CSP554 - Big Data Technologies - Final Project Report

Explore use of NoSQL databases in depth
NoSQL database technologies from Amazon AWS Cloud Platform

Yash Pradeep Gupte (A20472798)
Arvil Dey (A20404230)
Surya Thatee (A20473893)

INTRODUCTION

As the data volumes increased and the data structures became more complex, traditional databases started facing multiple challenges. Today's data has a broader profile: it's big, fast, and changing. The emergence of NoSQL databases occurred, which gave rise to various functionalities like horizontal scaling, flexible schemas, increasing the efficiency of huge datasets, and defining new data models that handle complex queries with faster read/write throughput[1]. We intend to explore four NoSQL databases technologies in depth and compare them based on their performance with large volumes of data. Amazon (AWS) Cloud platform offers a range of cloud services related to NoSQL databases. We will perform experiments specifically on **AWS DocumentDB, AWS Keyspaces,** and **AWS Neptune**. We extensively present major NoSQL databases like **MongoDB, Cassandra,** and **Graph Databases** which are highly used in industries all over the world[1][2].

SEARCH OF LITERATURE

AWS DocumentDB:

MongoDB compatibility is limited in Amazon DocumentDB, a managed proprietary NoSQL JSON document database service. As a document database on Amazon Web Services, Amazon DocumentDB can store, query, and index JSON data. The MongoDB database server is not used by the DocumentDB database. It instead uses Amazon's Aurora backend infrastructure to simulate the MongoDB API. DocumentDB was added to the Amazon Aurora relational database system. The system's architecture divides storage and compute such that each tier can scale independently, despite the fact that it's limited to a single writing master.

Now let's talk about some advantages of DocumentDB. ACID (atomicity, consistency, isolation, and durability) is a set of database transaction qualities designed to ensure data validity in the face of errors, power outages, and other calamities. Amazon DocumentDB now supports ACID transactions across multiple documents, statements, collections, and databases, thanks to the addition of MongoDB 4.0 compatibility. DocumentDB also features Automatic Provisioning and Setup because it is completely Managed, making it simple to get started with Amazon DocumentDB. DocumentDB delivers Amazon CloudWatch measurements for your cloud database instances, as well as monitoring and metrics. Over 40 critical operational metrics for your cluster can be viewed via the AWS Management Console, including CPU, memory, storage, query throughput, MongoDB opcounters, and active connections. Automatic Software Patching is also available in DocumentDB. Because DocumentDB offers a configurable JSON document model, data types, and efficient indexing, it has performance at scale and provides High Throughput and Low Latency for Document Queries. To provide for quick query assessment over huge document sets, the service employs a scale-up, in-memory

optimized design. Database Compute Resources and Automatic Storage may also be scaled simply using DocumentDB.

Now let's talk about some limitations that DocumendDB has. We will be mainly talking about limitations in 2 key areas - security and performance.

Let's first talk about security limitations. RBAC is a set of predefined roles that cannot be changed to meet the needs of a specific application. A user can be given read and write access, for example, but there is no way to limit such privileges so that they can only input new data and never alter or delete it. Users that have read/write access can also delete entire collections and indexes. Another security concern is when encryption at rest is only available at the time of cluster setup. If you want to encrypt data after creating your cluster unencrypted, you must first dump the database and then reload it into a freshly formed cluster. The third security concern would be that DocumentDB only works with the AWS Key Management Service for encryption. Customers do not have the option of bringing their own KMS.

Now let's talk about the performance limitations. Within a transaction, Amazon DocumentDB does not support cursors, and you cannot build new collections or query/update against non-existing collections. Another performance limitation is the one-minute timeout applied to document-level write locks, which is not adjustable by the user. Sessions have a 30-minute timeout and transactions have a one-minute execution limit. A transaction will be aborted if it times out, and any additional instructions sent within the session for the existing transaction will result in an error. The third performance limitation would be that all DocumentDB clusters must be located in the same region. When compared to Atlas, which allows replica sets to span the world and provide low-latency reads to application nodes wherever they are, this is a significant limitation[3].

AWS KeySpaces:

Apache Cassandra is a free open source NoSQL database where it stores records in key value pairs. It is highly robust as it has a masterless replication where it works on the concept of clustering and is horizontally scalable . Querying results come from nodes which can be configured according to specific requirements. Data can be synced to these nodes. Cassandra was created by Facebook which was later on moved to Apache License. Cassandra is a distributed database where you can run multiple nodes. These nodes communicate with a protocol called Gossip, used to transfer information about one another. These nodes may belong to the same data center. Cassandra scales linearly and indefinitely, increasing the performance. Cassandra divides the data into partitions distributed amongst multiple nodes. Partition is done based on the partition keys. This distribution is completely handled by cassandra, we just need to choose the partition key. Every node is assigned a partition token. Replication is also a feature of cassandra where it replicates these partitions amongst the nodes. Cassandra also employs Immediate Consistency meaning you can read immediately what you just wrote.

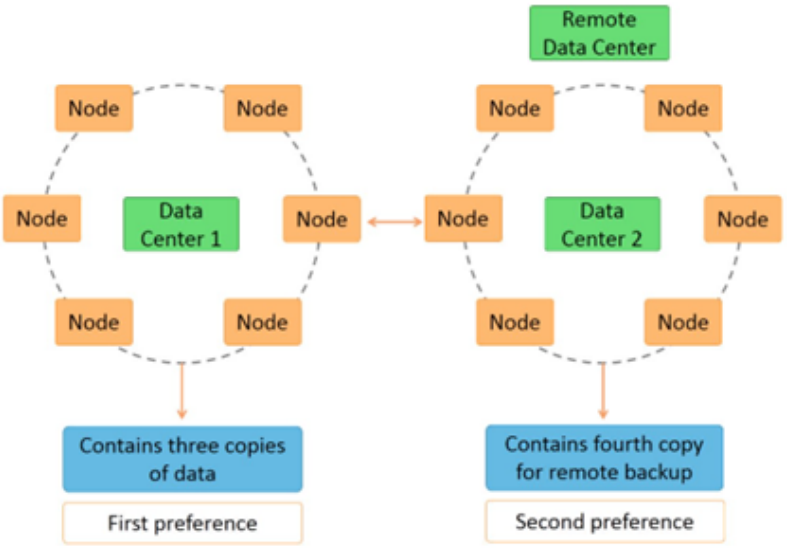


Figure 1. Cassandra Architecture

Amazon AWS Keyspaces is a scalable, highly available, and managed Apache Cassandra - compatible database service. It is serverless, so payment is done for only the resources we use, and the service automatically scales tables up and down in response to application traffic. AWS keyspaces makes it easier to migrate, run and scale Cassandra workloads in the AWS Cloud. There is no need to install any software or deploy any infrastructure to create keyspaces and tables in aws keyspaces.

We can create tables employing Partition keys, Composite keys and Clustering Keys. Amazon Keyspaces can be accessed by Amazon Keyspaces Console, CQLSH and Cassandra Drivers. It supports all commonly used Cassandra data-plane operations, such as creating keyspaces and tables, reading and writing data. The settings such as replication factor and consistency level are configured automatically to provide high availability, durability and single digit millisecond performance. Quorum consistency of Cassandra determines how many nodes will respond when we set the read and write consistency[2].

Let’s have a look at the Functional differences between AWS Keyspaces and Apache Cassandra.
1.Asynchronous creation and deletion of keyspaces and tables: AWS keyspaces performs data definition language (DDL) operations such as creating and deleting keyspaces and tables, asynchronously.
2.Authentication and authorization: AWS Identity and Access Management (IAM) is used by Amazon Keyspaces (for Apache Cassandra) for user authentication and authorisation, and it supports the same authorization policies as Apache Cassandra. As a result, the security configuration commands of Apache Cassandra are not supported by Amazon Keyspaces.
3.Batch: Unlogged batch commands with up to 30 commands are supported by Amazon Keyspaces. In a batch, only unconditional INSERT, UPDATE, or DELETE commands are allowed. Batches that have been logged are not supported.
4.Cluster Configuration: Amazon Keyspaces supports up to 3,000 CQL queries per TCP connection per second, however a driver can initiate an unlimited number of connections.
The majority of open-source Cassandra drivers create a connection pool to Cassandra and load balance queries across that pool. The default action of most drivers is to make a single connection to each peer IP address that

Amazon Keyspaces provides to them. As a result, utilizing the default parameters, a driver's maximum CQL query throughput will be 27,000 CQL queries per second.
5.Empty Strings: Empty strings and blob values are supported by Amazon Keyspaces. Empty strings and blobs, on the other hand, are not supported as clustering column values.
6.Lightweight Transactions:On INSERT, UPDATE, and DELETE operations, which are known in Apache Cassandra as lightweight transactions (LWTs), Amazon Keyspaces (for Apache Cassandra) fully supports comparison and set capability. Amazon Keyspaces (for Apache Cassandra) guarantees constant performance at any size, even for light transactions, as a serverless option. Using lightweight transactions with Amazon Keyspaces has no performance effect.
7.Load balancing: The entries in the `system.peers` table refer to Amazon Keyspaces load balancers. We recommend utilizing a round robin load-balancing scheme and tailoring the amount of connections per IP to your application's demands for the best results.
8.Pagination: The number of rows read to process a request, not the number of rows returned in the result set, is used to paginate Amazon Keyspaces results. As a result, certain pages for filtered queries may have less rows than you set in PAGE SIZE.
9.Prepared statements: Prepared statements can be used for data manipulation language (DML) activities like reading and writing data in Amazon Keyspaces. Prepared statements are not presently supported by Amazon Keyspaces for data definition language (DDL) activities like establishing tables and keyspaces. Prepared statements cannot be used to run DDL procedures.
10.Range Delete: Delete rows in a range is possible with Amazon Keyspaces. Within a partition, a range is a contiguous set of rows. A WHERE clause is used to indicate a range in a DELETE action. You can give a full partition as the range.You can also designate a range to be a subset of contiguous rows within a partition by including the partition key and omitting one or more clustering columns, or by using relational operators (for example, '>', "). You can delete up to 1,000 rows in a range with Amazon Keyspaces in a single action. Range deletions are also atomic, but not isolated.
11.System Table: Amazon Keyspaces populates the Apache 2.0 open source Cassandra drivers' system tables. The system tables that a client can see contain information that is specific to the authenticated user. Amazon Keyspaces has complete control over the system tables, which are read-only[4].
Limitations of Apache Cassandra: Cassandra does not support ACID properties and relational data properties. It does not support aggregates. Latency issues arise as making excessive requests and reading more data slows down transaction time. Cassandra does not support joins or subquery operations, it encourages you to denormalize your data for greater speed, because it's the only way to handle relational queries like complex joins.It is not feasible to group and order by the same column in Cassandra. If we want to conduct an aggregation query, we usually group by a column. Aggregation in Cassandra is done at the partition level, therefore we group by partition key.Because each partition can only be kept in one node machine, if you use the wrong partition keys, you may experience hot spots where one node receives the majority of the load.

AWS Neptune:

Relationships are first class citizens in graph databases, that is Graph databases are designed for depicting *relationships* between data points. Graph databases use nodes to store data entities, and edges to store relationships between entities. Property graphs and RDF graphs are two popular models of Graph Database. Property graphs focus on analytics and querying, these model graphs are used in sectors such as finance, manufacturing, retails, public safety, etc. On the other hand RDF(Resource Description Framework) emphasizes

data integration, they are used by Government statistics agencies, healthcare organizations, pharmaceutical companies, etc[5].

One such high-performance Graph Database engine is Amazon Neptune. Amazon Neptune is a fast, reliable, fully managed graph database service. With Neptune it is easy to build and run applications that work with highly connected datasets. Using Neptune, we can store billions of relationships and with a milliseconds latency we can query the graph. From availability over many AWS regions, Thanks to Amazon Web services! Neptune is also highly available, with read replicas, backup to Amazon S3, etc. Another interesting area is neural networks in Machine Learning, Neural Networks are computing systems with interconnected nodes that work much like neurons in the human brain. With Amazon promoting Neptune ML, graph neural networks(GNNs) come into light. By modeling data as a graph we can model to train Deep Learning algorithms[6].

Amazon Neptune is a fully-managed cloud based high-performance graph database, it uses graph query languages such as Gremlin and SPARQL to query connected data. And it is fully managed by AWS, So Aws handles hardware provisioning, software patching, backups, recovery, setup, or configuration.Amazon Neptune divides data into chunks and is spread across many disks. Now each chunk is replicated six ways across three availability zones. Since Amazon Neptune is vertically scalable instead of being horizontally scalable, to increase availability you have to increase replicas.Amazon Neptune uses ACID transactions (Atomicity, Consistency, Isolation and Durability).One of the drawbacks of Amazon Neptune is that it lacks graph data visualization. But it offers visualization through partners which is an additional cost. It also doesn't support advanced data analytics where you will find difficulty to integrate and move data.

COMPARISON

MongoDB vs Cassandra : The YCSB Benchmark also known as Yahoo Cloud Service Benchmark is employed to evaluate and compare NoSQL databases. It has a client that consists of two parts that is a workload generator and a set of scenarios which are a sequence of read, write and update operations on randomly selected records. Observing the throughput metrics, furthermore after analyzing read/update performance with various workloads, it is able to infer that Cassandra is quicker than MongoDB for update operations, delivering reduced execution time regardless of database size employed in the tests[7].

Cassandra vs Neptune: A transaction in a database refers to any operation which is considered to have a single unit of work whether it is completed fully or does not complete at all. So ACID(Atomicity, Consistency, Isolation, Durability) refers to four properties of a transaction which guarantee that each read, write, or modification of a table in the database have these properties. In this context, the main comparison between Neptune and Cassandra is that Neptune supports ACID transactions but Casandra supports only Atomicity and isolation that too for single operations.

MongoDB vs Neptune : There are many similarities and differences between MongoDB and Neptune. Both Neptune and MongoDB have schema-free data schemes. Their transactions are done using the ACID properties and both support concurrency. The primary database model for Neptune is graph DBMS and RDF store. The primary database model for MongoDB is a document store. MongoDB supports a wider variety of

programming languages compared to Neptune. Neptune has no partitioning methods available, but MongoDB has sharding as its partitioning method[8].

EXPERIMENTS

AWS DocumentDB

Environment setup

*Step 1: Create an AWS Cloud9 environmen*t

I can connect to and query my Amazon DocumentDB cluster using the mongo shell utilizing AWS Cloud9's web-based terminal.

Step 2: Create a security group

From my AWS Cloud9 environment, this security group will allow me to connect to my Amazon DocumentDB cluster.

Step 3: Create an Amazon DocumentDB cluster

Using the security group I formed in the previous phase, I created an Amazon DocumentDB cluster in this step.

Step 4: Install the mongo shell

In my AWS Cloud9 environment that I setup in Step 1, I installed the mongo shell. I use the mongo shell to connect to and query my Amazon DocumentDB cluster from the command line.

Step 5: Connect to my Amazon DocumentDB cluster

I use the mongo shell that I installed in Step 4 to access to my Amazon DocumentDB cluster.

1. Go to the Amazon DocumentDB administration console and look for my cluster under Clusters. By clicking on the cluster identification, you can select the cluster I established.
2. Copy the connection string displayed under Connect to this cluster using the mongo shell in the Connectivity and Security tab.
3. Paste the connection string into my AWS Cloud9 environment.

Importing Data

Using the wget command, download 4 json files from github to the Cloud9 environment. Using the mongoimport query, import data from Cloud9 into AWS DocumentDB. Please consult the appendix for the relevant code.

Data

Json file size & download performance in Cloud9

Wget	**Books**	**Covers**	**city_inspections**	**countries-big**
Length/Size	528,769	493,268	25,463,582	2,497,231

Time (s)	0.06	0.06	0.4	0.08
Speed (MB/S)	11.1 MB/s	10.6 MB/s	68.8.4 MB/s	30.9 MB/s

Number of datasets and import performance in Cloud9

mongoimport	**Books**	**Covers**	**city_inspections**	**countries-big**
Connected timestamp	2021-12-07T 22:08:12.699 +0000	2021-12-08T 00:07:49.775 +0000	2021-12-07T 22:52:54.151 +0000	2021-12-08T 00:03:59.212 +0000
Imported timestamp	2021-12-07T 22:08:12.930 +0000	2021-12-08T 00:07:50.089 +0000	2021-12-07T 22:52:56.913 +0000	2021-12-08T 00:03:59.988 +0000
Imported Data time usage	0.243	0.325	2.811	0.793
Documents number/rows	429	5102	80985	22103

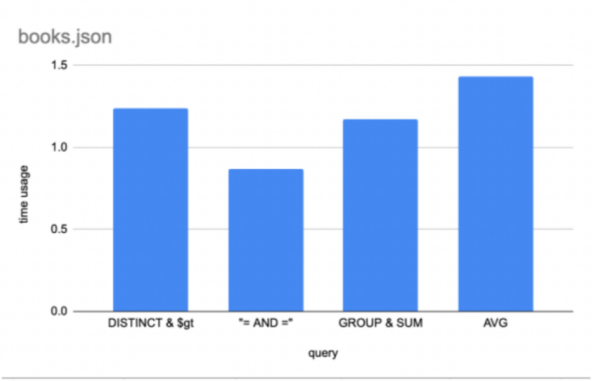
Read Performance

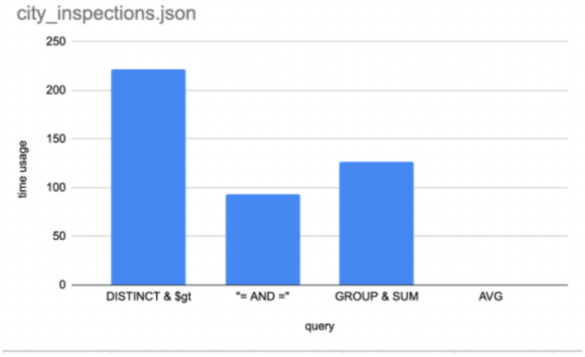
	Books	**Covers**	**city_inspections**	**countries-big**
Length/Size	528,769	493,268	25,463,582	2,497,231
Documents number/rows	429	5102	80985	22103
executionTimeMillis	0.387	1.563	26.224	6.9

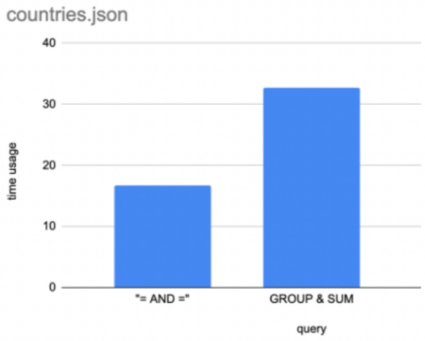
Query Performance

	Books	**Covers**	**city_inspections**	**countries-big**
DISTINCT & $gt	1.237	6.171	221.247	-
"= AND ="	0.869	4.087	93.511	16.686

GROUP & SUM	1.174	36.04	127.059	32.639
GROUP & AVG	1.431	11.202	-	-







Performance Analysis

I chose four different datasets from github, each with its own size, data format, and data structure. As we saw in the previous section, a document database can be used to store user profiles efficiently by storing only the attributes that are unique to each person. Some datasets are modest, such as "covers," which has about 5000 rows, while others are enormous, such as "city inspections," which has 80985 rows. I also imported the "books" dataset, which has just 429 rows. I run a variety of queries in DocumentDB to explore how different data structures and formats effect query performance.

Download efficiency and data size are positively associated, according to my tables and charts. Importing files to Cloud9 is around ten times slower than downloading them. In DocumentDB, read speed differs significantly from import and download performance. The number of rows in a file has a greater impact on read time than the file size. For example, the documents "books" and "covers" are both roughly 500kb in size. Covers, on the other hand, take 5 times longer to read than books. This is due to the fact that "covers" contain more rows than books." Because the column "description" comprises long content, the size of "books" is 5 times larger than "grade," although the reading time is practically the same. As a result, we can deduce that DocumentDB is better suited to data with fewer columns.

For tiny collections with few rows, the time utilization of different searches does not differ significantly. The shortest time is used by the find query, which means it must go over two columns and identify targets that

meet two conditions. The "Group and sum" query takes the most time in the "cover" and "countries" collections. This query returns item numbers from the same group, such as London restaurant numbers. DISTINCT & $gt are used for the longest time in the "city inspections" and "cover" collections. This query is used to locate target items that are greater than a specified number without repeating. We can deduce from this that target numbers have an impact on the performance of a single collection query. The number of documents in a collection has an impact on query performance (rows). The amount of documents (rows) and query efficiency are positively connected.

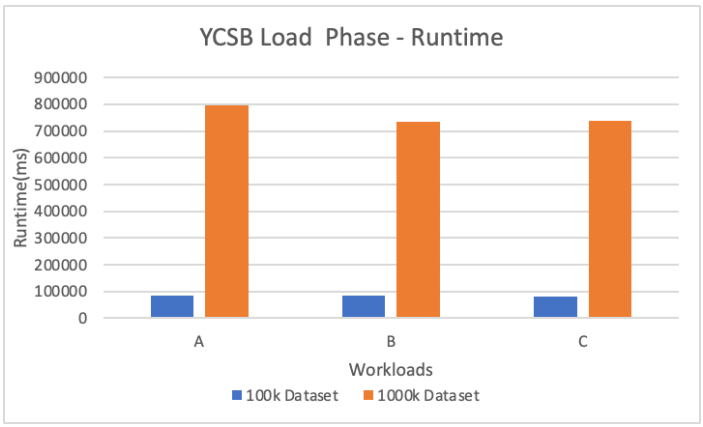
AWS Keyspaces and Apache Cassandra

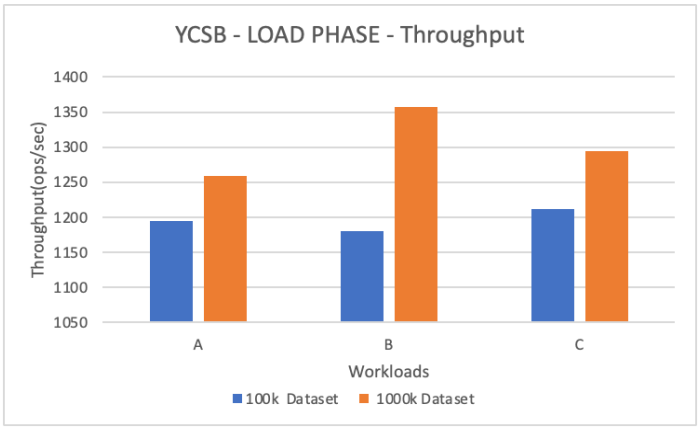
We start working with Apache Cassandra by initializing an EMR cluster with one master and one client server from the AWS EMR utility.Instead of Core Hadoop we will choose the Spark configuration. Once connected to the cluster we install Apache Cassandra 3.11.2 version and query the cqlsh prompt. Our goal here is to explore Apache Cassandra NoSQL database and its migration to AWS Keyspaces.

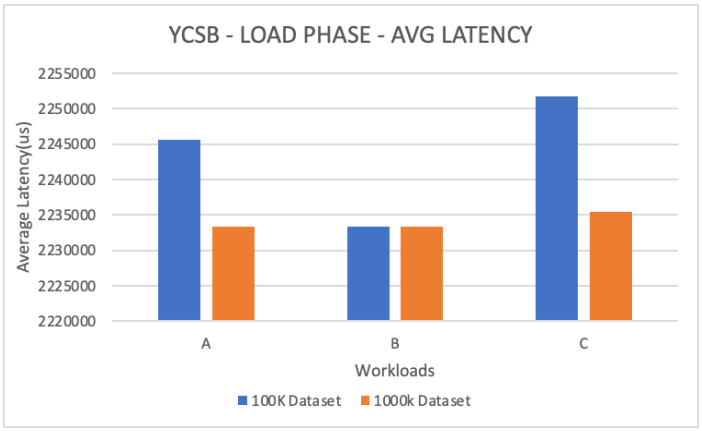
To understand the data modeling and performance optimization, we will be implementing the YCSB - Yahoo! Cloud Serving Benchmark utility for Apache Cassandra. The Yahoo Cloud Serving Benchmark (YCSB) project's purpose was to create a framework and a collection of common workloads for measuring the performance of various "key-value" and "cloud" serving stores [13].A common use of the tool is to benchmark multiple systems and compare them. In the initial experiment, we will generate three workloads A, B, and C against two different datasets containing 100k and 1000k records respectively.
A workload defines the data that will be loaded into the database during the loading phase, and the operations that will be executed against the data set during the transaction phase.
Workload A: Update Heavy Workload - This workload has a mix of 50/50 reads and writes
Workload B: Read mostly workload - This workload has a 95/5 reads/write mix
Workload C : Read Only workload - This workload is 100% read only.
YCSB is installed on the EMR Cluster with specified configurations for Cassandra database technology[14].

Then we start the cassandra server and enter into the cqlsh prompt mode to create a keyspace for yscb with a usertable. The queries for this step are demonstrated in the appendix. The usertable has 11 columns in which the yscb will load the data. We then start with load three types of workloads - Workload A, Workload B and Workload C for 100k records of dataset. The ycsb load command for each type of workload , loads data into our usertable which we just created in cassandra. After loading the dataset, we will run workload transactions on each of these datasets. The commands for load and run are mentioned below in the appendix section. YCSB provides detailed statistics about these operations. Performance measures like Runtime, Throughput (ops/sec) , Operations, Latency(Average, Minimum and Maximum), 95th percentile Latency and 99th percentile Latency. The run command is performed with 10 threads attempting a total of 100 ops/sec. We achieved this by passing thread and target parameters to the run command.
We have considered two types of datasets with 100k and 1000k records. The statistics of which are graphically represented as below.

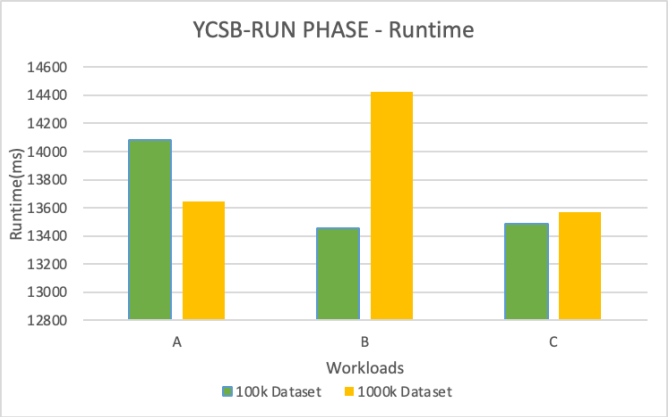
Data Loading statistics :

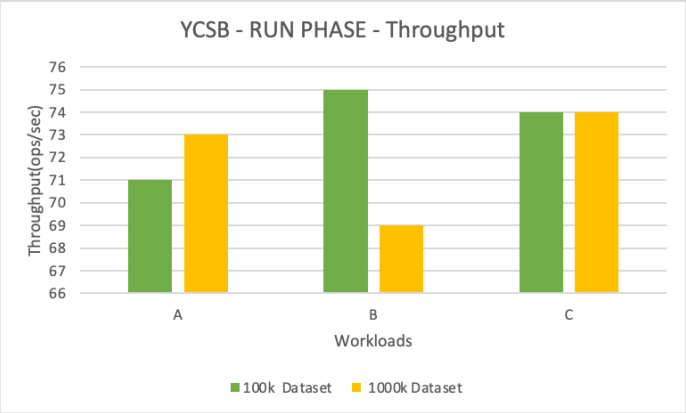


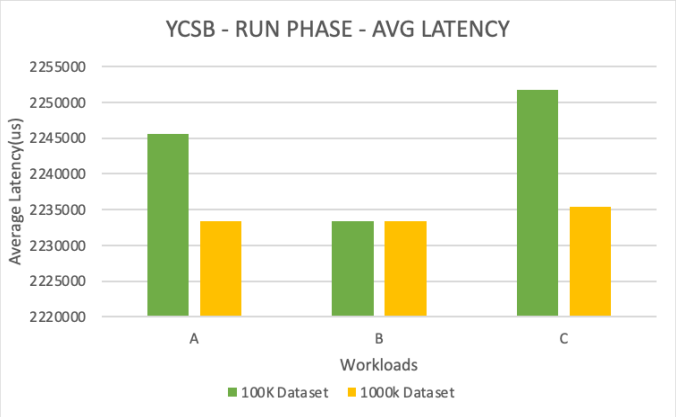




Run Transactions Statistics :







YCSB is an industry standard benchmarking tool to evaluate and compare multiple NoSQL database technologies. The above graph presents a brief overview of how apache cassandra performs over Big Data. The run phase is an important part which shows how different kinds of workloads - update, write and read perform over large datasets [15].

In the following section, we will be looking at the Migration of Apache Cassandra to AWS Keyspaces[16]. The AWS ecosystem provides tools to connect and migrate to AWS Keyspaces. We follow the instructions mentioned by AWS CLI. As in the previous section, we create an EMR cluster with two nodes, master and client. On the master node we install apache cassandra.
On the AWS Management console we have created an IAM user with access to AWS Keyspaces and some administrative privileges. We will use this IAM user to generate our Service specific credentials. Next we will configure our aws cli with these credentials[17]. Detailed steps will be mentioned in the appendix. Once we have our credentials in place, on the EMR cluster we will then download the Starfield digital certificate. Once we get the certificate, we will import the keyspaces credential key to our EMR cluster. Now we import a csv file which contains 10 records , to our EMR cluster using the scp command. Before we start with the migration task, we will need to connect to AWS Keyspaces. For this we need to create a .cassandra directory at the location /home/hadoop inside our EMR cluster. In this directory we will copy the str-root-2.crt (certificate) file and also create a cqlshrc file containing our cql configuration. In this file we set the connection parameters and cql parameters for AWS Keyspaces. The configurations are mentioned in detail below in the appendix. Once we

have done this setup we will start our local cassandra server and from another terminal we will write a command to connect AWS Keyspaces. The following command will be majorly used for connection ,

```
apache-cassandra-3.11.2/bin/cqlsh host 9142 -u ServiceUserName -p
ServicePassword --ssl
```

In my case, the command as follows:

```
apache-cassandra-3.11.2/bin/cqlsh cassandra.us-east-2.amazonaws.com 9142
-u "Keyspaces-cassandra-at-200332471694" -p
"ofTbjsdbdl85bIh8tz3XALYvA8psML6jv3D/fUZqPvg=" --ssl
```

This will make a connection to AWS Keyspaces and here after we can create keyspaces and tables on the AWS ecosystem for cassandra.
Next steps include creating a keyspace and table in AWS Keyspaces. We follow the data modelling technique of cql and write the following query to create a keyspace.

```
CREATE KEYSPACE IF NOT EXISTS "mykeyspace" WITH
REPLICATION={'class':'SimpleStrategy', 'replication_factor': 1};
```

Once we have our keyspace in place, we will create a table and keep the data type similar to the table we want to migrate from apache cassandra to AWS Keyspace. In my example, I have created a small database with 10 rows containing book awards details. We create a table in AWS Keyspaces,

```
CREATE TABLE mykeyspace.mytable_new( award text, year int, category text,
rank int, author text, book_title text, published text, PRIMARY
KEY((award,year), category,rank));
```

A csv file keyspace_sample_table.csv is a file which contains a book awards dataset. We will place this file at the same location where we have installed our cassandra server. For large files with more than 1000 records we can set the throughput capacity and prepare data for the same. We start by randomizing and shuffling the rows in our dataset to spread the writes across the partitions evenly. The following command does the above task for us,

award	year	category	rank	author	book_title	publisher
Kwesi Manu Prize	2020	Fiction	1	Akua Mansa	Where did you go?	SomePublisher
Kwesi Manu Prize	2020	Fiction	2	John Stiles	Yesterday	Example Books
Kwesi Manu Prize	2020	Fiction	3	Nikki Wolf	Moving to the Chateau	AnyPublisher
Wolf	2020	Non-Fiction	1	Wang Xiulan	History of Ideas	Example Books

Wolf	2020	Non-Fiction	2	Ana Carolina Silva	Science Today	SomePublisher
Wolf	2020	Non-Fiction	3	Shirley Rodriguez	The Future of Sea Ice	AnyPublisher
Richard Roe	2020	Fiction	1	Alejandro Rosalez	Long Summer	SomePublisher
Richard Roe	2020	Fiction	2	Arnav Desai	The Key	Example Books
Richard Roe	2020	Fiction	3	Mateo Jackson	Inside the Whale	AnyPublisher

```
tail -n +2 keyspaces_sample_table.csv | shuf -o keyspace.table.csv
(head -1 keyspaces_sample_table.csv && cat keyspace.table.csv ) >
keyspace.table.csv1 && mv keyspace.table.csv1 keyspace.table.csv
```

To analyze the data we run the awk command to determine the maximum and average row size

```
awk -F, 'BEGIN
{samp=10000;max=-1;}{if(NR>1){len=length($0);t+=len;avg=t/NR;max=(len>max
? len : max)}}NR==samp{exit}END{printf("{lines: %d, average: %d bytes,
max: %d bytes}\n",NR,avg,max);}' keyspace.table.csv
```

The output of the above script gives us the average row size to provision the write capacity needed for data upload.

The next important task is to configure the cqlshrc for the cql COPY FROM settings. The cqlsh COPY command lacks internal logic for allocating work evenly among its workers. We can, however, manually configure it to ensure that the work is distributed evenly. Start by analyzing the following key cqlsh parameters:

- DELIMITER - Usually a comma for csv files. If your file has fields separated in another formats you can define something else here.
- INGESTERATE - The maximum number of rows that cqlsh COPY FROM can process per second. If it is not set, it defaults to 100,000.
- NUMPROCESSES - The number of child worker processes generated by cqlsh for COPY FROM tasks. This setting has a maximum value of 16, and the default value is num cores - 1, where num cores is the number of processing cores on the host running cqlsh.
- MAXBATCHSIZE - The batch size specifies how many rows can be inserted into the destination table in a single batch. If this option is not set, cqlsh inserts rows in batches of 20.
- CHUNKSIZE - The size of the work unit that the child worker is assigned to. It is set to 5,000 by default.
- MAXATTEMPTS - The maximum number of times a failed worker chunk can be retried. When the maximum number of attempts is reached, the failed records are written to a new CSV file, which you can run again after investigating the failure.

The configuration file cqlshrc for my task is defined in detail in the appendix section.
Migration is now set to go. We start the cassandra utility and following the prior connection command, we will connect to AWS Keyspaces. We have created a keyspace and a table, which matches the format of keyspace_sample_table.csv file. Now, we use the COPY command to move our data into AWS Keyspaces.

```
COPY mytable_new FROM ‘./keyspace.table.csv’ WITH HEADER=true;
```

The migration is completed using cqlsh and we can perform multiple queries as the data resides on AWS Keyspaces.

I have performed the normal CRUD Operations on AWS Keyspaces(in appendix).

1. CREATE
To insert values into the dataset,

```
INSERT INTO mytable_new( award, year, category, rank, author, book_title, published) VALUES ('Academy 2022,'Sci-Fi','Alex','Into the space','SciFi Publications');
```

2. READ

```
SELECT * FROM mytable_new;

SELECT award,category,author FROM mytable_new WHERE rank=1;
```

3. UPDATE

```
UPDATE mytable_new SET published='Adventure' WHERE category = 'Fiction';
```

4. DELETE

```
DELETE FROM mytable_new WHERE rank = 3;
```

We can open the AWS Keyspaces console and access keyspaces and tables as well. Another task which can be performed in future work is to load huge amounts of data into AWS Keyspaces and perform some benchmarking as we did for apache cassandra in the previous section. This will definitely present a statistically significant comparison between AWS Keyspaces and Apache Cassandra.

AWS Neptune

It's all about connections!. In this world, all the things are linked to other things which eventually form connections/relationships between them. Graphs are one of the best ways to show those connections. By predicting proper correlations among these entities we can use them in practical applications like fraud detection, fake news detection, traffic prediction, predicting group membership in a social network and many more. And by using neural networks we can recognize underlying relationships in the given data.

Neptune being a noSql database has a new feature Neptune ML which uses graph neural network(GNN) technology to build and train machine learning models on graph data.And Neptune ML also supports Knowledge graph Embedding (KGE) models along with User defined models. In the graph database the data is stored as edges and nodes. Where nodes store collection of entities and relations as edges. So Neptune ML can train machine learning models to support five different categories of inference which are: 1)Node classification, 2) Node regression, 3) Edge classification, 4) Edge regression, 5) Link prediction.

Let's talk about Link prediction. To predict whether an edge exists between two nodes we use link prediction, which is an unsupervised machine learning task. By using GNN based models in Neptune ML, link prediction is performed by combining the connectivity and features of the local neighborhood of a node to create a more predictive model. Some of the applications where we can efficiently use link predictions are movie/product recommendations, knowledge graph completion, entity resolution in an identity graph etc.

<u>Process to create GNN models in Neptune ML are :</u>

1) Load Data: Using Gremlin drivers or Neptune Bulk Loader we can load data into the Neptune cluster.

2) Export Data: By specifying the type of ML model and model configuration parameters we make service calls such that the data and model configuration parameters are exported.

3) Model Training: For preprocessing the data , training the ML model and to generate an Amazon SageMaker endpoint that exposes the model we again give service calls.

4)Run Queries: Using the machine learning model we infer data using inference endpoints within our Gremlin queries.

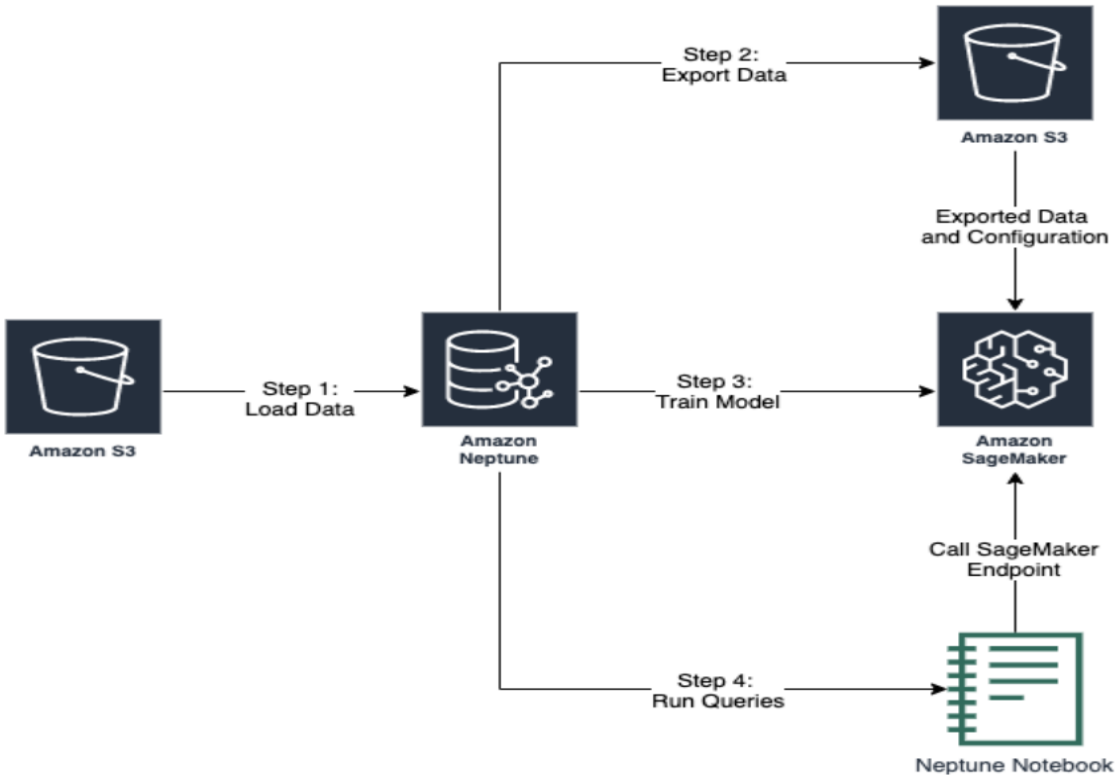


Fig 2: Four step process for creating GNN models

By using MovieLens dataset which contains movies, users and rating of those movies by users, now we predict the movies a user would be most likely to rate as well as which users are most likely to rate a particular selected movie.

Steps involved in loading data, exporting data, configuring data and training data:

- Create a bucket using the AWS Management Console and upload the dataset into the bucket.
- Now using the Neptune console service of AWS, let us create a Neptune DB cluster using AWS cloudformation template.[10]
- Now using Neptune console service of AWS, let us create a Neptune DB cluster using AWS cloudformation template
- Creating an IAM user with permission for Neptune is important for loading the data into Neptune ML.
- Create an Amazon EC2 key pair to use for launching a neptune cluster using AWS Cloudformation
- Add IAM policies to grant permissions needed to use the AWS CloudFormation template
- Setup the Amazon VPC where your Amazon Neptune DB cluster is located.[9]
- Open neptune notebooks by creating one.
- Loading Data:
- First check whether the cluster is configured to Neptune ML or not.
- By using Bulk loader or Gremlin drivers get data into AWS neptune Db. Need to provide S3 bucket URI.
- Now download data and format data such that it is compatible with Neptune.
- The dataset consists of data of 1682 movies, 19 genres, 100k ratings and 943 users.[11]

- For proper validation to build the model we remove vertices such that later the trained model can predict these missing vertices.
- Now before exporting data we need to configure Neptune Export Service which specifies the type of ML model to build.
- Configuring features is an important step where we specify details about types of data being used and how the model should interpret the data. All the properties are considered as features by the model.Here the Neptune ML considers numerical and category features. So some properties in the data does not come under these feature types thus we need configuration.
- By using standard feature preprocessing techniques the data is prepared for Deep Graph library(DGL) use. Thus, the data will be ready for model training. Here we use SageMaker for data processing
- By continuing with SageMaker processing we generate a model training strategy. Which includes configurations for type of model and model hyperparameter that will be used for model training.
- The final step is to create an inference endpoint which is an amazon sagemaker endpoint instance.
- Now with the trained model, we perform respective predictions such as what movies a user will rate, the top users most likely to rate a movie etc.

CONCLUSION

AWS Keyspaces provides a serverless service to host apache cassandra workloads and perform data modeling. Apache cassandra is a key value store NoSQL database which handles data efficiently and distributes it into partitions and clusters which provide better consistency. From our experiments we performed on apache cassandra demonstrated how big data is handled , different kinds of workloads and their transactions. The YCSB benchmark is an industry standard for analyzing and comparing NoSQL database technologies , which we have utilized to evaluate the performance of cassandra on various workloads and operations. It provides detailed statistics which help us to understand the mechanism behind these technologies. Furthermore, we also carry out experiments to present a smooth migration from Apache Cassandra to AWS Keyspaces. When we have an entire keyspace or cassandra environment set up , we can establish a connection to AWS Keyspaces and configure accordingly for migration achieved by cqlsh utility. The future work for this specific part of the project would be to benchmark performance on AWS Keyspaces and compare it with Apache Cassandra. Benchmarking tools guide an application developer to decide the type of environment suitable for the kind of data and its modeling indeed.

Working with graph databases in Neptune ML using Graph Neural networks is solving key problems of link predictions for network-structured data.With the ability to infer unknown connections within graph it allows us to work on new and interesting use cases within Amazon Neptune.Considering that working with highly connected datasets is a complex problem due to the hidden connections which link to numerous networks linked to one another it would provide us with ample number of opportunities to deep dig into this field. With Neptune ML Toolkit we are enabled to work on our custom models to obtain predictions through graph queries which is a plus.

Businesses are bombarded with massive amounts of data from a variety of sources at breakneck speeds every day. For processing and evaluating massive amounts of unstructured and dynamic data, relational databases are inefficient. When it comes to storing, accessing, and processing huge amounts of data,

incorporating NoSQL into a database management strategy helps business agility and flexibility. Regardless of the preset schema architecture, NoSQL quickly handles large data files and sets, resulting in better performance, scalability, and real-time availability. We have performed various implementations on the selected database technologies which provided us with better understanding Big Data and how it is handled using the NoSQL technologies. AWS DocumentDB, AWS Keyspaces and AWS Neptune are crucial NoSQL technologies which are implemented in different ways to server users with top notch applications with brilliant efficiencies

CONTRIBUTIONS

Overview, Comparison and Conclusion: Yash Pradeep Gupte , Arvil Dey , Surya Thatee

Search of Literature, Experiments and Appendix:

AWS DocumentDB: Arvil Dey

AWS KeySpaces: Yash Pradeep Gupte

AWS Neptune: Surya Thatee

REFERENCES

[1] Miles Ward, NoSQL Database in the Cloud: MongoDB on AWS, March 2013, Available at https://docs.huihoo.com/amazon/aws/whitepapers/NoSQL-Database-in-the-Cloud-MongoDB-on-AWS-March-2013.pdf

[2] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 2 (April 2010), 35–40. DOI:https://doi.org/10.1145/1773912.1773922

[3] What Is Amazon DocumentDB (with MongoDB Compatibility). Available at: https://docs.aws.amazon.com/documentdb/latest/developerguide/what-is.html

[4] Functional differences: Amazon Keyspaces vs. Apache Cassandra. Available at : https://docs.aws.amazon.com/keyspaces/latest/devguide/functional-differences.html

[5] R. kumar Kaliyar, "Graph databases: A survey," International Conference on Computing, Communication & Automation, 2015, pp. 785-790, doi: 10.1109/CCAA.2015.7148480

[6]What Is Amazon Neptune?. Available at : https://docs.aws.amazon.com/neptune/latest/userguide/intro.html

[7]Veronika Abramova and Jorge Bernardino. 2013. NoSQL databases: MongoDB vs cassandra. In Proceedings of the International C* Conference on Computer Science and Software Engineering (C3S2E '13). Association for Computing Machinery, New York, NY, USA, 14–22. DOI:https://doi.org/10.1145/2494444.2494447

[8] System Properties Comparison Amazon Neptune vs. MongoDB. Available at: https://db-engines.com/en/system/Amazon+Neptune%3BMongoDB

[9]Set up Amazon VPC : https://docs.aws.amazon.com/neptune/latest/userguide/get-started-vpc.html

[10]To create Neptune DB cluster: https://docs.aws.amazon.com/neptune/latest/userguide/get-started-create-cluster.html

[11] Dataset for MovieLens 100k: https://grouplens.org/datasets/movielens/100k/

[12] GitHub Link to Get started with Neptune ML: https://github.com/aws/graph-notebook/tree/main/src/graph_notebook/notebooks/04-Machine-Learning

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10). Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[14] Apache Cassandra 2.x CQL binding. Available at: https://github.com/brianfrankcooper/YCSB/tree/master/cassandra
[15] YCSB . Available at: https://github.com/brianfrankcooper/YCSB
[16] Migrating to Amazon Keyspaces. Available at: https://docs.aws.amazon.com/keyspaces/latest/devguide/migrating.html
[17] Accessing Amazon Keyspaces (for Apache Cassandra). Available at: https://docs.aws.amazon.com/keyspaces/latest/devguide/accessing.html

APPENDIX

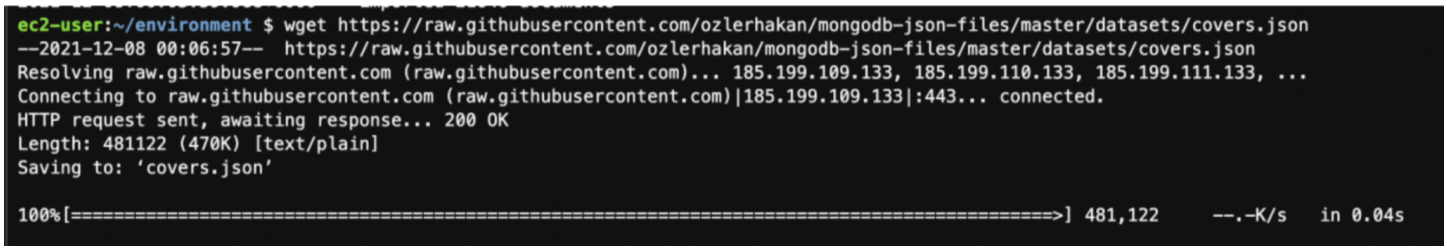
AWS DocumentDB

Download 4 json files from github to Cloud9 environment, using wget command:

Wget
https://raw.githubusercontent.com/ozlerhakan/mongodb-json-files/master/datasets/countries-big.json

Import data into AWS DocumentDB from Cloud9, using mongoimport query:

```
mongoimport --ssl \
--host="docdb-2021-12-07-05-53-21.cluster-ckfaszad0ezr.us-east-2.docdb.amazonaws.com:27017" \
--collection=countries \
--db=CS554project \
--file=countries-big.json \
--numInsertionWorkers 4 \
--username=ychen319 \
--password=Hide980502 \
--sslCAFile rds-combined-ca-bundle.pem
```



Read Query:
db.countries_big.find().explain("executionStats")

Aggregate AVG Query Sample:
db.covers.explain("executionStats").aggregate([{$group : {_id : "$book_id", num_r : {$avg : "$ratingval"}}}])

AWS Keyspaces and Apache Cassandra

YCSB

For YCSB benchmarking and implementation, we start by installing Cassnadra and YCSB on the EMR cluster following the command below,

Install Cassandra :

```
wget
https://archive.apache.org/dist/cassandra/3.11.2/apache-cassandra-3.11.2-bin.tar.gz
tar -xzvf apache-cassandra-3.11.2-bin.tar.gz
```

Install YCSB:

```
curl -O --location
https://github.com/brianfrankcooper/YCSB/releases/download/0.17.0/ycsb-0.17.0.tar.gz
tar xfvz ycsb-0.17.0.tar.gz
cd ycsb-0.17.0
```

Continuing with creating a keyspace and a table in cassandra, we first start cassandra

```
apache-cassandra-3.11.2/bin/cassandra &
```

On another terminal start the CQLSH query interpreter,

```
apache-cassandra-3.11.2/bin/cqlsh
```

Creating a keyspace ycsb and a usertable,

```
CREATE KEYSPACE ycsb WITH REPLICATION =
{'class':'SimpleStrategy','replication_factor': 3 };

USE ycsb;

CREATE TABLE ycsb.usertable(y_id VARCHAR, field0 VARCHAR,field1 VARCHAR,
field2 VARCHAR, field3 VARCHAR, field4 VARCHAR, field5 VARCHAR, field6
VARCHAR, field7 VARCHAR, field8 VARCHAR, field9 VARCHAR, PRIMARY
KEY(y_id));
```

Now go to your home directory and start loading data into our table. We create a dataset file - large.data and specify the record count parameter for ycsb inside that. The first dataset we will be benchmarking is the 100k records dataset.

large.dat - 100 k records

```
recordcount=100000
~
~
~
~
~
~
~
```

WORKLOAD A

1. LOAD 100K RECORDS IN CASSANDRA CQL - WORLOAD A

```
./bin/ycsb load cassandra-cql -p hosts="127.0.0.1" -P workloads/workloada
-P ../large.dat -s > load.dat
```

```
[hadoop@ip-172-31-9-214 ycsb-0.17.0]$ ./bin/ycsb load cassandra-cql -p hosts="127.0.0.1" -P workloads/workloada -P ../large.dat -s > load.dat
/etc/alternatives/jre/bin/java -cp /home/hadoop/ycsb-0.17.0/cassandra-binding/conf:/home/hadoop/ycsb-0.17.0/conf:/home/hadoop/ycsb-0.17.0/lib/core-0.17.0.jar:
/home/hadoop/ycsb-0.17.0/lib/htrace-core4-4.1.0-incubating.jar:/home/hadoop/ycsb-0.17.0/lib/jackson-mapper-asl-1.9.4.jar:/home/hadoop/ycsb-0.17.0/lib/jackson-
core-asl-1.9.4.jar:/home/hadoop/ycsb-0.17.0/lib/HdrHistogram-2.1.4.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-common-4.0.33.Final.jar:/home/hadoo
p/ycsb-0.17.0/cassandra-binding/lib/netty-transport-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/slf4j-api-1.7.25.jar:/home/hadoop/ycsb-0.
17.0/cassandra-binding/lib/netty-codec-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-buffer-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/
cassandra-binding/lib/cassandra-driver-core-3.0.0.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-handler-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/c
assandra-binding/lib/metrics-core-3.1.2.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/guava-16.0.1.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/cas
sandra-binding-0.17.0.jar site.ycsb.Client -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloada -P ../large.dat -s -load
Command line: -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloada -P ../large.dat -s -load
YCSB Client 0.17.0

Loading workload...
Starting test.
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
2022-05-01 16:40:44:983 0 sec: 0 operations; est completion in 0 second
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
2022-05-01 16:40:54:953 10 sec: 7261 operations; 726.03 current ops/sec; est completion in 2 minutes [INSERT: Count=7261, Max=123391, Min=406, Avg=1194.3, 90=
1677, 99=5579, 99.9=12951, 99.99=37375]
2022-05-01 16:41:04:953 20 sec: 17941 operations; 1068 current ops/sec; est completion in 1 minute [INSERT: Count=10681, Max=174079, Min=343, Avg=922.08, 90=1
277, 99=4287, 99.9=8455, 99.99=30095]
2022-05-01 16:41:14:953 30 sec: 30025 operations; 1208.4 current ops/sec; est completion in 1 minute [INSERT: Count=12083, Max=169471, Min=308, Avg=819.43, 90
=1045, 99=3591, 99.9=7739, 99.99=14759]
2022-05-01 16:41:24:953 40 sec: 45203 operations; 1517.8 current ops/sec; est completion in 49 seconds [INSERT: Count=15178, Max=106495, Min=298, Avg=641.64,
90=751, 99=2379, 99.9=6447, 99.99=10359]
2022-05-01 16:41:34:953 50 sec: 59294 operations; 1409.1 current ops/sec; est completion in 35 seconds [INSERT: Count=14091, Max=197375, Min=300, Avg=713.42,
90=872, 99=2761, 99.9=7039, 99.99=113087]
2022-05-01 16:41:44:955 60 sec: 70934 operations; 1163.88 current ops/sec; est completion in 25 seconds [INSERT: Count=11641, Max=148351, Min=332, Avg=853.77,
 90=1098, 99=3873, 99.9=8215, 99.99=18383]
2022-05-01 16:41:54:954 70 sec: 84775 operations; 1384.1 current ops/sec; est completion in 13 seconds [INSERT: Count=13841, Max=168959, Min=368, Avg=717.78,
90=755, 99=1775, 99.9=4463, 99.99=8743]
2022-05-01 16:42:04:953 80 sec: 98322 operations; 1354.84 current ops/sec; est completion in 2 second [INSERT: Count=13546, Max=158079, Min=365, Avg=733.14, 9
0=803, 99=2185, 99.9=5671, 99.99=8695]
2022-05-01 16:42:08:661 83 sec: 100000 operations; 452.54 current ops/sec; [CLEANUP: Count=1, Max=2246655, Min=2244608, Avg=2245632, 90=2246655, 99=2246655, 9
9.9=2246655, 99.99=2246655] [INSERT: Count=1678, Max=7359, Min=333, Avg=864.69, 90=1122, 99=3821, 99.9=5647, 99.99=7359]
```

Saving the output to a load.dat file which looks as follows,

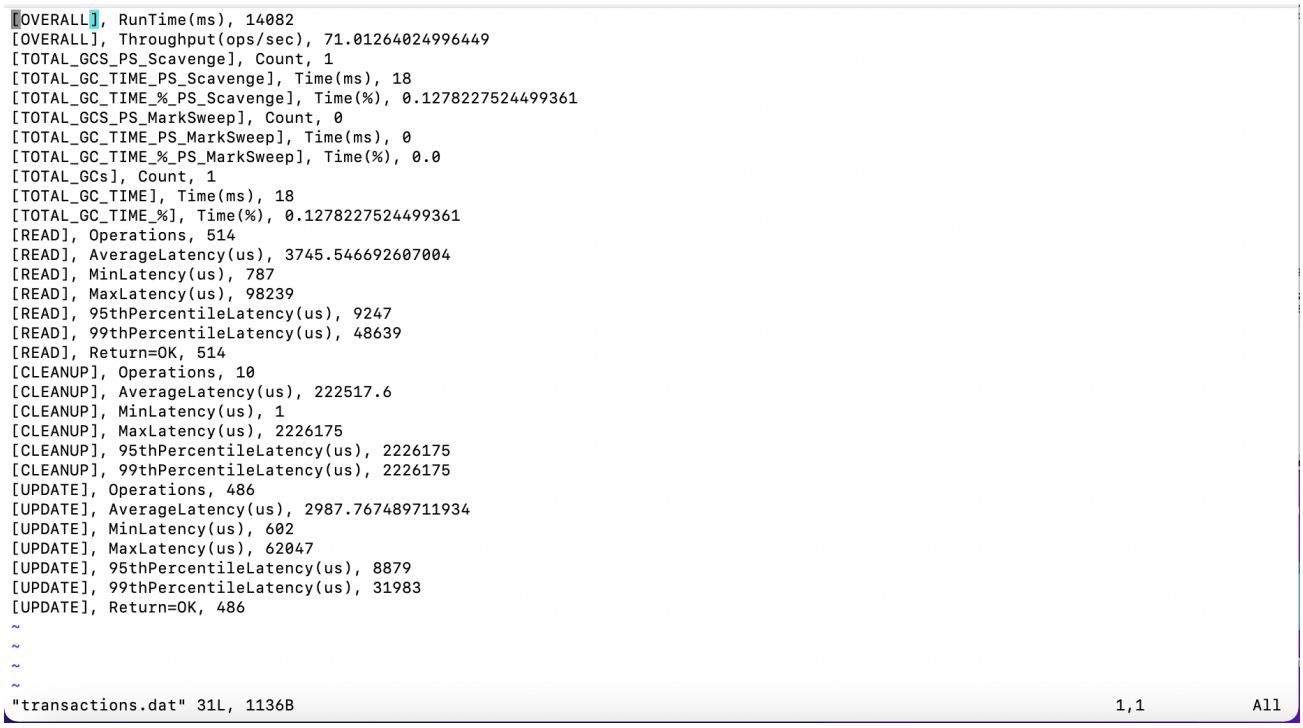
```
[OVERALL], RunTime(ms), 83711
[OVERALL], Throughput(ops/sec), 1194.5861356333098
[TOTAL_GCS_PS_Scavenge], Count, 51
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 247
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.29506277550142757
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 51
[TOTAL_GC_TIME], Time(ms), 247
[TOTAL_GC_TIME_%], Time(%), 0.29506277550142757
[CLEANUP], Operations, 1
[CLEANUP], AverageLatency(us), 2245632.0
[CLEANUP], MinLatency(us), 2244608
[CLEANUP], MaxLatency(us), 2246655
[CLEANUP], 95thPercentileLatency(us), 2246655
[CLEANUP], 99thPercentileLatency(us), 2246655
[INSERT], Operations, 100000
[INSERT], AverageLatency(us), 794.6882
[INSERT], MinLatency(us), 298
[INSERT], MaxLatency(us), 197375
[INSERT], 95thPercentileLatency(us), 1380
[INSERT], 99thPercentileLatency(us), 3351
[INSERT], Return=OK, 100000
~
~
~
~
~
~
~
~
~
~
~
"load.dat" 24L, 914B                                                      1,1          All
```

2. RUN 100K RECORDS FOR WORKLOAD A - 10 THREADS, 100 OPERATIONS PER SEC - 10 operations/sec per thread

```
./bin/ycsb run cassandra-cql -p hosts="127.0.0.1" -P workloads/workloada
-P ../large.dat -s -threads 10 -target 100 > transactions.dat
```

```
[hadoop@ip-172-31-9-214 ycsb-0.17.0]$ ./bin/ycsb run cassandra-cql -p hosts="127.0.0.1" -P workloads/workloada -P ../large.dat -s -threads 10 -target 100 > tr
ansactions.dat
/etc/alternatives/jre/bin/java -cp /home/hadoop/ycsb-0.17.0/cassandra-binding/conf:/home/hadoop/ycsb-0.17.0/conf:/home/hadoop/ycsb-0.17.0/lib/core-0.17.0.jar:
/home/hadoop/ycsb-0.17.0/lib/htrace-core4-4.1.0-incubating.jar:/home/hadoop/ycsb-0.17.0/lib/jackson-mapper-asl-1.9.4.jar:/home/hadoop/ycsb-0.17.0/lib/jackson-
core-asl-1.9.4.jar:/home/hadoop/ycsb-0.17.0/lib/HdrHistogram-2.1.4.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-common-4.0.33.Final.jar:/home/hado
op/ycsb-0.17.0/cassandra-binding/lib/netty-transport-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/slf4j-api-1.7.25.jar:/home/hadoop/ycsb-0.
17.0/cassandra-binding/lib/netty-codec-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-buffer-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/
cassandra-binding/lib/cassandra-driver-core-3.0.0.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-handler-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/c
assandra-binding/lib/metrics-core-3.1.2.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/guava-16.0.1.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/cas
sandra-binding-0.17.0.jar site.ycsb.Client -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloada -P ../large.dat -s -threads 10 -targe
t 100 -t
Command line: -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloada -P ../large.dat -s -threads 10 -target 100 -t
YCSB Client 0.17.0

Loading workload...
Starting test.
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
2022-05-01 16:47:29:953 0 sec: 0 operations; est completion in 0 second
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
2022-05-01 16:47:39:898 10 sec: 825 operations; 82.5 current ops/sec; est completion in 3 second [READ: Count=416, Max=62495, Min=787, Avg=3410.31, 90=6627, 9
9=17791, 99.9=62495, 99.99=62495] [UPDATE: Count=410, Max=59231, Min=622, Avg=2741.7, 90=5179, 99=25647, 99.9=59231, 99.99=59231]
2022-05-01 16:47:43:980 14 sec: 1000 operations; 42.87 current ops/sec; [READ: Count=98, Max=98239, Min=797, Avg=5168.57, 90=5399, 99=96383, 99.9=98239, 99.99
=98239] [CLEANUP: Count=10, Max=2226175, Min=1, Avg=222517.6, 90=8, 99=2226175, 99.9=2226175, 99.99=2226175] [UPDATE: Count=76, Max=62047, Min=602, Avg=4315.2
1, 90=5791, 99=40671, 99.9=62047, 99.99=62047]
```

Saving the output to a transactions.dat file which looks as follows,



WORKLOAD B

1. LOAD 100K RECORDS IN CASSANDRA CQL - WORLOAD B

```
./bin/ycsb load cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadb
-P ../large.dat -s > loadb.dat
```

```
[hadoop@ip-172-31-9-214 ycsb-0.17.0]$ ./bin/ycsb load cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadb -P ../large.dat -s > loadb
.dat
/etc/alternatives/jre/bin/java -cp /home/hadoop/ycsb-0.17.0/cassandra-binding/conf:/home/hadoop/ycsb-0.17.0/conf:/home/hadoop/ycsb-0.17.0/
lib/core-0.17.0.jar:/home/hadoop/ycsb-0.17.0/lib/htrace-core4-4.1.0-incubating.jar:/home/hadoop/ycsb-0.17.0/lib/jackson-mapper-asl-1.9.4.j
ar:/home/hadoop/ycsb-0.17.0/lib/jackson-core-asl-1.9.4.jar:/home/hadoop/ycsb-0.17.0/lib/HdrHistogram-2.1.4.jar:/home/hadoop/ycsb-0.17.0/ca
ssandra-binding/lib/netty-common-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-transport-4.0.33.Final.jar:/home/ha
doop/ycsb-0.17.0/cassandra-binding/lib/slf4j-api-1.7.25.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-codec-4.0.33.Final.jar:/h
ome/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-buffer-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/cassandra-driver
-core-3.0.0.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-handler-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/l
ib/metrics-core-3.1.2.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/guava-16.0.1.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/c
assandra-binding-0.17.0.jar site.ycsb.Client -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloadb -P ../large.dat
 -s -load
Command line: -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloadb -P ../large.dat -s -load
YCSB Client 0.17.0

Loading workload...
Starting test.
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
2022-05-01 17:05:38:331 0 sec: 0 operations; est completion in 0 second
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
2022-05-01 17:05:48:307 10 sec: 10033 operations; 1003.3 current ops/sec; est completion in 1 minute [INSERT: Count=10033, Max=46111, Min=
325, Avg=847.84, 90=1067, 99=3823, 99.9=10223, 99.99=43615]
2022-05-01 17:05:58:296 20 sec: 21153 operations; 1112 current ops/sec; est completion in 1 minute [INSERT: Count=11120, Max=99967, Min=31
7, Avg=888.13, 90=1182, 99=4807, 99.9=9527, 99.99=15255]
2022-05-01 17:06:08:297 30 sec: 32985 operations; 1183.2 current ops/sec; est completion in 1 minute [INSERT: Count=11832, Max=120255, Min
=342, Avg=837.03, 90=1057, 99=3573, 99.9=6483, 99.99=14887]
2022-05-01 17:06:18:296 40 sec: 49739 operations; 1675.4 current ops/sec; est completion in 41 seconds [INSERT: Count=16754, Max=105151, M
in=304, Avg=589.77, 90=724, 99=1187, 99.9=5215, 99.99=14407]
2022-05-01 17:06:28:296 50 sec: 61506 operations; 1176.7 current ops/sec; est completion in 32 seconds [INSERT: Count=11768, Max=143103, M
in=334, Avg=842.52, 90=1022, 99=4211, 99.9=8407, 99.99=129279]
2022-05-01 17:06:38:296 60 sec: 73355 operations; 1184.9 current ops/sec; est completion in 22 seconds [INSERT: Count=11848, Max=17663, Mi
n=338, Avg=826.84, 90=1046, 99=3491, 99.9=6943, 99.99=10167]
2022-05-01 17:06:48:296 70 sec: 86419 operations; 1306.4 current ops/sec; est completion in 12 seconds [INSERT: Count=13064, Max=148991, M
in=314, Avg=769.58, 90=937, 99=3301, 99.9=7983, 99.99=75519]
2022-05-01 17:06:58:296 80 sec: 96938 operations; 1051.9 current ops/sec; est completion in 3 second [INSERT: Count=10519, Max=97599, Min=
331, Avg=944.42, 90=1022, 99=6215, 99.9=12359, 99.99=39903]
2022-05-01 17:07:02:998 84 sec: 100000 operations; 651.21 current ops/sec; [CLEANUP: Count=1, Max=2234367, Min=2232320, Avg=2233344, 90=22
34367, 99=2234367, 99.9=2234367, 99.99=2234367] [INSERT: Count=3062, Max=50495, Min=352, Avg=800.23, 90=961, 99=3155, 99.9=7707, 99.99=504
95]
[hadoop@ip-172-31-9-214 ycsb-0.17.0]$
```

Saving the output to a loadb.dat file which looks as follows,

```
[OVERALL], RunTime(ms), 84702
[OVERALL], Throughput(ops/sec), 1180.609666831952
[TOTAL_GCS_PS_Scavenge], Count, 58
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 248
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.29279119737432413
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 58
[TOTAL_GC_TIME], Time(ms), 248
[TOTAL_GC_TIME_%], Time(%), 0.29279119737432413
[CLEANUP], Operations, 1
[CLEANUP], AverageLatency(us), 2233344.0
[CLEANUP], MinLatency(us), 2232320
[CLEANUP], MaxLatency(us), 2234367
[CLEANUP], 95thPercentileLatency(us), 2234367
[CLEANUP], 99thPercentileLatency(us), 2234367
[INSERT], Operations, 100000
[INSERT], AverageLatency(us), 803.16814
[INSERT], MinLatency(us), 304
[INSERT], MaxLatency(us), 148991
[INSERT], 95thPercentileLatency(us), 1372
[INSERT], 99thPercentileLatency(us), 4001
[INSERT], Return=OK, 100000
~
~
~
```

2. RUN 100K RECORDS FOR WORKLOAD B - 10 THREADS, 100 OPERATIONS PER SEC - 10 operations/sec per thread

```
./bin/ycsb run cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadb
-P ../large.dat -s -threads 10 -target 100 > transactions_b.dat
```

```
[hadoop@ip-172-31-9-214 ycsb-0.17.0]$ ./bin/ycsb run cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadb -P ../large.dat -s -threads
 10 -target 100 > transactions_b.dat
/etc/alternatives/jre/bin/java -cp /home/hadoop/ycsb-0.17.0/cassandra-binding/conf:/home/hadoop/ycsb-0.17.0/conf:/home/hadoop/ycsb-0.17.0/
lib/core-0.17.0.jar:/home/hadoop/ycsb-0.17.0/lib/htrace-core4-4.1.0-incubating.jar:/home/hadoop/ycsb-0.17.0/lib/jackson-mapper-asl-1.9.4.j
ar:/home/hadoop/ycsb-0.17.0/lib/jackson-core-asl-1.9.4.jar:/home/hadoop/ycsb-0.17.0/lib/HdrHistogram-2.1.4.jar:/home/hadoop/ycsb-0.17.0/ca
ssandra-binding/lib/netty-common-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-transport-4.0.33.Final.jar:/home/ha
doop/ycsb-0.17.0/cassandra-binding/lib/slf4j-api-1.7.25.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-codec-4.0.33.Final.jar:/h
ome/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-buffer-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/cassandra-driver
-core-3.0.0.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-handler-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/l
ib/metrics-core-3.1.2.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/guava-16.0.1.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/c
assandra-binding-0.17.0.jar site.ycsb.Client -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloadb -P ../large.dat
 -s -threads 10 -target 100 -t
Command line: -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloadb -P ../large.dat -s -threads 10 -target 100 -t
YCSB Client 0.17.0

Loading workload...
Starting test.
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
2022-05-01 17:12:06:236 0 sec: 0 operations; est completion in 0 second
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
2022-05-01 17:12:16:171 10 sec: 889 operations; 88.89 current ops/sec; est completion in 2 second [READ: Count=840, Max=60959, Min=660, Av
g=2387.67, 90=3973, 99=16751, 99.9=49823, 99.99=60959] [UPDATE: Count=49, Max=57535, Min=747, Avg=3312.37, 90=4675, 99=57535, 99.9=57535,
99.99=57535]
2022-05-01 17:12:19:623 13 sec: 1000 operations; 32.16 current ops/sec; [READ: Count=106, Max=2067, Min=688, Avg=1012.44, 90=1335, 99=1788
, 99.9=2067, 99.99=2067] [CLEANUP: Count=10, Max=2224127, Min=1, Avg=222313, 90=12, 99=2224127, 99.9=2224127, 99.99=2224127] [UPDATE: Coun
t=5, Max=1975, Min=848, Avg=1221, 90=1975, 99=1975, 99.9=1975, 99.99=1975]
[hadoop@ip-172-31-9-214 ycsb-0.17.0]$
```

Saving the output to a transactions_b.dat file which looks as follows,

```
[OVERALL], RunTime(ms), 13453
[OVERALL], Throughput(ops/sec), 74.33286255853713
[TOTAL_GCS_PS_Scavenge], Count, 1
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 10
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.07433286255853713
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 1
[TOTAL_GC_TIME], Time(ms), 10
[TOTAL_GC_TIME_%], Time(%), 0.07433286255853713
[READ], Operations, 946
[READ], AverageLatency(us), 2233.571881606765
[READ], MinLatency(us), 660
[READ], MaxLatency(us), 60959
[READ], 95thPercentileLatency(us), 5123
[READ], 99thPercentileLatency(us), 16135
[READ], Return=OK, 946
[CLEANUP], Operations, 10
[CLEANUP], AverageLatency(us), 222313.0
[CLEANUP], MinLatency(us), 1
[CLEANUP], MaxLatency(us), 2224127
[CLEANUP], 95thPercentileLatency(us), 2224127
[CLEANUP], 99thPercentileLatency(us), 2224127
[UPDATE], Operations, 54
[UPDATE], AverageLatency(us), 3118.722222222222
[UPDATE], MinLatency(us), 747
[UPDATE], MaxLatency(us), 57535
[UPDATE], 95thPercentileLatency(us), 5207
[UPDATE], 99thPercentileLatency(us), 7031
[UPDATE], Return=OK, 54
~
~
~
~
~
~
~
~
~
~
"transactions_b.dat" 31L, 1135B                                                            1,1           All
```

WORKLOAD C

1. LOAD 100K RECORDS IN CASSANDRA CQL - WORLOAD C

```
./bin/ycsb load cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadc
-P ../large.dat -s > loadc.dat
```

```
[hadoop@ip-172-31-9-214 ycsb-0.17.0]$ ./bin/ycsb load cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadc -P ../large.dat -s > loadc
.dat
/etc/alternatives/jre/bin/java -cp /home/hadoop/ycsb-0.17.0/cassandra-binding/conf:/home/hadoop/ycsb-0.17.0/conf:/home/hadoop/ycsb-0.17.0/
lib/core-0.17.0.jar:/home/hadoop/ycsb-0.17.0/lib/htrace-core4-4.1.0-incubating.jar:/home/hadoop/ycsb-0.17.0/lib/jackson-mapper-asl-1.9.4.j
ar:/home/hadoop/ycsb-0.17.0/lib/jackson-core-asl-1.9.4.jar:/home/hadoop/ycsb-0.17.0/lib/HdrHistogram-2.1.4.jar:/home/hadoop/ycsb-0.17.0/ca
ssandra-binding/lib/netty-common-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-transport-4.0.33.Final.jar:/home/ha
doop/ycsb-0.17.0/cassandra-binding/lib/slf4j-api-1.7.25.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-codec-4.0.33.Final.jar:/h
ome/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-buffer-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/cassandra-driver
-core-3.0.0.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-handler-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/l
ib/metrics-core-3.1.2.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/guava-16.0.1.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/c
assandra-binding-0.17.0.jar site.ycsb.Client -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloadc -P ../large.dat
 -s -load
Command line: -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloadc -P ../large.dat -s -load
YCSB Client 0.17.0

Loading workload...
Starting test.
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
2022-05-01 17:16:07:607 0 sec: 0 operations; est completion in 0 second
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
2022-05-01 17:16:17:576 10 sec: 10569 operations; 1056.9 current ops/sec; est completion in 1 minute [INSERT: Count=10569, Max=81151, Min=
330, Avg=822.83, 90=1038, 99=3659, 99.9=9983, 99.99=33311]
2022-05-01 17:16:27:576 20 sec: 22472 operations; 1190.3 current ops/sec; est completion in 1 minute [INSERT: Count=11903, Max=231935, Min
=308, Avg=829.67, 90=1059, 99=4207, 99.9=8167, 99.99=14111]
2022-05-01 17:16:37:577 30 sec: 34052 operations; 1157.88 current ops/sec; est completion in 59 seconds [INSERT: Count=11581, Max=125375,
Min=352, Avg=855.59, 90=1108, 99=3711, 99.9=7211, 99.99=8807]
2022-05-01 17:16:47:576 40 sec: 50047 operations; 1599.66 current ops/sec; est completion in 40 seconds [INSERT: Count=15994, Max=92159, M
in=304, Avg=617.98, 90=752, 99=2119, 99.9=6911, 99.99=12687]
2022-05-01 17:16:57:576 50 sec: 63523 operations; 1347.6 current ops/sec; est completion in 29 seconds [INSERT: Count=13476, Max=147327, M
in=308, Avg=734.21, 90=867, 99=4423, 99.9=8655, 99.99=13335]
2022-05-01 17:17:07:581 60 sec: 74258 operations; 1072.96 current ops/sec; est completion in 21 seconds [INSERT: Count=10736, Max=137087,
Min=326, Avg=926.29, 90=1194, 99=4779, 99.9=11239, 99.99=34879]
2022-05-01 17:17:17:576 70 sec: 87153 operations; 1290.15 current ops/sec; est completion in 11 seconds [INSERT: Count=12894, Max=105087,
Min=334, Avg=769.46, 90=911, 99=3431, 99.9=8383, 99.99=18831]
2022-05-01 17:17:27:576 80 sec: 99730 operations; 1257.7 current ops/sec; est completion in 1 second [INSERT: Count=12577, Max=89151, Min=
360, Avg=789.28, 90=850, 99=4739, 99.9=18559, 99.99=32895]
2022-05-01 17:17:30:053 82 sec: 100000 operations; 109 current ops/sec; [CLEANUP: Count=1, Max=2252799, Min=2250752, Avg=2251776, 90=22527
99, 99=2252799, 99.9=2252799, 99.99=2252799] [INSERT: Count=270, Max=8199, Min=411, Avg=824.72, 90=964, 99=4319, 99.9=8199, 99.99=8199]
[hadoop@ip-172-31-9-214 ycsb-0.17.0]$
```

Saving the output to a loadb.dat file which looks as follows,

```
[OVERALL], RunTime(ms), 82479
[OVERALL], Throughput(ops/sec), 1212.4298306235526
[TOTAL_GCS_PS_Scavenge], Count, 82
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 283
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.3431176420664654
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 82
[TOTAL_GC_TIME], Time(ms), 283
[TOTAL_GC_TIME_%], Time(%), 0.3431176420664654
[CLEANUP], Operations, 1
[CLEANUP], AverageLatency(us), 2251776.0
[CLEANUP], MinLatency(us), 2250752
[CLEANUP], MaxLatency(us), 2252799
[CLEANUP], 95thPercentileLatency(us), 2252799
[CLEANUP], 99thPercentileLatency(us), 2252799
[INSERT], Operations, 100000
[INSERT], AverageLatency(us), 782.74382
[INSERT], MinLatency(us), 304
[INSERT], MaxLatency(us), 231935
[INSERT], 95thPercentileLatency(us), 1377
[INSERT], 99thPercentileLatency(us), 3987
[INSERT], Return=OK, 100000
~
~
~
```

2. RUN 100K RECORDS FOR WORKLOAD C - 10 THREADS, 100 OPERATIONS PER SEC - 10 operations/sec per thread

```
./bin/ycsb run cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadc
-P ../large.dat -s -threads 10 -target 100 > transactions_c.dat
```

```
[[hadoop@ip-172-31-9-214 ycsb-0.17.0]$ ./bin/ycsb run cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadc -P ../large.dat -s -threads
 10 -target 100 > transactions_c.dat
/etc/alternatives/jre/bin/java -cp /home/hadoop/ycsb-0.17.0/cassandra-binding/conf:/home/hadoop/ycsb-0.17.0/conf:/home/hadoop/ycsb-0.17.0/
lib/core-0.17.0.jar:/home/hadoop/ycsb-0.17.0/lib/htrace-core4-4.1.0-incubating.jar:/home/hadoop/ycsb-0.17.0/lib/jackson-mapper-asl-1.9.4.j
ar:/home/hadoop/ycsb-0.17.0/lib/jackson-core-asl-1.9.4.jar:/home/hadoop/ycsb-0.17.0/lib/HdrHistogram-2.1.4.jar:/home/hadoop/ycsb-0.17.0/ca
ssandra-binding/lib/netty-common-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-transport-4.0.33.Final.jar:/home/ha
doop/ycsb-0.17.0/cassandra-binding/lib/slf4j-api-1.7.25.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-codec-4.0.33.Final.jar:/h
ome/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-buffer-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/cassandra-driver
-core-3.0.0.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/netty-handler-4.0.33.Final.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/l
ib/metrics-core-3.1.2.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/guava-16.0.1.jar:/home/hadoop/ycsb-0.17.0/cassandra-binding/lib/c
assandra-binding-0.17.0.jar site.ycsb.Client -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloadc -P ../large.dat
 -s -threads 10 -target 100 -t
Command line: -db site.ycsb.db.CassandraCQLClient -p hosts=127.0.0.1 -P workloads/workloadc -P ../large.dat -s -threads 10 -target 100 -t
YCSB Client 0.17.0

Loading workload...
Starting test.
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
2022-05-01 17:20:08:373 0 sec: 0 operations; est completion in 0 second
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
2022-05-01 17:20:18:334 10 sec: 885 operations; 88.5 current ops/sec; est completion in 2 second [READ: Count=885, Max=80639, Min=715, Avg
=2712.91, 90=4029, 99=50399, 99.9=79423, 99.99=80639]
2022-05-01 17:20:21:821 13 sec: 1000 operations; 32.99 current ops/sec; [READ: Count=115, Max=12559, Min=650, Avg=1160.23, 90=1288, 99=591
9, 99.9=12559, 99.99=12559] [CLEANUP: Count=10, Max=2220031, Min=2, Avg=221903.5, 90=11, 99=2220031, 99.9=2220031, 99.99=2220031]
[hadoop@ip-172-31-9-214 ycsb-0.17.0]$
```

Saving the output to a transactions_c.dat file which looks as follows,

```
[OVERALL], RunTime(ms), 13487
[OVERALL], Throughput(ops/sec), 74.14547341884779
[TOTAL_GCS_PS_Scavenge], Count, 1
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 10
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.07414547341884778
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 1
[TOTAL_GC_TIME], Time(ms), 10
[TOTAL_GC_TIME_%], Time(%), 0.07414547341884778
[READ], Operations, 1000
[READ], AverageLatency(us), 2534.348
[READ], MinLatency(us), 650
[READ], MaxLatency(us), 80639
[READ], 95thPercentileLatency(us), 6147
[READ], 99thPercentileLatency(us), 43615
[READ], Return=OK, 1000
[CLEANUP], Operations, 10
[CLEANUP], AverageLatency(us), 221903.5
[CLEANUP], MinLatency(us), 2
[CLEANUP], MaxLatency(us), 2220031
[CLEANUP], 95thPercentileLatency(us), 2220031
[CLEANUP], 99thPercentileLatency(us), 2220031
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"transactions_c.dat" 24L, 885B                                                         1,1           All
```

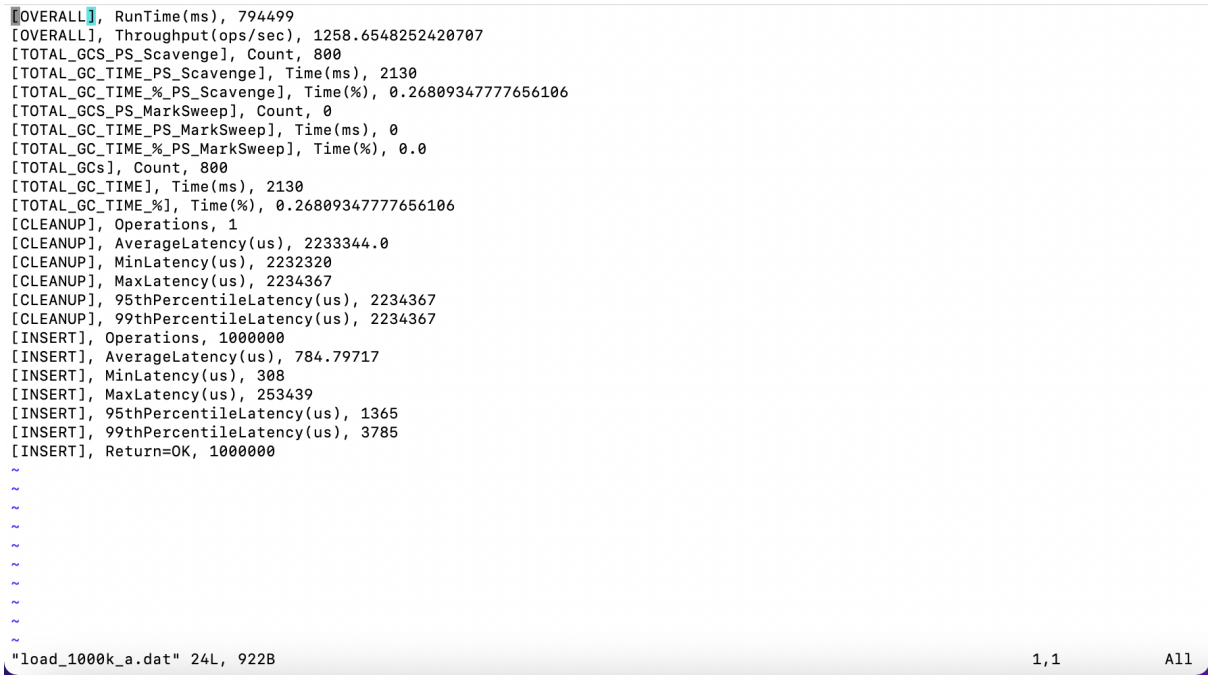
Now we will follow the similar steps for 1000k dataset by changing the large.dat file contents with recordcount as 1000k

WORKLOAD A

1. LOAD 1000K RECORDS IN CASSANDRA CQL - WORLOAD A

```
./bin/ycsb load cassandra-cql -p hosts="127.0.0.1" -P workloads/workloada
-P ../large.dat -s > load_1000k_a.dat
```

Saving the output to a load_1000k_a.dat file which looks as follows,



```
[OVERALL], RunTime(ms), 794499
[OVERALL], Throughput(ops/sec), 1258.6548252420707
[TOTAL_GCS_PS_Scavenge], Count, 800
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 2130
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.26809347777656106
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 800
[TOTAL_GC_TIME], Time(ms), 2130
[TOTAL_GC_TIME_%], Time(%), 0.26809347777656106
[CLEANUP], Operations, 1
[CLEANUP], AverageLatency(us), 2233344.0
[CLEANUP], MinLatency(us), 2232320
[CLEANUP], MaxLatency(us), 2234367
[CLEANUP], 95thPercentileLatency(us), 2234367
[CLEANUP], 99thPercentileLatency(us), 2234367
[INSERT], Operations, 1000000
[INSERT], AverageLatency(us), 784.79717
[INSERT], MinLatency(us), 308
[INSERT], MaxLatency(us), 253439
[INSERT], 95thPercentileLatency(us), 1365
[INSERT], 99thPercentileLatency(us), 3785
[INSERT], Return=OK, 1000000
~
~
~
~
~
~
~
~
~
~
"load_1000k_a.dat" 24L, 922B                                                  1,1           All
```

2. RUN 1000K RECORDS FOR WORKLOAD A - 10 THREADS, 100 OPERATIONS PER SEC - 10 operations/sec per thread

```
./bin/ycsb run cassandra-cql -p hosts="127.0.0.1" -P workloads/workloada
-P ../large.dat -s -threads 10 -target 100 > transactions_1000k_a.dat
```

Saving the output to a transactions_1000k_a.dat file which looks as follows,

```
[OVERALL], RunTime(ms), 13648
[OVERALL], Throughput(ops/sec), 73.27080890973036
[TOTAL_GCS_PS_Scavenge], Count, 1
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 10
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.07327080890973037
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 1
[TOTAL_GC_TIME], Time(ms), 10
[TOTAL_GC_TIME_%], Time(%), 0.07327080890973037
[READ], Operations, 483
[READ], AverageLatency(us), 7001.130434782609
[READ], MinLatency(us), 861
[READ], MaxLatency(us), 113599
[READ], 95thPercentileLatency(us), 13055
[READ], 99thPercentileLatency(us), 91391
[READ], Return=OK, 483
[CLEANUP], Operations, 10
[CLEANUP], AverageLatency(us), 221903.3
[CLEANUP], MinLatency(us), 1
[CLEANUP], MaxLatency(us), 2220031
[CLEANUP], 95thPercentileLatency(us), 2220031
[CLEANUP], 99thPercentileLatency(us), 2220031
[UPDATE], Operations, 517
[UPDATE], AverageLatency(us), 3012.1257253384915
[UPDATE], MinLatency(us), 666
[UPDATE], MaxLatency(us), 109567
[UPDATE], 95thPercentileLatency(us), 7571
[UPDATE], 99thPercentileLatency(us), 37439
[UPDATE], Return=OK, 517
~
```

WORKLOAD B

1. LOAD 1000K RECORDS IN CASSANDRA CQL - WORLOAD B

```
./bin/ycsb load cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadb
-P ../large.dat -s > load_1000k_b.dat
```

Saving the output to a load_1000k_b.dat file which looks as follows,

```
[OVERALL], RunTime(ms), 736272
[OVERALL], Throughput(ops/sec), 1358.1937110198405
[TOTAL_GCS_PS_Scavenge], Count, 814
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 2195
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.298123519568855
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 814
[TOTAL_GC_TIME], Time(ms), 2195
[TOTAL_GC_TIME_%], Time(%), 0.298123519568855
[CLEANUP], Operations, 1
[CLEANUP], AverageLatency(us), 2233344.0
[CLEANUP], MinLatency(us), 2232320
[CLEANUP], MaxLatency(us), 2234367
[CLEANUP], 95thPercentileLatency(us), 2234367
[CLEANUP], 99thPercentileLatency(us), 2234367
[INSERT], Operations, 1000000
[INSERT], AverageLatency(us), 726.701254
[INSERT], MinLatency(us), 282
[INSERT], MaxLatency(us), 254335
[INSERT], 95thPercentileLatency(us), 1253
[INSERT], 99thPercentileLatency(us), 3441
[INSERT], Return=OK, 1000000
~
```

2. RUN 1000K RECORDS FOR WORKLOAD B - 10 THREADS, 100 OPERATIONS PER SEC - 10 operations/sec per thread

```
./bin/ycsb run cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadb
-P ../large.dat -s -threads 10 -target 100 > transactions_1000k_b.dat
```

Saving the output to a transactions_1000k_b.dat file which looks as follows,

```
[OVERALL], RunTime(ms), 14425
[OVERALL], Throughput(ops/sec), 69.32409012131716
[TOTAL_GCS_PS_Scavenge], Count, 1
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 22
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.15251299826689774
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 1
[TOTAL_GC_TIME], Time(ms), 22
[TOTAL_GC_TIME_%], Time(%), 0.15251299826689774
[READ], Operations, 941
[READ], AverageLatency(us), 5761.360255047822
[READ], MinLatency(us), 809
[READ], MaxLatency(us), 181247
[READ], 95thPercentileLatency(us), 12983
[READ], 99thPercentileLatency(us), 46815
[READ], Return=OK, 941
[CLEANUP], Operations, 10
[CLEANUP], AverageLatency(us), 222313.4
[CLEANUP], MinLatency(us), 2
[CLEANUP], MaxLatency(us), 2224127
[CLEANUP], 95thPercentileLatency(us), 2224127
[CLEANUP], 99thPercentileLatency(us), 2224127
[UPDATE], Operations, 59
[UPDATE], AverageLatency(us), 4290.237288135593
[UPDATE], MinLatency(us), 887
[UPDATE], MaxLatency(us), 56063
[UPDATE], 95thPercentileLatency(us), 13703
"transactions_1000k_b.dat" 31L, 1139B                                    1,1           Top
```

WORKLOAD C

1. LOAD 1000K RECORDS IN CASSANDRA CQL - WORLOAD C

```
./bin/ycsb load cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadc
-P ../large.dat -s > load_1000k_c.dat
```

Saving the output to a load_1000k_c.dat file which looks as follows,

```
[OVERALL], RunTime(ms), 239579
[OVERALL], Throughput(ops/sec), 1393.9702561576767
[TOTAL_GCS_PS_Scavenge], Count, 249
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 682
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.28466601830711374
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 249
[TOTAL_GC_TIME], Time(ms), 682
[TOTAL_GC_TIME_%], Time(%), 0.28466601830711374
[CLEANUP], Operations, 1
[CLEANUP], AverageLatency(us), 2235392.0
[CLEANUP], MinLatency(us), 2234368
[CLEANUP], MaxLatency(us), 2236415
[CLEANUP], 95thPercentileLatency(us), 2236415
[CLEANUP], 99thPercentileLatency(us), 2236415
[INSERT], Operations, 333966
[INSERT], AverageLatency(us), 698.7809507554662
[INSERT], MinLatency(us), 291
[INSERT], MaxLatency(us), 186239
[INSERT], 95thPercentileLatency(us), 1102
[INSERT], 99thPercentileLatency(us), 2969
[INSERT], Return=OK, 333966
[INSERT], Return=ERROR, 1
[INSERT-FAILED], Operations, 1
[INSERT-FAILED], AverageLatency(us), 471680.0
[INSERT-FAILED], MinLatency(us), 471552
[INSERT-FAILED], MaxLatency(us), 471807
"load_1000k_c.dat" 31L, 1211B                                1,1           Top
```

2. RUN 1000K RECORDS FOR WORKLOAD C - 10 THREADS, 100 OPERATIONS PER SEC - 10 operations/sec per thread

```
./bin/ycsb run cassandra-cql -p hosts="127.0.0.1" -P workloads/workloadc
-P ../large.dat -s -threads 10 -target 100 > transactions_1000k_c.dat
```

Saving the output to a transactions_1000k_c.dat file which looks as follows,

```
[OVERALL], RunTime(ms), 13570
[OVERALL], Throughput(ops/sec), 73.69196757553426
[TOTAL_GCS_PS_Scavenge], Count, 1
[TOTAL_GC_TIME_PS_Scavenge], Time(ms), 10
[TOTAL_GC_TIME_%_PS_Scavenge], Time(%), 0.07369196757553427
[TOTAL_GCS_PS_MarkSweep], Count, 0
[TOTAL_GC_TIME_PS_MarkSweep], Time(ms), 0
[TOTAL_GC_TIME_%_PS_MarkSweep], Time(%), 0.0
[TOTAL_GCs], Count, 1
[TOTAL_GC_TIME], Time(ms), 10
[TOTAL_GC_TIME_%], Time(%), 0.07369196757553427
[READ], Operations, 0
[READ], AverageLatency(us), NaN
[READ], MinLatency(us), 9223372036854775807
[READ], MaxLatency(us), 0
[READ], 95thPercentileLatency(us), 0
[READ], 99thPercentileLatency(us), 0
[READ], Return=NOT_FOUND, 1000
[CLEANUP], Operations, 10
[CLEANUP], AverageLatency(us), 224566.0
[CLEANUP], MinLatency(us), 2
[CLEANUP], MaxLatency(us), 2246655
[CLEANUP], 95thPercentileLatency(us), 2246655
[CLEANUP], 99thPercentileLatency(us), 2246655
[READ-FAILED], Operations, 1000
[READ-FAILED], AverageLatency(us), 2688.526
[READ-FAILED], MinLatency(us), 778
[READ-FAILED], MaxLatency(us), 96447
[READ-FAILED], 95thPercentileLatency(us), 5947
[READ-FAILED], 99thPercentileLatency(us), 14415
~
~
~
                                                              1,1           All
```

AMAZON AWS KEYSPACES - MIGRATION

For Migration to AWS Keyspaces, we will initialize an EMR cluster and install Apache cassandra similar to above YCSB implementation. The EMR cluster is pre-installed with aws cli which we will use to generate Service Specific Credentials for an IAM user which I had created previously on AWS IAM console. We generate our credentials with the following command. My IAM user name is Keyspaces-cassandra in this case.

Command

```
aws iam create-service-specific-credential \
    --user-name Keyspaces-cassandra \
    --service-name cassandra.amazonaws.com
```

O/p:

```
{
    "ServiceSpecificCredential": {
        "CreateDate": "2022-05-02T14:54:31+00:00",
        "ServiceName": "cassandra.amazonaws.com",
        "ServiceUserName": "Keyspaces-cassandra-at-200332471694",
        "ServicePassword": "hidden",
        "ServiceSpecificCredentialId": "ACCAS5JF65WHA2YAPGL64",
        "UserName": "Keyspaces-cassandra",
        "Status": "Active"
    }
}
```

We need to save these credentials somewhere as this is the only time we will see this output. We will import keyspaces migration key which we generated during creation of the IAM user to our EMR cluster. This key is named `keyspaces-migration.pem` . We will also create a keyspaces file mentioned in the above Experiments section of this report , which will be imported to the EMR cluster as well. This is the working home directory of the EMR cluster.

```
[hadoop@ip-172-31-27-74 ~]$ ls -al
total 37552
drwxr-xr-x  6 hadoop hadoop      218 May  3 14:00 .
drwxr-xr-x  4 root   root         36 Feb 24 01:22 ..
drwxrwxr-x 10 hadoop hadoop      208 May  3 13:56 apache-cassandra-3.11.2
-rw-rw-r--  1 hadoop hadoop 38436262 Feb 19  2018 apache-cassandra-3.11.2-bin.tar.gz
drwxr-xr-x  2 hadoop hadoop       39 May  3 13:57 .aws
-rw-r--r--  1 hadoop hadoop       86 Feb 19 02:55 .bash_profile
-rw-r--r--  1 hadoop hadoop      626 Feb 19 02:55 .bashrc
drwx------  3 hadoop root         25 May  3 13:56 .cache
-rw-------  1 hadoop hadoop     1674 May  3 14:00 keyspaces-migration.pem
-rw-rw-r--  1 hadoop hadoop      702 May  3 13:58 keyspaces_sample_table.csv
drwx------  2 hadoop hadoop       29 May  3 13:56 .ssh
[hadoop@ip-172-31-27-74 ~]$
```

Now we will download the Starfield digital certificate and save it in a new directory .cassandra and also crate a configuration file cqlshrc.

```
[hadoop@ip-172-31-27-74 ~]$ mkdir .cassandra
[hadoop@ip-172-31-27-74 ~]$ curl https://certs.secureserver.net/repository/sf-class2-root.crt -O
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1468  100  1468    0     0   9984      0 --:--:-- --:--:-- --:--:--  9986
[hadoop@ip-172-31-27-74 ~]$ ls
apache-cassandra-3.11.2             keyspaces-migration.pem     sf-class2-root.crt
apache-cassandra-3.11.2-bin.tar.gz  keyspaces_sample_table.csv
[hadoop@ip-172-31-27-74 ~]$ cp sf-class2-root.crt .cassandra/
[hadoop@ip-172-31-27-74 ~]$ cd .cassandra
[hadoop@ip-172-31-27-74 .cassandra]$ ls
sf-class2-root.crt
[hadoop@ip-172-31-27-74 .cassandra]$ vim cqlshrc
```

The contents of cqlshrc file will be as follows,

```
[connection]
port = 9142
factory = cqlshlib.ssl.ssl_transport_factory

[ssl]
validate = true
certfile =  ~/.cassandra/sf-class2-root.crt
version = TLSv1_2

[copy]
NUMPROCESSES=16
MAXATTEMPTS=25

[copy-from]
CHUNKSIZE=30
INGESTRATE=1500
MAXINSERTERRORS=-1
MAXPARSEERRORS=-1
MINBATCHSIZE=1
MAXBATCHSIZE=10

[csv]
field_size_limit=999999
~
~
                                                                    7,14          All
```

Configure aws cli to utilize the service specific credentials which we created for cassandra keysapces.

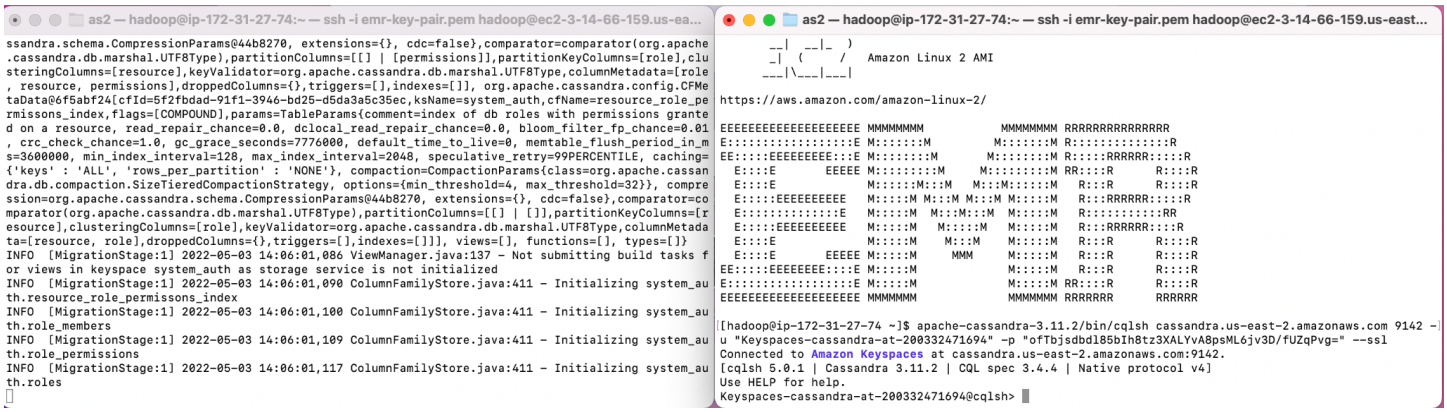
```
aws configure
```

We will enter service specific credentials along with the region as us-east-2 and format as json

```
[hadoop@ip-172-31-27-74 ~]$ aws configure list
      Name                    Value             Type    Location
      ----                    -----             ----    --------
   profile                <not set>             None    None
access_key     ****************GL64 shared-credentials-file
secret_key     ****************Pvg= shared-credentials-file
    region                   us-east-2              env    AWS_DEFAULT_REGION
[hadoop@ip-172-31-27-74 ~]$
```

We are set to connect to AWS Keyspaces now,

`apache-cassandra-3.11.2/bin/cqlsh` *`host`* `9142 -u` *`ServiceUserName`* `-p` *`ServicePassword`* `--ssl`



The next step in the experiments section is to create a keysapce and table on AWS keysapces,

```
Keyspaces-cassandra-at-200332471694@cqlsh> CREATE KEYSPACE IF NOT EXISTS "mykeyspace" WITH REPLICATION={'class':'SimpleStrategy', 'replication_factor': 1};
Keyspaces-cassandra-at-200332471694@cqlsh> DESCRIBE KEYSPACES;

mykeyspace  system_schema  system_schema_mcs  system

Keyspaces-cassandra-at-200332471694@cqlsh> USE mykeyspace;
Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace> CREATE TABLE mykeyspace.mytable( award text, year int, category text, rank int, author text, book_title text, publishe
d text, PRIMARY KEY((year,award), category,rank));
Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace> DESCRIBE TABLES;

<empty>

Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace> DESCRIBE TABLE mytable;

CREATE TABLE mykeyspace.mytable (
    year int,
    award text,
    category text,
    rank int,
    author text,
    book_title text,
    published text,
    PRIMARY KEY ((year, award), category, rank)
) WITH CLUSTERING ORDER BY (category ASC, rank ASC)
    AND bloom_filter_fp_chance = 0.01
    AND caching = {'class': 'com.amazonaws.cassandra.DefaultCaching'}
    AND comment = ''
    AND compaction = {'class': 'com.amazonaws.cassandra.DefaultCompaction'}
    AND compression = {'class': 'com.amazonaws.cassandra.DefaultCompression'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.0
    AND default_time_to_live = 0
    AND gc_grace_seconds = 7776000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 3600000
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';

Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace>
```

To prepare data, we randomize and shuffle the keyspces_sample_table.csv file and analyze it in the following steps of commands,

```
[hadoop@ip-172-31-27-74 ~]$ tail -n +2 keyspaces_sample_table.csv | shuf -o keyspace.table.csv
[hadoop@ip-172-31-27-74 ~]$ (head -1 keyspaces_sample_table.csv && cat keyspace.table.csv ) > keyspace.table.csv1 && mv keyspace.table.csv1 keyspace.table.csv
[hadoop@ip-172-31-27-74 ~]$ ls
apache-cassandra-3.11.2             keyspaces-migration.pem     keyspace.table.csv
apache-cassandra-3.11.2-bin.tar.gz  keyspaces_sample_table.csv  sf-class2-root.crt
[hadoop@ip-172-31-27-74 ~]$ vim keyspace.table.csv
[hadoop@ip-172-31-27-74 ~]$ awk -F, 'BEGIN {samp=10000;max=-1;}{if(NR>1){len=length($0);t+=len;avg=t/NR;max=(len>max ? len : max)}}NR==samp{exit}END{printf("{lines: %d, average: %d bytes, max: %d bytes}\n",NR,avg,max);}' keyspace.table.csv
{lines: 10, average: 63 bytes, max: 78 bytes}
[hadoop@ip-172-31-27-74 ~]$ awk -F, 'BEGIN {samp=10000;max=-1;}{if(NR>1){len=length($0);t+=len;avg=t/NR;max=(len>max ? len : max)}}NR==samp{exit}END{printf("{lines: %d, average: %d bytes, max: %d bytes}\n",NR,avg,max);}' keyspace.table.csv
{lines: 10, average: 63 bytes, max: 78 bytes}
```

For demonstration purpose, I have also loaded a data.csv file containing details about tweets from the Twitter dataset which was available on Kaggle Open Source dataset repository.

```
[hadoop@ip-172-31-27-74 ~]$ ls
apache-cassandra-3.11.2             keyspaces-migration.pem     sf-class2-root.crt
apache-cassandra-3.11.2-bin.tar.gz  keyspaces_sample_table.csv
data.csv                            keyspace.table.csv
[hadoop@ip-172-31-27-74 ~]$ tail -n +2 data.csv | shuf -o data.table.csv
[hadoop@ip-172-31-27-74 ~]$ (head -1 data.csv && cat data.table.csv ) > data.table.csv1 && mv data.table.csv1 data.table.csv
[hadoop@ip-172-31-27-74 ~]$ awk -F, 'BEGIN {samp=10000;max=-1;}{if(NR>1){len=length($0);t+=len;avg=t/NR;max=(len>max ? len : max)}}NR==samp{exit}END{printf("{lines: %d, average: %d bytes, max: %d bytes}\n",NR,avg,max);}' keyspace.table.csv
{lines: 10, average: 63 bytes, max: 78 bytes}
[hadoop@ip-172-31-27-74 ~]$ awk -F, 'BEGIN {samp=10000;max=-1;}{if(NR>1){len=length($0);t+=len;avg=t/NR;max=(len>max ? len : max)}}NR==samp{exit}END{printf("{lines: %d, average: %d bytes, max: %d bytes}\n",NR,avg,max);}' data.table.csv
{lines: 10000, average: 79 bytes, max: 930 bytes}
[hadoop@ip-172-31-27-74 ~]$
```

We then change and configure the provisioned capacity settings for our ‘mykeyspace.mytable’ that we just created in our AWS Keyspace, using the ALTER command.

```
Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace> ALTER TABLE mykeyspace.mytable WITH custom_properties={'capacity_mode':{'throughput_mode': 'PROVISIONED', 'read_capacity_units': 100, 'write_capacity_units':3 }} ;
Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace>
```

Our final step in migration is to migrate our dataset using the COPY command. Before that we set the Consistency level of cassandra to LOCAL_QUORUM.

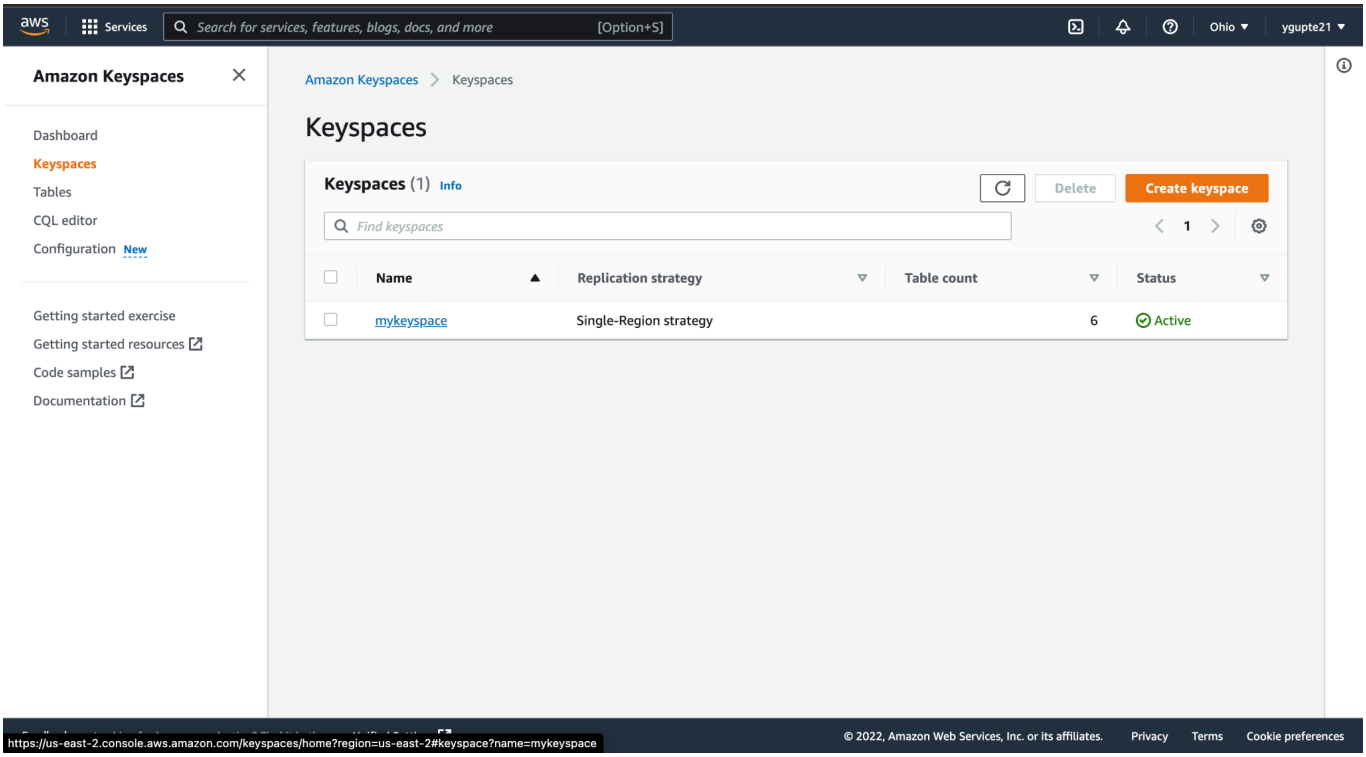
```
[hadoop@ip-172-31-18-20 ~]$ apache-cassandra-3.11.2/bin/cqlsh cassandra.us-east-2.amazonaws.com 9142 -u
 "Keyspaces-cassandra-at-200332471694" -p "ofTbjsdbdl85bIh8tz3XALYvA8psML6jv3D/fUZqPvg=" --ssl
Connected to Amazon Keyspaces at cassandra.us-east-2.amazonaws.com:9142.
[cqlsh 5.0.1 | Cassandra 3.11.2 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
Keyspaces-cassandra-at-200332471694@cqlsh> USE mykeyspace;
Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace> DESCRIBE TABLES;

mytable2  mytable_new  data  mytable

Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace> CONSISTENCY LOCAL_QUORUM;
Consistency level set to LOCAL_QUORUM.
Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace> COPY mytable_new FROM './keyspace.table.csv' WITH
 HEADER=true;
Reading options from /home/hadoop/.cassandra/cqlshrc:[copy]: {'maxattempts': '25', 'numprocesses': '16'
}
Reading options from /home/hadoop/.cassandra/cqlshrc:[copy-from]: {'minbatchsize': '1', 'chunksize': '3
0', 'maxparseerrors': '-1', 'maxinserterrors': '-1', 'ingestrate': '1500', 'maxbatchsize': '10'}
Reading options from the command line: {'header': 'true'}
Using 16 child processes

Starting copy of mykeyspace.mytable_new with columns [award, year, category, rank, author, book_title,
published].
Processed: 9 rows; Rate:       6 rows/s; Avg. rate:      11 rows/s
9 rows imported from 1 files in 0.814 seconds (0 skipped).
Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace>
```

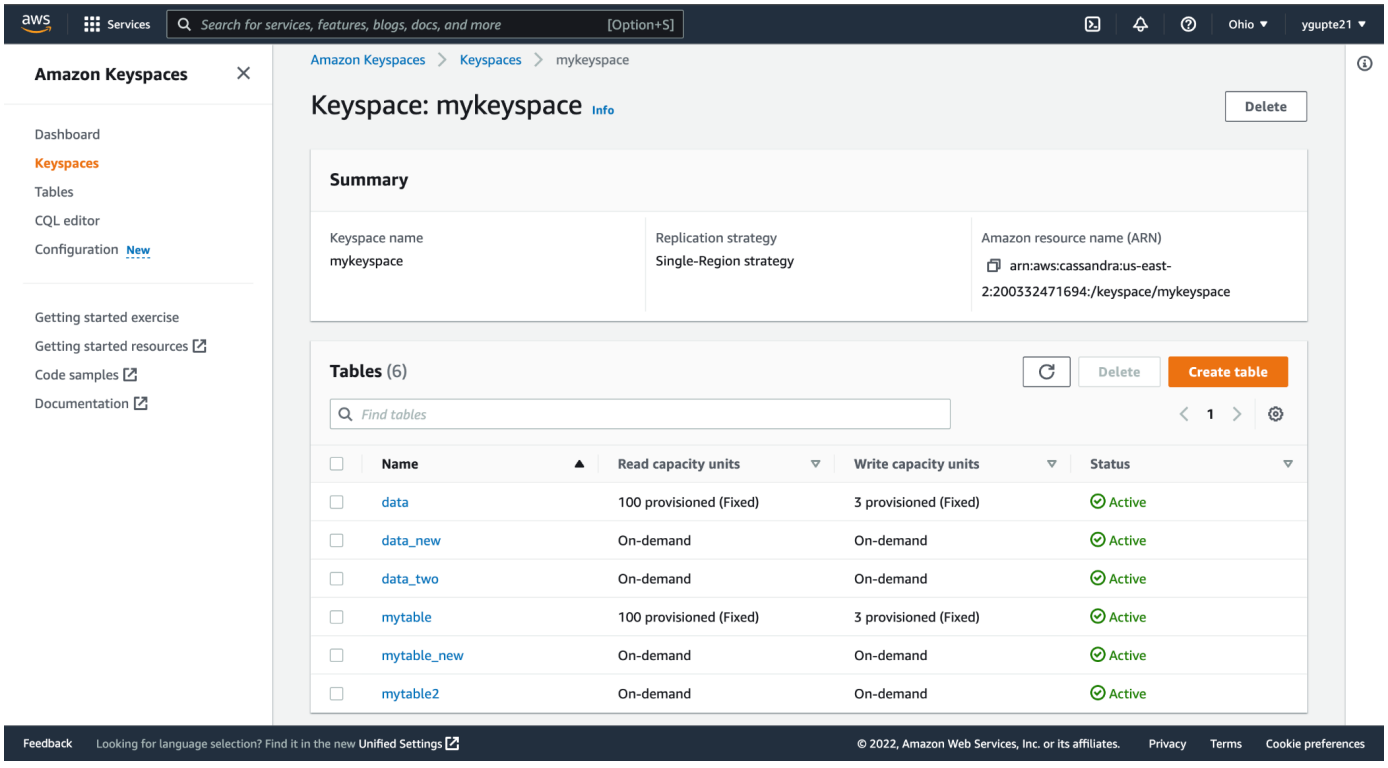
We have successfully imported our data from Apache cassandra to AWS Keyspaces using the cqlsh scripting. Next we perform a CRUD operations on the imported / migrated data present in AWS Keysapces

```
Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace> INSERT INTO mytable_new( award, year, category,rank,author,book_title,published) VALUES('Academy',
2022, 'Sci-Fi', 2, 'Alex', 'Into the space', 'SciFi Puclications');
Unable to fetch query trace: Error from server: code=2200 [Invalid query] message="unconfigured table system_traces.sessions"
Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace> SELECT * FROM mytable_new;

 award            | year | category     | rank | author             | book_title            | published
------------------+------+--------------+------+--------------------+-----------------------+--------------------
          Academy | 2022 |       Sci-Fi |    2 |               Alex |        Into the space | SciFi Puclications
 Kwesi Manu Prize | 2020 |      Fiction |    1 |         Akua Mansa |     Where did you go? |      SomePublisher
 Kwesi Manu Prize | 2020 |      Fiction |    2 |        John Stiles |             Yesterday |      Example Books
 Kwesi Manu Prize | 2020 |      Fiction |    3 |         Nikki Wolf | Moving to the Chateau |       AnyPublisher
             Wolf | 2020 |  Non-Fiction |    1 |        Wang Xiulan |      History of Ideas |      Example Books
             Wolf | 2020 |  Non-Fiction |    2 | Ana Carolina Silva |         Science Today |      SomePublisher
             Wolf | 2020 |  Non-Fiction |    3 |  Shirley Rodriguez | The Future of Sea Ice |       AnyPublisher
      Richard Roe | 2020 |      Fiction |    1 |  Alejandro Rosalez |           Long Summer |      SomePublisher
      Richard Roe | 2020 |      Fiction |    2 |        Arnav Desai |                The Key |      Example Books
      Richard Roe | 2020 |      Fiction |    3 |      Mateo Jackson |      Inside the Whale |       AnyPublisher

(10 rows)
Unable to fetch query trace: Error from server: code=2200 [Invalid query] message="unconfigured table system_traces.sessions"
Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace>
```

```
Keyspaces-cassandra-at-200332471694@cqlsh:mykeyspace> SELECT author, book_title, category FROM mytable_new WHERE published='SomePublisher' ALLOW FILTERIN
G;

 author             | book_title        | category
--------------------+-------------------+-------------
         Akua Mansa | Where did you go? |      Fiction
 Ana Carolina Silva |     Science Today | Non-Fiction
  Alejandro Rosalez |       Long Summer |      Fiction

(3 rows)
```

We can manage keysapces and tables and query cassandra queries from AWS Keyspaces dashboard as well. As we have migrated data from Apache Cassandra to AWS Keysapces.


aws
Services
Search for services, features, blogs, docs, and more
[Option+S]
Ohio
ygupte21
Amazon Keyspaces
Dashboard
Keyspaces
Tables
CQL editor
Configuration New
Getting started exercise
Getting started resources
Code samples
Documentation
Amazon Keyspaces > Keyspaces
Keyspaces
Keyspaces (1) Info
Delete
Create keyspace
Find keyspaces
1
Name
Replication strategy
Table count
Status
mykeyspace
Single-Region strategy
6
Active
https://us-east-2.console.aws.amazon.com/keyspaces/home?region=us-east-2#keyspace?name=mykeyspace
© 2022, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


aws
Services
Search for services, features, blogs, docs, and more
[Option+S]
Ohio
ygupte21
Amazon Keyspaces
Dashboard
Keyspaces
Tables
CQL editor
Configuration New
Getting started exercise
Getting started resources
Code samples
Documentation
Amazon Keyspaces > Keyspaces > mykeyspace
Keyspace: mykeyspace Info
Delete
Summary
Keyspace name
mykeyspace
Replication strategy
Single-Region strategy
Amazon resource name (ARN)
arn:aws:cassandra:us-east-2:200332471694:/keyspace/mykeyspace
Tables (6)
Delete
Create table
Find tables
1
Name
Read capacity units
Write capacity units
Status
data
100 provisioned (Fixed)
3 provisioned (Fixed)
Active
data_new
On-demand
On-demand
Active
data_two
On-demand
On-demand
Active
mytable
100 provisioned (Fixed)
3 provisioned (Fixed)
Active
mytable_new
On-demand
On-demand
Active
mytable2
On-demand
On-demand
Active
Feedback
Looking for language selection? Find it in the new Unified Settings
© 2022, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

AWS Neptune

Import Neptune ML module

```
import neptune_ml_utils as neptune_ml
neptune_ml.check_ml_enabled()
```

Using Bulk Loader for Ingesting Data

```
s3_bucket_uri="s3://<INSERT S3 BUCKET OR PATH>"
# remove trailing slashes
s3_bucket_uri = s3_bucket_uri[:-1] if s3_bucket_uri.endswith('/') else s3_bucket_uri
```

```
%load -s {response} -f csv -p OVERSUBSCRIBE --run
```

Configuring Features

```
export_params={
"command": "export-pg",
"params": { "endpoint": neptune_ml.get_host(),
            "profile": "neptune_ml",
            "cloneCluster": False
            },
"outputS3Path": f'{s3_bucket_uri}/neptune-export',
"additionalParams": {
        "neptune_ml": {
          "version": "v2.0",
         "features": [
            {
                "node": "movie",
                "property": "title",
                "type": "word2vec"
            },
            {
                "node": "user",
                "property": "age",
                "type": "bucket_numerical",
                "range" : [1, 100],
                "num_buckets": 10
            }
         ]
        }
      },
"jobSize": "medium"}
```

Data Processing

```
# The training_job_name can be set to a unique value below, otherwise one will be auto generated
training_job_name=neptune_ml.get_training_job_name('link-prediction')

processing_params = f"""
--config-file-name training-data-configuration.json
--job-id {training_job_name}
--s3-input-uri {export_results['outputS3Uri']}
--s3-processed-uri {str(s3_bucket_uri)}/preloading """
```

```
%neptune_ml dataprocessing start --wait --store-to processing_results {processing_params}
```

Training Model

```
training_params=f"""
--job-id {training_job_name}
--data-processing-id {training_job_name}
--instance-type ml.p3.2xlarge
--s3-output-uri {str(s3_bucket_uri)}/training
--max-hpo-number 2
--max-hpo-parallel 2 """
```

```
%neptune_ml training start --wait --store-to training_results {training_params}
```

Example of Prediction

```
%%gremlin
g.with("Neptune#ml.endpoint","${endpoint}").
    with("Neptune#ml.limit",10).
      V().has('title', 'Apollo 13 (1995)').
        in('rated').with("Neptune#ml.prediction").hasLabel('user').id()
```

By using link predictions we will be able to show how to predict edges and vertices.