decentralized means that there is no single point of failure.

A single point of failure means that if a part of a system stops working, it will stop the entire system from working. This is what a highly available system tries to reduce.

## 2.2 Elastic Scalability

Scalability is a feature of a system that can continue to serve requests with little effect on performance. The easiest form of scaling is vertical scaling, where more hardware capacity and memory is added to your existing system. Horizontal scaling is adding more machines to a system with some or all of the data thereby reducing the burden on just one machine.

Elastic scalability is a special property of horizontal scalability where you can scale up and scale down without any problems. Scaling up means to add more machines to a system where the system can receive some or all of the data and begin processing new requests without disrupting the whole system. On the other hand, scaling down means removing a system from the cluster, thus reducing processing power, without any disruption to the system. Cassandra's feature of having no downtime is based around its elastically scalable nature.[3]

---

[3]Hewitt, Eben (2010) Cassandra: The Definitive Guide, O'Reilly Media, Inc. Retrieved: December 2010

## 2.3 Tunable Consistency

Consistency means that a read operation always returns the most recently written write value.[4]

Cassandra provides the tools necessary to configure a system to be eventually consistent, casually consistent or strictly consistent. In this essay, I will touch upon the eventual consistency and the strict consistency setup. The topic of consistency is considerably deep and beyond the scope of this essay. However, here is a brief description of eventually and strict consistency.

Strict consistency requires that any read will always return the most recently written value.[5]

Eventual consistency means on the surface that all updates will propagate throughout all of the replicas in a distributed system, but that this may take some time.[6]

## 2.4 Cloud

For my experiments, I will use Amazon's Elastic Compute Cloud (EC2) to set up a cluster of nodes on which I will install Apache Cassandra to mimic its use by a large scale web application. "Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud." [7]

The reason for using the cloud, which is an abstraction for virtualized computing nodes provided by software companies such as Amazon, Microsoft and Google, is because of its ease of use and readily available resources.

[4]Hewitt, Eben (2010) Cassandra: The Definitive Guide, O'Reilly Media, Inc. Retrieved: December 2010
[5]Ibid
[6]Ibid
[7]Amazon (Year unknown) Amazon Elastic Compute Cloud, Amazon Web Services, Retrieved: June 2010 from http://aws.amazon.com/ec2/

# 3. Investigation

## 3.1 Setting up Apache Cassandra in the Cloud

### 3.1.1 Amazon Elastic Compute Cloud (EC2)

"Amazon EC2 is a web service that enables you to launch and manage server instances in Amazon's data centers using Application Programming Interfaces (APIs) or available tools and utilities." [8]

The instances are available in different sizes and configurations depending upon the user's needs.

### 3.1.2 Setting up an instance in the cloud

Amazon provides users with a set of API tools to communicate with Amazon Web Services, to operate and manage computing instances in the cloud. Using these tools and reading the documentation provided by Amazon, I was able to create instances in the cloud, on which I would set up Apache Cassandra.

Instances can be created in one of the following regions:

| **Region** | **Endpoint** |
|---|---|
| US-East (Northern Virginia) Region | ec2.us-east-1.amazonaws.com |
| US-West (Northern California) Region | ec2.us-west-1.amazonaws.com |
| EU (Ireland) Region | ec2.eu-west-1.amazonaws.com |
| Asia Pacific (Singapore) Region | ec2.ap-southeast-1.amazonaws.com |

[8]Amazon (Year unknown) Amazon Elastic Compute Cloud, Amazon Web Services, Retrieved: June 2010 from http://aws.amazon.com/ec2/

Amazon provides these various regions to serve customers around the world and to provide an adequate response time for requests coming from different parts of the world.

To try and understand the region that would best suit my needs from the perspective of a customer wanting to set up an application in the cloud, I tried to test the response time of a request sent to an instance in each region.

Given that I am located in Dubai, United Arab Emirates and the approximate distances are shown below, I hypothesized that the Asia Pacific region would have the least response time and thereby, be the best region for me to set up my instances.

I hypothesize that the order of the response times will be in the following ascending order:

Picture C: Dubai ==> AP-Southeast Region

Picture B: Dubai ==> EU-West Region

Picture D: Dubai ==> US-East Region

Picture A: Dubai ==> US-West Region

A

B

C

D

In order to test the response times from the different locations, I created an instance in each of the above regions. I created these instances using the following command as an example:

```
mihir@mihir-desktop:~$ ec2-run-instances ami-1481707d -k jham-aws-keypair -z us-east-1a --region us-
east-1

RESERVATION   r-3575455f      121794904095  default
INSTANCE      i-5115ec3d      ami-1481707d                    pending         jham-aws-keypair
        0             m1.small        2011-01-08T12:31:19+0000        us-east-1a      aki-407d9529
              monitoring-disabled                                     instance-store
```

Command syntax:

```
ec2-run-instances <ami-code> -k <keypair name> -z <availability zone>  --region <region name>
```

Here is the breakdown of each component of the command:

- ec2-run-instances: Command which creates an instance in the cloud.

- ami-code: The ID of an Amazon Machine Image (AMI). An AMI is a snapshot of an operating system which can be used to create instances from. I chose this one in particular because it is a version of Ubuntu, a very common Linux Distribution.

- keypair: A Key-pair is a form of authentication that enables the user to connect to instances through the SSH protocol. I registered the jham-aws-keypair on the Amazon Web Services Console, so that instances could be created via that keypair.

- availability-zone: Availability zones are separate data centers in the same geographic region. This helps customers avoid a single point of failure, in case there is a failure in one of the data centers.

- region: Regions are separate geographic areas(E.g. US and Ireland) as mentioned above.

To measure the response time to each instance from my location, I used to the common ping command line tool available on every Linux distribution.

Here are the results I collected when from a 100 ping requests to each instance:

| Request path | Response |
|---|---|
| Dubai ==> US-East Region | -ec2-67-202-58-205.compute-1.amazonaws.com ping statistics -<br>100 packets transmitted, 100 received, 0% packet loss, time 99061ms<br>rtt min/avg/max/mdev = 256.856/264.245/276.565/4.179 ms |
| Dubai ==> EU-West Region | --- ec2-46-137-0-143.eu-west-1.compute.amazonaws.com ping statistics ---<br>100 packets transmitted, 100 received, 0% packet loss, time 99021ms<br>rtt min/avg/max/mdev = 215.532/218.677/222.826/1.647 ms |
| Dubai ==> US-West Region | --- ec2-204-236-140-5.us-west-1.compute.amazonaws.com ping statistics ---<br>100 packets transmitted, 100 received, 0% packet loss, time 99016ms<br>rtt min/avg/max/mdev = 307.315/311.346/315.323/3.211 ms |
| Dubai ==> AP-Southeast Region | --- ec2-175-41-156-204.ap-southeast-1.compute.amazonaws.com ping statistics ---<br>100 packets transmitted, 100 received, 0% packet loss, time 99135ms<br>rtt min/avg/max/mdev = 94.529/103.178/111.703/6.070 ms |

*The highlighted sections show the average response time in milliseconds.

As hypothesized, the response times were in the following ascending order:

Dubai ==> AP-Southeast Region

Dubai ==> EU-West Region

Dubai ==> US-East Region

Dubai ==> US-West Region

### 3.1.3 Installing Apache Cassandra on an instance

After creating 3 instances in the AP-Southeast region, I had to install and set up Apache Cassandra on each of these instances.

After reading a blog post on how to set up Apache Cassandra in the cloud, a storage configuration file had to be copied over to the instance before installing Apache Cassandra.

Using the scp command line tool of the Ubuntu Linux Distribution, I successfully copied over the file.

```
sudo scp -i /home/mihir/Desktop/ec2-api-tools-1.3-62308/jham-aws-keypair-euwest.pem
 /home/mihir/Desktop/Ec2_cassandra_setup/storage-conf.xml ec2-46-137-0-143.eu-west-
1.compute.amazonaws.com:~/
```

Command syntax:

```
sudo scp -i <keypair-location> <storage-config-file location> <destination-folder-location>
```

Response:

```
storage-conf.xml                                        100%   17KB  17.1KB/s
 00:00
```

After the file had been copied over, I logged into the instance to begin installing Apache Cassandra on the instance.

Using the ssh command line tools of the Ubuntu Linux Distribution, I logged onto the instance.

```
sudo ssh -i /home/mihir/Desktop/ec2-api-tools-1.3-62308/jham-aws-keypair-euwest.pem ec2-46-137-0-
143.eu-west-1.compute.amazonaws.com
```

Command syntax:

```
sudo ssh -i <keypair-location> <public-dns-instance-address>
```

Firstly, I had to update the instance. This was done using the following command line tools.

```
sudo -E apt-get update  -y
sudo -E apt-get upgrade -y
```

Apache Cassandra requires a Java Runtime Environment (JRE) to be present for it to work. The following command line tool was used to download and install the Java 6 runtime environment.

```
sudo apt-get install openjdk-6-jre
```

After installing Java, I had to install Apache's Ant.

Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. (Apache Ant)

The following command line tool was used to download and install ant on the instance.

```
sudo -E apt-get install -y ant
```

I had to then configure the default Cassandra directories to point to and use the brief storage located at /mnt.

```
sudo -E mkdir /var/lib/cassandra
sudo -E chown ubuntu /var/lib/cassandra
sudo -E mkdir /var/log/cassandra
sudo -E chown ubuntu /var/log/cassandra
mkdir /var/lib/cassandra/commitlog /var/lib/cassandra/data
sudo -E mkdir /mnt/cassandra
sudo -E chown ubuntu /mnt/cassandra
mkdir /mnt/cassandra/commitlog /mnt/cassandra/data /mnt/cassandra/logs
echo "/mnt/cassandra/logs /var/log/cassandra     none bind" | sudo -E tee -a /etc/fstab
sudo -E mount /var/log/cassandra
echo "/mnt/cassandra/commitlog /var/lib/cassandra/commitlog     none bind" | sudo -E tee -a
/etc/fstab
sudo -E mount /var/lib/cassandra/commitlog
echo "/mnt/cassandra/data /var/lib/cassandra/data     none bind" | sudo -E tee -a /etc/fstab
sudo -E mount /var/lib/cassandra/data
```

I had to download the actual Apache Cassandra file and unzip it on the instance's home directory.

```
wget -q http://www.apache.org/dist/cassandra/0.6.2/apache-cassandra-0.6.2-bin.tar.gz
tar -zxf apache-cassandra-0.6.2-bin.tar.gz
echo "export CASSANDRA_HOME=~/apache-cassandra-0.6.2" | sudo -E tee -a ~/.bashrc
rm apache-cassandra-0.6.2-bin.tar.gz
```

After installing, I had to copy the storage configuration file to the Apache Cassandra directory. But before that, I had to specify the seed instance.

The Seed instance is the super node in the cluster. A super node is needed so that whenever a node joins the cluster, it knows which node to contact and understand which other nodes are present in the cluster and who to communicate and replicate its data with.

To do this, we open up the storage configuration file via Vim. Vim is a command-line text editor that is pre-installed on the Ubuntu Linux Distribution.

```
vi ~/storage-config.xml
```

Once I was within vim, I searched for the location in the file where the “Seed” configuration is specified, by typing the following command inside vim:

/Seed

```
<Seeds>
<Seed>domU-12-31-39-0B-F2-11</Seed>
</Seeds>
```

I replaced this sample IP address (domU-12-31-39-0B-F2-11) with the IP address of the instance, because that is what the EC2 nodes use to communicate with each other. After editing and saving the storage configuration file, I copied the file over to the Apache Cassandra directory.

Cassandra was set up and ready.

## 3.2 Experiments

### 3.2.1 Experiment 1: Apache Cassandra's Replication in its simplest form

Aim: To show the how data is accessible from every node in the cluster with a replication factor of 1.

A replication factor specifies how many copies of each piece of data will be stored and distributed throughout the Cassandra cluster. With a replication factor of 1, the data will exist only on a single node in the cluster. Losing that node would mean that the data becomes unavailable.

Set up:

- Three terminals, each with a different instance running Apache Cassandra.

Node A - Running the Super node instance.

Node B

Node C

Step 1: Inserted values into the Cassandra database, via Node A(Super Node)

```
cassandra> set Keyspace1.Standard2['jsmith']['first'] = 'John'
Value inserted.
cassandra> set Keyspace1.Standard2['jsmith']['last'] = 'Smith'
Value inserted.
cassandra> set Keyspace1.Standard2['jsmith']['age'] = '42'
Value inserted.
```

Step 2: Retrieved values from the database, via Node A and B.

```
get Keyspace1.Standard2['jsmith']
=> (column=last, value=Smith, timestamp=1294511248354000)
=> (column=first, value=John, timestamp=1294511240083000)
=> (column=age, value=42, timestamp=1294511255618000)
Returned 3 results.
```

Step 3: Inserted another set of values, via Node C.

```
cassandra> set Keyspace1.Standard2['mjham']['first'] = 'Mihir'
Value inserted.
```

Step 4: Retrieved values via Node A(Super Node)

```
cassandra> get Keyspace1.Standard2['mjham']
=> (column=first, value=Mihir, timestamp=1294511372516000)
Returned 1 results.
```

## Implications

This experiment proved that every node in a Cassandra cluster can accept and consistently respond to read and write requests, regardless of replication factor. This shows the highly scalable nature of Cassandra, where every node in the cluster plays a vital role ensuring high throughput for an application.

With a minimum replication factor of 1, data written to one node was read from another node, even though the data was not present on the node it was being read from. This is because the Cassandra nodes communicate with each other and forward requests to the appropriate node holding the data.

### 3.2.2 Experiment 2: Unavailability in the light of failure with a replication factor of 1

Aim: To show how a replication factor of 1 is not sufficient to ensure high availability of a Cassandra cluster.

Set up:

-Three terminals, each with a different instance running Apache Cassandra.

Node A - Running the Super node instance.

Node B

Node C

I used the iptables command to simulate node failures by blocking incoming and outgoing requests to and fro port(virtual component, part of a network address) 7000 on a particular node. It is to be noted that port 7000 is what Cassandra uses to communicate in between nodes. I happened to come across this command on the Internet when searching for ways to block requests to a particular port.

Command syntax:

```
iptables -A <Input/Output> -p <protocol> --destination-port <port-to-block> -j <Action>
```

Step 1: Write data to Node B

```
cassandra> set Keyspace1.Standard2['jdoe']['first'] = 'John'
Value inserted.
```

Step 2: Disconnect Node A from the cluster

```
sudo iptables -A INPUT -p tcp --destination-port 7000 -j DROP
sudo iptables -A OUTPUT -p tcp --destination-port 7000 -j DROP

./nodetool ring -h 10.128.49.240

Address         Status    Load        Range                                      Ring
                                      99310391769213970408383852238529550311
10.130.71.178 Down      1.75 MB     31451908490063309301104707230901568533      |<--|
10.128.49.240 Up        1.03 MB     69199037029076226094078387851362683938      |   |
10.128.50.108 Down      442.13 KB   99310391769213970408383852238529550311      |-->|
```

Step 3: Read data from all 3 nodes

Node A:

```
cassandra> get Keyspace1.Standard2['jdoe']
=> (column=first, value=John, timestamp=1294514909490000)
Returned 1 results.
```

Node B:

```
cassandra> get Keyspace1.Standard2['jdoe']
Exception null
```

Node C:

```
cassandra> get Keyspace1.Standard2['jdoe']
Exception null
```

Step 4: Write similar data to Node B

```
cassandra> set Keyspace1.Standard2['jdoe']['last'] = 'Doe'
Exception null
```

Step 5: Write similar data to Node C

```
cassandra> set Keyspace1.Standard2['jdoe']['last'] = 'Doe'
Exception null
```

## Implications

This experiment shows how if the node that is holding the value being requested fails, the entire system becomes unavailable for that particular value. This is seen in the fact that even though Node A was the only one that was simulated to have a failure, Node B and Node C could not read or write any value to the [‘jdoe’] column in the database.

This experiment shows that with a replication factor of 1, Cassandra is very fragile and becomes unavailable in the event of failure.

### 3.2.3 Experiment 3: Cassandra as a highly available system with a replication factor greater than 1

Aim: To show that even in the event of a single failure, Cassandra continues to service requests for all possible values when the replication factor is greater than 1.

Set up:

-Three terminals, each with a different instance running Apache Cassandra.

Node A - Running the Super node instance.

Node B

Node C

Step 1: Write data to Node A

```
cassandra> set Keyspace1.Standard2 ['jdoe']['first'] = 'John'
Value inserted.
```

Step 2: Disconnect Node B from the cluster

```
iptables -A INPUT -p tcp --destination-port 7000 -j DROP
iptables -A OUTPUT -p tcp --destination-port 7000 -j DROP

./nodetool ring -h 10.128.47.191

Address      Status    Load        Range                              Ring
                       148041545435619621215649485105470834280
10.128.129.120Down     2.97 MB     970702895556872459753251773607659563      |<--|
10.128.129.205 Up      2.97 MB     116645277562376155986754090513264663897   |   |
10.128.47.191 Up       2.97 MB     148041545435619621215649485105470834280   |-->|
```

Step 3: Write data to Node C

```
cassandra> set Keyspace1.Standard2 ['jdoe']['last'] = 'Doe'
Value inserted.
```

Step 4: Read data from Node A, Node B and Node C

Node A:

```
cassandra> get Keyspace1.Standard2['jdoe']
=> (column=last, value=Doe, timestamp=1294516942560000)
=> (column=first, value=John, timestamp=1294516798519000)
Returned 2 results.
```

Node B:

```
cassandra> get Keyspace1.Standard2['jdoe']
=> (column=first, value=John, timestamp=1294516798519000)
Returned 1 results.
```

Node C:

```
cassandra> get Keyspace1.Standard2['jdoe']
=> (column=last, value=Doe, timestamp=1294516942560000)
=> (column=first, value=John, timestamp=1294516798519000)
Returned 2 results.
```

Step 5: Disconnect Node C (Disconnecting all 3 nodes)

```
iptables -A INPUT -p tcp --destination-port 7000 -j DROP
iptables -A OUTPUT -p tcp --destination-port 7000 -j DROP

./nodetool ring -h 10.128.47.191

Address      Status    Load        Range                              Ring
                       148041545435619621215649485105470834280
10.128.129.120Down      2.97 MB      970702895556872459753251773607659563      |<--|
10.128.129.205Down      2.97 MB      116645277562376155986754090513264663897    |   |
10.128.47.191 Up      2.97 MB      148041545435619621215649485105470834280    |-->|
```

Step 6: Read from all 3 nodes

Node A:

```
cassandra> get Keyspace1.Standard2['jdoe']
=> (column=last, value=Doe, timestamp=1294516942560000)
=> (column=first, value=John, timestamp=1294516798519000)
Returned 2 results.
```

Node B:

```
cassandra> get Keyspace1.Standard2['jdoe']
=> (column=first, value=John, timestamp=1294516798519000)
Returned 1 results.
```

Node C:

```
cassandra> get Keyspace1.Standard2['jdoe']
=> (column=last, value=Doe, timestamp=1294516942560000)
=> (column=first, value=John, timestamp=1294516798519000)
Returned 2 results.
```

Step 7: Write to Node A

```
cassandra> set Keyspace1.Standard2 ['jdoe']['age'] = '27'
Value inserted.
```

## Implications

This experiment shows that even in the event of a single failure, with a replication factor of 3, Node A and Node C were able to service requests for the [‘jdoe’] column. This is evident from the fact that I was able to write values to Node C and read from both, Node A and Node C, even when Node B was down.

Furthermore, when Node C failed, Node A still continued to service requests for all possible values. This was evident from the fact that Node A was able to write values to the [‘jdoe’] column, even when Node B and Node C were down.

This shows that Cassandra as a system is able to handle two independent failures with a replication factor of 3, and thus shows the highly available nature of Cassandra.

### 3.2.4 Experiment 4: Impact of increasing replication factor on response time of requests

Aim: To write 10,000 values to a cluster of replication factor of 1 and to a cluster of replication factor of 3, and show its impact on response time of requests.

Set up:

-Three terminals, each with a different instance running Apache Cassandra.

Node A- Running the Super node instance.

Node B

Node C

I used the following program I wrote using the Python programming language, to insert 10,000 values into a Cassandra cluster.

All the program does is loop through values ranging from 1 to 10,000, and insert test values into a [‘mihir’] column.

```
import pycassa
from pycassa import *
client = pycassa.connect(['10.130.71.178:9160'])
cf = pycassa.ColumnFamily(client, 'Keyspace1', 'Standard2')
for i in range(1,10000):
    cf.insert('mihir', {str(i) : str(i)})
```

Step 1: Run the program 3 times against the cluster with a replication factor of 1 and get an approximate understanding of how long it takes to service these requests

```
time python cassandraInserts.py; time python cassandraInserts.py; time python cassandraInserts.py;

real	0m26.685s
user	0m0.960s
sys	0m0.240s

real	0m27.228s
user	0m0.884s
sys	0m0.312s

real	0m26.861s
user	0m0.876s
sys	0m0.288s
```

Step 2: Run the program 3 times against the cluster with a replication factor of 3 and get an approximate understand of how long it takes to service these requests

```
time python cassandraInserts.py; time python cassandraInserts.py; time python cassandraInserts.py;

real	0m23.979s
user	0m0.832s
sys	0m0.400s

real	0m23.546s
user	0m0.900s
sys	0m0.316s

real	0m23.174s
user	0m0.892s
sys	0m0.368s
```

Step 3: Run the program 3 times against the cluster with a replication factor of 1 and get an approximate understand of how long it takes to service these requests. This time, change the column being written to from [‘mihir’] to [‘test’].

Note that the decision to change the column from [‘mihir’] to [‘test’] was based on trial and error to bring out the results below.

```
time python cassandraInserts.py; time python cassandraInserts.py; time python cassandraInserts.py;

real	0m6.010s
user	0m1.840s
sys	0m0.460s

real	0m6.027s
user	0m2.020s
sys	0m0.452s

real	0m5.986s
user	0m1.984s
sys	0m0.456s
```

Step 4: Run the program 3 times against the cluster with a replication factor of 3 and get an approximate understand of how long it takes to service these requests. This time, change the column being written to from [‘mihir’] to [‘test’].

Note that the decision to change the column from [‘mihir’] to [‘test’] was based on trial and error to bring out the results below.

```
time python cassandraInserts.py; time python cassandraInserts.py; time python cassandraInserts.py;

real	0m22.445s
user	0m0.932s
sys	0m0.340s

real	0m22.969s
user	0m1.016s
sys	0m0.260s

real	0m22.858s
user	0m0.888s
sys	0m0.320s
```

## Implications

This experiment shows that the variability in response time on the cluster with a replication factor of 1 is due to the nodes coordinating amongst each other on who is responsible for the incoming data. When the incoming data is present on the node that is connected to by the Python client, the response time is quicker, as seen in the case of inserting data in the [‘test’] column. On the other hand, when the incoming data is not present on the node connected to the python client, the node forwards the request to the node responsible for the data, which adds to the response time, as in the case of inserting data into the [‘mihir’] column.

In the case of the cluster with a replication factor of 3, there is minimal variability in the response time, since every node is responsible for the incoming data, as the number of nodes is equal to the replication factor.

This experiment also shows the performance benefit of having a lower replication factor when nodes do not need to wait for data to be replicated to other nodes. However, as seen in experiments above, this low replication factor comes at the cost of low availabilty.

### 3.2.5 Experiment 5: Impact of consistency level on response time

Aim: To write data with 3 different consistency levels of ONE, QUORUM and ALL, to both clusters of replication factors of 1 and 3, and observe its impact on response time to service requests.

Set up:

-Three terminals, each with a different instance running Apache Cassandra.

Node A - Running the Super node instance.

Node B

Node C

I will be using the same program as in the above experiment, with the following modifications

```
import pycassa
from pycassa import *
client = pycassa.connect(['10.130.71.178:9160'])
cf = pycassa.ColumnFamily(client, 'Keyspace1', 'Standard2')
for i in range(1,10000):
    cf.insert('mihir', {str(i) : str(i)}, ConsistencyLevel.QUORUM)
```

I will be changing the ConsistencyLevel from QUORUM to ONE and ALL to see its impact.

ONE:  Ensure that the write has been written to at least 1 node's commit log and memory table

QUORUM: Ensure that the write has been written to <ReplicationFactor> / 2 + 1 nodes

ALL: Ensure that the write is written to <ReplicationFactor> nodes before responding to the client.

(Pycassa Thrift Object Types)

Step 1: Run program with consistency level ONE against the cluster with a replication factor of 1.

```
time python cassandraInserts.py; time python cassandraInserts.py; time python cassandraInserts.py;

real	0m5.531s
user	0m1.960s
sys	0m0.532s

real	0m6.126s
user	0m1.820s
sys	0m0.416s

real	0m5.979s
user	0m1.836s
sys	0m0.396s
```

Step 2: Run program with consistency level ONE against the cluster with a replication factor of 3.

```
time python cassandraInserts.py; time python cassandraInserts.py; time python cassandraInserts.py;

real	0m22.898s
user	0m0.892s
sys	0m0.316s

real	0m23.349s
user	0m0.976s
sys	0m0.324s

real	0m23.220s
user	0m0.888s
sys	0m0.304s
```

Step 3: Run program with consistency level QUORUM against the cluster with a replication factor of 1.

```
time python cassandraInserts.py; time python cassandraInserts.py; time python cassandraInserts.py;

real	0m5.531s
user	0m1.960s
sys	0m0.532s

real	0m6.126s
user	0m1.820s
sys	0m0.416s

real	0m5.979s
user	0m1.836s
sys	0m0.396s
```

Step 4: Run program with consistency level QUORUM against the cluster with a replication factor of 3.

```
time python cassandraInserts.py; time python cassandraInserts.py; time python cassandraInserts.py;

real	0m28.431s
user	0m0.904s
sys	0m0.292s

real	0m28.726s
user	0m1.016s
sys	0m0.208s

real	0m28.778s
user	0m0.984s
sys	0m0.260s
```

Step 5: Run program with consistency level ALL against the cluster with a replication factor of 1.

```
time python cassandraInserts.py; time python cassandraInserts.py; time python cassandraInserts.py;

real    0m5.531s
user    0m1.960s
sys     0m0.532s

real    0m6.126s
user    0m1.820s
sys     0m0.416s

real    0m5.979s
user    0m1.836s
sys     0m0.396s
```

Step 6: Run program with consistency level ALL against the cluster with a replication factor of 3.

```
time python cassandraInserts.py; time python cassandraInserts.py; time python cassandraInserts.py;

real    0m34.210s
user    0m0.920s
sys     0m0.304s

real    0m35.037s
user    0m1.032s
sys     0m0.244s

real    0m34.076s
user    0m1.048s
sys     0m0.428s
```

## Implications

This experiment shows that in a case with a replication factor of 1 a change in the consistency level has no effect on the response time, since there is only one node in consideration, even though there are 3 nodes present in the cluster. This is because Cassandra bases its consistency levels off the replication factor and not the number of nodes present in the cluster.

In the case of the cluster with a replication factor of 3, a increase in the consistency level causes a corresponding increase in the repsonse time of the service requests. This is because Cassandra has to wait until the request is written to as many nodes specified by the consistency level, before it responds back to the client.

Therefore, choosing a higher consistency level ensures that the data is available on all nodes as soon as it is written. However, increased consistency comes at the cost of slower response time and thereby, decreased performance.

# 4. Conclusion

By conducting the following experiments, I was able to show the following features of Apache Cassandra:

- **Tunably Consistent**: I saw that by changing consistency levels, I was able to configure a system that is eventually consistent (in the case of replication factor of 3 and ConsistencyLevel of ONE) as well as a system that is strictly consistent (in the case of replication factor of 3 and ConsistencyLevel of ALL). I was also able to show the advantages of a eventually consistent system against a strictly consistent system, in terms of response time of service requests.
- **Elastically Scalable**: I saw that every node in the cluster was able to accept writes, regardless of whether or not it was responsible for the data being requested. This gives Cassandra the ability to have each node play the role of a Master, and thereby guarantee throughput.
- **Highly Available**: I saw that by simulating node failures, Cassandra was still able to service requests, in the case of a replication factor greater than 1.

# 5. Bibliography

## 5.1 Books

Hewitt, Eben (2010) Cassandra: The Definitive Guide, O'Reilly Media, Inc.

## 5.2 Websites

Vogels, Werner (2007) Amazon's Dynamo. All Things Distributed, Retrieved: May 2010 from http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html

Apache (2009) Cassandra Wiki from http://wiki.apache.org/cassandra/ArticlesAndPresentations

Hulen, Corey (2010) Multi-machine EC2 Cassandra Setup in 30 minutes from http://www.coreyhulen.org/?p=277

Amazon (Year unknown) Amazon Elastic Compute Cloud, Amazon Web Services, Retrieved: June 2010 from http://aws.amazon.com/ec2/

Natarajan, Ramesh (2009) Ping Tutorials: 15 Effective Ping Command Examples, The Geek Stuff, from http://www.thegeekstuff.com/2009/11/ping-tutorial-13-effective-ping-command-examples/

## 5.3 Pictures

Unknown, Retrieved: December 2010 from http://www.freemaptools.com/how-far-is-it-between.htm