Explore the use of NoSQL Document databases Technologies

AWS DynamoDB and Apache Cassandra

Atharva Tanaji Kadam (A20467229)

Overview:

Developers now have a choice between two main categories of databases: RDBMS and NoSQL (Not Only SQL). NoSQL databases, like DynamoDB and MongoDB, sacrifice some robustness to gain speed and scalability. Such databases support horizontal scaling, efficiently process huge amounts of data, and perform well in a distributed environment. I am going to compare 3 major NoSQL databases from various aspects like scalability, availability, and performance on large volumes of data.
The use of NoSQL databases has a number of benefits. It might be helpful when handling challenging queries. Any application that needs reliable, single-digit millisecond latency can benefit from its tailored services.
We will explore the use of cloud 2 NoSQL databases in-depth and compare AWS DynamoDB, and Apache Cassandra from many aspects including query language, data representation, data types, architecture, security, scalability, availability, data consistency, and other capabilities.

AWS Dynamo DB:

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets you offload the administrative burdens of operating and scaling a distributed database so that you don't have to worry about hardware provisioning, setup, configuration, replication, software patching, or cluster scaling. DynamoDB also offers encryption at rest, which eliminates the operational burden and complexity involved in protecting sensitive data. It is designed to deliver single-digit millisecond response at any scale and predictable performance. It is a key-value document database with a collection of items that follow a Peer-to-Peer system.

Core Components of AWS DynamoDB:

Data is stored in tables using key-value or document paradigm in NoSQL databases. The three basic data models utilized by DynamoDB are tables, items, and attributes. Items are collections of attributes and tables are collections of items. Attributes behave like fields from RDBMS and might be one-valued or multivalued. Items are similar to rows and each item in a table can be identified by a primary key.
There are 2 types of primary keys in DynamoDB:

1. **Partition Key**: A simple primary key, composed of one attribute known as the partition key.DynamoDB uses the partition key's value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to

DynamoDB) in which the item will be stored. No two items can have the same partition key value in a table with only a partition key.

2. **Partition key and sort key:** Referred to as a composite primary key, this type of key is composed of two attributes. The first attribute is the partition key, and the second attribute is the sort key.DynamoDB uses the partition key value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored. All items with the same partition key value are stored together, in sorted order by sort key value.In a table that has a partition key and a sort key, it's possible for multiple items to have the same partition key value. However, those items must have different sort key values.

On a table, you can add one or more secondary indexes. In addition to queries against the primary key, a secondary index enables you to query the data in the database using an alternative key. Although indexes are not required by DynamoDB, they do provide your apps additional flexibility when accessing your data. Following creating a secondary index on a table, reading data from the index is quite similar to reading data from the table.

DynamoDB supports two kinds of indexes:

1. Global secondary index – An index with a partition key and sort key that can be different from those on the table.
2. Local secondary index – An index that has the same partition key as the table, but a different sort key.

An optional feature that records data update events in DynamoDB tables is called DynamoDB Streams. These events' data are displayed in the stream in almost real-time and in the chronological order in which they happened.

Each event is represented by a stream record. If you enable a stream on a table, DynamoDB Streams writes a stream record whenever one of the following events occurs:

1. A new item is added to the table: The stream captures an image of the entire item, including all of its attributes.
2. An item is updated: The stream captures the "before" and "after" images of any attributes modified in the item.
3. An item is deleted from the table: The stream captures an image of the entire item before it was deleted.

Each stream record also contains the name of the table, the event timestamp, and other metadata. Stream records have a lifetime of 24 hours; after that, they are automatically removed from the stream.

Benefits and Use Cases for DynamoDB:

Benefits:

1. **Performance and Scalability:** Scaling databases can be risky and challenging, as anyone who has worked in the IT business knows. By keeping track of how close your consumption is to the top bounds, DynamoDB gives you the option to auto-scale. This can help you prevent performance problems and cut expenses by enabling your system to react in accordance with the volume of data traffic.

2. **Access to control rules:** Effective access control becomes more crucial as data grows more particular and personal. You want to be able to quickly grant access control to the appropriate individuals without impeding other people's productivity. The table owner can exert more control over the data in the table thanks to DynamoDB's fine-grained access control.
3. **Persistence of event stream data:** Developers can receive and update item-level data before and after changes to that data using DynamoDB streams. This is since DynamoDB streams offer a time-ordered list of changes made to the data over the previous 24 hours. With streams, you can quickly update a read-cache, send incremental backups to Amazon S3, or make changes to a full-text search data store like Elasticsearch using the API.
4. **Time to Live:** Time-to-Live, also known as TTL, is a procedure that enables you to define timestamps for erasing out-of-date data from your tables. The data that has been designated to expire is removed from the database as soon as the timestamp expires. Developers can keep track of expired data and erase it automatically thanks to this functionality. This procedure aids in decreasing storage requirements and lowering the price of manual data deletion labor.
5. **Storage of inconsistent schema items:** DynamoDB can handle it if your data objects must be stored in inconsistent schemas. Since DynamoDB is a NoSQL data model, it is more efficient at handling less structured data than a relational data model, making it easier to handle query volumes and provide high-performance queries for item storage in ad hoc schemas.

Use Cases: Netflix and AB Testing in UI using Dynamo DB

In 2000 Netflix had only 300,000 subscribers but just within 20 years customer base increased exponentially. Netflix has more than 126 original series and films which is more than any other network and cable channel.

Netflix uses microservice architecture. Each application or microservice has its own code and resources in a microservice architecture. By nature, it won't share any of it with any other apps. According to estimates, Netflix uses 700 microservices to manage all of its various components as a whole: One microservice keeps track of all the shows you've watched, another takes the monthly fee from your credit card, another looks at your viewing patterns and employs algorithms to predict a list of films you'll likely enjoy, and a third provides the titles and pictures of these films to be displayed in a list on the main menu.

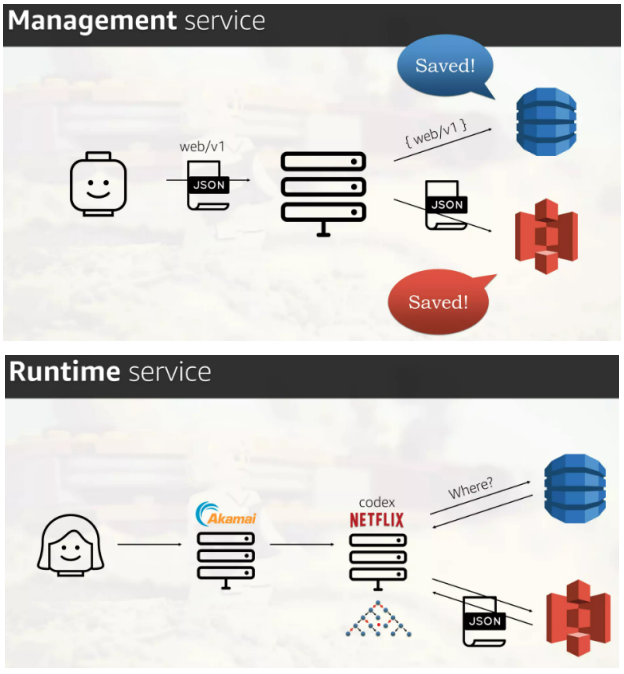
To run all this one has to have a massive network of servers which was once Netflix-owned. But as customers and content grew it became a back pain for Netflix. So they decided to run all this on someone else servers who will also deal with maintaining the hardware. The someone they chose is nothing but AWS (Amazon Web Services).

Here, the focus is mainly on how Netflix utilized A/B testing in its UI using DynamoDB. These tests were conducted on various elements in the UI like rows and columns, search engines, etc. Consider testing the search engine where the original testing variable bundle in jQuery and the new version of the original in react. Changing the file format will result in users having to download a larger bundle, increased time to download, more memory usage, and time to interact. So considering that depending on the experience one chooses, that particular bundle

will be downloaded. Even though that was the case, only one element of the UI is considered. That paired with other regional and engine-specific dependencies still led to large chunks of data being downloaded. So, how do we deal with conditional dependencies? What if we generate on-demand?

To generate on-demand, a dependency graph is generated. Repositories are fetched to a codex which generates a Json Artifact. A single Json artifact can generate multiple dependency graphs specific to that build. Parallel, dynamic inputs specific to the user form a set of flagging conditions. With the artifact and a set of dynamic inputs, we apply the truths to the conditional graphs to generate UI on-demand.

Scaling was still a problem. With so many artifacts and other such builds for platforms like TV, PS4, and Xbox, build management was needed. Primary issues which arose were, where to store so many artifacts. How to find them? As the codex was mission-critical, it shouldn't crash. So, to tackle all these issues, Netflix employs AWS. For storage, they utilized Amazon S3 Buckets and for metadata, they used AWS DynamoDB. One particular attribute of DynamoDB which was its stream, was very crucial in choosing it, as the streams showed which records were changed and how they were changed. In this built management service, the management plan created records for each artifact and stored them in AWS DynamoDB while the raw artifact was saved in Amazon's S3 bucket. Whenever there was a user request, the codex queries DynamoDB asking where the artifact is, which it then fetches from the S3 bucket.



One such approach was implemented, in which a particular build was activated by setting a flag in DynamoDB to state this build may take traffic very soon. DynamoDB returns a stream of builds that had a flag. The codex would then prime all those artifacts stored in the S3. Unfortunately, this wasn't possible as streams couldn't be consumed by the node.js client. Instead of the streams, a list of active services was returned to the codex which was then primed and used.

Environment Setup:

There are three ways of accessing DynamoDB:

1. AWS Management Console
2. AWS Command Line Interface(AWS CLI) or
3. DynamoDB API

I have done it via AWS CLI. AWS Command Line Interface (AWS CLI) is used to do all CRUD tasks, such as creating, reading, updating, and deleting tables, as well as to control and automate a variety of AWS services via the command line. It can also be used to combine utility scripts and Amazon DynamoDB activities. The AWS CLI with DynamoDB requires both an access key ID and a secret access key, which you must first get. The parameters for the DynamoDB operation are listed on the command line after the operation's name. JSON files can be read by the AWS CLI.

Step 1: AWS CLI set up
First, we need to install the AWS CLI on the local system which helps to interact with the AWS Management Console by the use of access key ID and secret access key.

Step 2: AWS configuration
We must set the command line interface's configuration after installing the AWS CLI on the local machine. We must specify the Access Key ID, Secret Access Key, AWS Region, and output format while configuring the AWS configuration. The AWS CLI establishes a connection with the specific AWS Account for the specified AWS Region based on the data to carry out the CRUD operations.

Apache Cassandra:

Apache Cassandra is an open-source NoSQL distributed database trusted by thousands of companies for scalability and high availability without compromising performance. Linear scalability and proven fault tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data. Masterless architecture and low latency mean Cassandra will withstand an entire data center outage with no data loss—across public or private clouds and on-premises. Cassandra’s support for replicating across multiple data centers is best-in-class, providing lower latency for your users and the peace of mind of knowing that you can survive regional outages. Failed nodes can be replaced with no downtime. To ensure reliability and stability, Cassandra is tested on clusters as large as 1,000 nodes and with hundreds of real-world use cases and schemas tested with replay, fuzz, property-based, fault-injection, and performance tests. Choose between synchronous or asynchronous replication for each update. Highly available asynchronous operations are optimized with features like Hinted Handoff and Read Repair. The audit logging feature for operators tracks the DML, DDL, and DCL activity with minimal impact on normal workload performance, while the faldstool allows the capture and replay of production workloads for analysis. Cassandra is

suitable for applications that can't afford to lose data, even when an entire data center goes down. There are no single points of failure. There are no network bottlenecks. Every node in the cluster is identical. Read and write throughput both increase linearly as new machines are added, with no downtime or interruption to applications. Cassandra streams data between nodes during scaling operations such as adding a new node or data center during peak traffic times. Zero Copy Streaming makes this up to 5x faster without nodes for a more elastic architecture, particularly in cloud and Kubernetes environments.

Core Components:

The following are the core components of Apache Cassandra:

1. Node: Node is the location where all the data and files are stored.
2. Datacenter: A data center is a group or collection of nodes. Together these nodes are categorized as data centers.
3. Cassandra Cluster: Cluster is another component of Cassandra. It is a group or collection of one or more data centers.
4. Commit log: All the write operations are written to the commit log in Cassandra, used for crash recovery.
5. Mem table: In Cassandra, the mem table has a memory resident data structure. Once the data is written to the commit log, it is written to the mem table on a temporary basis.
6. SSTable: is a disk file to which all the data from the mem table flushes as it reaches a certain threshold value.
7. Bloom filter: is a cache used for testing after every query. Bloom filters are quick, nondeterministic, algorithms in Cassandra.
8. CQL Table: Cassandra Query Language (CQL) is a collection of of ordered columns in a table. Each table comprises columns and primary key.
9. Gossip Protocol: These are communication protocols that aid in discovering, sharing location and data about the different nodes present in the cluster.

Benefits and Use Cases:

Benefits:

1. Peer-to-Peer Architecture: Instead of using a master-slave architecture, Cassandra uses a peer-to-peer architecture. As a result, Cassandra does not have a single point of failure. Additionally, every Cassandra cluster in any of the data centers can have any number of servers or nodes added to it. Any server can accept requests from any client since all the machines are on an equal playing field. Without a doubt, Cassandra has set the bar much higher than other databases with its solid architecture and exceptional features.
2. Elastic Scalability: The ability to elastically scale Cassandra is one of its main benefits. Scaling up or down a Cassandra cluster is simple. It's interesting that the Cassandra cluster may have any number of nodes added or removed without much disruption.

While scaling up or down, you don't need to restart the cluster or modify queries linked to Cassandra applications. This is the reason Cassandra is well known for having a very high throughput for most nodes. There is no downtime or other interruption to the applications when scaling takes place; read and write throughput both rise at the same time.

3. High Availability and Fault Tolerance: Data replication, which makes Cassandra highly available and fault-tolerant, is another impressive aspect of the system. Replication is the process of storing each piece of data several times. This is due to the fact that even if one node fails, the user ought to be able to easily obtain the data from a different place. Each row in a Cassandra cluster is replicated using the row key. You can choose how many replicas you want to make. Data replication can take place across various data centers, just like scaling. Cassandra gains high-level backup and recovery capabilities as a result.
4. High Performance: Cassandra was created with the fundamental goal of unlocking the potential of several multicore computers. This wish has come true thanks to Cassandra! Cassandra has shown outstanding performance when dealing with massive data volumes. Therefore, Cassandra is adored by businesses that deal with massive amounts of data every day and can't afford to lose that data.
5. Column Oriented: Cassandra features a very high-level, column-oriented data model. This indicates that Cassandra saves columns based on the names of the columns, enabling very rapid slicing. In Cassandra, column names may also contain the actual data, in contrast to typical databases where column names only contain metadata. In contrast to relational databases, which only have a small number of columns, Cassandra rows can have many columns. Cassandra has a robust data model at its disposal.
6. Tunable Consistency: Cassandra is a unique database due to features like Tunable Consistency. Consistency in Cassandra can be of two types: Strong consistency and Eventual consistency. You can choose any of these, depending on your needs. Once the cluster accepts the write, eventual consistency ensures that the client is approved. Strong consistency, on the other hand, means that any update is broadcast to every system or node where the specific data is located. Additionally, you are able to mix eventual and powerful consistency. For instance, you can choose eventual consistency for local data centers with low latency and strong consistency for faraway data centers with high latency.

Use Cases: Activision, keeping up with Cassandra

The Weather Channel, British Gas, Activision, a games developer, and William Hill all work in various industries with various clients and business strategies. The companies do, however, share one thing in common: they all rely on the Apache Cassandra NoSQL data platform to manage ever-growing data volumes and personalize client experiences.

Companies may now collect and use more data than ever thanks to the internet of things (IoT) and new technology. Higher expectations and more intense rivalry to remain ahead of the curve come with more data. The company began a project to improve the way it used data to deliver a better customer experience in 2011. It tried many different databases, including Oracle,

MongoDB, and Infobright, and often switched while it tried to find one that solved all its challenges.
In 2014, it tried out Apache Cassandra – and for the first time in five years, it didn't switch to something else, Kanouse says. Cassandra is an open-source, scalable NoSQL database that acts as a platform for applications that require fast performance and no downtime. With the release of Call of Duty: Advanced Warfare in 2014, Activision tried out a new system where, based on real-time data, it could message players with highly personalized communication to enhance the experience of the player and increase engagement. This involved large amounts of data, and according to Kanouse, the company "couldn't have done it without Cassandra".

Environment Setup:

We start by creating an EMR cluster, then installing Cassandra and YCSB as follows:

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

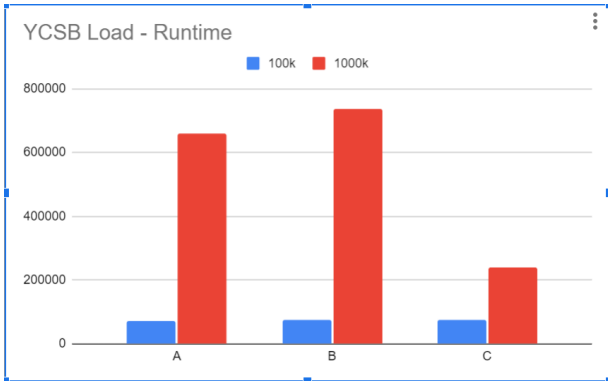
Install YCSB:
curl -O --location
https://github.com/brianfrankcooper/YCSB/releases/download/0.17.0/ycsb-0.17.0.tar.gz
tar xfvz ycsb-0.17.0.tar.gz

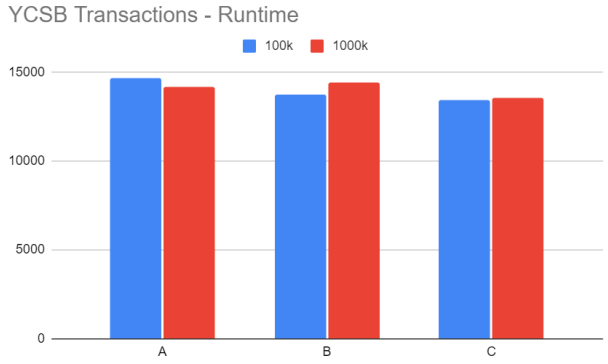
Experiments and Benchmarking:

Cassandra:
By setting up an EMR cluster from the AWS EMR tool with a master server and a client-server, we can begin working with Apache Cassandra. We will choose the Spark configuration rather than Core Hadoop. Installing Apache Cassandra 3.11.2 is done once the cluster connection is made, and the cqlsh prompt is queried.
We are going to use YCSB - Yahoo! Cloud Serving Benchmark utility to learn data modeling and performance optimization. The goal of the Yahoo Cloud Serving Benchmark (YCSB) project was to develop a methodology and a set of typical workloads for evaluating the efficiency of different "key-value" and "cloud" serving stores. The tool is frequently used to benchmark and contrast various systems. In the first experiment, we will create three workloads—A, B, and C—and apply them to two distinct datasets, each of which contains 100k and 1000k records.
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.

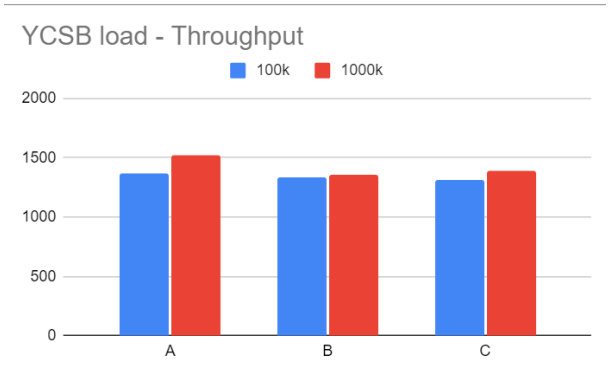
YCSB is installed on the EMR Cluster with specified configurations for Cassandra database technology.
The Cassandra server is then started, and the cqlsh prompt mode is used to build a keyspace with a user table for the YSCB. The yscb will load the data into the user table's 11 columns. Then, with a dataset of 100k records, we begin by loading three different types of workloads: Workload A, Workload B, and Workload C. The ycsb load command loads data into the user table we just created in Cassandra for each type of workload. We will perform workload transactions on each of these datasets after the dataset has been loaded. I have added all the queries and screenshots I have taken to output file.
The resultant data can be graphically represented as follows:

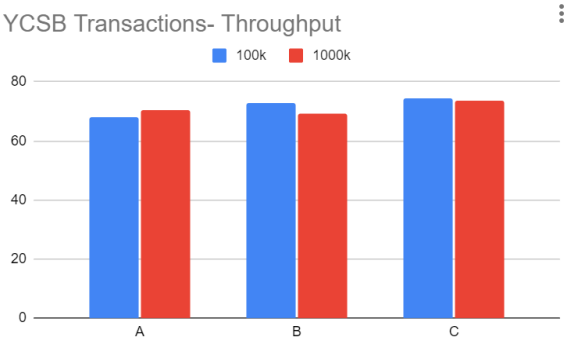
1. Runtime:



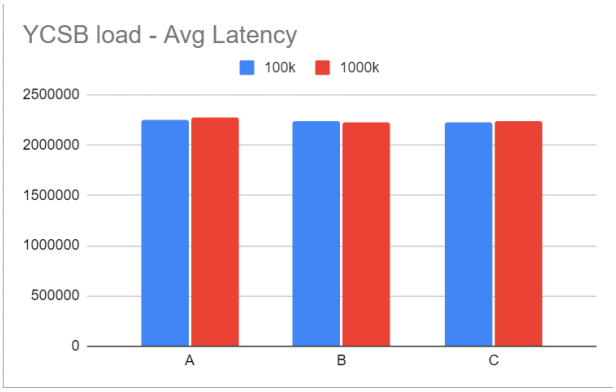


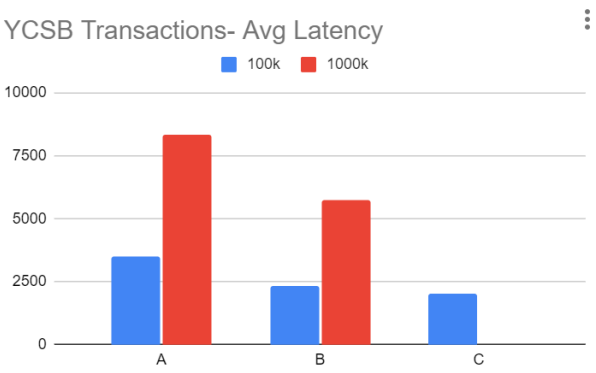
2. Throughput





3. Avg Latency :





The graphs shown above give us a brief overview of how apache cassandra would perform over Big Data. The run phase is an important part that shows how different kinds of workloads - update, write and read perform over large datasets.

DynamoDB:

Here, I have used 2 json files (movieratings and applemusic). Creation, insertion, scanning, selection, and other tasks are performed via python code. Using my AWS IAM user access, secret key, and region, I connected to dynamo DB. Boto3 was the python package that was used to interact with AWS DynamoDB.

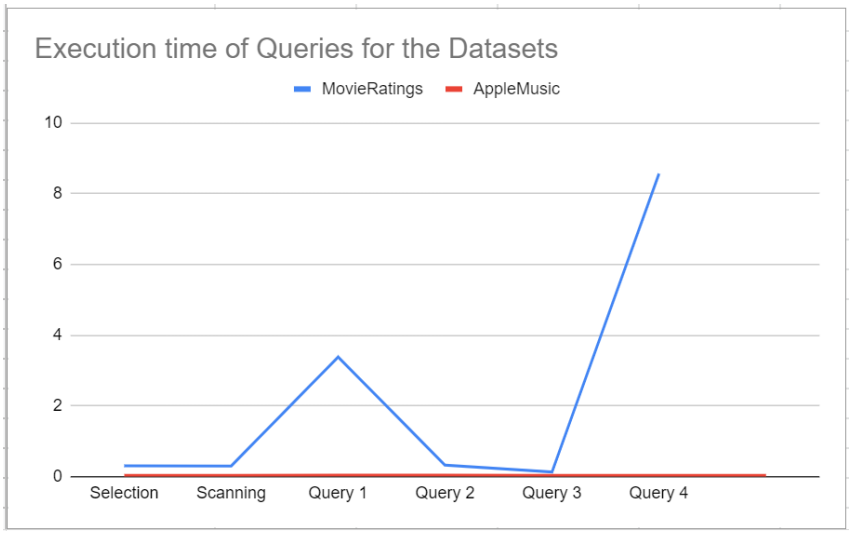
Following were the Tasks and Queries which was performed:

1. Table creation
2. Loading the dataset into the table
3. Insertion of single row
4. Selection of single row
5. Scanning
6. Query 1: collectionID/movieId = x
7. Query 2: CONTAINS
8. Query 3: release date/timestamp = x
9. Query 4: BEGINS WITH

Following is the data gathered:

DynamoDB	MovieRatings	AppleMusic
Item Count	500	92
Loading the dataset	42.9303711	7.89918828
Insertion	0.3102452755	0.03627824783
Selection	0.2999868393	0.03620409966
Scanning	3.383961201	0.04250836372
Query 1	0.3266015053	0.04335618019
Query 2	0.1343064308	0.03907489777
Query 3	8.57014513	0.04030299187
Query 4	-	0.03979444504

Above data represented in a graph:



Conclusion:

From our experiments on Apache cassandra, we can how it handles Big data with different types of workloads. We have used the YCSB benchmark to assess the performance of cassandra on various workloads and operations. This benchmark is an industry-standard for studying and comparing NoSQL database technologies. It offers thorough statistics that aid in our comprehension of the workings of these technologies. From the data, we can see how evenly Apache Cassandra manages 2 workloads with almost the same Throughput, transaction phase runtime, and load phase Average Latency.

The future work for this specific part of the project would be to benchmark performance on AWS DynamoDB 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.

Every day, businesses are inundated with large amounts of data coming from several sources at fast speeds. Relational databases are ineffective for processing and analyzing large amounts of unstructured and dynamic data. Incorporating NoSQL into a database management strategy improves corporate agility and flexibility when it comes to storing, accessing, and processing enormous amounts of data. NoSQL processes big data files and sets fast, improving performance, scalability, and real-time availability regardless of the predetermined schema architecture. We have carried out a variety of implementations on the chosen database technologies, which has improved our comprehension of Big Data and how NoSQL technologies are used to handle it.

References:

[1] Veronika Abramova and Jorge Bernardino. 2013. NoSQL databases: MongoDB vs Cassandra. DOI: https://doi.org/10.1145/2494444.2494447

[2] Douglas Kunda and Hazael Phiri. 2017. A comparative study of NoSQL and relational database. https://www.researchgate.net/publication/326019759_A_Comparative_Study_of_NoSQL_and_Relational_Database

[3] K. Kaur and R. Rani, "Modeling and querying data in NoSQL databases," in 2013 IEEE International Conference on Big Data, Silicon Valley, CA, USA, 2013 pp. 1-7.

[4]https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

[5]https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html

[6] https://rockset.com/blog/5-use-cases-for-dynamodb/

[7] https://docs.aws.amazon.com/documentdb/latest/developerguide/what-is.html

[8] https://aws.amazon.com/documentdb/features/

[9] https://www.linkedin.com/pulse/netflix-case-study-how-aws-made-work-seamless-janak-sawale/

[10] https://www.youtube.com/watch?v=k8PTetgYzLA&ab_channel=AmazonWebServices

[11] https://www.slideshare.net/AmazonWebServices/a-story-of-netflix-and-ab-testing-in-the-user-interface-using-dynamodb-dat308-reinvent-2017

[12] https://www.researchgate.net/publication/360743335_MongoDB_Case_Study_on_Forbes

[13] https://dl.acm.org/doi/pdf/10.1145/1807128.1807152

[14] https://github.com/brianfrankcooper/YCSB/wiki

[15] https://www.computerweekly.com/news/4500254443/Keeping-up-with-Cassandra-the-NoSQL-database

[16] https://cassandra.apache.org/_/index.html

[17] https://www.computerweekly.com/news/4500254443/Keeping-up-with-Cassandra-the-NoSQL-database