GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Practical Course Report

Performance analysis/measurements with Cassandra and HBase

Abdul Rafay

MatrNr: 18413875

Supervisor: Prof. Dr. Julian Kunkel

Georg-August-Universität Göttingen
Institute of Computer Science

March 24, 2023

Abstract

HBase and Cassandra are both popular NoSQL databases used for handling large amounts of structured and semi-structured data. However, choosing between them can be challenging, as both have unique features and strengths. Benchmarking these two databases can help in making an informed decision. The specific problem we are trying to solve is to compare the performance of HBase and Cassandra in terms of read and write throughput, latency, and scalability. The aim is to identify which database is better suited for specific use cases based on their performance characteristics. There have been several benchmark studies conducted in the past to compare the performance of HBase and Cassandra. However, most of them are outdated and do not reflect the current state of these databases. Moreover, the benchmarking methodology and the workload used in these studies vary, making it difficult to compare the results. In this paper we evaluate the performance of HBase and Cassandra databases with YCSB benchmarking tool by generating 500000 records of 1000 bytes. We record the benchmarks for read and write workloads. We run the clusters of Hbase and Cassandra in docker containers and record the benchmarks while deploying the containerized clusters on AWS ECS cluster. At last we compare the benchmark results of both the databases with respect to latency and throughput.

Contents

List of Tables iv

List of Figures iv

List of Listings iv

List of Abbreviations v

1 Introduction: HBase and Cassandra the two NoSQL Databases 1
- 1.1 HBase . . . 1
- 1.2 Cassandra . . . 1
- 1.3 Use Cases . . . 1
- 1.4 Applications . . . 2
- 1.5 Benefits . . . 2

2 HBase Insights 2
- 2.1 Data Model . . . 2
- 2.2 HBase Architecture . . . 3
- 2.3 HBase Components . . . 3
- 2.4 HBase Internal Working . . . 4
  - 2.4.1 Reads and Writes . . . 4
  - 2.4.2 Communication with HDFS . . . 5
  - 2.4.3 Log Flusher and Log Roller in HLog . . . 5

3 Cassandra Insights 6
- 3.1 Data Model . . . 6
- 3.2 Cassandra Architecture . . . 7
- 3.3 Cassandra Internal Working . . . 7
  - 3.3.1 Reads and Writes . . . 7

4 Project Architecture 9

5 How to Deploy 9
- 5.1 Step 1: Clone the Project Repository . . . 9
- 5.2 Step 2: Configure Infrastructure Parameters . . . 10
- 5.3 Step 3: Configure the Shell script to deploy Hbase cluster or Cassandra cluster . . . 10
- 5.4 Step 4: Execute Terraform Scripts to Provision AWS Infrastructure . . . 11

6 Project File Insight 12
- 6.1 Cassandra Docker Setup . . . 12
  - 6.1.1 Cassandra Cluster Setup with Docker File . . . 12
  - 6.1.2 Cassandra Benchmark Shell File . . . 14
- 6.2 HBase Docker Setup . . . 15
  - 6.2.1 HBase Cluster Setup with Docker File . . . 15
  - 6.2.2 HBase Benchmark Shell File . . . 17

7 Dataset **18**
7.1 Dataset 1: YCSB Benchmark Framework Dataset 18

8 Benchmarks and Results **18**
8.1 Metrics and Testing Strategy 18
8.2 Workload Types in Benchmarking with YCSB 19
8.3 YCSB Performance KPIs 20
8.4 Evaluation and Results 20
8.4.1 Test Setup 20
8.4.2 Workload A: Heavy Update Workload 21
8.4.3 Workload B: Read Mostly Workload 22

9 Conclusion **23**

References **24**

A Code samples **A1**

List of Tables

List of Figures

1 HBase Architecture 3
2 HBase File Mapping to HDFS 4
3 HBase Write Operation 5
4 Cassandra Architecture 8
5 Project Architecture 9
6 AWS EC2 Instance look 11
7 AWS ECS Cluster look 12
8 YCSB Benchmark Results File 20
9 Read-Write Latency and Throughput for HBase and Cassandra: Workload A 21
10 Read Operation Mean Latency compared with the Read Counts for HBase and Cassandra: Workload A 22
11 Read-Write Latency and Throughput for HBase and Cassandra: Workload A 23
12 Read Operation Mean Latency compared with the Read Counts for HBase and Cassandra: Workload A 24

List of Listings

List of Abbreviations

HPC High-Performance Computing

AWS Amazon Web Services

ECS Elastic Container Service

YCSB Yahoo Cloud Serving Benchmark

1 Introduction: HBase and Cassandra the two NoSQL Databases

As the world is moving towards big data, distributed database systems and clusters are becoming more popular for their ability to handle large amounts of data and provide high availability and fault tolerance. These systems distribute data across multiple nodes in a cluster to allow for parallel processing and to prevent data loss in the event of a node failure. In the realm of distributed databases, NoSQL databases have become an increasingly popular choice due to their flexibility, scalability, and ability to handle unstructured data.

NoSQL databases differ from traditional relational databases by not enforcing a strict schema, allowing for more flexible data storage and retrieval. They provide powerful features such as scalability, high availability, and fault tolerance that are essential for modern distributed applications.

1.1 HBase

HBase is a distributed, scalable, and column-oriented NoSQL database that runs on top of the Hadoop Distributed File System (HDFS). It is modeled after Google's Bigtable and provides random access to large amounts of structured and semi-structured data. HBase is designed to provide low latency, high throughput, and scalability while ensuring data consistency. It lies in the consistency and partition tolerance (CP) side of the CAP theorem, sacrificing availability to ensure data consistency. [al11]

HBase is widely used in applications that require real-time data processing such as social media, online gaming, and financial services. Its ability to handle massive amounts of data with low latency makes it suitable for use cases such as real-time analytics, fraud detection, and recommendation engines.

1.2 Cassandra

Cassandra is another distributed, scalable, and column-oriented NoSQL database that was originally developed by Facebook. It provides high availability, fault tolerance, and scalability and is designed to handle large amounts of unstructured data across multiple commodity servers. Cassandra is built on the principles of Amazon's Dynamo and Google's Bigtable, making it a highly scalable and performant database. It lies in the availability and partition tolerance (AP) side of the CAP theorem, sacrificing consistency to ensure high availability. [Avi10]

Cassandra is widely used in applications that require high availability and scalability such as online retail, social media, and IoT. Its ability to handle massive amounts of data with high availability makes it suitable for use cases such as real-time analytics, content management, and messaging applications.

1.3 Use Cases

HBase and Cassandra are both commonly used in big data applications that require high scalability, availability, and fault tolerance. HBase is best suited for use cases that require strong consistency such as financial services and online gaming. Cassandra, on the other

hand, is best suited for use cases that require high availability and scalability such as IoT and online retail.

1.4 Applications

HBase and Cassandra are used in a wide range of applications such as real-time analytics, content management, fraud detection, recommendation engines, and messaging applications. They are also used for log processing, social media, and online gaming.

1.5 Benefits

By using HBase or Cassandra, companies can achieve significant benefits such as scalability, high availability, fault tolerance, and low latency. These benefits translate into increased efficiency, improved customer experience, and reduced downtime. Companies that adopt HBase or Cassandra can achieve competitive advantages by leveraging their capabilities to handle large amounts of data and provide real-time processing capabilities.

2 HBase Insights

HBase is a distributed, scalable, and column-oriented NoSQL database that provides low latency, high throughput, and scalability while ensuring data consistency. It is designed to run on top of the Hadoop Distributed File System (HDFS) and provides random access to large amounts of structured and semi-structured data. This paper provides an overview of the HBase database architecture, its components, and how they work together.

2.1 Data Model

The data model for HBase is designed to support the storage and retrieval of large amounts of structured data. The HBase data model is based on a column-family-oriented approach, where data is organized into tables that consist of one or more column families. Each column family is a group of related columns that share a common prefix and storage characteristics. The data within a column family is stored in key/value pairs, where the key is a combination of the row key, column family, and column qualifier. The HBase data model has the following components:

- Tables: HBase tables are the top-level container for storing data. A table consists of one or more column families and is identified by a unique name.
- Column Families: A column family is a group of related columns that share a common prefix and storage characteristics. All columns within a column family have the same data type and are stored together on disk. Column families are defined when a table is created, and new column families can be added later if needed.
- Columns: A column is the smallest unit of data in HBase. Each column belongs to a column family and has a unique name within that column family. Columns are identified by a column qualifier, which is a string that follows the column family name.

- Row Keys: The row key is the primary means of accessing data in HBase. It is a unique identifier for each row in a table and is used to order the data within the table. Row keys can be of variable length and can be composed of any combination of characters.
- Row: It contains the row key and many other values of columns.
- Cell: It contains the value in byte array with timestamp and it maps to a certain column and row.
- Timestamps: HBase supports the storage of multiple versions of data within a row. Each version is identified by a timestamp, which is stored as part of the column qualifier. Timestamps can be used to filter or sort the data within a column family.

2.2 HBase Architecture

HBase is designed as a distributed system with a master-slave architecture. The master node is responsible for managing the cluster, while the slave nodes store the data. The architecture consists of several components, each responsible for a specific task.

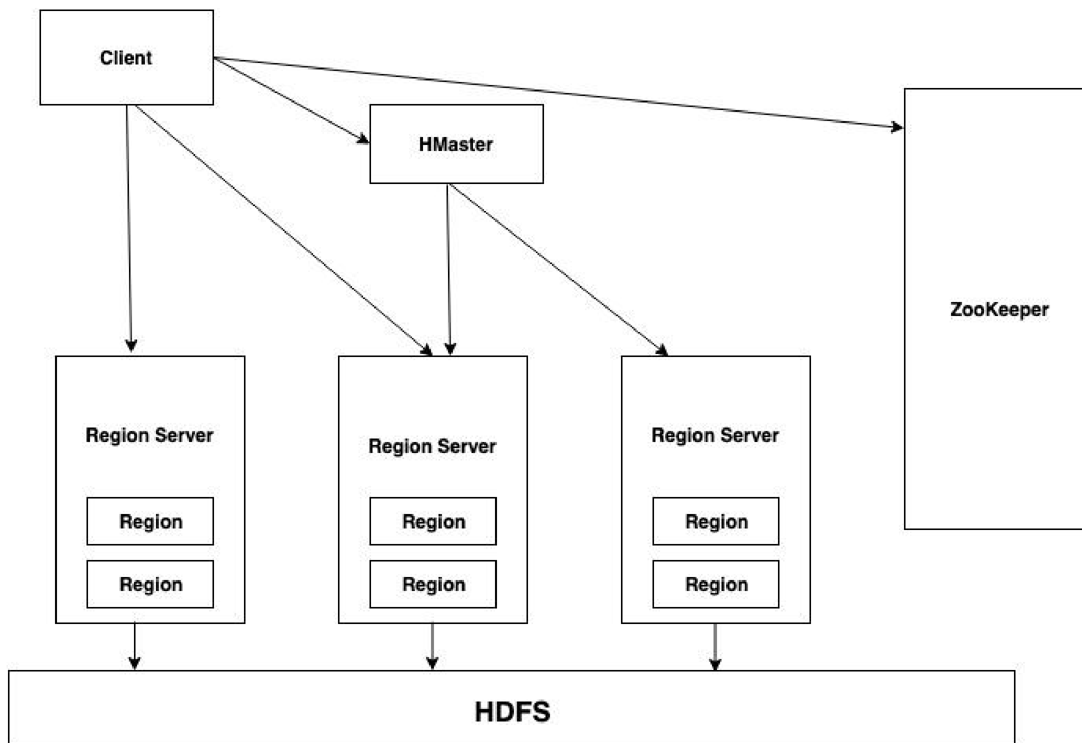

Figure 1: HBase Architecture

2.3 HBase Components

- HMaster: The HMaster is the central coordinating node in HBase. It is responsible for managing the cluster, including assigning regions to region servers, load balancing, and monitoring cluster health.

- Region Server: The Region Server is responsible for serving data for a set of regions. Each Region Server is responsible for a subset of the regions in the cluster.
- Zookeeper: Zookeeper is a distributed coordination service that is used to manage and synchronize the HBase cluster. It is responsible for managing the master failover, coordinating region server failover, and maintaining the cluster state.
- HDFS: HDFS is the underlying distributed file system that HBase runs on. HBase uses HDFS for storing the actual data.
- HBase Client: The HBase client provides an API for applications to interact with the HBase cluster. Applications can read and write data to HBase using the HBase client.

2.4 HBase Internal Working

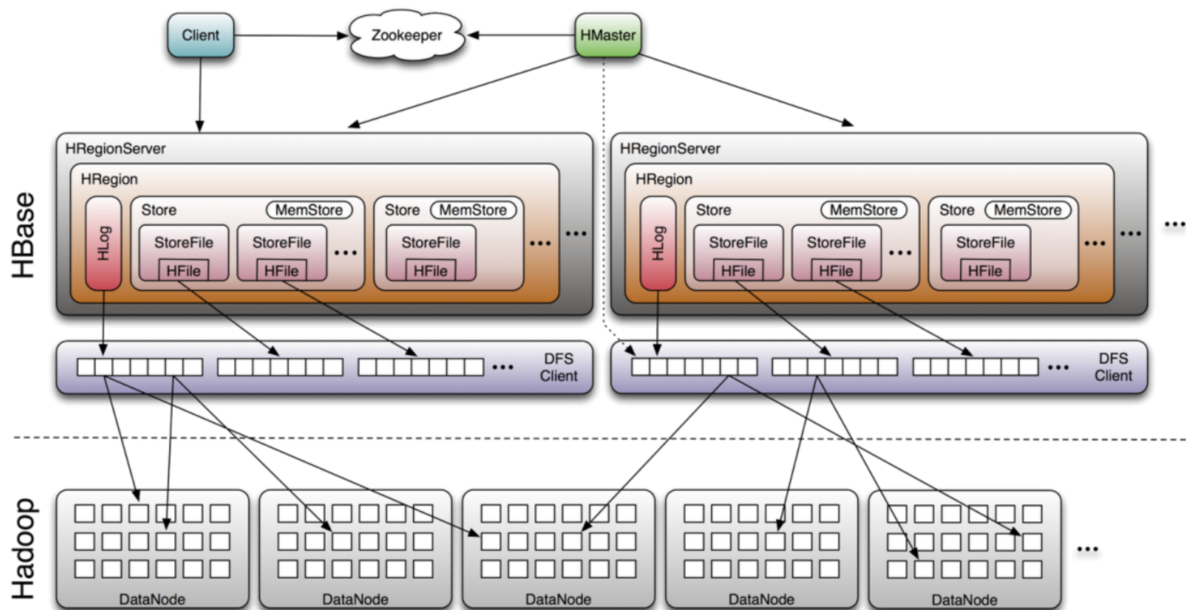

Figure 2: HBase File Mapping to HDFS [Geo09]

HBase is a distributed database that uses the Hadoop Distributed File System (HDFS) as its underlying storage mechanism. The internal workings of HBase involve several components working together to provide efficient storage and retrieval of large amounts of structured data.

2.4.1 Reads and Writes

When a client application submits a read or write request to HBase, the request is first processed by the HBase Client API. The client API communicates with the HBase RegionServer responsible for the data being accessed. The RegionServer is responsible for serving read and write requests for one or more regions of a table. Each region is stored on a separate RegionServer.

When a write request is received, the RegionServer first writes the data to the write-ahead log (WAL). The WAL is an append-only log file that records all changes made to the data in the region. Once the data is written to the WAL, it is then written to an in-memory store called the MemStore. The MemStore is periodically flushed to disk to create a new store file on HDFS. The store files are sorted by row key, making it easy to perform range queries.

When a read request is received, the RegionServer first checks if the data is in the MemStore. If the data is not in the MemStore, the RegionServer looks up the data in the HFiles stored on HDFS. The HFiles are sorted by row key, and the RegionServer uses a binary search to find the requested data.

2.4.2 Communication with HDFS

HBase communicates with HDFS to store data by creating HFiles on HDFS. HFiles are immutable files that are written once and never modified. HBase uses the Hadoop Distributed Copy (DistCP) tool to copy the HFiles from the RegionServer to HDFS. DistCP is a MapReduce-based tool that provides scalable and fault-tolerant copying of large data sets.

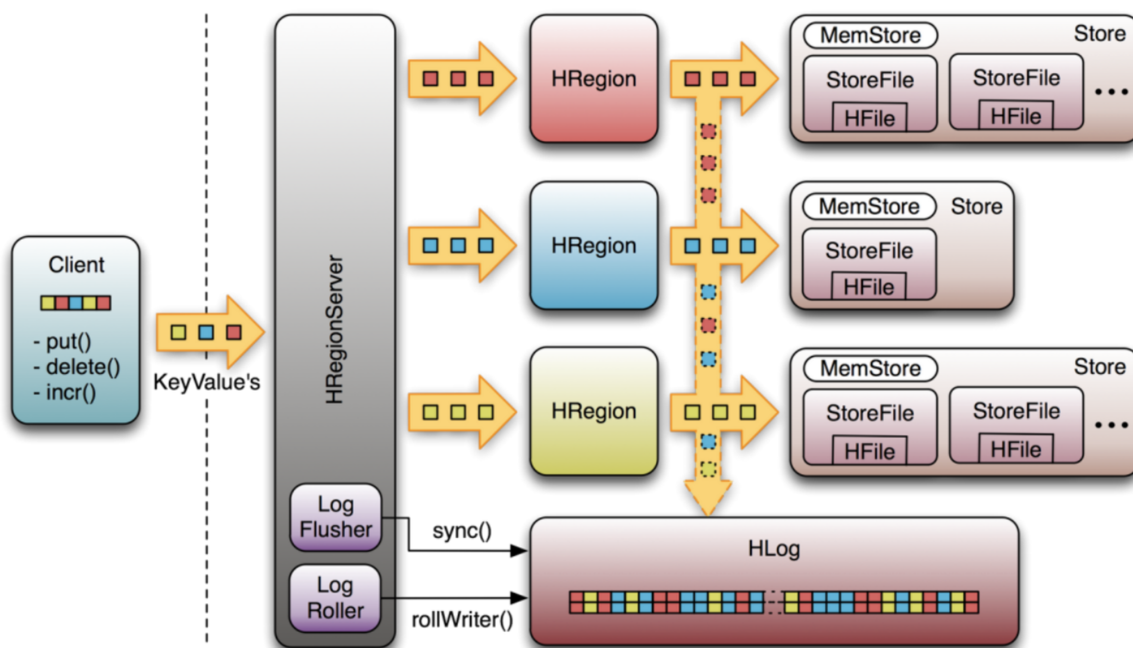

Figure 3: HBase Write Operation [Zoo]

2.4.3 Log Flusher and Log Roller in HLog

The write-ahead log (WAL) is a critical component of HBase, as it ensures that data is not lost in the event of a RegionServer failure. The log flusher is responsible for periodically flushing the MemStore to disk, creating new store files on HDFS. The log roller is responsible for rolling over the WAL to a new file when the current file reaches a certain size or age. The log roller also archives old log files to prevent them from taking up too much disk space.

In summary, HBase's internal workings involve several components working together to provide efficient storage and retrieval of large amounts of structured data. The write-ahead log (WAL) ensures that data is not lost in the event of a RegionServer failure, and the log flusher and log roller are responsible for managing the WAL. HBase communicates with HDFS to store data by creating HFiles on HDFS, and read and write requests are processed by the HBase Client API and the RegionServer.

3 Cassandra Insights

3.1 Data Model

Cassandra's data model is unique and different from the traditional relational databases. Cassandra's data model is column-family based, which is more flexible than the rigid structure of the relational model. The data model is designed to meet the demands of distributed systems, providing high scalability and fault tolerance. The column-family data model has the following components:

- Keyspace: Keyspace in Cassandra is similar to a schema in a traditional database. It acts as a container for tables and defines the replication strategy used for data storage. Keyspaces in Cassandra are used to group related column families and tables together.

- Column Families: Column families in Cassandra are used to organize and store data. They act as containers for rows of data, which are composed of a set of columns. Each column has a unique name and holds a value. The column families in Cassandra can be defined with varying degrees of complexity, depending on the application's needs. There are two types of column families in Cassandra: standard column families and super column families.

- Rows: Rows in Cassandra are similar to rows in traditional databases, but with a key difference. In Cassandra, the rows are uniquely identified by a row key, which is a string that serves as the primary key for the row. The row key is used to locate the data in the column family.

- Columns: Columns in Cassandra hold the actual data. They are identified by a unique name within a row and can have a value of any data type. Each column in Cassandra can be defined with a timestamp, which allows for the creation of time-series data.

- Secondary Indexes: Cassandra also provides support for secondary indexes. Secondary indexes allow for the creation of additional indexes on the data, beyond the primary key. This feature allows for more flexible querying and filtering of data.

In conclusion, Cassandra's data model is a flexible, column-family based model that provides high scalability and fault tolerance. It consists of keyspaces, column families, rows, columns, and secondary indexes. Cassandra's data model is designed to meet the demands of modern distributed systems, providing efficient data storage and retrieval for large datasets [Cas].

3.2 Cassandra Architecture

Cassandra's architecture is designed to provide high scalability, fault tolerance, and linear scalability, even when dealing with large amounts of data. The architecture is built on a peer-to-peer model, where all nodes are equal, and there is no single point of failure. Cassandra's architecture has the following components:

- Node: A node in Cassandra is a single instance of the database that runs on a physical or virtual machine. Nodes are the fundamental building blocks of a Cassandra cluster, and each node can act as both a coordinator and a replica for data. Cassandra's architecture is designed to scale horizontally by adding more nodes to the cluster, which allows for increased processing power and storage capacity.
- Cluster: A cluster in Cassandra is a group of nodes that work together to store and manage data. Clusters are designed to provide fault tolerance and high availability, even in the face of hardware failures or network outages. Cassandra's architecture allows for the dynamic addition and removal of nodes from the cluster, which allows for the seamless expansion and contraction of the database.
- Datacenter: A datacenter in Cassandra is a logical grouping of nodes that are geographically close together. Datacenters are used to provide fault tolerance and disaster recovery capabilities. Cassandra's architecture allows for the creation of multiple datacenters, which can be spread across different geographic regions.
- Coordinator: A coordinator in Cassandra is responsible for routing read and write requests to the appropriate nodes in the cluster. Each node in Cassandra can act as a coordinator, which allows for load balancing and fault tolerance.
- Replication: Cassandra's architecture utilizes replication to ensure data durability and high availability. Each node in the cluster can act as a replica for data, which means that data is replicated across multiple nodes. Cassandra's architecture allows for configurable replication factors, which determine the number of replicas that are created for each piece of data.

In conclusion, Cassandra's architecture is designed to provide high scalability, fault tolerance, and high availability. The architecture is built on a peer-to-peer model, where nodes are organized into clusters, and each node can act as both a coordinator and a replica for data. Cassandra's architecture utilizes replication to ensure data durability and high availability, and it allows for the dynamic addition and removal of nodes from the cluster [Dat].

3.3 Cassandra Internal Working

3.3.1 Reads and Writes

In Apache Cassandra, reads and writes are optimized for high throughput and happen asynchronously. Cassandra's distributed architecture, partitioning scheme, and replication strategy are designed to handle large amounts of data and provide high availability and fault tolerance [ORe].

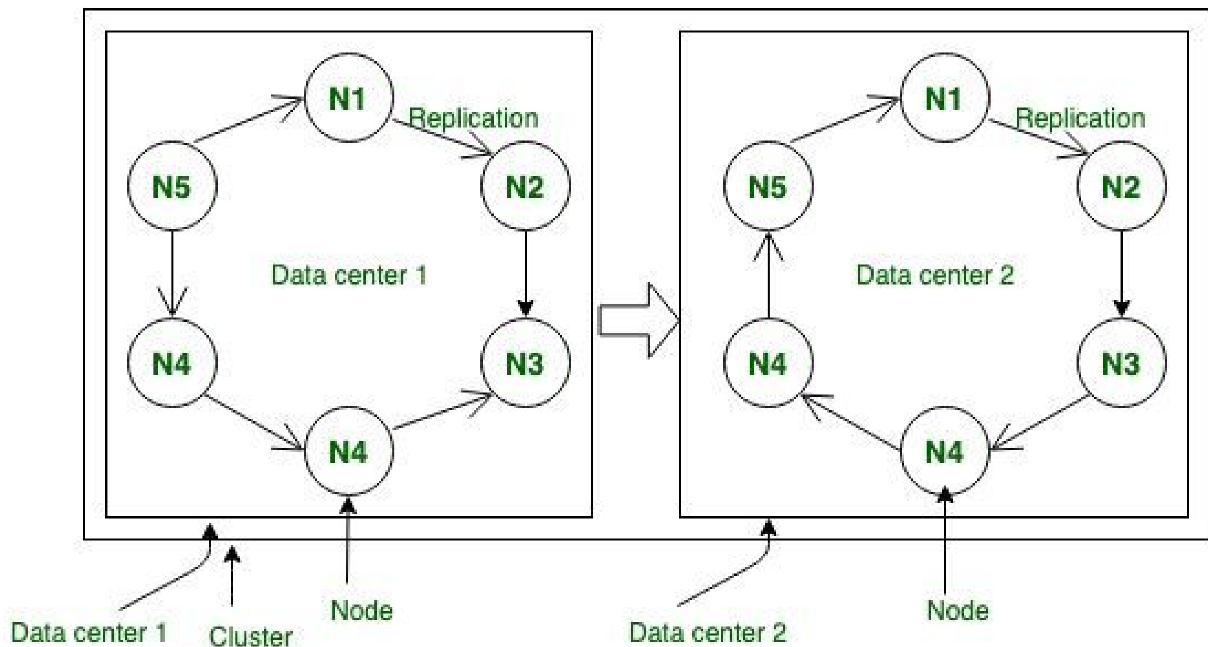

Figure 4: Cassandra Architecture [Zoo]

- Partitioning: Cassandra partitions data across multiple nodes in a cluster. Each node is responsible for a portion of the data, and data is distributed across the cluster based on a partition key. Cassandra's partitioning scheme is designed to distribute data evenly across the cluster, which allows for high scalability and performance.

- Replication: Cassandra utilizes replication to ensure high availability and fault tolerance. Each piece of data is replicated across multiple nodes in the cluster, which means that data is available even if one or more nodes fail. Cassandra's replication scheme is configurable, and it allows for the creation of replicas across multiple datacenters for disaster recovery purposes.

- Consistency Levels: Cassandra's consistency levels determine how many nodes must acknowledge a read or write operation before it is considered successful. Cassandra's consistency levels are configurable, and they range from ONE (where only one node needs to acknowledge the operation) to ALL (where all nodes must acknowledge the operation). Cassandra's consistency levels allow for fine-grained control over the trade-off between consistency and performance.

- Write Path: When a write operation is performed in Cassandra, the data is first written to a commit log on disk for durability. After the write is acknowledged, the data is written to an in-memory structure called the memtable. Once the memtable reaches a certain size, it is flushed to disk as an immutable SSTable (sorted string table).

- Read Path: When a read operation is performed in Cassandra, the partition key is used to determine which nodes in the cluster hold the data. Cassandra then performs a parallel read from the nodes that hold the data, and the results are merged and returned to the client.

- Tuning: Cassandra provides a number of tuning options that allow for fine-grained control over read and write performance. Tuning options include compression, caching, and tuning of garbage collection settings.

4 Project Architecture

the project architecture involves the use of a Git repository that contains shell scripts for Apache Cassandra and Apache HBase, along with Terraform scripts. The Terraform scripts are responsible for provisioning the AWS Infrastructure for an ECS cluster in the Frankfurt region, which includes a VPC and a security group. The ECS cluster consists of three nodes, each with an instance type of t2.2xlarge. This instance type has 8 CPUs and 32 GB of RAM, along with moderate network performance of up to 5 Gbps. The three nodes run three containers, which are responsible for running either the HBase or Cassandra database. The containers are deployed using Docker, allowing for easy management and scalability. This architecture provides a robust and scalable solution for running HBase or Cassandra databases on AWS infrastructure with the use of Terraform and Docker containers as shown in the Figure. 5.

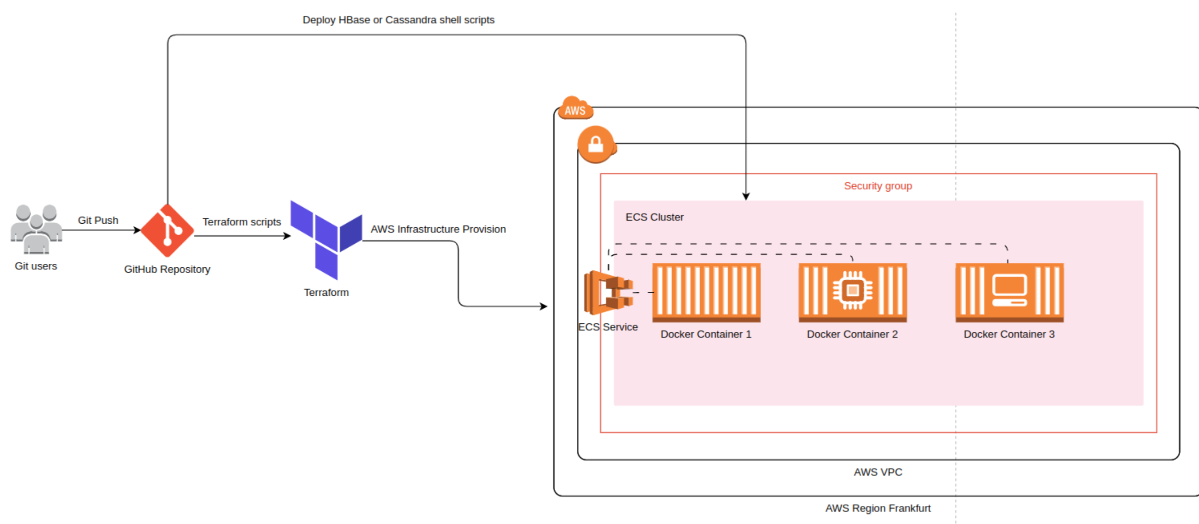

Figure 5: Project Architecture

5 How to Deploy

5.1 Step 1: Clone the Project Repository

```
# clone the project
git clone https://github.com/Rafay007/HPCSA-Project.git
```

5.2 Step 2: Configure Infrastructure Parameters

```
# Change the parameters of the project
nano terraform.tfvars
```

```
# Configure the AWS infrastructure parameters based on your requirements
# Region on which we want to provision our cluster
region = "eu-central-1"
# Instance types represent the VM configuration, if you have heavy workload then you
instance_type  = "t2.2xlarge"
# user for the connection instances
ssh_connection_user = "ubuntu"
# which base image to use on VMs
ami = "ubuntu/images/hvm-ssd/ubuntu-*-20.04-amd64-server-*"
# VPC Private CIDR and how many IP addresses you want
vpc_cidr = "178.0.0.0/16"
# VPC Public CIDR and how many IP addresses you want
public_subnet_cidr = "178.0.10.0/24"
# Provide path to your AWS Account credentials
creds = "~/.aws/credentials"
# Project Tag
tag = "terraform-test"
```

5.3 Step 3: Configure the Shell script to deploy Hbase cluster or Cassandra cluster

When the infrastructure is provisioned, it is going to execute the shell scripts on start. Below part in the 'main.tf' terraform file which is responsible to execute the shell script.

We need to make `all_in_one.sh` shell script to point the Cassandra or HBase database shell scripts.

For Cassandra Database:

```
# Create a shell script to point which database files
nano all_in_one.sh
# Put the below content in the shell file in case of Cassandra database
cassandra_on_aws/run-cassandra.sh # shell script to spawn cassandra cluster on AWS EC
cassandra_on_aws/run-cassandra-benchmark.sh # shell script to run benchmarks on cassa
```

For HBase Database:

```
# Create a shell script to point which database files
nano all_in_one.sh
# Put the below content in the shell file in case of HBase database
hbase_on_aws/run-hbase.sh # shell script to spawn cassandra cluster on AWS ECS
hbase_on_aws/run-hbase-benchmark.sh # shell script to run benchmarks on cassandra clu
```

```
# Create a task definition for the shell script
resource "aws_ecs_task_definition" "ecs_task_definition" {
  family                   = "my-task"
  container_definitions    = jsonencode([{
    name      = "my-container"
    image     = "my-docker-image"
    cpu       = 512
    memory    = 1024
    essential = true
    command   = ["/bin/bash", "all_in_one.sh"] # this is the script to run on start
  }])
  requires_compatibilities = ["EC2"]
}
```

5.4 Step 4: Execute Terraform Scripts to Provision AWS Infrastructure

```
# Install Terraform on Linux if not installed
sudo apt-get install terraform
# Initialize Terraform Project
terraform init
# Validate terraform modules
terraform validate
# Deploy modules on AWS
terraform apply --auto-approve
```

Wait for few minutes 5 minutes for infrastructure to get provisioned, and verify if the ECS cluster exist on AWS, Figure. 6 illustrates the creation of ECS cluster on AWS with 3 EC2 Nodes in different Availability Zones as shown in Figure. 7.

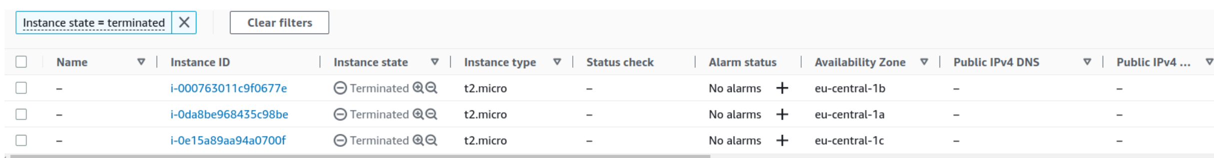

Figure 6: AWS EC2 Instance look

The AWS UI shows the cluster is up and running.

Figure 7: AWS ECS Cluster look

6 Project File Insight

This section of project provide a insights to the project files by demonstrating, how the Cassandra or HBase docker containers are made and communicating to each other.

6.1 Cassandra Docker Setup

6.1.1 Cassandra Cluster Setup with Docker File

`cassandra_on_aws/run-cassandra.sh` is responsible to spawn a Cassandra docker containers.

```

# Set the AWS region and ECS cluster name
AWS_REGION="eu-central-1"
ECS_CLUSTER_NAME="cassandra-cluster"

# Set the Docker image and container name
DOCKER_IMAGE="cassandra:latest"
DOCKER_CONTAINER_NAME="cassandra-container"


# Set the Docker network name
DOCKER_NETWORK_NAME="cassandra-network"

```

```
14 # Set the Cassandra keyspace and table name
15 KEYSPACE="mykeyspace"
16 TABLE_NAME="mytable"
17
18 # Set the SUSY dataset URL and file name
19 SUSY_DATASET_URL="https://archive.ics.uci.edu/ml/machine-learning-databases/00279/
20 SUSY.csv.gz"
21 SUSY_DATASET_FILE="SUSY.csv.gz"
22
23 if [ -x "$(command -v docker)" ]; then
24     echo "Update docker"
25     echo "Current docker version installed is $(command docker -v)"
26     echo "Total number of nodes to be: $1"
27 else
28     echo "Install docker"
29 fi
30
31 if [[ -n "$(docker images -q cassandra:latest)" ]]; then
32   echo "Official latest Cassandra docker image exists"
33 else
34     echo "Official latest Cassandra docker needs to be installed"
35     echo "Installing cassandra docker image $(command docker pull cassandra:latest)"
36 fi
37
38
39
40 # Set the path to the cqlsh command-line tool
41 CQLSH_PATH="/usr/bin/cqlsh"
42
43 # Set the path to the ccm command-line tool
44 CCM_PATH="/usr/local/bin/ccm"
45
46 # Create a new ECS cluster
47 echo "Creating new ECS cluster..."
48 aws ecs create-cluster --cluster-name $ECS_CLUSTER_NAME --region $AWS_REGION
49
50 # Create the Docker network
51 docker network create $DOCKER_NETWORK_NAME
52
53 # Launch the Cassandra nodes
54 echo "Launching $CASSANDRA_NODES Cassandra nodes..."
55 for (( i=1; i<=$CASSANDRA_NODES; i++ ))
56 do
57     docker run -d \
58         --name $DOCKER_CONTAINER_NAME-$i \
59         --network $DOCKER_NETWORK_NAME \
60         -e CASSANDRA_SEEDS=$DOCKER_CONTAINER_NAME-1 \
61         -e CASSANDRA_CLUSTER_NAME=$ECS_CLUSTER_NAME \
```

```
62          -e CASSANDRA_DC=dc1 \
63          -e CASSANDRA_RACK=rack1 \
64          $DOCKER_IMAGE
65  done
66
67  # Wait for the Cassandra nodes to start up
68  echo "Waiting for Cassandra nodes to start up..."
69  for (( i=1; i<=$CASSANDRA_NODES; i++ ))
70  do
71      until docker exec $DOCKER_CONTAINER_NAME-$i nodetool status | grep -q "^UN"
72      do
73          sleep 10
74      done
75  done
```

6.1.2 Cassandra Benchmark Shell File

`cassandra_on_aws/run-cassandra-benchmark.sh` file is responsible to download the dataset and run benchmarks on the Cassandra cluster.

```
1
2   # Set the AWS region and ECS cluster name
3   AWS_REGION="eu-central-1"
4   ECS_CLUSTER_NAME="cassandra-cluster"
5   CASSANDRA_CONTACT_POINT=$(aws ecs describe-clusters --region $AWS_REGION
6   --clusters $ECS_CLUSTER_NAME --query "clusters[0].registeredContainerInstances[0].ec2
7   --output text | xargs aws ec2 describe-instances --region $AWS_REGION --instance-ids
8   | jq -r ".Reservations[].Instances[].PrivateIpAddress")
9
10  # Set the Docker image and container name
11  DOCKER_IMAGE="cassandra:latest"
12  DOCKER_CONTAINER_NAME="cassandra-container"
13  SUSY_DATASET_URL="https://archive.ics.uci.edu/ml/machine-learning-databases/00279/
14  SUSY.csv.gz"
15
16  # Set the Docker network name
17  DOCKER_NETWORK_NAME="cassandra-network"
18
19
20
21
22  # Create a keyspace and table in Cassandra
23  echo "Creating keyspace and table in Cassandra..."
24  docker exec $DOCKER_CONTAINER_NAME-1 cqlsh -e "CREATE KEYSPACE $KEYSPACE WITH
25  replication = {'class': 'SimpleStrategy', 'replication_factor': $CASSANDRA_NODES};"
26
27  docker exec $DOCKER_CONTAINER_NAME-1 cqlsh -e "CREATE TABLE $KEYSPACE.$TABLE_NAME
28  (id int primary key, value text);"
```

```
29
30
31 # Run the benchmark on YCSB benchmark framework and dataset
32 git clone https://github.com/brianfrankcooper/YCSB.git
33
34
35 # Run the below commands in cqlsh shell
36 docker exec $DOCKER_CONTAINER_NAME-1 cqlsh -e "create keyspace ycsb
37 WITH REPLICATION = {'class' : 'SimpleStrategy', 'replication_factor': 3 };
38 USE ycsb;
39 create table usertable ( y_id varchar primary key, field0 varchar, field1 varchar,
40 field2 varchar, field3 varchar, field4 varchar, field5 varchar,
41 field6 varchar, field7 varchar, field8 varchar, field9 varchar);"
42
43
44
45 workloads=("workloadd" "workloade")
46
47 repeatrun=3
48 records=5000000
49 operations=300000
50 threads=400
51 driver="cassandra-cql"
52 hosts=$TASK_IP
53
54 for work in "${workloads[@]}"
55 do
56         echo "Loading data for" $work
57         ./bin/ycsb load $driver -P ./workloads/$work -p hosts=$hosts -p
58         recordcount=$records -threads 40 > $work"_load.log"
59
60         echo "Running tests"
61         for r in `seq 1 $repeatrun`
62         do
63                 ./bin/ycsb run $driver -P ./workloads/$work -p hosts=$hosts
64                 -p recordcount=$records -p operationcount=$operations -threads $threa
65         done
66         #Truncate table and start over
67         cqlsh -f cassandra_truncate $hosts
68 done
```

6.2 HBase Docker Setup

6.2.1 HBase Cluster Setup with Docker File

`/hbase_on_aws/run-hbase.sh` is responsible to spawn a 3 node Zookeeper, Hadoop and HBase docker containers.

```
...
...
# start hadoop by getting into hadoop master container
docker exec -it hadoop-master.1.$(docker service ps hadoop-master --no-trunc |
tail -n 1 | awk '{print $1}' ) /bin/sh -c
"sbin/stop-yarn.sh;sbin/stop-dfs.sh;bin/hadoop namenode
-format;sbin/start-dfs.sh;sbin/start-yarn.sh;"



# Launch the HBase nodes
for ((i=1;i<=$1;i++)); do
    echo "Launching HBase node $i..."
    docker run -d \
        --name "$DOCKER_CONTAINER_NAME-$i" \
        --network $DOCKER_NETWORK_NAME \
        -p "16010:$((16010 + $i - 1))" \
        -p "9090:$((9090 + $i - 1))" \
        -p "2181:$((2181 + $i - 1))" \
        -e "HBASE_MANAGES_ZK=false" \
        -e "HBASE_REGIONSERVERS=$1" \
        -e "HBASE_HEAPSIZE=2G" \
        -e "HBASE_MASTER_HOST=$DOCKER_CONTAINER_NAME-1" \
        -e "HBASE_MASTER_PORT=16000" \
        -e "ZOOKEEPER_QUORUM=$DOCKER_CONTAINER_NAME-1,$DOCKER_CONTAINER_NAME-2
        ,$DOCKER_CONTAINER_NAME-3" \
        -e "ZOOKEEPER_CLIENT_PORT=$((2181 + $i - 1))" \
        "$DOCKER_IMAGE" \
        "/usr/local/hbase/bin/hbase-region-server start"
done

# Launch the HBase master
echo "Launching HBase master..."
docker run -d \
    --name "$DOCKER_CONTAINER_NAME-1" \
    --network $DOCKER_NETWORK_NAME \
    -p "16000:16000" \
    -p "16010:16010" \
    -p "9090:9090" \
    -p "2181:2181" \
    -e "HBASE_MANAGES_ZK=true" \
    -e "HBASE_REGIONSERVERS=$1" \
    -e "HBASE_HEAPSIZE=4G" \
    -e "ZOOKEEPER_QUORUM=$DOCKER_CONTAINER_NAME-1,$DOCKER_CONTAINER_NAME-2
    ,$DOCKER_CONTAINER_NAME-3" \
    -e "ZOOKEEPER_CLIENT_PORT=2181" \
    "$DOCKER_IMAGE" \
    "/usr/local/hbase/bin/hbase master start"
```

```
49
50 # Wait for the HBase nodes to start up
51 echo "Waiting for HBase nodes to start up..."
52 for ((i=1;i<=$1;i++)); do
53     while ! docker exec "$DOCKER_CONTAINER_NAME-$i" /usr/local/hbase/bin/hbase shell
54         sleep 10
55     done
56 done
57
```

6.2.2 HBase Benchmark Shell File

`hbase_on_aws/run-hbase-benchmark.sh` is responsible to run benchmarks on HBase cluster

```
1  # Set the AWS region and ECS cluster name
2  AWS_REGION="eu-central-1"
3  ECS_CLUSTER_NAME="hbase-cluster"
4
5  # Set the Docker image and container name
6  DOCKER_IMAGE="newnius/hbase:1.2.6"
7  DOCKER_CONTAINER_NAME="hbase-container"
8
9  # Set the Docker network name
10 DOCKER_NETWORK_NAME="hbase-network"
11
12 # Download the YCSB benchmark framework and dataset repo
13 git clone https://github.com/brianfrankcooper/YCSB.git
14
15 #copy the hbase-site.xml into YCSB config folder
16 cp /etc/hbase/hbase.conf/hbase-site.xml ./YCSB/hbase10/conf
17
18
19 # Run the below command in hbase shell
20 create 'usertable', 'cf', {SPLITS => (1..200).map
21 {|i| "user#{1000+i*(9999-1000)/200}"}, MAX_FILESIZE =>
22 4*1024**3}
23
24 workloads=("workloada" "workloadb")
25
26 repeatrun=3
27 records=5000000
28 operations=300000
29 threads=400
30 driver="hbase10"
31
32 for work in "${workloads[@]}"
33 do
```

```
34          echo "Loading data for" $work
35          ./bin/ycsb load $driver -P ./workloads/$work -p columnfamily=cf -p
36          hbase.zookeeper.znode.parent=/hbase-unsecure
37          -p recordcount=$records -threads 40 > $work"_load.log"
38          echo "Running tests"
39          for r in `seq 1 $repeatrun`
40          do
41                  ./bin/ycsb run $driver -P ./workloads/$work -p columnfamily=cf -p
42                  hbase.zookeeper.znode.parent=/hbase-unsecure
43                  -p recordcount=$records -p operationcount=$operations -threads
44                  $threads > $work"_run_"$r".log"
45          done
46          #Truncate table and start over
47          hbase shell ./hbase_truncate
48  done
49
50
```

7 Dataset

7.1 Dataset 1: YCSB Benchmark Framework Dataset

The Yahoo! Cloud Serving Benchmark (YCSB) is an open-source framework designed to test the performance and scalability of NoSQL databases and other data stores. The YCSB benchmark allows users to test different workloads against various database management systems, including Apache Cassandra, HBase, MongoDB, and many others. The benchmark generates a series of database operations based on different workloads and measures the system's performance in terms of throughput, latency, and scalability. YCSB supports a variety of workloads, including read-dominated, write-dominated, update heavy, read mostly, read latest workload and mixed workloads, allowing users to evaluate database performance under different conditions. The YCSB benchmark is widely used in academia and industry to compare the performance of different data stores and to evaluate the suitability of a database for a particular application.

The YCSB Client is a Java program that creates the data to be loaded into the database and generates the operations that constitute the workload. Using the YCSB client Java program, we created 400GB of data.

8 Benchmarks and Results

8.1 Metrics and Testing Strategy

To run the benchmark in YCSB (Yahoo! Cloud Serving Benchmark), there are several parameters that need to be set to configure the benchmark. Here are some of the commonly used benchmark parameters:

- **db**: specifies the name of the database to be benchmarked.
- **threads**: specifies the number of client threads to be used for the benchmark. To prevent latencies caused by clients, it is important to carefully choose the number of client test threads in YCSB. If a heavy benchmark workload is used with a small number of test threads, each thread will have too many requests to handle, resulting in a rise in request latency for reasons unrelated to the database.
- **target**: specifies the target number of operations per second to achieve during the benchmark.
- **recordcount**: specifies the total number of records to be inserted into the database. To avoid a local trap in YCSB, it is important to have a large enough number of records. The local trap occurs when only a few cluster nodes handle most of the operations, making it impossible to obtain an accurate overall performance of the cluster. This is often due to a lack of test data.
- **operationcount**: specifies the total number of operations to perform during the benchmark. To ensure a substantial and stable load across all nodes in YCSB, it is important to have a large enough number of operations. This will also ensure that the test can run for a sufficient amount of time to overcome any cold start and memory garbage collection effects.
- **workload**: specifies the workload to be used for the benchmark, such as workloada, workloadb, workloadc, etc. Whereas workloada means read-only workload that generates uniform random reads across the entire database, and so goes on for others.

In this project, we defined the number of records to 5 Million of 1000 bytes, and 0.3 Million number of operations, 400 threads mentioned in `hbase_on_aws/run-hbase-benchmark.sh` and `cassandra_on_aws/run-cassandra-benchmark.sh` files.

8.2 Workload Types in Benchmarking with YCSB

In YCSB (Yahoo! Cloud Serving Benchmark), a workload is a set of operations that simulate a specific application behavior. A workload is composed of a mix of different operations such as reads, writes, and updates that are executed on the database being tested. The goal is to measure the performance of the database under different workload conditions.

YCSB comes with several standard workloads, including:

- **workloada**: a mixture of read and write operations, generating 50% reads and 50% writes, with a higher write-to-read ratio.
- **workloadb**: a read-only workload that generates uniform random reads across the entire database.
- **workloadc**: a mixture of read and write operations, generating 95% reads and 5% updates, and intended to test the caching mechanisms of the database.
- **workloadd**: a range query workload that generates queries over a range of records in the database.

- **workloade**: a read-modify-write workload that reads a record, modifies it, and writes it back to the database.
- **workloadf**: a read-modify-write workload that also includes a scan operation.

In this project we make use of workload a and b. These workloads are designed to simulate different types of application behaviors and generate different types of loads on the database. By running multiple workloads on a database and measuring its performance, users can gain a better understanding of its behavior under various conditions.

8.3 YCSB Performance KPIs

YCSB provides several performance metrics or KPIs (Key Performance Indicators) to evaluate the performance of these systems. Some of the KPIs provided by YCSB are:

- **Throughput[ops/sec]**: It measures the number of operations (read/write) per second that a system can perform.
- **Latency**: It measures the time taken by a system to respond to an operation (read/write). The latency is usually reported as an average or a percentile value (e.g., 99th percentile).
- **CPU utilization**: It measures the percentage of CPU time used by the system during the benchmark.
- **Memory utilization**: It measures the amount of memory used by the system during the benchmark.

After the end of the benchmark test, the results txt file in timeseries format is created.

```
13:31:31:668 1040 sec: 7664680 operations; 5918.7 current ops/sec; est completion in 5 minutes [READ: Count=11879, Max=2922495, Min=330, Avg=12876.18, 90=25839, 99=60447, 99.9=172415, 99.99=1323007] [INSERT: Count=47281, Max=3092479, Min=305, Avg=10295.08, 90=11023, 99=74879, 99.9=1500159, 99.99=3063807]
13:31:41:668 1050 sec: 7741902 operations; 7722.2 current ops/sec; est completion in 5 minutes [READ: Count=15307, Max=1859583, Min=333, Avg=11976.89, 90=23247, 99=54495, 99.9=179327, 99.99=1321983] [INSERT: Count=61920, Max=1967103, Min=308, Avg=7367.05, 90=10903, 99=79295, 99.9=237311, 99.99=1921023]
13:31:51:669 1060 sec: 7794664 operations; 5276.2 current ops/sec; est completion in 5 minutes [READ: Count=10558, Max=2746367, Min=341, Avg=13512.93, 90=22863, 99=76927, 99.9=742399, 99.99=2412543] [INSERT: Count=42194, Max=2930687, Min=321, Avg=11016.03, 90=11151, 99=122943, 99.9=991231, 99.99=2869247]
13:32:01:668 1070 sec: 7868025 operations; 7336.1 current ops/sec; est completion in 4 minutes [READ: Count=14708, Max=979455, Min=340, Avg=11746.66, 90=20927, 99=41215, 99.9=145791, 99.99=914431] [INSERT: Count=58680, Max=1460223, Min=321, Avg=8516.51, 90=11239, 99=73215, 99.9=571903, 99.99=773119]
13:32:11:668 1080 sec: 7923094 operations; 5506.9 current ops/sec; est completion in 4 minutes [READ: Count=10985, Max=891391, Min=333, Avg=12269.99, 90=22159, 99=50463, 99.9=441599, 99.99=845823] [INSERT: Count=44057, Max=1021951, Min=331, Avg=10755.84, 90=12839, 99=71295, 99.9=986111, 99.99=1013247]
13:32:21:668 1090 sec: 7999816 operations; 7672.2 current ops/sec; est completion in 4 minutes [READ: Count=15167, Max=665599, Min=340, Avg=10811.36, 90=20463, 99=37535, 99.9=86847, 99.99=189823] [INSERT: Count=61582, Max=989183, Min=329, Avg=8206.56, 90=11759, 99=71295, 99.9=553471, 99.99=982015]
13:32:31:668 1100 sec: 8074843 operations; 7502.7 current ops/sec; est completion in 4 minutes [READ: Count=15155, Max=602623, Min=328, Avg=11692.42, 90=22127, 99=44351, 99.9=132479, 99.99=289023] [INSERT: Count=59880, Max=1144831, Min=330, Avg=7745.87, 90=11047, 99=57855, 99.9=488191, 99.99=965631]
13:32:41:668 1110 sec: 8153534 operations; 7869.1 current ops/sec; est completion in 4 minutes [READ: Count=15500, Max=980479, Min=343, Avg=12033.66, 90=22495, 99=49535, 99.9=148223, 99.99=627199] [INSERT: Count=63194, Max=1089535, Min=324, Avg=7178.6, 90=12351, 99=74367, 99.9=150143, 99.99=860159]
13:32:51:668 1120 sec: 8222357 operations; 6882.3 current ops/sec; est completion in 4 minutes [READ: Count=13641, Max=878079, Min=328, Avg=12024.18, 90=22655, 99=51551, 99.9=109951, 99.99=814079] [INSERT: Count=55144, Max=996351, Min=319, Avg=8614.67, 90=11751, 99=86207, 99.9=567807, 99.99=985087]
13:33:01:668 1130 sec: 8294780 operations; 7242.3 current ops/sec; est completion in 3 minutes [READ: Count=14574, Max=891391, Min=339, Avg=12161.64, 90=22959, 99=56927, 99.9=229503, 99.99=600063] [INSERT: Count=57849, Max=1712127, Min=324, Avg=7999.76, 90=12967, 99=64351, 99.9=250367, 99.99=1495039]
13:33:11:668 1140 sec: 8368021 operations; 7324.1 current ops/sec; est completion in 3 minutes [READ: Count=14699, Max=183295, Min=352, Avg=10821.66, 90=21007, 99=38111, 99.9=63391, 99.99=127167] [INSERT: Count=58561, Max=1610751, Min=320, Avg=8213.15, 90=11871, 99=54559, 99.9=618495, 99.99=1473535]
13:33:21:668 1150 sec: 8435153 operations; 6713.2 current ops/sec; est completion in 3 minutes [READ: Count=13331, Max=707583, Min=340, Avg=11439.87, 90=20927, 99=64575, 99.9=178559, 99.99=691711] [INSERT: Count=53801, Max=1379327, Min=326, Avg=8761.67, 90=13919, 99=77311, 99.9=551935, 99.99=1044479]
13:33:31:668 1160 sec: 8509944 operations; 7479.1 current ops/sec; est completion in 3 minutes [READ: Count=15045, Max=876543, Min=336, Avg=12530.66, 90=22735, 99=45375, 99.9=158463, 99.99=825855] [INSERT: Count=59748, Max=1224703, Min=327, Avg=7817.64, 90=12839, 99=72255, 99.9=215039, 99.99=1011199]
13:33:41:668 1170 sec: 8579768 operations; 6982.4 current ops/sec; est completion in 3 minutes [READ: Count=13930, Max=961023, Min=340, Avg=11884.78, 90=22831, 99=51391, 99.9=126911, 99.99=926719] [INSERT: Count=55873, Max=1207295, Min=313, Avg=8387.73, 90=12599, 99=58847, 99.9=525823, 99.99=909311]
```

Figure 8: YCSB Benchmark Results File

8.4 Evaluation and Results

In order to get more reliable results, we perform each workload (workloada, workloadb etc) 3 times and record the mean value.

8.4.1 Test Setup

As discussed in Project Architecture section 4, we use AWS cloud platform to run our benchmark tests. In addition, we deployed the HBase and Cassandra cluster with 3 docker containers running on 3 different machines/nodes EC2 instances in 3 different availability zones in EU (Frankfurt) region. The EC2 instance type used is recognized as t2.2xlarge. Each EC2 instance/Node has 8 vCPU and 32 GB of RAM. The network performance is measured between the nodes is upto 5 GB/s. The used Nodes has up to 3.3 GHz Intel Xeon Scalable processor (Haswell E5-2676 v3 or Broadwell E5-2686 v4).

8.4.2 Workload A: Heavy Update Workload

This workload is to be considered as Update- Heavy workload. There is a mix of 50% Read and 50% Write Operations.

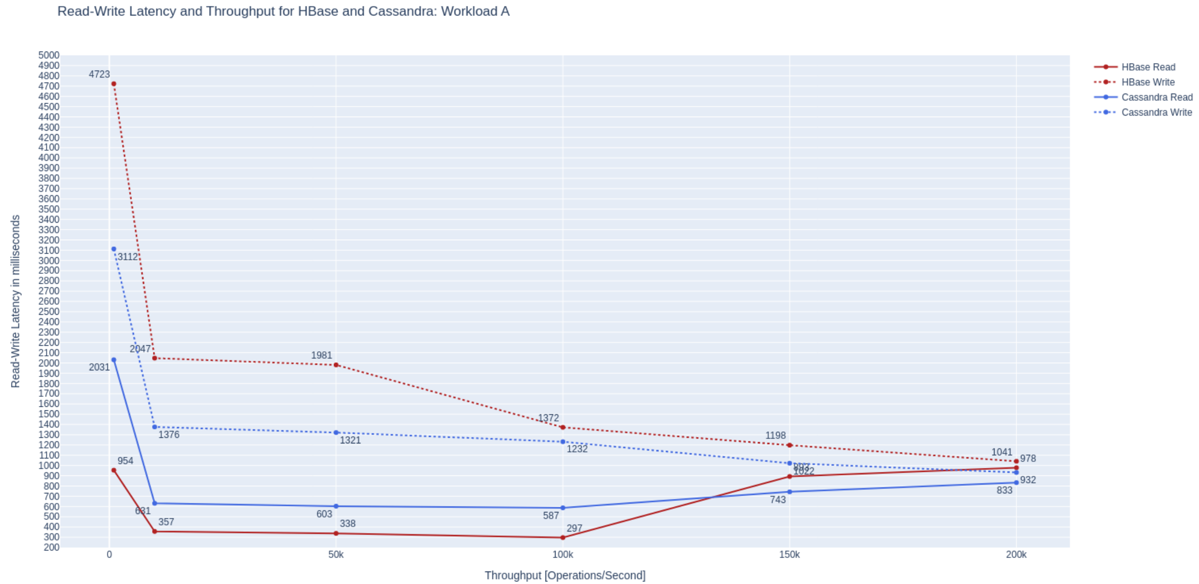

Figure 9: Read-Write Latency and Throughput for HBase and Cassandra: Workload A

Fig. 9 represents the Read and Write latency in milliseconds on Y axis, and Throughput on X axis. From the figures of both the databases HBase and Cassandra, it can be seen that Cassandra performs better than HBase. With 200k Operations, Cassandra read latency lies at 833ms which is lower as compared to HBase read latency at 978ms. On the other hand if we see the write latency, here also Cassandra performs well with 932ms latency at 200K Operations as compared to higher latency of HBase with 1041ms. It can be seen that with less operations per second HBase read latency is lower and better than Cassandra, and when the operations count is greater than Cassandra has a better read and write latency.

Fig. 10 represents Read Operation Mean Latency compared with the Read Counts for HBase and Cassandra. The left Y axis represents the Read Counts and the right Y axis represents the Read Operation Mean Latency. On X axis we have Operations Count. The bar graph illustrates the Read Counts whereas the line graph depicts the latency. As shown in the Fig. 10, we can see that nearly 90% of read counts for workload A has been performed with 200K out of 250K Total read operations, and Cassandra has a less average latency with 833ms whereas HBase lies at 978ms. Here also, we can see with higher operations count Cassandra performs better than HBase.

When we have a count of 200k operations, the average read latency indicates the average time it takes for all 200k read operations to complete. For example, if the average read latency is 10 milliseconds, it means that on average, each read operation takes 10 milliseconds to complete, and the 200k operations would take approximately 2 million milliseconds (200k x 10) or 33.3 minutes to complete.

To conclude, Cassandra outperforms HBase with many write operations.

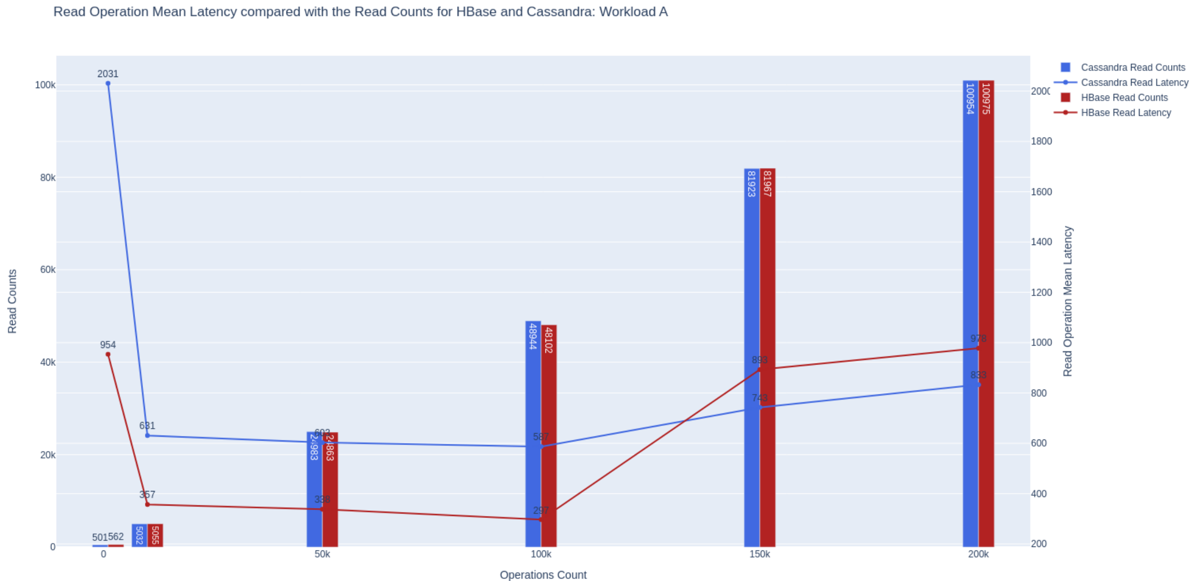

Figure 10: Read Operation Mean Latency compared with the Read Counts for HBase and Cassandra: Workload A

8.4.3 Workload B: Read Mostly Workload

This workload is to be considered as Read Mostly workload. There is a mix of 95% Read and 5% Write Operations.

Fig. 11 represents the Read latency in milliseconds on Y axis, and Throughput on X axis. From the figures of both the databases HBase and Cassandra, it can be seen that HBase performs better than Cassandra. With 200k Operations, Cassandra read latency lies at 598ms which is higher as compared to HBase read latency at 286ms. It can be seen that whether the operations count is lower or higher, HBase performs better than Cassandra.

Fig. 12 represents Read Operation Mean Latency compared with the Read Counts for HBase and Cassandra. The left Y axis represents the Read Counts and the right Y axis represents the Read Operation Mean Latency. On X axis we have Operations Count. The bar graph illustrates the Read Counts whereas the line graph depicts the latency. As shown in the Fig. 12, we can see that nearly 40% of read counts for workload B has been performed with 200K out of 500K Total operations, and HBase has a less average latency with 286ms whereas Cassandra lies at 598ms.

When we have a count of 200k operations, the average read latency indicates the average time it takes for all 200k read operations to complete. For example, if the average read latency is 10 milliseconds, it means that on average, each read operation takes 10 milliseconds to complete, and the 200k operations would take approximately 2 million milliseconds (200k x 10) or 33.3 minutes to complete.

To conclude, HBase outperforms Cassandra with many read operations.

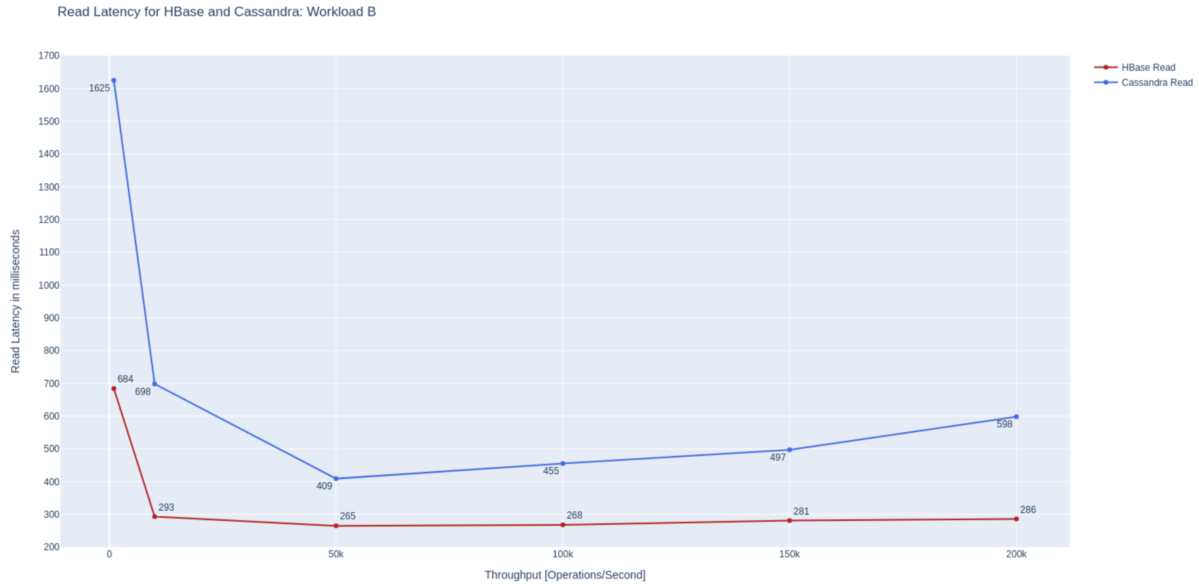

Figure 11: Read-Write Latency and Throughput for HBase and Cassandra: Workload A

9 Conclusion

We use Docker and AWS ECS cluster to deploy HBase and Cassandra cluster to better isolate the resources. The Yahoo! Cloud Serving Benchmark was introduced to enable fair comparisons between different data storage systems. The benchmark includes an adaptable workload generator called the YCSB Client, which can be utilized to load data sets and execute workloads in various data serving systems. HBase and Cassandra are two data storage systems with distinct architectures, and their performance varies depending on read and write operations. Workload A and workload B were used to test the systems, with workload A consisting of 50% read and 50% write operations, and workload B consisting of 95% read operations. The test included a range of operations, from as low as 1000 ops count to as high as 200000 ops count. The results showed that HBase outperformed Cassandra when there were more read operations and fewer write operations. However, when there were more write operations for a higher ops count, Cassandra performed better. HBase had higher throughput and lower runtime for data insertion, making it a better choice for applications with a lower operation count and mostly read operations. On the other hand, Cassandra is preferred for larger operations with mostly write cases.

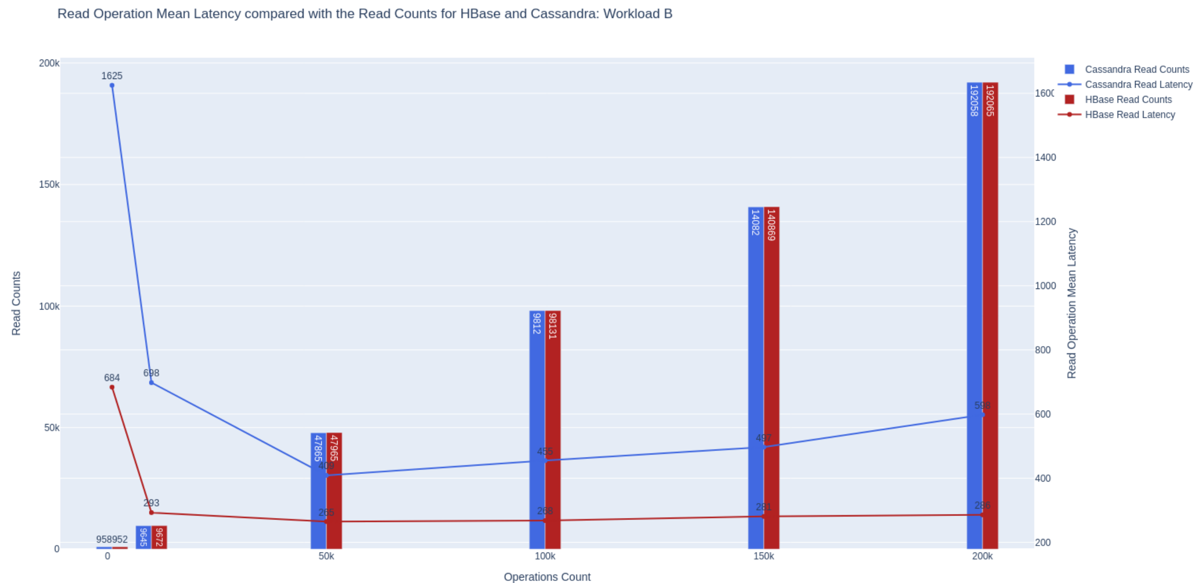

Figure 12: Read Operation Mean Latency compared with the Read Counts for HBase and Cassandra: Workload A

References

[al11] George Porter et al. "HBase: The Definitive Guide". In: *O'Reilly Media, Inc.* (2011).

[Avi10] Prashant Malik Avinash Lakshman. "Cassandra: A Decentralized Structured Storage System". In: *ACM SIGOPS Operating Systems Review* 44.2 (Apr. 2010).

[Cas] Cassandra.org. "Apache Cassandra, https://cassandra.apache.org/". In: *Cassandra.apache.org* ().

[Dat] DataStax. "DataStax, https://www.datastax.com/". In: *datastax.com* ().

[Geo09] Lars George. "HBase Architecture 101 - Storage". In: *larsgeorge* (2009).

[ORe] O'Reilly. ""Cassandra: The Definitive Guide" by Jeff Carpenter and Eben Hewitt. O'Reilly Media, Inc., 2016." In: *oreilly* ().

[Zoo] ZooKeeper. "Welcome to Apache ZooKeeper". In: *ZooKeeper* ().

A Code samples

The project code is available on: https://github.com/Rafay007/HPCSA-Project