Evaluation of Cloud Storage on Edge

Dhruv Rauthan 2019A7PS0095G f20190095@goa.bits-pilani.ac.in
Priyam Loganathan 2019A7PS0108G f20190108@goa.bits-pilani.ac.in

1. Problem Statement

Edge Computing is here with its distributed architecture, low latency connectivity and bandwidth relief. Unlike cloud computing which is familiar and known, the edge computing services and support for application development is still in early stages. In the cloud we have numerous database services to efficiently store data whereas no such service exists at the edge yet.

In this project, we analyze what unique challenges and issues are posed by edge networking to cloud-based storage services. Many edge projects start with a decentralized architecture like the one used in Cassandra. Hence, in this study we start with understanding and quantifying the impact that edge networking has on the workings of a cloud storage system like Cassandra.

This study revolves around the following research topics :

1. Study and calculation of Cassandra read/write operation delay from total delay.
2. Changing which latencies (endpoint-edge or edge-edge) has a greater impact on the overall RTT?
3. Impact of changing packet loss on Cassandra read/write performance.

2. Introduction

In recent years, we have seen a rise in shifting of data from data centers to the network edge, inculcating a new paradigm called Edge Computing. It is a distributed information technology architecture in which client data is processed at the periphery of the network, as close to the originating source as possible.

- Edge computing moves some portion of storage and compute resources out of the central data center and closer to the source of the data itself.
- Rather than transmitting raw data to a central data center for processing and analysis, that work is instead performed where the data is actually generated -- whether that's a retail store, a factory floor, a sprawling utility or across a smart city.
- Only the result of that computing work at the edge, such as real-time business insights, equipment maintenance predictions or other actionable answers, is sent back to the main data center for review.

As shown in the picture below (Figure 1), the Edge Gateway Servers are distributed across a vast geographic location which enables latency critical services like railway systems, wind turbines, smart cars, IoT devices to work more efficiently compared to a cloud computing model where the latencies would be high.

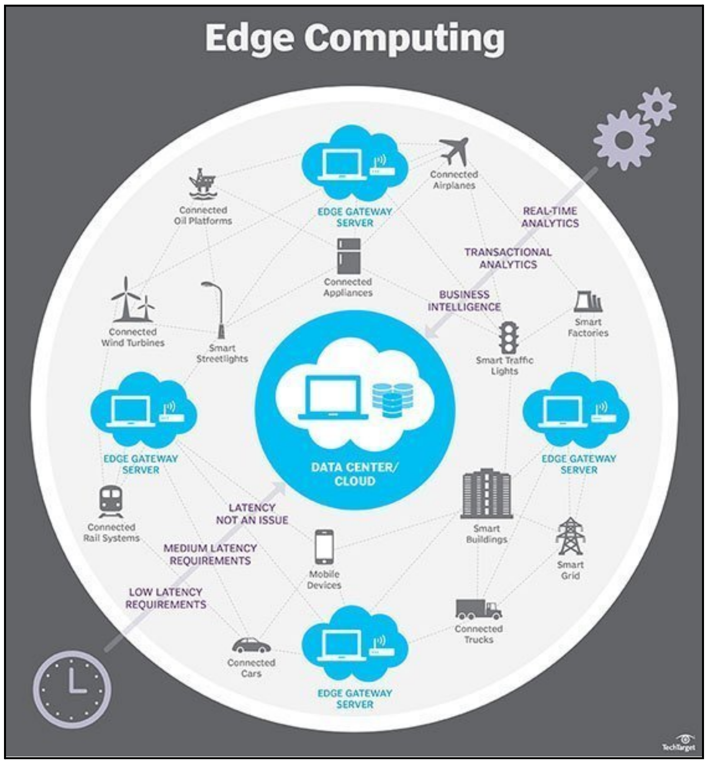


Figure 1

Many applications such as real-time video processing, augmented/virtual reality gaming, environment sensing, benefit from such decentralized, close-to-user deployments where low-latency, real-time results are expected.

2.1 Edge Computing vs Cloud Computing

The research community has started questioning the general applicability of cloud computing with respect to emerging enabling technologies and novel applications, such as augmented reality, industrial Internet of Things, etc. The primary motivating assumption within the edge computing community is rather long end-to-end cloud access latency due to limited and sparse deployment of data centers across the globe. Since 2009, there has been a drastic increase in network coverage by major Cloud Providers. [1]

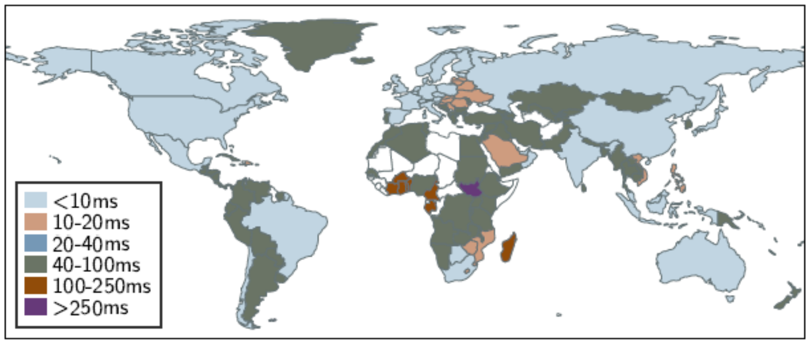


Figure 2

According to a paper which evaluated the current state of cloud connectivity globally, it was found that the majority of the world population can access a cloud facility within 100ms [4] (Figure 2). Low latency devices can function efficiently only in continents like the US and Europe where there is an abundance of cloud provider data centers. Extensive datacenter deployment is necessary to make cloud access latencies consistently compatible with requirements of next-generation applications, especially for Asia, South America, and Africa. An alternative to this expensive solution is edge computing where full fledged data centers are not mandatory to ensure low latency.

2.2 Challenges in Edge Computing

An edge platform consists of edge sites which can be gateway node(s) and/or local server machines that connect to the Internet. An edge site can contain multiple edge nodes. End-devices, such as mobile devices, wearable sensors, IoT nodes, and smart cars, will connect to the edge nodes for storage/compute services. As we move closer to the end-devices, resources are restricted to few CPU cores, little storage and limited DRAM. In this environment, a data center state or storage service faces multiple challenges because of the following properties [5] :

- Distributed : Edges sites are distributed geographically and do not have a centralized node to run the service. Previous studies have shown that such a system makes metadata management and data consistency a challenging task.
- Heterogenous : Edge nodes can be heterogeneous in terms of their storage, networking and computing power. A large database of metadata to track edge node heterogeneity is required which needs to be handled efficiently and in a scalable manner.
- Dynamic : The Edge domain has to be constantly monitored and updated as any slip up in the configuration might lead to devastating cyber attacks. Managing numerous heterogeneous nodes can be a daunting task.

3. Cassandra

Apache Cassandra is a NoSQL, distributed database management system. It is designed to handle large amounts of data across multiple servers, providing high availability without a single point of failure. It achieves this using a ring-type architecture, where the smallest unit is a node. [2]

3.1 Cassandra Components

- Node
  - Basic infrastructure component
  - Connects with other nodes using an internal network (ring)
  - Uses the Gossip Protocol to exchange data
- Rack
  - Logical grouping of nodes within the ring
  - Database uses racks to ensure that replicas are distributed amongst different logical groupings
- Database
  - Logical set of racks
- Cluster
  - Components which contains 1 or more datacenters
  - Outermost storage container

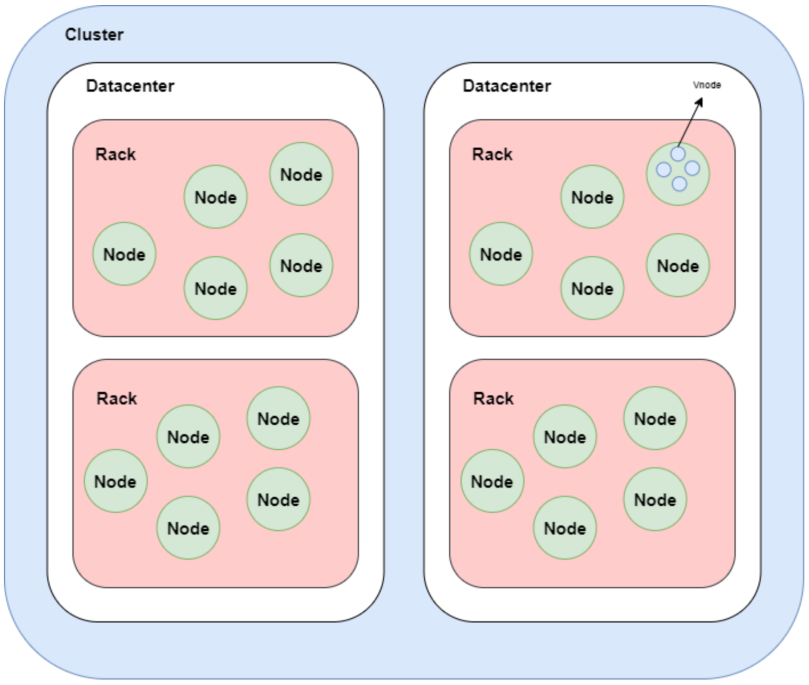


Figure 3

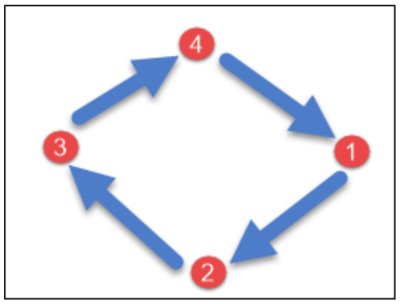
3.2 Replication Factor

This is the total number of replicas across the cluster. For example, if this value is set to 1, then only 1 copy of each row exists in the cluster. It can also be set at the rack or the datacenter level.

3.3 Replication Strategy

This controls how the replicas are chosen. There are various replication strategies available:

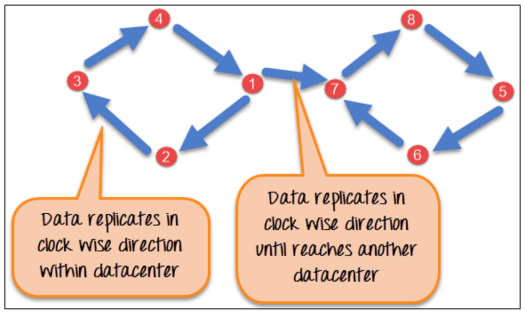
- SimpleStrategy (Figure 4)
  - Used when there is only 1 datacenter
  - Replica is placed on the first node selected by the partitioner
  - The remaining replicas are placed in a clockwise direction on the node ring



SimpleStrategy in Cassandra

Figure 4

- NetworkTopologyStrategy (Figure 5)
  - Used when there is more than 2 datacenters
  - Replicas are set for each datacenter separately
  - Places replicas in the clockwise direction in the ring until reaches the first node in another rack


4
3
1
2
8
7
5
6
Data replicates in clock wise direction within datacenter
Data replicates in clock wise direction until reaches another datacenter

NetworkTopologyStrategy in Cassandra

Figure 5

4. Continuum

This is the deployment and benchmarking framework for an edge network [3]. It automates the setting up and configuration of the cloud, edge, and endpoint hardware and networks. Further, it manages the installation of software inside the emulated environment and can perform benchmarks as well. The execution consists of 3 phases namely, infrastructure, installation and benchmarking.

4.1 Software Used

4.1.1 KVM

Kernel-based Virtual Machine (KVM) is an open source virtualization technology built into Linux. It lets us turn Linux into a hypervisor that allows a host machine to run multiple, isolated virtual environments called virtual machines (VMs). We will need KVMs to emulate the cloud and edge worker as well as the endpoints

4.1.2 Libvirt

This is an open source management tool for managing platform virtualization. The virsh console uses the interface provided by libvirt to interact with the virtual machines. This will be used to manage the KVMs deployed.

4.1.3 QEMU

This is a free and open source emulator. It emulates the machine's processor and provides a set of different hardware and device models for the machine, enabling it to run a variety of guest OSes. This will interoperate with the KVMs to run the virtual machines at native speeds

4.1.4 Ansible

An open source software which provides a framework that is used to automate IT operations. This will be used to install the required software (such as Docker, Kubernetes, KubeEdge etc) on the virtual machines.

4.2 Architecture

1. Infrastructure configuration: Libvirt configuration files are created.
2. Infrastructure execution: The configuration files are executed, creating QEMU/KVM virtual machines connected through network bridges.
3. Software configuration: Ansible is configured for software installation.
4. Software execution: Ansible playbooks are executed, installing operating services and resource management software on each machine.
5. Benchmark configuration The benchmark is configured and prepared.
6. Benchmark execution: Applications in docker containers are executed on resource management software running on the emulated infrastructure. Meanwhile, application- and system-level metrics are captured, processed and presented to the user.

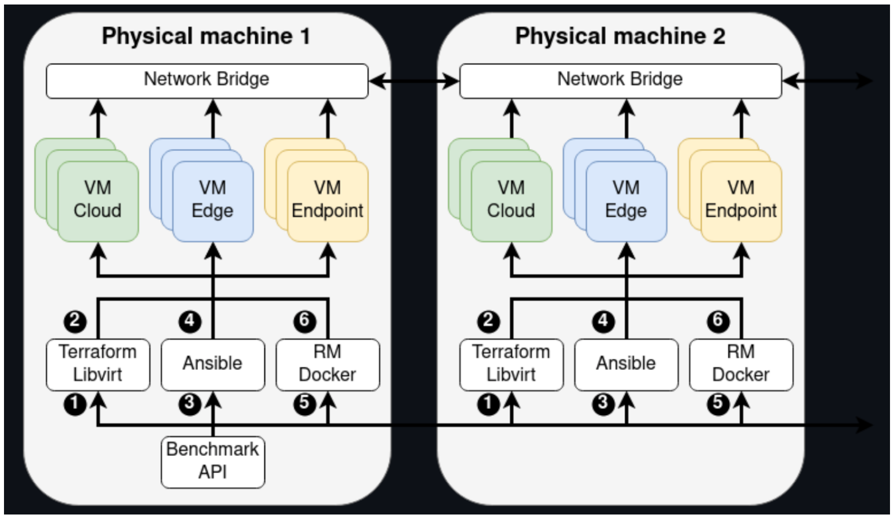

Physical machine 1
Network Bridge
VM Cloud
VM Edge
VM Endpoint
❷
❹
❻
Terraform Libvirt
Ansible
RM Docker
❶
❸
❺
Benchmark API
Physical machine 2
Network Bridge
VM Cloud
VM Edge
VM Endpoint
❷
❹
❻
Terraform Libvirt
Ansible
RM Docker
❶
❸
❺

Figure 6

5. Simulation Setup

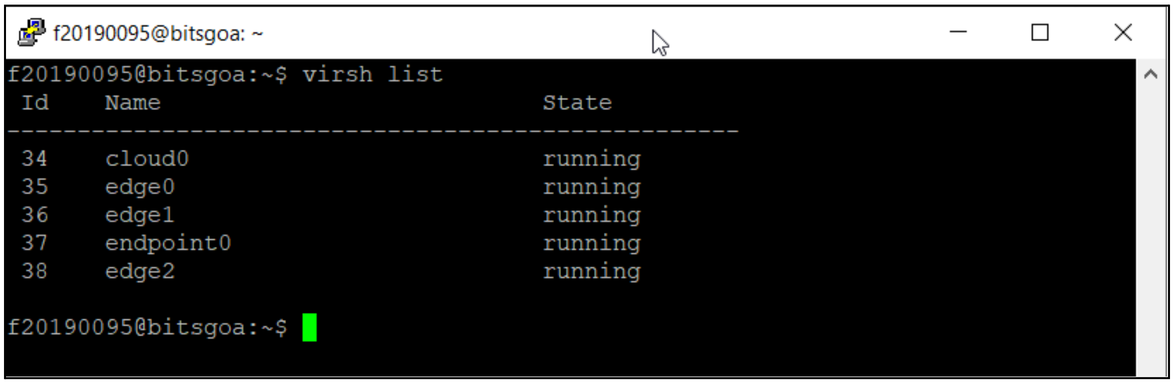
Our end goal is to use the Edge Continuum Framework to simulate a real time edge computing network and use it to benchmark link latencies between Cassandra nodes in the cluster.
We used the framework to automate the task of building VMs and setting them up on the same network.

The command given below creates 1 cloud controller node (cloud0), 3 edge nodes (edge0, edge1, edge2) and 1 endpoint (endpoint0) :

```
python3 main_without_benchmark.py --edgenodes 3 --endpoints 1 -m edge
image-classification
```

Libvirt is used to interact with the VMs. The command given below lists all the VMs running on the machine (Screenshot 1) :

```
virsh list
```



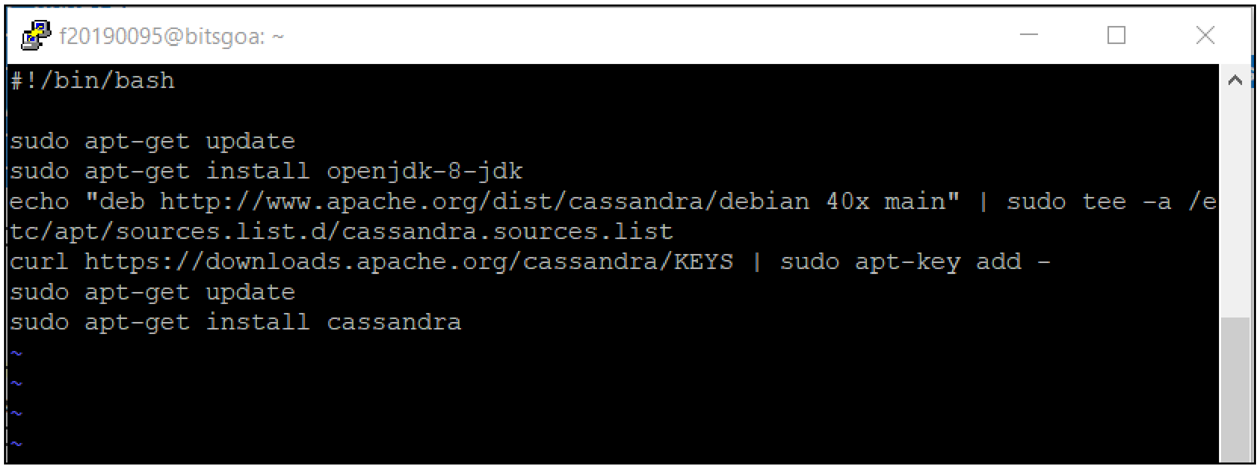
```
f20190095@bitsgoa:~$ virsh list
 Id    Name                           State
----------------------------------------------------
 34    cloud0                         running
 35    edge0                          running
 36    edge1                          running
 37    endpoint0                      running
 38    edge2                          running

f20190095@bitsgoa:~$
```

Screenshot 1

We used SSH (Secure Shell) to login into each VM :

```
ssh edge0@192.168.122.11 -i /home/f20190095/.ssh/id_rsa_benchmark
```

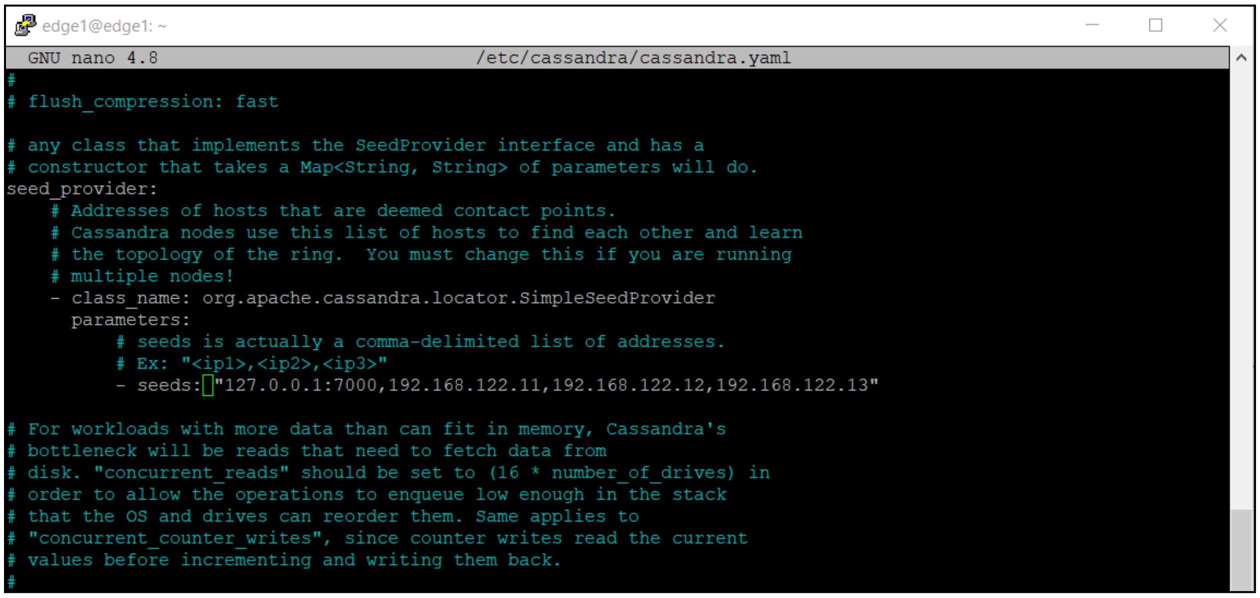
The following screenshot (Screenshot 2) shows the script we made to install Cassandra on each edge node. In the script, we also included installing an old version of Java JDK instead of the latest one since the latest version caused some problems in Cassandra.



```
f20190095@bitsgoa: ~
#!/bin/bash

sudo apt-get update
sudo apt-get install openjdk-8-jdk
echo "deb http://www.apache.org/dist/cassandra/debian 40x main" | sudo tee -a /e
tc/apt/sources.list.d/cassandra.sources.list
curl https://downloads.apache.org/cassandra/KEYS | sudo apt-key add -
sudo apt-get update
sudo apt-get install cassandra
~
~
~
~
```

Screenshot 2

After successfully installing Cassandra, we edited the cassandra.yaml file (Screenshot 3) and specified a list of IP addresses of other nodes in the seeds field. This is to ensure that each node can communicate with every other node. A seed node is used to bootstrap the gossip process for new nodes joining a cluster.



```
edge1@edge1: ~
  GNU nano 4.8                     /etc/cassandra/cassandra.yaml
#
# flush_compression: fast

# any class that implements the SeedProvider interface and has a
# constructor that takes a Map<String, String> of parameters will do.
seed_provider:
    # Addresses of hosts that are deemed contact points.
    # Cassandra nodes use this list of hosts to find each other and learn
    # the topology of the ring.  You must change this if you are running
    # multiple nodes!
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          # seeds is actually a comma-delimited list of addresses.
          # Ex: "<ip1>,<ip2>,<ip3>"
          - seeds: "127.0.0.1:7000,192.168.122.11,192.168.122.12,192.168.122.13"

# For workloads with more data than can fit in memory, Cassandra's
# bottleneck will be reads that need to fetch data from
# disk. "concurrent_reads" should be set to (16 * number_of_drives) in
# order to allow the operations to enqueue low enough in the stack
# that the OS and drives can reorder them. Same applies to
# "concurrent_counter_writes", since counter writes read the current
# values before incrementing and writing them back.
#
```

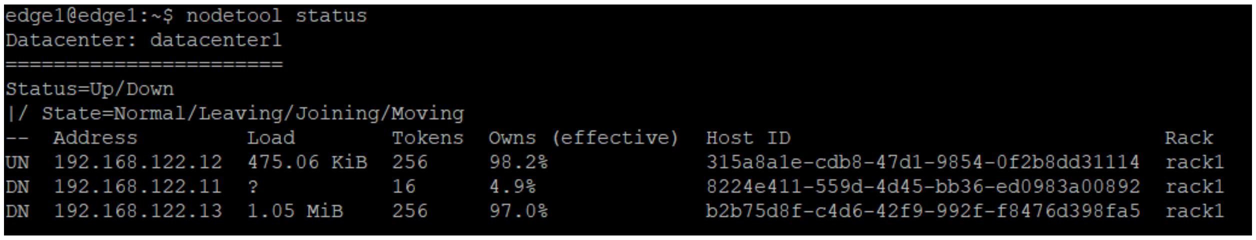
Screenshot 3

After configuring each Cassandra node in their respective VM, we start the service using (Screenshot 5) :

```
service cassandra start
```

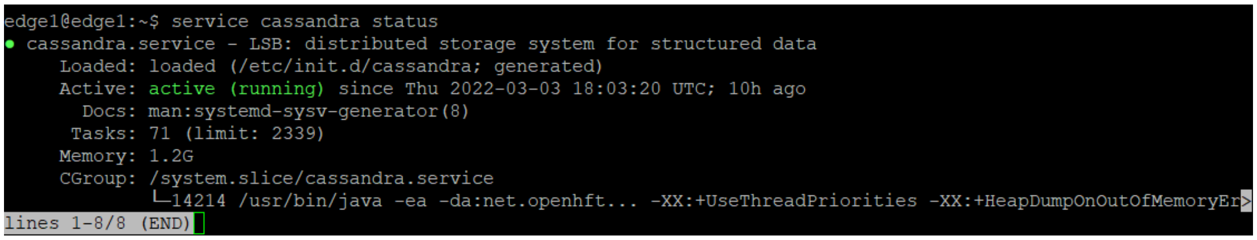
We check the status of the Cassandra cluster in the screenshots given below (Screenshot 5) using :

```
nodetool status
```



```
edge1@edge1:~$ nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address          Load        Tokens  Owns (effective)  Host ID                               Rack
UN  192.168.122.12  475.06 KiB  256     98.2%             315a8a1e-cdb8-47d1-9854-0f2b8dd31114  rack1
DN  192.168.122.11  ?           16      4.9%              8224e411-559d-4d45-bb36-ed0983a00892  rack1
DN  192.168.122.13  1.05 MiB    256     97.0%             b2b75d8f-c4d6-42f9-992f-f8476d398fa5  rack1
```

Screenshot 4



```
edge1@edge1:~$ service cassandra status
● cassandra.service - LSB: distributed storage system for structured data
     Loaded: loaded (/etc/init.d/cassandra; generated)
     Active: active (running) since Thu 2022-03-03 18:03:20 UTC; 10h ago
       Docs: man:systemd-sysv-generator(8)
      Tasks: 71 (limit: 2339)
     Memory: 1.2G
     CGroup: /system.slice/cassandra.service
             └─14214 /usr/bin/java -ea -da:net.openhft... -XX:+UseThreadPriorities -XX:+HeapDumpOnOutOfMemoryEr>
lines 1-8/8 (END)
```

Screenshot 5

5.1 Adding delays between VMs

To simulate a real world edge computing setup, we had to add delays between the VMs using the **tcconfig (**Python) library. **tcconfig** is a tc command wrapper. Makes it easy to set up traffic control of network bandwidth/latency/packet-loss/packet-corruption/etc. to a network-interface.

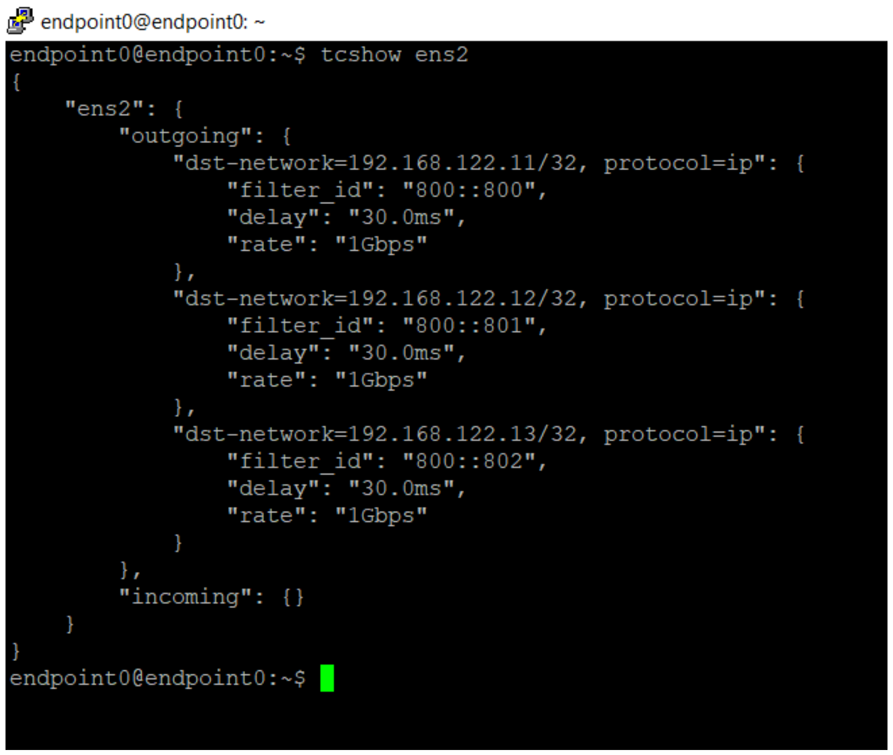
We used the following commands to add delay between the endpoint and the edge nodes :

```
tcset --add ens2 --delay 30ms --network 192.168.122.11
tcset --add ens2 --delay 30ms --network 192.168.122.12
tcset --add ens2 --delay 30ms --network 192.168.122.13
```

These commands, when executed on the endpoint add an outgoing delay of 30ms to every packet in the *ens2* network interface whose destination is *192.168.122.11, 192.168.122.12 or 192.168.122.13.*

The modifications to the network interface can be seen by the following command (Screenshot 6) :

```
tcshow ens2
```


endpoint0@endpoint0: ~

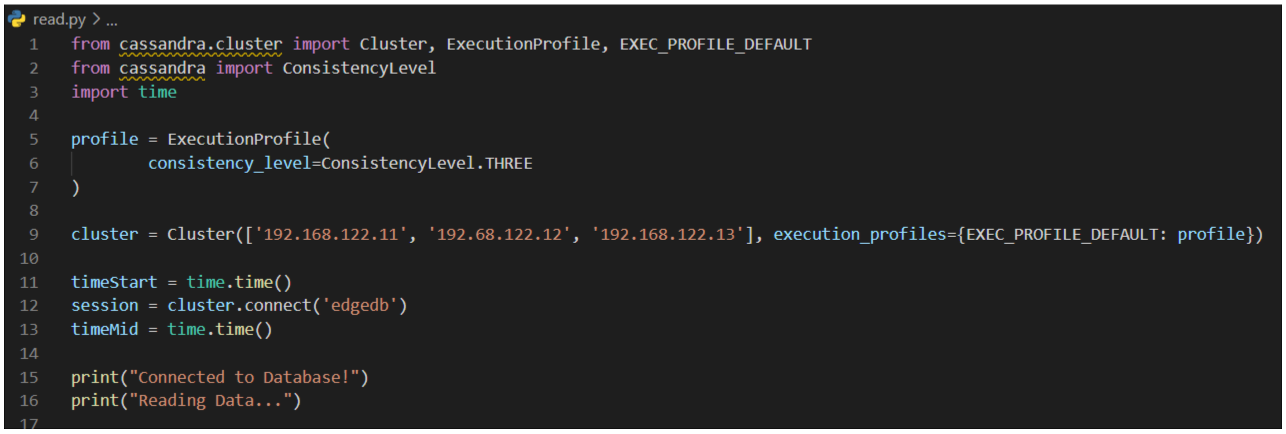
```
endpoint0@endpoint0:~$ tcshow ens2
{
    "ens2": {
        "outgoing": {
            "dst-network=192.168.122.11/32, protocol=ip": {
                "filter_id": "800::800",
                "delay": "30.0ms",
                "rate": "1Gbps"
            },
            "dst-network=192.168.122.12/32, protocol=ip": {
                "filter_id": "800::801",
                "delay": "30.0ms",
                "rate": "1Gbps"
            },
            "dst-network=192.168.122.13/32, protocol=ip": {
                "filter_id": "800::802",
                "delay": "30.0ms",
                "rate": "1Gbps"
            }
        },
        "incoming": {}
    }
}
endpoint0@endpoint0:~$
```

Screenshot 6

Similarly, we will be altering packet loss using tcset to test Cassandra's limits.

6. Code Explanation

The code written to test Cassandra's performance is written in Python. We used the DataStax Python Driver to connect and perform operations on the database. Four different codes were written to test other parameters in Cassandra. All the four codes consisted of a common code in the beginning when the python driver connects to the database. The code is given below (Screenshot 7) :



read.py > ...

```python
from cassandra.cluster import Cluster, ExecutionProfile, EXEC_PROFILE_DEFAULT
from cassandra import ConsistencyLevel
import time

profile = ExecutionProfile(
        consistency_level=ConsistencyLevel.THREE
)

cluster = Cluster(['192.168.122.11', '192.68.122.12', '192.168.122.13'], execution_profiles={EXEC_PROFILE_DEFAULT: profile})

timeStart = time.time()
session = cluster.connect('edgedb')
timeMid = time.time()

print("Connected to Database!")
print("Reading Data...")
```

Screenshot 7

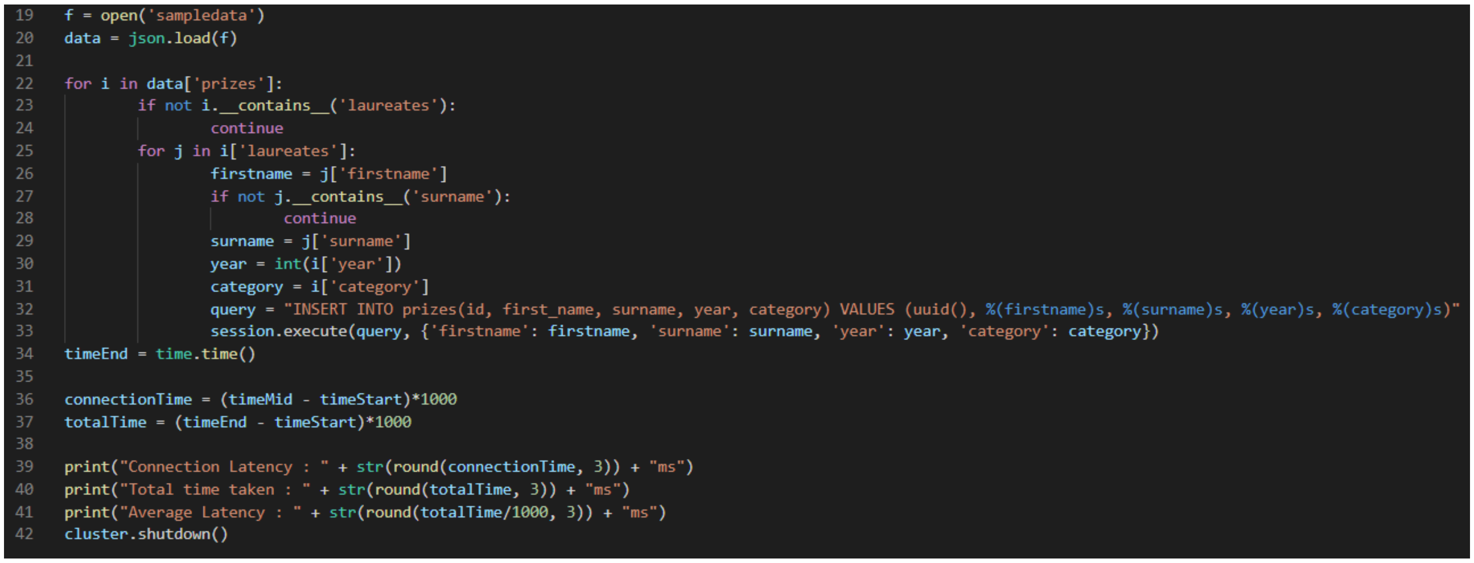
Firstly, we import the libraries that enable us to interact with Cassandra. Next, we create a Cassandra cluster object in which we specify the edge nodes IP addresses and the execution profile. In the execution profile, we mention the consistency level for that particular experiment.

Secondly, the connect method is called on the cluster object and the name of the keyspace (*edgedb*) is specified as the parameter. This returns a session object with which we will be able to execute queries and other operations on the database. The time taken to connect to the database is recorded as the Connection Latency.

Lastly, the sample data used for the experiments is a list of all the novel prize winners in all categories from 1900 to 2021.

6.1 Thousand Writes

The sample data is converted from JSON format to a python dictionary. The data is then parsed and a query string is generated which is passed on as a parameter to the execute method of the session object. This inserts the data in the *prizes* table family. The time taken to insert all the thousand rows is recorded and displayed. (Screenshot 8)

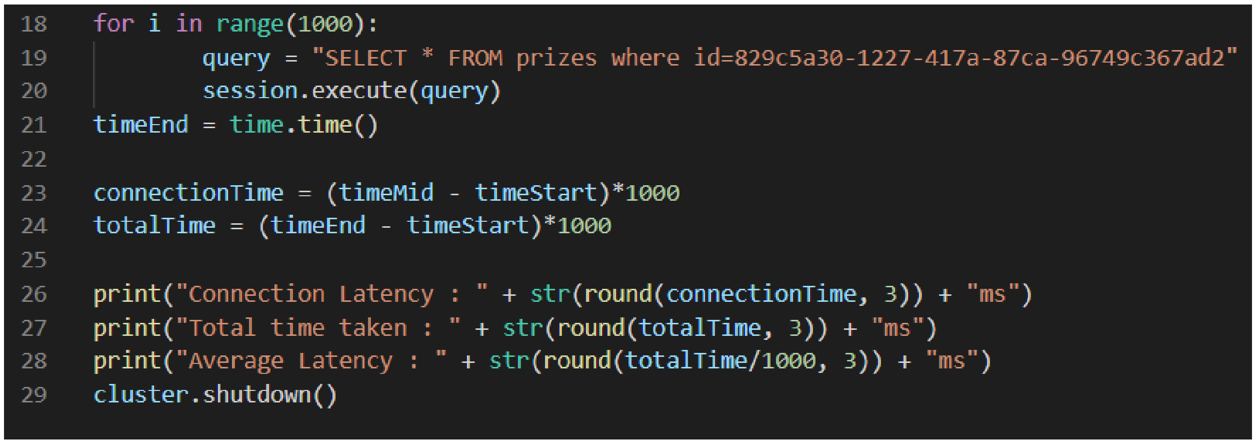


```python
f = open('sampledata')
data = json.load(f)

for i in data['prizes']:
        if not i.__contains__('laureates'):
                continue
        for j in i['laureates']:
                firstname = j['firstname']
                if not j.__contains__('surname'):
                        continue
                surname = j['surname']
                year = int(i['year'])
                category = i['category']
                query = "INSERT INTO prizes(id, first_name, surname, year, category) VALUES (uuid(), %(firstname)s, %(surname)s, %(year)s, %(category)s)"
                session.execute(query, {'firstname': firstname, 'surname': surname, 'year': year, 'category': category})
timeEnd = time.time()

connectionTime = (timeMid - timeStart)*1000
totalTime = (timeEnd - timeStart)*1000

print("Connection Latency : " + str(round(connectionTime, 3)) + "ms")
print("Total time taken : " + str(round(totalTime, 3)) + "ms")
print("Average Latency : " + str(round(totalTime/1000, 3)) + "ms")
cluster.shutdown()
```

Screenshot 8

6.2 Thousand Reads

The read code executes the same query thousand times and the time taken for this is recorded and displayed. (Screenshot 9)



```python
for i in range(1000):
        query = "SELECT * FROM prizes where id=829c5a30-1227-417a-87ca-96749c367ad2"
        session.execute(query)
timeEnd = time.time()

connectionTime = (timeMid - timeStart)*1000
totalTime = (timeEnd - timeStart)*1000

print("Connection Latency : " + str(round(connectionTime, 3)) + "ms")
print("Total time taken : " + str(round(totalTime, 3)) + "ms")
print("Average Latency : " + str(round(totalTime/1000, 3)) + "ms")
cluster.shutdown()
```

Screenshot 9

Similar codes were written for single read and write instead of thousand reads and writes. This was for testing purposes and all the code can be found in our github.

7. Findings Explanation

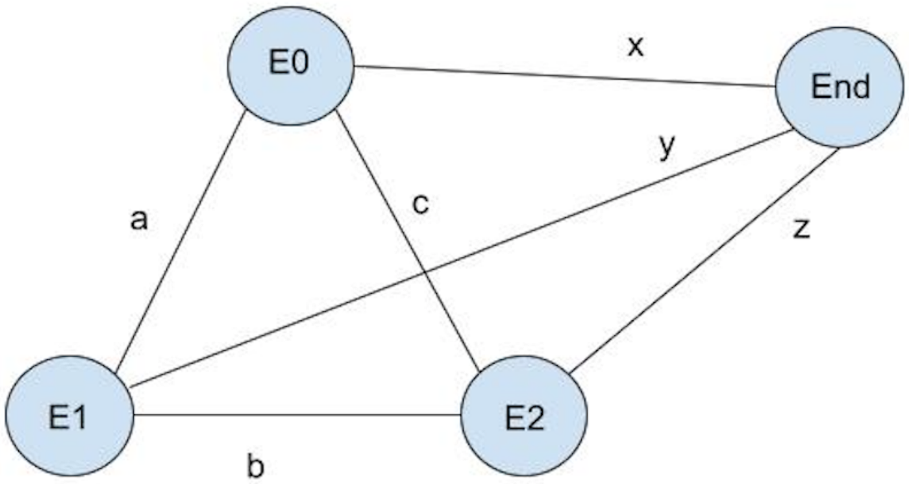


Figure 7

Our setup (Figure 7) consists of 3 nodes namely *E0*, *E1* and *E2* with 1 endpoint *End*. The latencies between the different machines are indicated in the above diagram. The nodes in our Cassandra cluster follow the SimpleSnitch protocol to exchange data between themselves. Initially, we set the consistency level as 2, and the replication factor as 3.

7.1 Constant edge latencies with fixed increments in endpoint latencies

Initially, we kept the latencies between the edges constant, i.e,
a = 10ms
b = 10ms
c = 10ms

We chose these initial values as per [https://par.nsf.gov/servlets/purl/10184999], where for the user to edge servers latency, more than 50% of the users experienced around 10ms latency. Hence, we believe this will serve as a realistic value for our experiments.

Now, we changed the edge-to-endpoint latency, i.e, x, y and z, in increments of 10. For example
x = y = z = 10ms
x = y = z = 20ms

This gave us the write RTT as 43.69ms and 64.5ms respectively. This behavior is very straightforward and as expected, since increasing the wire latency by 10ms for every link will increase the RTT by 20ms.

We did not pursue this experiment further, since the results did not produce anything new and were as expected, giving us a linear graph with a constant slope

<u>**7.2 Constant endpoint latencies with fixed increments in edge latencies**</u>

Initially, we kept the endpoint-to-edge latencies constant, i.e,
x = 10ms
y = 10ms
z = 10ms

Now, we changed the latencies between the edge nodes, i.e, a, b and c, in increments of 10.
For example
a = 5ms, b = 10ms, c = 15ms
a = 15ms, b = 20ms, c = 25ms

By changing the consistency levels, we found that any node can act as the coordinator, and at first requests will be sent to the nodes which our driver knows about (specified in the python program). But once it connects and understands our cluster, it may change to a "closer" coordinator.

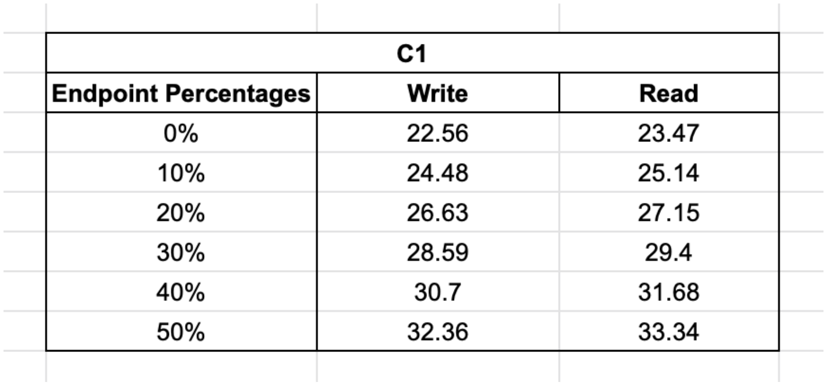
Again, for this experiment, we found that we got a linear graph on plotting the graph and this was as expected.

<u>**7.3 Constant edge/endpoint latencies with percentage increments in endpoint/edge latencies**</u>

To study which has a greater effect, changing the endpoint-to-edge latency or edge-to-edge latencies, we decided to increment either of these by 10% while keeping the other constant. Further, we also varied the consistency levels to study the impact of the link latencies in either case.
The base values are as follows:
a = 5ms, b = 10ms, c = 15ms
x = y = z = 10ms

7.3.1 Consistency Level = 1

a) Varying endpoint-to-edge latencies

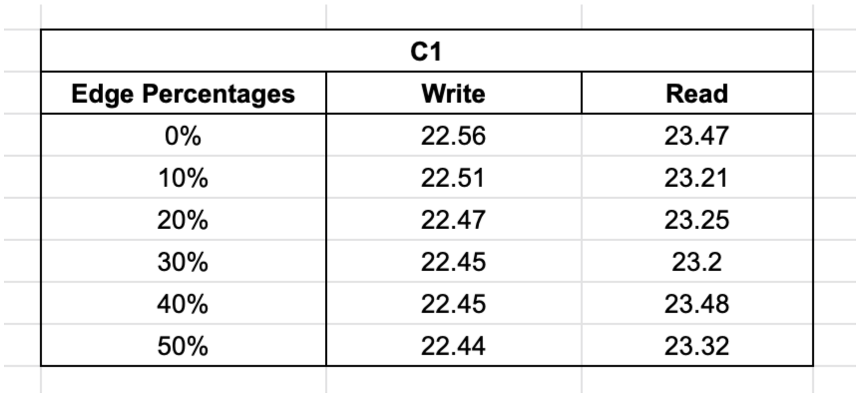


C1		
Endpoint Percentages	**Write**	**Read**
0%	22.56	23.47
10%	24.48	25.14
20%	26.63	27.15
30%	28.59	29.4
40%	30.7	31.68
50%	32.36	33.34

Table 1

The read latencies are slightly higher because in the case of write operations, the data is written in the memory and not on the disk. For a read operation, we need to go through a lot of filters to actually get the required object from the database, hence taking a slightly longer time.
We can clearly see a slight increase in the read/write operation RTT as we increase the endpoint-to-edge latencies.

b) Varying edge latencies



C1		
Edge Percentages	**Write**	**Read**
0%	22.56	23.47
10%	22.51	23.21
20%	22.47	23.25
30%	22.45	23.2
40%	22.45	23.48
50%	22.44	23.32

Table 2

Here, we observe that there is very little difference even when we change the latencies by up to 50% between the edge nodes. The read/write operations were happening with the same delay. This can be attributed to the fact that the consistency level for the cluster is 1. This means that the minimum number of nodes needed to acknowledge the read/write operation is 1. Since, the nodes do not need to exchange the data between themselves to acknowledge the request, the link latencies between the edge nodes does not affect the overall latency of the request. The endpoint simply contacts the cluster, and the node which receives the request, simply responds directly to the endpoint.

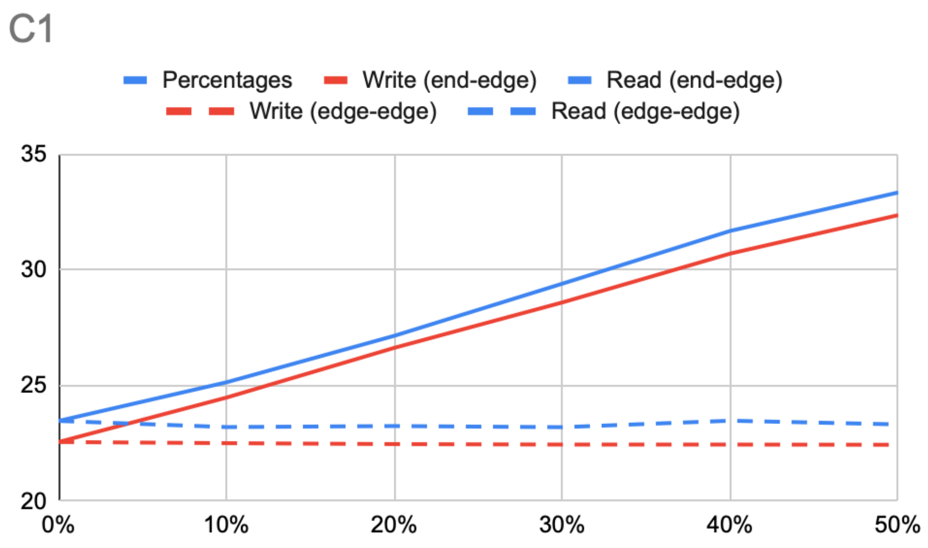


Figure 8

The solid line indicates the RTT when the endpoint-to-edge latencies are increased and the dotted line indicates the RTT when the edge latencies are increased. (Figure 8)

7.3.2 Consistency Level = 2

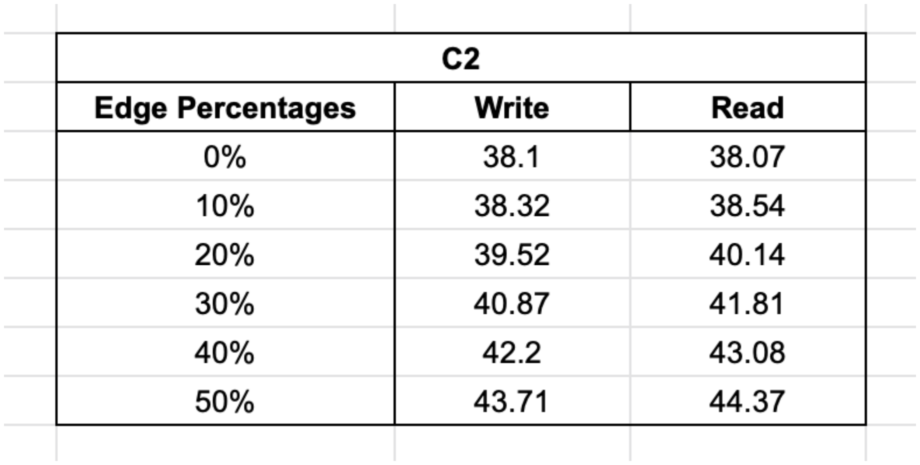
a) Varying endpoint-to-edge latencies

C2		
Endpoint Percentages	**Write**	**Read**
0%	38.1	38.07
10%	38.84	39.53
20%	40.9	41.57
30%	43.19	43.74
40%	45.14	45.86
50%	46.83	47.56

Table 3

The base RTT for the read/write operations, i.e, when the endpoint percentage is 0%, is more as compared to the previous result. This is expected because increasing the consistency level leads to more communication between the edge nodes. This leads to an increased RTT overall.

b) Varying edge latencies



C2		
Edge Percentages	**Write**	**Read**
0%	38.1	38.07
10%	38.32	38.54
20%	39.52	40.14
30%	40.87	41.81
40%	42.2	43.08
50%	43.71	44.37

Table 4

When varying the edge latencies, there is a slight increase in the read/write operation RTT. This is higher as compared to when the consistency level is 1. Again, this is as expected since there is communication between at least two edge nodes to acknowledge the requests.

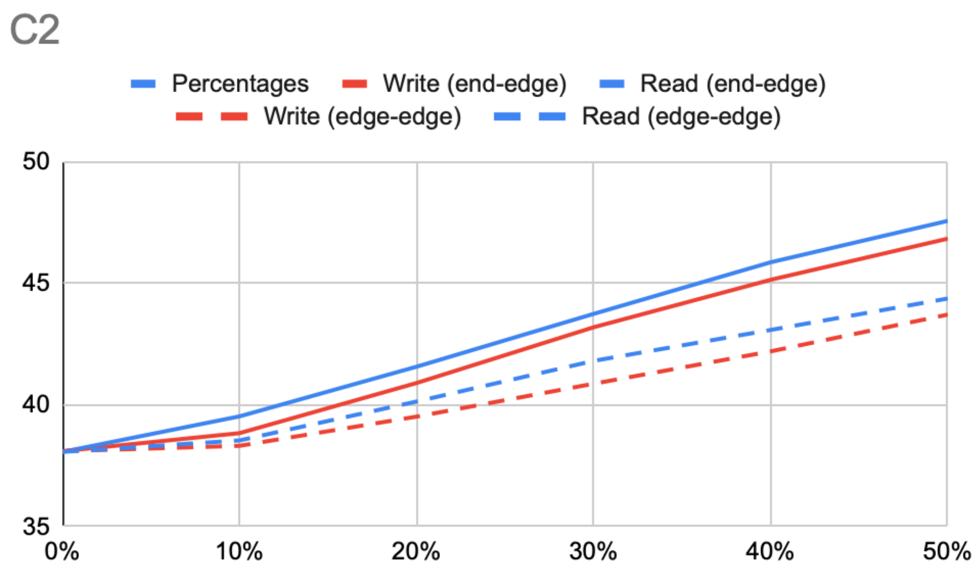
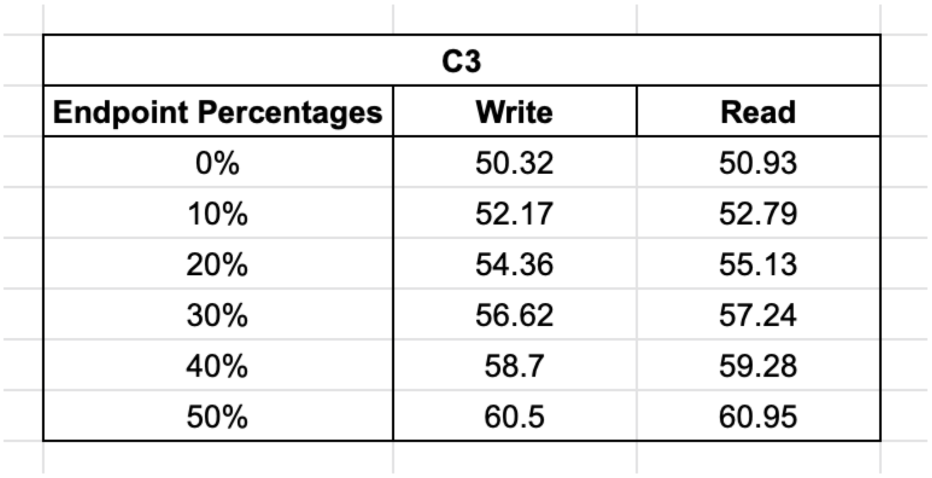


Figure 9

In Figure 9, we see the edge RTT "catching up" to the endpoint-to-edge RTT. The slope for the edge latencies has increased as compared to the previous graph (Figure 8) due to the increase in the consistency level.

7.3.3 Consistency Level = 3

a) Varying endpoint-to-edge latencies



C3		
Endpoint Percentages	**Write**	**Read**
0%	50.32	50.93
10%	52.17	52.79
20%	54.36	55.13
30%	56.62	57.24
40%	58.7	59.28
50%	60.5	60.95

Table 5

This configuration gives the highest RTT for a read/write operation yet. This is due to all 3 of the nodes communicating with each other to acknowledge a request. The values increase in a similar pattern to the others.

b) Varying edge latencies

C3		
Edge Percentages	**Write**	**Read**
0%	50.32	50.93
10%	52.87	53.24
20%	55.77	56.46
30%	58.89	59.16
40%	61.19	61.99
50%	64.077	64.47

Table 6

We see the maximum difference between the initial and final read/write operations as compared to the consistency levels. This is expected as per our previous explanations.

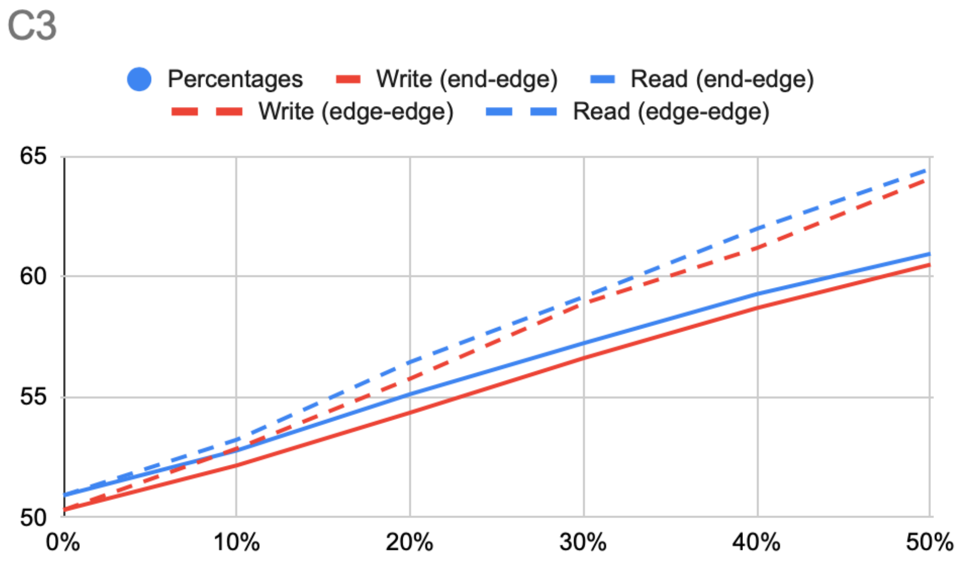


Figure 10

The slope for the edge RTT is more than the endpoint-to-edge RTT. This is an interesting observation, as we can conclude that for a consistency level of 3 for three nodes, increasing the edge latencies has a higher effect as compared to changing the endpoint-to-edge latencies.

8. Conclusion

We set up a Cassandra cluster on the virtual machines deployed using the Continuum framework. Using the Cassandra python driver, we write scripts to read and write a 1000 objects. These scripts are used to run experiments while changing the consistency levels of our cluster.
By changing the consistency levels, we found that any node can act as the coordinator, and at first requests will be sent to the nodes which our driver knows about. But once it connects and understands our cluster, it may change to a "closer" coordinator.
Further, we find that for lower consistency levels, the latencies between the endpoint and edge nodes have a larger effect on the total RTT. For higher consistency levels, the latencies between the edge nodes have a greater impact.

9. References

[1] Cloudy with a Chance of Short RTTs (https://dl.acm.org/doi/10.1145/3487552.3487854)
[2] Cassandra Architecture
(https://cassandra.apache.org/doc/latest/cassandra/architecture/overview.html)
[3] Continuum framework (https://github.com/atlarge-research/continuum)
[4] Surrounded by Clouds: A Comprehensive Cloud Reachability Study
(https://doi.org/10.1145/3442381.3449854)
[5] Sharing and Caring of Data at the Edge
(https://www.usenix.org/conference/hotedge20/presentation/trivedi)