Project

Comparative Performance Analysis between Cassandra and MongoDB in tracking efficiency of NBA players.

DATA 225: Database Systems for Analytics

Professor: Ming-Hwa Wang

Team 4

Iqra Bismi, Shilpa Shivarudraiah, Saniya Lande

San Jose State University, CA

Table of Contents

List Of Figures

List Of Tables

Acknowledgment

We would like to express our gratitude to Professor Ming-Hwa Wang who taught us Database systems and motivated us to do a project related to NoSQL as it will be useful in the future for our careers. It helped us in doing a lot of Research on this topic and we learned a lot of new concepts. We would like to thank Professor for his guidance as well as for providing necessary information and clearing our doubts regarding the project as well as the coursework.
We would also like to thank our friends from San Jose State University for their encouragement and kind co-operation which helped us in completion of this project.

Abstract

With the evolution of communication technologies, there has been a significant increase in the volumes of data generated daily. An alternative approach for relational databases for storing and analyzing such huge volumes of data led to the emergence and the growth of NoSQL. There is an ever-increasing demand for NoSQL due to its various features like high scalability, dealing with unstructured, semi-structured and structured data, etc. Hence, we have done a performance comparison between Cassandra and MongoDB, the two top databases used in the leading companies to understand the advantages as well as shortcomings of each database and to select an appropriate database as per the requirement for a particular application. We used the NBA player dataset to perform these performance comparisons between two databases as on-sport analytics is widely used for tracking the performance of each player, each team as well as the game and we were interested in understanding which database can be suitable for on-field sport analytics. Firstly, we measured execution time for data loading as well as CRUD operations where we observed that MongoDB performs best in most of the operations except Update operation. Consistency (CAP theorem) was also studied for each database by changing their consistency levels and observing the performance of each database on different consistency levels. Latency patterns were observed for both the databases which showed us that cassandra performs much better than MongoDB when the number of operations increases. The results for each experiment are presented in this report.

Keywords - NoSQL; Performance; Cassandra; MongoDB;

1 Introduction

1.1 Objective

The main objective of this project is to implement, analyze and compare the performance of Cassandra and MongoDB databases to examine their behavior for tracking the efficiency of NBA players.

1.2 What is the problem

There are four main types of NoSQL namely Key-value NoSQL, Document NoSQL, Column-family NoSQL and Graph NoSQL. Depending upon their data storage model, each of these NoSQL have different types of databases like Amazon DynamoDB, Redis, Cassandra, MongoDB, etc. Finding a right database for a specific workload or an application can be very challenging due to the availability of a huge number of NoSQL databases. Hence, evaluating the performance of NoSQL in-order to select a right database as per the requirement of each application is the need of the hour. Each application communicates with the database using the operations like CREATE, READ, UPDATE, INSERT, DELETE, SCAN, etc. The runtime for each of these operations may vary. The amount of data, the type of the data to be handled, the run-time for CRUD operations can affect the performance of the NoSQL. We have selected MongoDB and Cassandra for evaluating their performance in terms of data storage, query handling and run-time for each operation. We chose Cassandra and MongoDB for our analysis as they are the most widely used databases in the industry due to its applicability in various areas.

1.3 Why this is a project related to this class

This project relates to our Database Systems class since we will be comparing two NoSQL databases in terms of their performance in tracking the efficiency of NBA players. We are not sure exactly which database performs better in tracking the efficiency of these players. We only have a high-level idea and theoretical knowledge of the modern technology NoSQL databases, so database systems is well suited to this problem as to analyze which database suffices the given requirement.

We will be using MongoDB (Document Oriented) and Cassandra (Column Oriented) NoSQL databases in this project for performance comparison, which is one of the many topics discussed in our Database Systems course work.

1.4 Why other approach is no good

The other approach to solve the problem we have stated above is to use Relational Databases (SQL). But in the current world where a huge volume of data is being generated, SQL databases have several limitations compared to NoSQL databases in handling such data. Relational databases handle only structured data in the form of rows and columns. Whereas NoSQL databases are capable of handling various formats of data which includes structured, semi-structured and unstructured data.

The notable limitations of SQL databases are:

- **Hardware/Scalability:** The norm for SQL databases is to scale-up vertically, where capacity can only be expanded by increasing capabilities, like RAM, CPU, and SSD, on the prevailing server or by migrating to a bigger, expensive one. We'll have to continually

increase drive space as our data grows and we will need faster machines to run evolving and more sophisticated technologies.

- **Data Normalization:** Developed at a time when the cost of information storage was high, relational databases try to negate data duplication. Each table has different information, and that they are connected and queried using common values. However, as relational databases get large, the lookups and joins required between numerous tables can slow down query performance. Particularly, when there is more interrelationship in the data; it makes the traversal along the relationship difficult and costly.
- **Rigidity:** A SQL database schema must be defined before use. Once defined, they're inflexible, and modifications are typically difficult and resource intensive. For that reason, substantial time must be invested in upfront planning, before the database is ever put into production. So, it follows that they’re only appropriate when all our data is additionally structured and we don’t expect much change, either in volume or data types.
- **Performance:** The performance of the SQL databases depends on the number of tables. If there are a greater number of tables, the response given to the queries are going to be slower. Additionally, more data presence not only slows down the machine, but it also eventually makes it complex to search out information. Thus, a SQL database is thought to be a slower database.
- **Complexity:** Although the SQL database is free from complex structuring, occasionally it's going to become complex too. When the amount of data in a SQL database increase, it eventually makes the system more complicated. Each and every data has been complex since the information is arranged using common characteristics.

- **Structure Limitations:** The fields that are present on a SQL database are with limitations. Limitations in essence means it cannot accommodate more information. Despite this, if more information is provided, it may lead to data loss. Therefore, it's necessary to explain the precise amount of data volume which the data field is going to be given.

Also, the relational database has a predefined schema which makes it difficult to handle any data which is not structured as per the defined schema.

1.5 Why this approach is better

As there is tremendous development in communication technologies, massive amount of data is generated daily through social networks and mobile communication. This type of huge data, commonly known as big data, is sometimes unstructured or nested. Due to this, there is impedance mismatch in using the traditional relational database in which data has to be stored in the form of relations and tuples. To store it in relational structure, it has to be first converted into relations/tuples. Also, traditional relational databases don't support nested structure. As a result, this was quite frustrating for developers.

Therefore, with further research in this field non-relational databases have gained popularity in storing both structured and unstructured data efficiently. It has become the most common and suitable choice for data storage and processing. In other words, NoSQL databases are bringing a more robust, reliable and scalable data management alternatively. Particularly, Cassandra and MongoDB are used by many companies and institutions around the globe due to their excellence performance when it comes to the characteristics of flexibility and scalability in data processing and storage.

Following are the important features of NoSQL databases which makes it much better as compared to SQL:

- **Multi-model:**

  As discussed above, relational databases can store data in predefined structure and format. It may be possible that currently we chose relational databases as per the application requirement although later on, the data can grow dramatically which may cause scalability issues in future with relational ones. Also, relational databases have fixed schema, so every time there is an update in the data, we have to first change the format/structure as per relational database structure and sometimes this even leads to having unnecessary columns. However, non-relational databases offer much better flexibility in storing unstructured data. There is no need to define schema and developers can focus on other requirements of the application rather than worrying about data schema. In addition to this, NoSQL are not restricted to any specific data type i.e. they can store strings, integers, Boolean values etc. Such as Redis (a key value NoSQL) support lists, hashes, sets. Hence, NoSQL is best suited when there is an agile requirement in the business and fast implementation is essential.

- **Easily Scalable:**

  Scaling can be done in two ways: horizontal scaling (out) and vertical scaling (up). Vertical scaling means getting bigger machines, RAM, memory and processors which makes the overall process costly. On the other hand, horizontal scaling means dividing the data into smaller chunks, this process is known as sharding. In traditional databases scaling up would

make the overall process expensive. Also, relational databases can be scaled horizontally but this makes the overall process complicated. The application has to keep track of which server to talk to for every bit of data which increases the downtime. Moreover, there is no control over consistency in the data.

On the other hand, NoSQL's are built master-slave and peer to peer architecture. Data can easily be scaled and distributed across multiple servers without any downtime. This scalability improves overall performance and high availability for update/ read. Also, consistency, availability and partition tolerance can be modified as per the requirement.

- **Distributed:**

  Non-relational databases are designed to run on clusters using commodity hardware. Also, to improve performance NoSQL support replication i.e., either through peer-to-peer model or master-slave model. Peer-to peer is useful when there is more read while master-slave is useful for update consistency (as the update can be written to master node only).

  However, as we know, relational database is not suitable to run on clusters and they follow ACID property (atomicity, consistency, isolation, durability) i.e. one cannot see partial updates. Although this maintains consistency, it also leads to low availability and low speed. Previously, SQL didn't support transactions and were quite popular amongst websites due to their high speed. Currently, because of ACID transactions many websites such as eBay had to forgo transactions so as to maintain high speed.

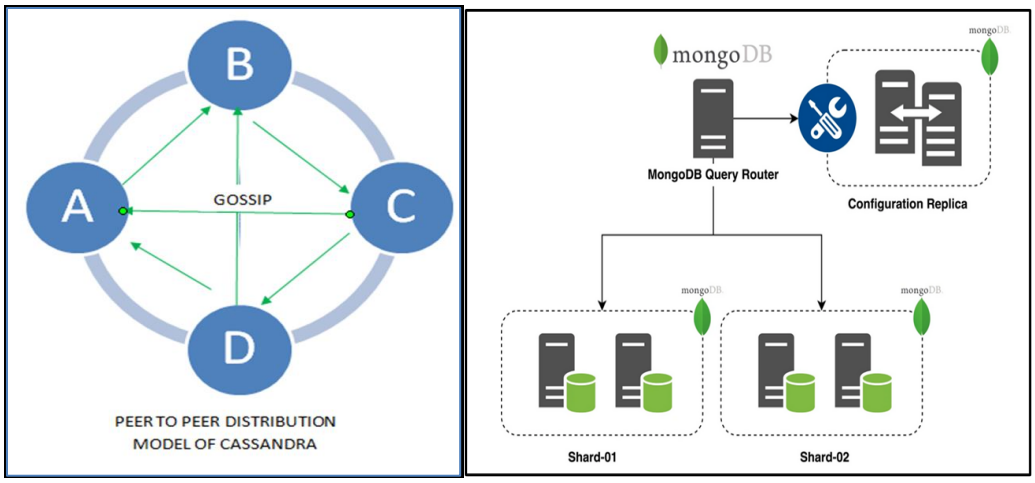


Figure 1: Peer-to-Peer Distribution for Cassandra and Master-Slave for MongoDB

- **Redundancy and zero downtime:**

  Hardware failure is a major concern for developers while defining an application. Nevertheless, the architecture of NoSQL is designed to handle network failure by allowing multiple copies of the data across various nodes. If one node is down, then data can be fetched from another node. This leads to zero downtime. Such as in MongoDB; even when the primary node is down, the remaining nodes in the replication set elect a new master. The application doesn't have to manage any issues related to node failure or electing a new primary node. Hence, with the help of replica sets MongoDB offers high availability.

  Similarly, in Cassandra the availability can be increased by configuring the quorum consistency i.e. write and read consistency level can be decreased to increase availability.

- **Big data applications:**

  As mentioned above, NoSQL databases don't put any restrictions on storing massive datasets. For this reason, they are much suitable for big data applications. It ensures that

storing data does not become a problem when all other aspects of the application are efficient and fast. This is particularly true in the case when most of the data is analytical and fast query performance is required.

While on the other hand, relational databases are much better in handling transactional and small data. They were never designed to handle data analytics along with huge data.

Therefore, because of these advantages of NoSQL most companies with huge data are choosing NoSQL over SQL such as Facebook is using Cassandra, Google is using Google big table: a column NoSQL. Also, many e-commerce websites are switching to document type NoSQL such as MongoDB where updating/removing product catalog is much easier.

1.6 Scope of Investigation

The scope of this project is to do an in-depth analysis on the performance of Cassandra and MongoDB as these two databases are widely used in many industrial applications. Also, the dataset consists of performance parameters of NBA players and other game measures. As we know, the NBA is a Professional Basketball League game in North America and is one of the most popular games. Hence, it is important to know on-field sports analytics in order to improve their game's statistics. The main purpose of on-field sports analytics is to focus on the statistics of games at various levels. Such as, the number of goals made by the teams, number of fouls made by a player, defensive or offensive rebounds etc. For this reason, on-sport analytics is widely used by many sport games as it helps teams and players to get insights about the game. In other words, it helps players to hone their skills, improve their winning strategies and goals percentage.

With reference to this, we shall be importing dataset in both the databases i.e., in Cassandra and MongoDB to explore which database is suitable as per the requirement. The comparison will be done on the following parameters:

- Data storage in both the databases shall be compared. In Cassandra, data will be stored in tables whereas in MongoDB data will be stored inside the collection. So, we shall be comparing which databases give a better understanding of the data with respect to the format of the dataset.
- Execution time of CRUD operation on both the databases shall be noted for comparison. Overall, the query performance of both databases shall be compared.
- Comparison with respect to Consistency and Availability. The CAP theorem of MongoDB is based on Consistency and fault tolerance while on the other hand, Cassandra has availability and fault tolerance. Suppose, if consistency of the data is top priority then MongoDB will have an upper edge as compared to Cassandra.
- Measure of average read and update latency for both the databases by increasing or decreasing the number of records.

2 Theoretical Bases & Literature Review

2.1 Theoretical background and Related Research to solve the problem

Research paper titled 'Distributed NoSQL Data Stores: Performance Analysis and a Case Study' evaluated and compared the performance of Cassandra, MongoDB and HBase in terms of their scalability, throughput and latency for the atomic operations on the structured dataset generated by YCSB benchmarking. Figure 1 below shows the comparison between throughput and latency for Cassandra, MongoDB and HBase. It was observed that as the number of nodes or workload increases, Cassandra shows highest throughput and the lowest latency for all the atomic operations (READ, WRITE, UPDATE) whereas MongoDB performs the best for low distributed setup.

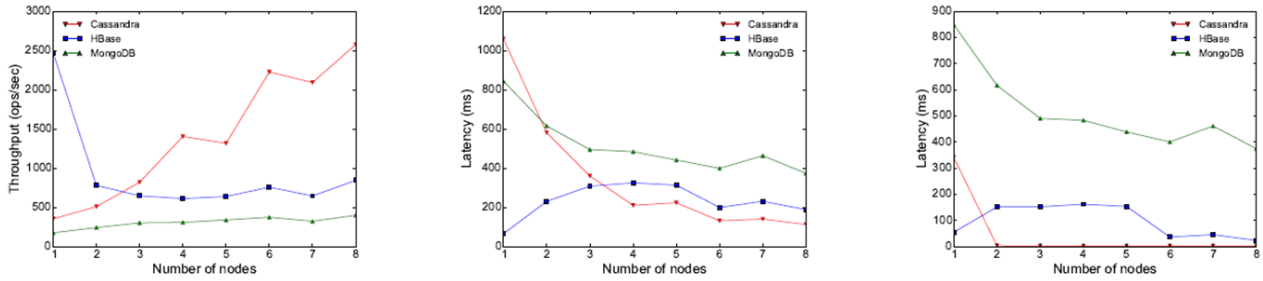


Figure 2: Performance comparison between Cassandra, HBase and MongoDB in terms of throughput and latency

Another paper titled 'Performance Benchmarking and Comparison of NoSQL Databases: Redis vs MongoDB vs Cassandra Using YCSB Tool' studies about MongoDB, Cassandra and Redis and compares their performance based on execution time for read, update, scan operations. After performing 100000 operations on 800000 inserted records, it was observed that a good cache memory directly impacts the execution time. Figure 2 shows overall run-time for write and read operations for these three databases. It was concluded that MongoDB outperformed Cassandra for

read operations as MongoDB loads its register mapping into the memory and scan operations. Cassandra showed difficulty for read and update operations.

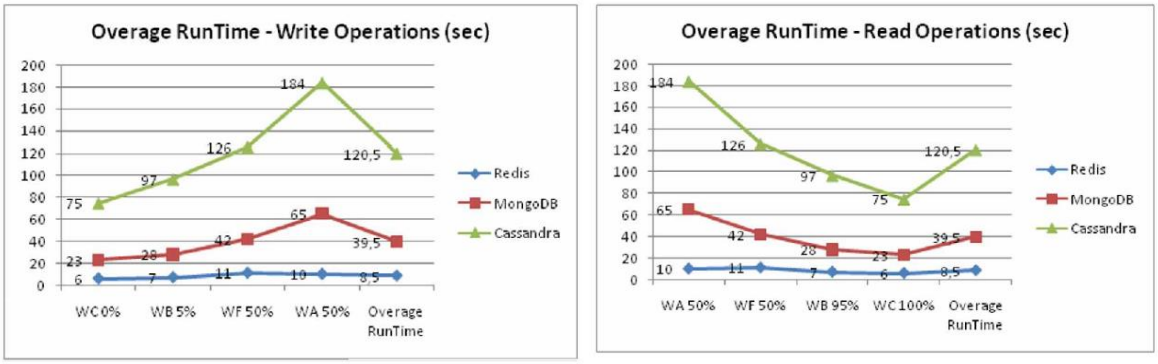


Figure 3: Performance comparison for read and write operations for Redis, MongoDB and Cassandra

Research paper titled 'Comparative study of MongoDB vs Cassandra in big data analytics' compares reads and write performance between MongoDB and Cassandra and has a conclusion that Reads are more efficient and effective in MongoDB's storage engine than they are in Cassandra. Cassandra's storage engine achieves very thorough writes since it stores data in an append-only arrangement. This makes extreme practice of spinning disk drives that have poor seek times, but can-do consecutive writes very rapidly.

But the limitation is that, for reads, we frequently need to scan over numerous varieties of an object to get the most recent version to return to the caller.

However, MongoDB updates data in place. This means it does extra random IO when writes are processed, but it has the benefit of being faster when processing reads, since we can find the precise location of an object on disk in one b-tree lookup

In MongoDB, all writes are noted in the master as the MongoDB cluster has only one master. Hence, its ability to write some novel data to the database is severely inadequate by the storage size of that solitary master node.

In the case of Cassandra, all the nodes in a cluster are masters as it follows a peer-to-peer paradigm. Hence, each node accepts multiple writes in parallel. So, to deal with huge data and multiple writes with an enhanced performance Cassandra is efficient.

The research paper titled 'Query driven implementation of Twitter base using Cassandra' suggested the advantages of using Cassandra for storing tweets. Currently, Twitter is using their internal database named "Manhattan" to store tweets. However, the authors of this paper had designed a data model for twitter to prove that read/ write operation is much more efficient in Cassandra. In that experiment, column families were created where each user had a unique column family storing all the tweets for that user. The column families had information such as id of the tweet, time of the tweet. Also, the tweets were sorted by decreasing order of time. Hence, after doing a few read operations it was concluded that Cassandra is efficient in read operations because of the sequential insertion of the data in the database. Likewise, for write operation it was observed that Cassandra is much more efficient because of peer-to-peer architecture and its upper hand on availability over consistency. Hence, node is always available for write operation. Therefore, as per the paper, Cassandra is the best option for storing huge time series data with high availability and performance.

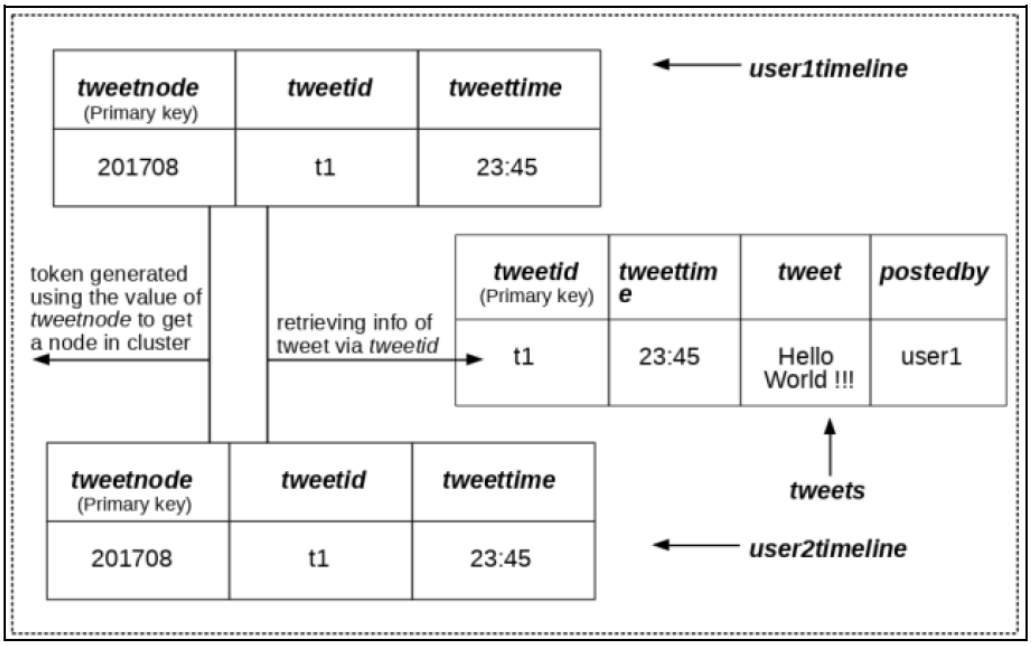


Figure 4: Storage and Retrieval of tweets

2.2 Advantages and disadvantages related to above research papers

The papers 'Distributed NoSQL Data Stores: Performance Analysis and a Case Study' and 'Performance Benchmarking and Comparison of NoSQL Databases: Redis vs MongoDB vs Cassandra Using YCSB Tool' studied the performance analysis between different databases which was done on a structured dataset generated by Yahoo's Cloud Serving Benchmark (YCSB). However, we can use real-world data to verify if we obtain similar results in terms of performance between these databases.

The paper 'Comparative study of MongoDB vs Cassandra in big data analytics' shows comparison in theoretical terms and not on the basis of any experimental research.

For the paper, 'Query driven implementation of Twitter base using Cassandra', research was mainly focused on tweets and no information about followers of each user was analyzed.

2.3 Solution to solve this problem

The research papers we studied showed performance analysis between different databases for synthetic data that was generated using YCSB benchmark. In our project, we are selecting a real-life scenario dataset to measure the efficiency of NBA players to make a performance comparison between MongoDB and Cassandra. This will give us better and meaningful insights about the effectiveness of both databases.

2.4 Literature review of NBA Players

Sports Analytics is an ever-growing field of data analytics which involves collection of historical and relevant data of the sports industry in-order to analyze the player performance, business performance, etc. which further can be used for improving decision making before and during the sports event.

National Basketball Player (NBA) Association is a men's professional basketball team in North America. It consists of 30 teams, 29 from the United States and 1 from Canada. Each team consists of 15 players with 5 players on the court at a time. It consists of two forwards (power forward, small forward), Shooting guard, point guard and a center. The game starts with a tip-off. Once a player is in possession of the ball, they have 24 seconds to shoot the ball in the opponent's basket. The team scores 2 points if they shoot the ball in opponents' basket from within the arc and score 3 points if they shoot the ball from outside of this arc. Any free throws are awarded a score of 1 point. If the team is unable to shoot the ball within 24 seconds, it results in a shot clock violation and the ball is passed on to the other team. The opposing team will try to take the ball off you by either blocking shots or by rebounding a missed shot or by stealing the ball in-order to score themselves. The game is played in 4 quarters each of 12 minutes duration. The team which gets

the highest points at the end of the game wins. There are no ties in basketball, hence overtime periods will be played to determine the winner. Figure 4 shows the basketball court layout.

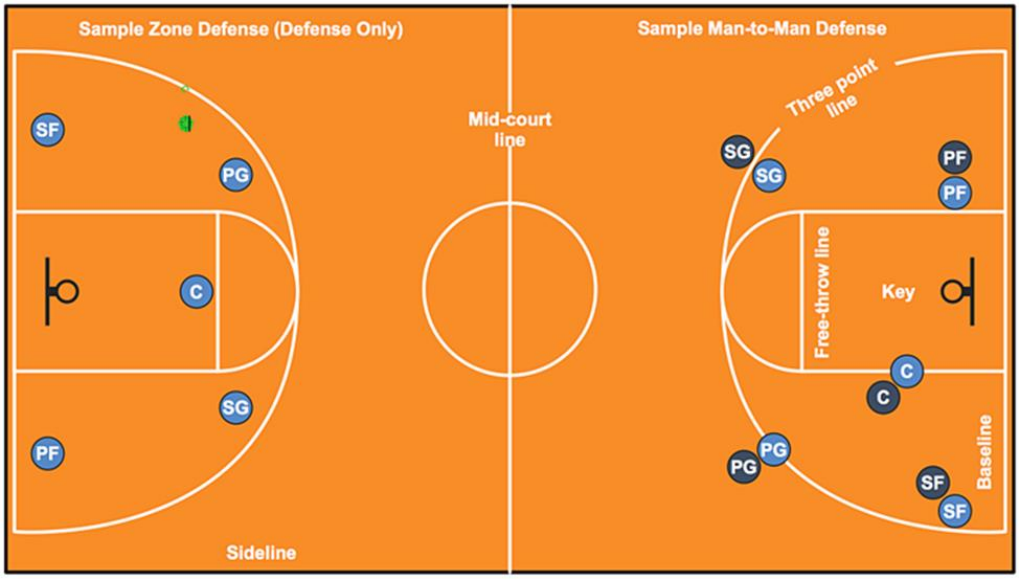


Figure 5: Basketball court layout

2.5 Where your solution different from others

- **Data Storage**

  In Cassandra, data is stored in the form of tables whereas in MongoDB, data is stored in collections/documents. Most research papers have not discussed data storage although we shall be exploring the advantages and disadvantages of types of data storage. Overall, we shall be analyzing which type of data (column oriented or document oriented) makes query easier.

- **CAP**

Previous research papers focus more on the performance of loading and atomic operations whereas in this project, we would like to see which database is a better option when it comes to focusing on consistency over availability or vice versa.

- **Real-life Scenario Dataset**

  Most papers did research on random data generated by using different tools such as YCSB or Google Colab Tool. In fact, in one of the benchmarking experiments data was structured and formatted for adequacy. However, in our project we shall be simulating a real-time dataset without any formatting and modification.

2.6 Why your solution is better

Most research papers have done comparisons between both databases on one or two parameters. Such as in the paper titled 'Comparative Performance Analysis of NoSQL Cassandra and MongoDB databases', researchers focused on CRUD and Update/read latency. Although in our project we shall be doing analysis at multiple levels which shall make our solutions unique. Following are the reasons which makes our solution different and better than others:

- Comparing the time taken to load data in both the databases.
- Doing analysis on the CAP Theorem: We shall be analyzing, which is best suited for NBA players dataset i.e. consistency or availability depending on our analytical requirements.
- Comparing the Execution time of CRUD operation on both the databases
- Exploring the average read and update latency of both the databases along with throughput rate.

3 Hypothesis

3.1 Single/multiple hypothesis

1. To test and verify if MongoDB performs better in CRUD operations and Cassandra performs better in write operations.

   From the research papers we studied, it was observed that MongoDB performs better for create, read and delete operations whereas Cassandra performs better for write operations. We would like to verify if the same results are obtained for our analysis.

2. To test and verify if Cassandra has low update and read latency as compared to MongoDB, when the load increases.

   As per the research paper, Cassandra has low average read/update latency and high throughput rate when the load increases. This is due to the fact that they have peer-peer replication. On the other hand, MongoDB latency increases exponentially when the number of operations is increased.

4 Methodology

4.1 How to generate/collect input data

The dataset on NBA players will be taken from Kaggle (https://www.kaggle.com). Kaggle dataset has good information regarding games statistics and players performance. It provides information about more than 60,000 games, 30 teams and 4.5k players and their statistics.We will be using the NBA player dataset to compare features of two most popular NoSQLs ie. Cassandra and MongoDB.

To get the relevant records on NBA player, sqlite zip file was downloaded from Kaggle (https://www.kaggle.com/agilesifaka/historic-nba-drafting-game-and-player analysis/data) which had different tables on NBA game (such as information about players, game statistics ie. number of games won/loss wrt year, data about different teams etc.)

To access the tables and records, sqlite file was uploaded in DB browser for SQLite and then multiple join queries were performed to get data on games, players, and teams. Also, to do an overall analysis on basketball,the above three tables were combined.Lastly, all these tables were downloaded in a local machine as csv files.

One file titled ‘Game_Analysis.csv’ contains the information related to each team and the matches played for a number of years along with their performance which includes the scores obtained, fouls made, which team won the game for which season, etc. Other file named ‘Player_Analysis.csv’ includes the data about each individual player such as his height, weight, the team he belongs to, the position in which he plays the game, etc. whereas the last file ‘Team_Analysis.csv’ includes the data about each team like the percentage win on home ground, percentage win on other ground, total losses, total salary for each team for different years, etc. All

this data can be used to do an overall on-field game analysis to increase the performance of each player, team as well as the game.

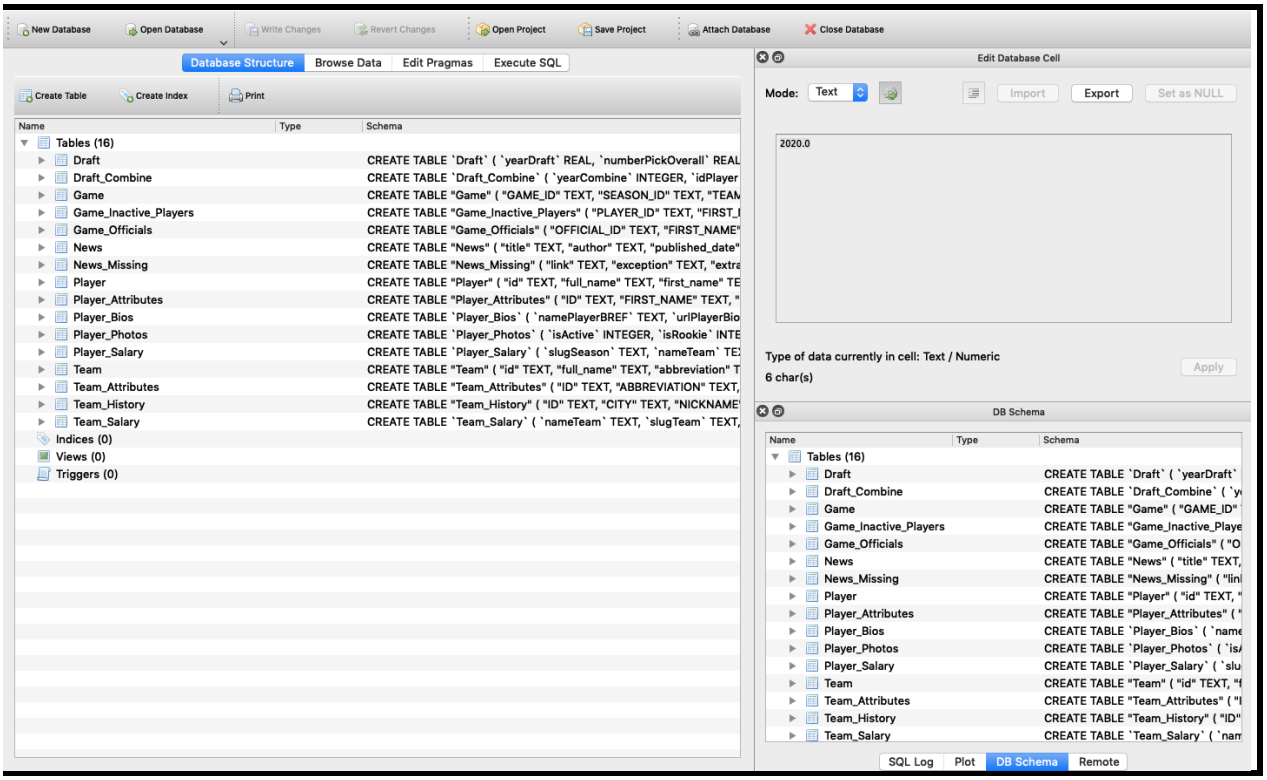
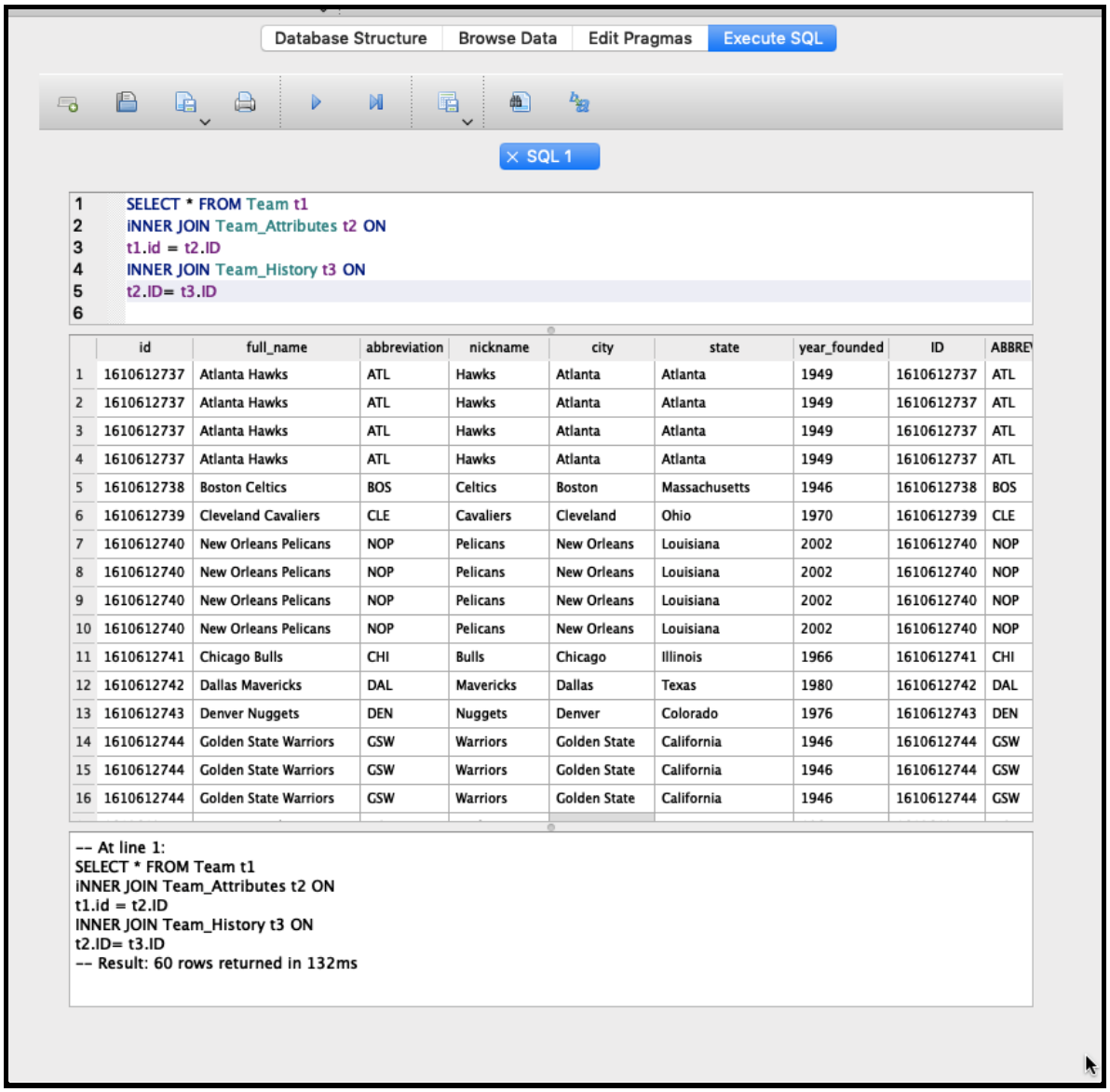


Figure 6: Viewing tables in SQLite browser



	id	full_name	abbreviation	nickname	city	state	year_founded	ID	ABBRE
1	1610612737	Atlanta Hawks	ATL	Hawks	Atlanta	Atlanta	1949	1610612737	ATL
2	1610612737	Atlanta Hawks	ATL	Hawks	Atlanta	Atlanta	1949	1610612737	ATL
3	1610612737	Atlanta Hawks	ATL	Hawks	Atlanta	Atlanta	1949	1610612737	ATL
4	1610612737	Atlanta Hawks	ATL	Hawks	Atlanta	Atlanta	1949	1610612737	ATL
5	1610612738	Boston Celtics	BOS	Celtics	Boston	Massachusetts	1946	1610612738	BOS
6	1610612739	Cleveland Cavaliers	CLE	Cavaliers	Cleveland	Ohio	1970	1610612739	CLE
7	1610612740	New Orleans Pelicans	NOP	Pelicans	New Orleans	Louisiana	2002	1610612740	NOP
8	1610612740	New Orleans Pelicans	NOP	Pelicans	New Orleans	Louisiana	2002	1610612740	NOP
9	1610612740	New Orleans Pelicans	NOP	Pelicans	New Orleans	Louisiana	2002	1610612740	NOP
10	1610612740	New Orleans Pelicans	NOP	Pelicans	New Orleans	Louisiana	2002	1610612740	NOP
11	1610612741	Chicago Bulls	CHI	Bulls	Chicago	Illinois	1966	1610612741	CHI
12	1610612742	Dallas Mavericks	DAL	Mavericks	Dallas	Texas	1980	1610612742	DAL
13	1610612743	Denver Nuggets	DEN	Nuggets	Denver	Colorado	1976	1610612743	DEN
14	1610612744	Golden State Warriors	GSW	Warriors	Golden State	California	1946	1610612744	GSW
15	1610612744	Golden State Warriors	GSW	Warriors	Golden State	California	1946	1610612744	GSW
16	1610612744	Golden State Warriors	GSW	Warriors	Golden State	California	1946	1610612744	GSW

Figure 7: Joining tables in SQLite browser

Firstly, data will be loaded in Python Pandas for better understanding of the data by following the step below:

- NBA file will be downloaded from kaggle website
- After unzipping the file the data set will be loaded into Python Pandas using python jupyter notebook and by using command “pd.read_csv”
- Then, we will do data wrangling for efficient analysis of our problem statement

Data wrangling methods consists of:

- ❖ Checking for null values/missing data
- ❖ Checking the data types for better understanding of the data
- ❖ Fixing the structural errors

Now the next step includes importing csv files in both the databases for further analysis.

For uploading data in MongoDB, following process shall be followed:

Sl.No.	Steps Involved to import data in MongoDB
Installation of MongoDB module	Installing pymongo through command prompt. This library will allow interaction with MongoDB database through python
Installation of MongoDB compass	MongoDB compass is required for making the connection to the local host. Also, it will be used for analysing MongoDB data.
Importing required modules in jupyter notebook	We shall be importing all necessary library in notebook MongoDB such as pandas, json and pymongo
Creating connection	After importing the modules, the next step is to connect to the mongo client in jupyter notebook by connecting to the default port and host.
Converting CSV files to Json format	All csv files will be converted to json format with the help of pandas and using the command **df.to_dict(orient="record")**

Creating database and collection	Next process, will be to create a database and collection in Mongo cluster
Inserting data inside the collection	All files will be imported into MongoDB database . Each file will be uploaded by creating a new collection in MongoDB compass and by using command **insert_many()**
View the imported data	Finally, we can view the imported data using MongoDB compass
Recording the time taken in inserting data	Also, the time taken to import files in MongoDB will be noted for further comparison.

Table 1: Steps involved to import data in MongoDB

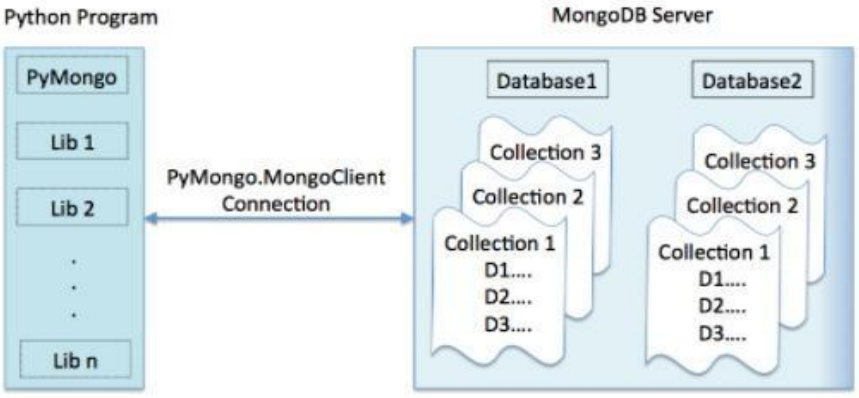


Figure 8: Data Loading in MongoDB

```
: 1 import pandas as pd
  2 import pymongo
  3 import json
```

```
: 1 client= pymongo.MongoClient("mongodb://localhost:27017")
```

```
: 1 df= pd.read_csv(r"/Users/iqrabismi/Desktop/Player_Analysis.csv")
```

```
: 1 df.head(5)
```

:

	PLAYER_ID	PLAYER_NAME	SCHOOL	COUNTRY	BIRTHDATE	PLAYER_CURRENT_AGE	HEIGHT	WEIGHT	SEASON_EXP	POSITION	...	AST	PLAYER_NM
0	203932	Aaron Gordon	Arizona	USA	1995-09-16	26	80	235	6	Forward	...	4.3	Aaron Gordon
1	1628988	Aaron Holiday	UCLA	USA	1996-09-30	25	72	185	2	Guard	...	1.7	Aaron Holiday
2	1627846	Abdel Nader	Iowa State	Egypt	1993-09-25	28	77	225	3	Forward	...	0.8	Abdel Nader
3	1629690	Adam Mokoka	Mega Basket	France	1998-07-18	23	76	190	1	Guard	...	0.4	Adam Mokoka
4	1629678	Admiral Schofield	Tennessee	United Kingdom	1997-03-30	24	77	241	1	Guard-Forward	...	0.5	Admiral Schofield

5 rows × 24 columns

```
In [11]: 1 Data = df.to_dict(orient="record")
```

```
/var/folders/fg/0cvyhxdj5v7503hv35z31yhr0000gn/T/ipykernel_47338/1688368244.py:1: FutureWarning: Using short name for
'orient' is deprecated. Only the options: ('dict', list, 'series', 'split', 'records', 'index') will be used in a fut
ure version. Use one of the above to silence this warning.
  Data = df.to_dict(orient="record")
```

```
In [14]: 1 database= client["project"]
```

```
In [15]: 1 print(database)
```

```
Database(MongoClient(host=['localhost:27017'], document_class=dict, tz_aware=False, connect=True), 'project')
```

```
In [16]: 1 database.test.insert_many(Data)
```

```
Out[16]: <pymongo.results.InsertManyResult at 0x115ea9040>
```

```
In [ ]: 1
```

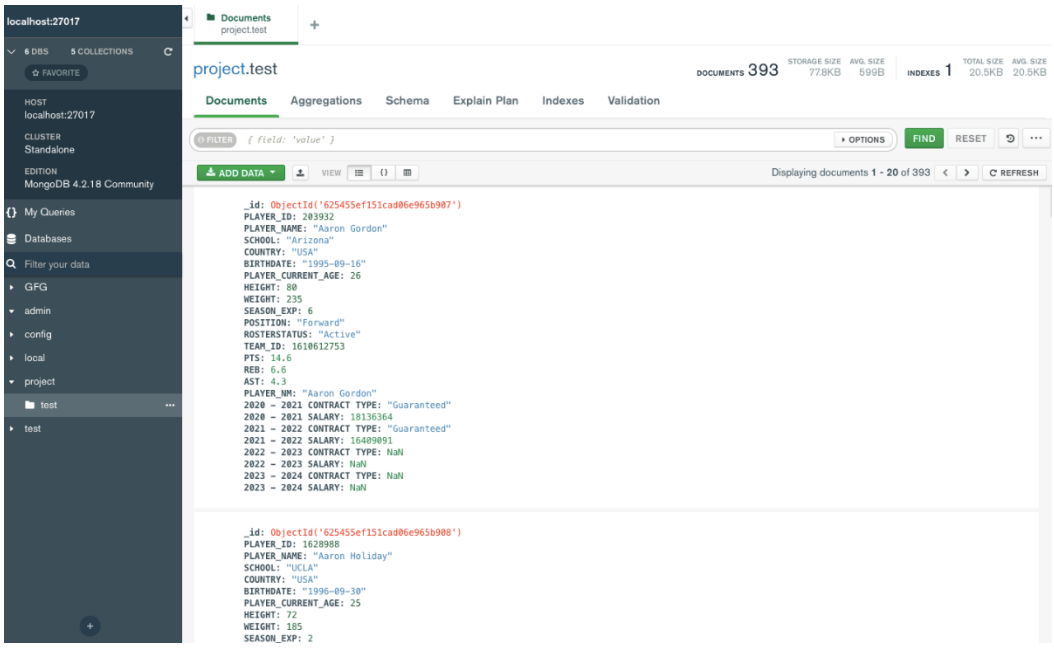


Figure 9: Loading .csv file in MongoDB

Cassandra doesn't have any configuration server for controlling the operation of other servers. There is no master-slave relationship. Although there is a ring of servers, each doing equal functions i.e. storing different parts of the data. Following are the steps for importing data into Cassandra.

S.No.	Steps Involved to import data in Cassandra
Installing Cassandra	Installation of Cassandra 4.0.3 on docker
Creating Keyspace and Column families	Will be creating a keyspace (preferably with a replication factor of 3) to store multiple column-families into it. Then the next step will be to create three column families specifying its primary key
Importing the CSV file	Import the data from .csv files to this column family using COPY command.
View the imported data	We can then view these column families using DESC command and selecting * from the keyspace and the specific column family.
Recording loading time	Cassandra shows the time required for loading the data directly in command prompt

Table 2: Steps Involved to import data in Cassandra

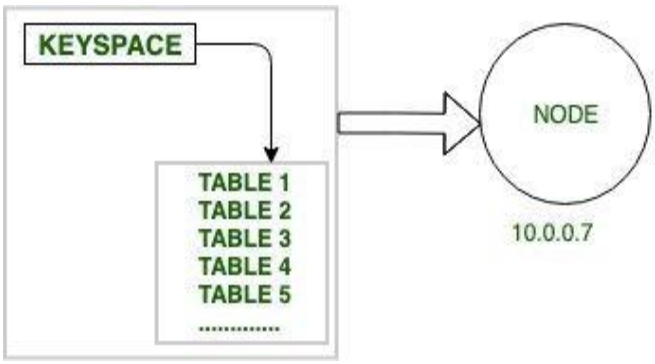


Figure 10: Data Loading in Cassandra

```
C:\Users\Saniya>
C:\Users\Saniya>docker-compose up -d
Recreating saniya-cassandra-1 ... done

C:\Users\Saniya>cqlsh -u cassandra -p cassandra

WARNING: console codepage must be set to cp65001 to support utf-8 encoding on Windows platforms.
If you experience encoding problems, change your console codepage with 'chcp 65001' before starting cqlsh.

Connected to My Cluster at 127.0.0.1:9042
[cqlsh 6.0.0 | Cassandra 4.0.3 | CQL spec 3.4.5 | Native protocol v5]
Use HELP for help.
WARNING: pyreadline dependency missing.  Install to enable tab completion.
cassandra@cqlsh> CREATE KEYSPACE BASKETBALL
   ...    WITH REPLICATION = {
   ...     'class' : 'NetworkTopologyStrategy',
   ...     'datacenter1' : 1
   ...    } ;
cassandra@cqlsh>
cassandra@cqlsh> use BASKETBALL;
cassandra@cqlsh:basketball> CREATE COLUMNFAMILY team_analysis (TEAM_ID int PRIMARY KEY, TEAM_NAME varchar, TEAM_CITY var
char) WITH COMPACT STORAGE;
cassandra@cqlsh:basketball> COPY basketball.team_analysis (TEAM_ID,TEAM_NAME,TEAM_CITY) FROM 'Team_Analysis.csv' WITH DE
LIMITER=',' AND HEADER=TRUE
              ...
              ... ;
Using 7 child processes

Starting copy of basketball.team_analysis with columns [team_id, team_name, team_city].
Processed: 29 rows; Rate:       5 rows/s; Avg. rate:       9 rows/s
29 rows imported from 1 files in 0 day, 0 hour, 0 minute, and 3.101 seconds (0 skipped).
cassandra@cqlsh:basketball> desc keyspaces;

basketball  system_auth         system_traces
calories    system_distributed  system_views
system      system_schema       system_virtual_schema

cassandra@cqlsh:basketball> select * from basketball.team_analysis;

 team_id    | team_city     | team_name
------------+---------------+-----------------------
 1610612739 |     Cleveland |    Cleveland Cavaliers
 1610612755 |  Philadelphia |     Philadelphia 76ers
 1610612761 |       Toronto |        Toronto Raptors
 1610612741 |       Chicago |         Chicago Bulls
```

Figure 11: Loading .csv file in Cassandra

4.2 How to solve the problem

4.2.1 Algorithm design

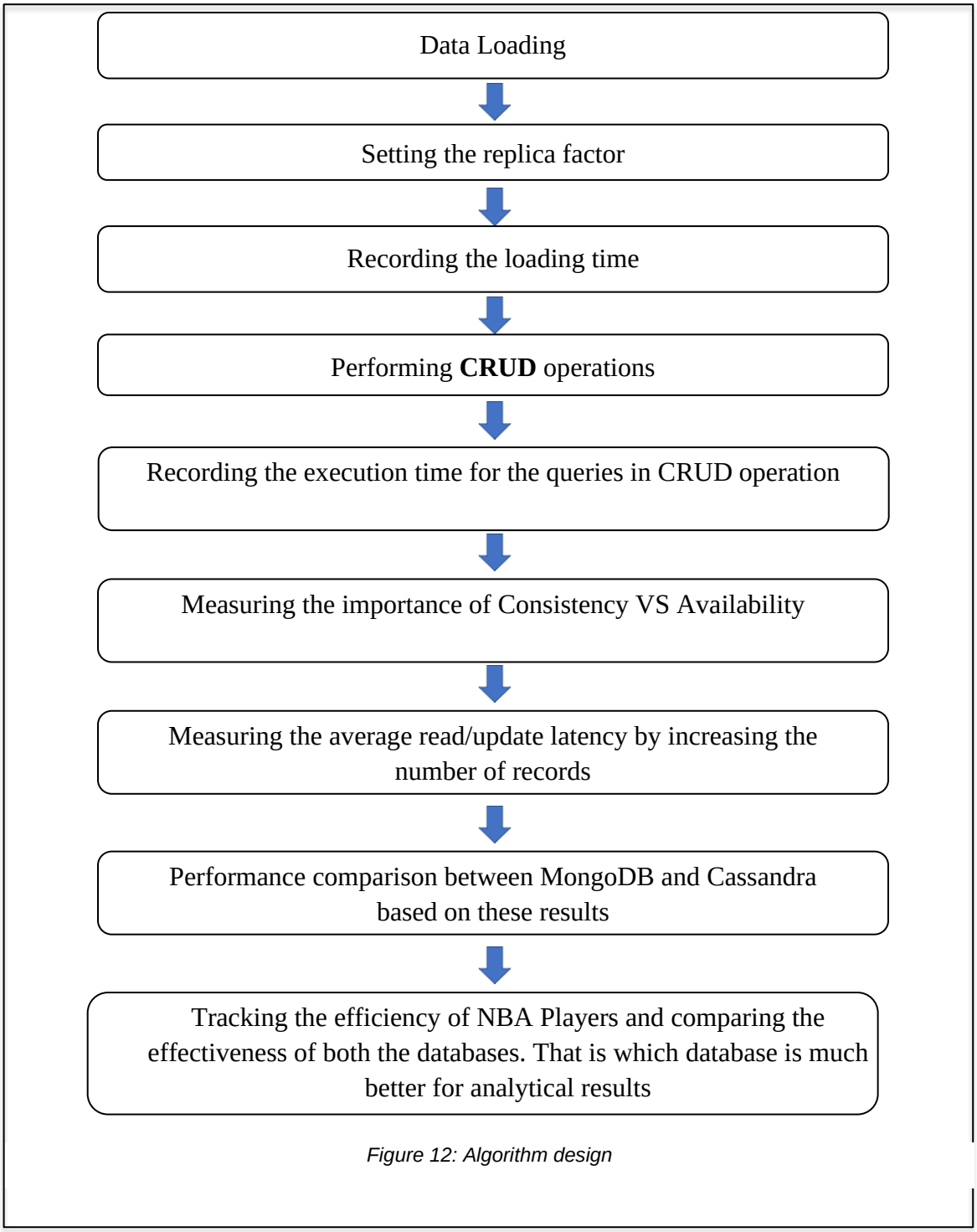


Figure 12: Algorithm design

4.2.2 Language used

Language	Purpose
CQL	For querying in Cassandra
Python - Pandas	For ETL and data wrangling
PyMongo	For querying and inserting data in MongoDB

Table 3: Languages used

PyMongo is a Python distribution containing tools for working with MongoDB, and is the recommended way to work with MongoDB from Python.

4.2.3 Tools used

Tools	Purpose
Jupyter notebook	For converting csv file to json format, data visualization and data cleaning
MongoDB Compass	Used for managing collection and document. Also, for querying, aggregating, and analyzing MongoDB data in a visual environment
MongoDB Atlas	To access the data virtually in the cloud, we will establish a connection to the cloud using MongoDB Atlas
Docker	Docker Image is one of the methods of installing Cassandra
Cassandra	For data storing and data analysis

Table 4: Tools used

Docker: (Cassandra Installation Method)

Docker Image is one of the methods of installing Cassandra. If you are a current Docker user, installing a Docker image is simple. You'll need to install Docker Desktop for Mac, Docker

Desktop for Windows, or have docker installed on Linux. Pull the appropriate image and then start Cassandra with a run command.

MongoDB Atlas:

MongoDB Atlas simplifies deploying and managing your databases while offering the versatility you need to build resilient and performant global applications on the cloud providers of your choice.

4.3 How to generate output

- **MongoDB**

After the data is loaded into MongoDB cluster, CRUD and other operations shall be performed as mentioned above. Execution time taken by each operation shall be noted. This will include two commands; explain() and db.collection.explain() methods that will give us information on query statistics. Also, the same can be seen in mongo compass on the "explain plan" tab. The time is given in milliseconds.

- **Cassandra**

When the .csv files are loaded into the created keyspaces, the time required to load each row from these files is displayed on cqlsh. Also, the rate at which each row is updated and the average rate of loading is displayed.Similarly, by using the command 'TRACING ON', the timestamp for each of the steps followed internally to execute the query operations like READ, UPDATE, DELETE will be displayed on cqlsh. We can then calculate execution time by taking the difference between the time-stamp from when the execution started and the time-stamp when the execution is completed. After the comparison based on various performance parameters between Cassandra and MongoDB, the efficiency of NBA players

will also be analyzed from the three datasets obtained from Kaggle by connecting the database to python.

4.4 How to test against hypotheses

We will compare the loading time and the execution time for each of the query operations like WRITE, READ, UPDATE, DELETE for Cassandra and MongoDB on the NBA player dataset to see which database processes the data faster in terms of the query performance. The throughput and the latency can also be compared based on these query operations. We will also study which database is more suitable for consistency and availability which will be based on the sharding and replication factor used. To test and analyze which database performs better on NBA dataset

.

5 . Implementation

5.1 Code

Initially, we uploaded csv files in Cassandra and MongoDB and implemented various CRUD operations to measure performance. Queries for each operation are as follows:

5.1.1 Data loading

Cassandra

Firstly, we created a keyspace called basketball using a query as shown in fig 13 and created a column family (table) for game_analysis, team_analysis, player_analysis, bb_analysis with all the column names and the datatype for each column. We then inserted the respective csv files into this column family. Fig 14 and 15 shows the query for creating table and copying data for game_analysis respectively.

```
CREATE KEYSPACE BASKETBALL
  WITH REPLICATION = {
   'class' : 'NetworkTopologyStrategy',
   'datacenter1' : 3
  } ;
```

Figure 13: Query for creating keyspace in Cassandra

```
CREATE COLUMNFAMILY game_analysis
(TEAM_ID int PRIMARY KEY,SEASON int,LOCATION varchar,WIN_COUNT int, AVERAGE_2_POINT_GOAL_EFFICIENCY float,AVERAGE_3_POINT_GOAL_EFFICIENCY float,AVERAGE_2_POINT_GOAL_PERCENTAGE float,
AVERAGE_3_POINT_GOAL_PERCENTAGE float,FREE_THROUGH_GOAL_EFFICIENCY float,FREE_THROUGH_GOAL_PERCENTAGE float,OFFENSIVE_REBOUND_PERCENTAGE float,AVERAGE_ASSISTS float,AVERAGE_PAINT_POINTS
float,AVERAGE_2ND_CHANCE_POINTS float,DEFENSIVE_REBOUND_PERCENTAGE float,AVERAGE_NUMBER_OF_STEALS float,AVERAGE_NUMBER_OF_BLOCKS float,AVERAGE_POINTS_AFTER_TURNOVER_PERCENTAGE
float,AVERGAE_FOULS float,AVERAGE_TURNOVER float,GAME_COUNT float,TOTAL_WIN_COUNT float,TOTAL_AVERAGE_2_POINT_GOAL_EFFICIENCY float,TOTAL_AVERAGE_3_POINT_GOAL_EFFICIENCY
float,TOTAL_AVERAGE_2_POINT_GOAL_PERCENTAGE float,TOTAL_AVERAGE_3_POINT_GOAL_PERCENTAGE float,TOTAL_FREE_THROUGH_GOAL_EFFICIENCY float,TOTAL_FREE_THROUGH_GOAL_PERCENTAGE
float,TOTAL_OFFENSIVE_REBOUND_PERCENTAGE float,TOTAL_AVERAGE_ASSISTS float,TOTAL_AVERAGE_PAINT_POINTS float,TOTAL_AVERAGE_2ND_CHANCE_POINTS float,TOTAL_DEFENSIVE_REBOUND_PERCENTAGE
float,TOTAL_AVERAGE_NUMBER_OF_STEALS float,TOTAL_AVERAGE_NUMBER_OF_BLOCKS float,TOTAL_AVERAGE_POINTS_AFTER_TURNOVER_PERCENTAGE float,TOTAL_AVERGAE_FOULS float,TOTAL_AVERAGE_TURNOVER
float,TOTAL_GAME_COUNT float,TEAM_NAME varchar,TEAM_SLUG varchar);
```

Figure 14: Creating column-family for game_analysis

```
COPY basketball.game_analysis
(TEAM_ID,SEASON,LOCATION,WIN_COUNT, AVERAGE_2_POINT_GOAL_EFFICIENCY,AVERAGE_3_POINT_GOAL_EFFICIENCY,AVERAGE_2_POINT_GOAL_PERCENTAGE,
AVERAGE_3_POINT_GOAL_PERCENTAGE,FREE_THROUGH_GOAL_EFFICIENCY,FREE_THROUGH_GOAL_PERCENTAGE,OFFENSIVE_REBOUND_PERCENTAGE,AVERAGE_ASSISTS,AVERAGE_PAINT_POINTS,AVERAGE_2ND_CHANCE_POINTS
,DEFENSIVE_REBOUND_PERCENTAGE,AVERAGE_NUMBER_OF_STEALS,AVERAGE_NUMBER_OF_BLOCKS,AVERAGE_POINTS_AFTER_TURNOVER_PERCENTAGE,AVERGAE_FOULS,AVERAGE_TURNOVER,GAME_COUNT,TOTAL_WIN_COUNT
,TOTAL_AVERAGE_2_POINT_GOAL_EFFICIENCY,TOTAL_AVERAGE_3_POINT_GOAL_EFFICIENCY,TOTAL_AVERAGE_2_POINT_GOAL_PERCENTAGE,TOTAL_AVERAGE_3_POINT_GOAL_PERCENTAGE,TOTAL_FREE_THROUGH_GOAL_EFFICIENCY
,TOTAL_FREE_THROUGH_GOAL_PERCENTAGE,TOTAL_OFFENSIVE_REBOUND_PERCENTAGE,TOTAL_AVERAGE_ASSISTS,TOTAL_AVERAGE_PAINT_POINTS,TOTAL_AVERAGE_2ND_CHANCE_POINTS,TOTAL_DEFENSIVE_REBOUND_PERCENTAGE
,TOTAL_AVERAGE_NUMBER_OF_STEALS,TOTAL_AVERAGE_NUMBER_OF_BLOCKS,TOTAL_AVERAGE_POINTS_AFTER_TURNOVER_PERCENTAGE,TOTAL_AVERGAE_FOULS,TOTAL_AVERAGE_TURNOVER,TOTAL_GAME_COUNT,TEAM_NAME
,TEAM_SLUG) FROM 'Game_Analysis.csv' WITH NULL='NULL' AND DELIMITER = ',' AND HEADER=TRUE;
```

Figure 15: Copying the csv file into column-family

```
Starting copy of basketball.game_analysis with columns [team_id, season, location, win_count, average_2_point_goal_efficiency, average_3_point_goal_efficiency, average_2_point
_goal_percentage, average_3_point_goal_percentage, free_through_goal_efficiency, free_through_goal_percentage, offensive_rebound_percentage, average_assists, average_paint_poi
nts, average_2nd_chance_points, defensive_rebound_percentage, average_number_of_steals, average_number_of_blocks, average_points_after_turnover_percentage, avergae_fouls, aver
age_turnover, game_count, total_win_count, total_average_2_point_goal_efficiency, total_average_3_point_goal_efficiency, total_average_2_point_goal_percentage, total_average_3
_point_goal_percentage, total_free_through_goal_efficiency, total_free_through_goal_percentage, total_offensive_rebound_percentage, total_average_assists, total_average_paint_
points, total_average_2nd_chance_points, total_defensive_rebound_percentage, total_average_number_of_steals, total_average_number_of_blocks, total_average_points_after_turnove
r_percentage, total_avergae_fouls, total_average_turnover, total_game_count, team_name, team_slug].
Processed: 2720 rows; Rate:     327 rows/s; Avg. rate:     636 rows/s
2720 rows imported from 1 files in 4.275 seconds (0 skipped).
cqlsh:basketball>
```

Figure 16: Showing that the data is being copied into column-family

MongoDB

Query for data Loading (PyMongo):

First, the csv file for game_analysis was uploaded into pandas dataframe using jupyter notebook which is shown in fig 17. After uploading and data cleaning, data was converted into json format. Then database and collection was created in MongoDB compass (commands explained below for reference). Finally, after creating the collection, the json file was uploaded in MongoDB compass using insert_Many command. Similar steps were repeated for other datasets as well.

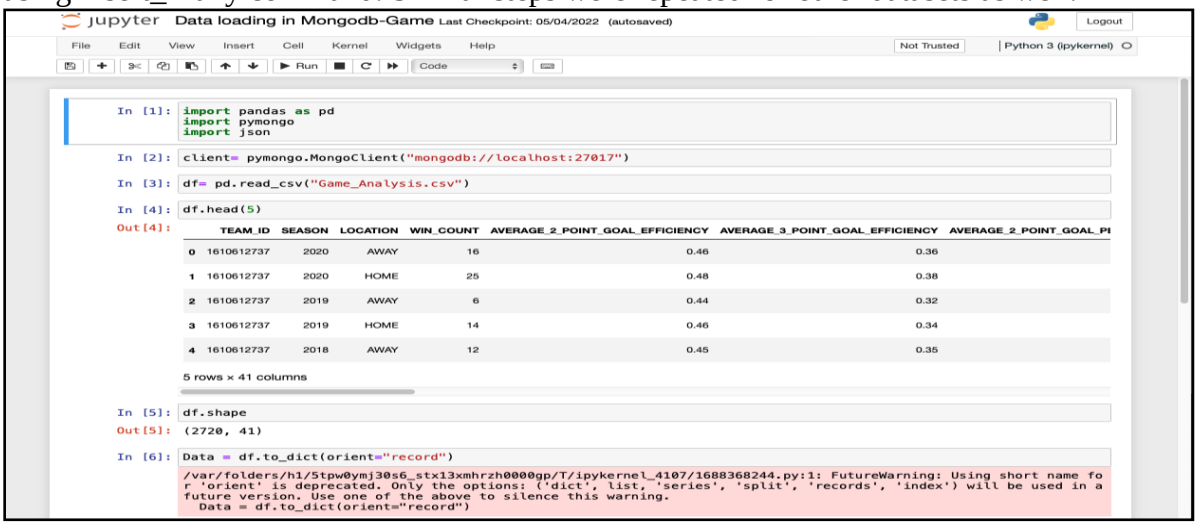


Figure 17: Data wrangling using Jupyter notebook

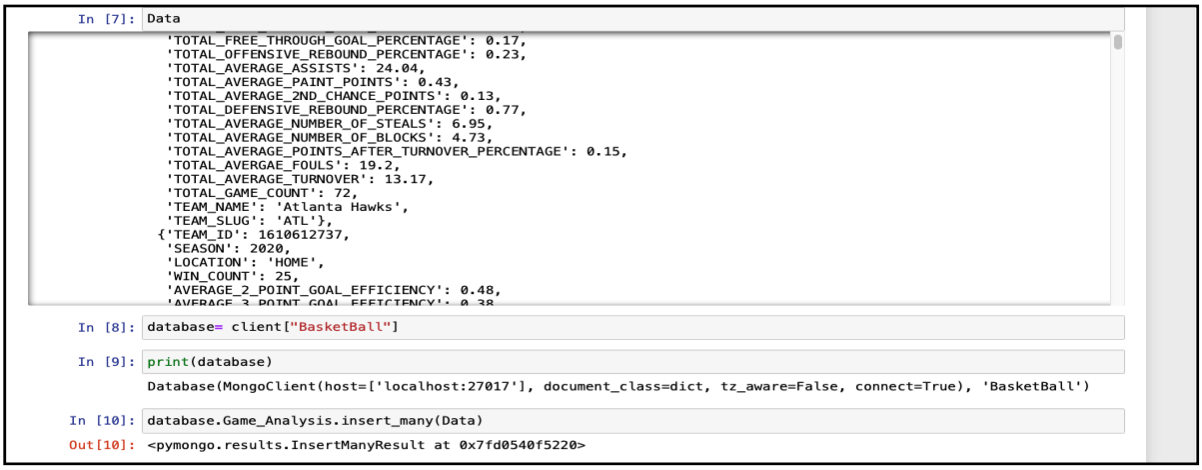


Figure 18:Creating database and collection and uploading JSON document in MongoDB compass.

5.1.2 Creating replica nodes on databases

Cassandra

We created three nodes for a cluster using commands as shown in fig 19.

```
Last login: Thu May  5 16:05:54 on ttys004

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ docker run --name cassandra-1 -p 9042:9042 -d cassandra:3.7
Unable to find image 'cassandra:3.7' locally
3.7: Pulling from library/cassandra
6a5a5368e0c2: Pull complete
97e4c6575710: Pull complete
d8288e3be5a2: Pull complete
d111f542073e: Pull complete
2549cfb76ce6: Pull complete
a375ee20c601: Pull complete
2e678e60bfc4: Pull complete
c2d5e7ed7dfc: Pull complete
21015df69ccb: Pull complete
a5b3a5d43f72: Pull complete
Digest: sha256:bce66127d99ff99020f2c5370b585744c851df7a21b42e7843195a0b143b94e7
Status: Downloaded newer image for cassandra:3.7
e9b3321e0c893c69fb66944639833378a9bfce5f8fb0fdb2fa6e3acb3aba5287
docker: Error response from daemon: driver failed programming external connectivity on endpoint cassandra-1 (98a4a494675dcec515256ab77e97b0749963ff5b63780595bb21520ef7bfde8e): Bind for
0.0.0.0:9042 failed: port is already allocated.
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ docker run --name cassandra-1 -p 9042:9042 -d cassandra:3.7
8b09f8f55f2d63973340175aa20a6044ee4353fade919e4110b5f0926e209875
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ INSTANCE1=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}" cassandra-1)
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ echo "Instance 1: ${INSTANCE1}"
Instance 1: 172.17.0.2
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ docker run --name cassandra-2 -p 9043:9042 -d -e CASSANDRA_SEEDS=$INSTANCE1 cassandra:3.7
9344d2f2cc63f5b513947f121df29095f5b62882ecb8867535f50c3809743ddc
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ INSTANCE2=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}" cassandra-2)
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ echo "Instance 2: ${INSTANCE2}"
Instance 2: 172.17.0.3
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ echo "Wait 60s until the second node joins the cluster"
Wait 60s until the second node joins the cluster
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ sleep 60
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ docker run --name cassandra-3 -p 9044:9042 -d -e CASSANDRA_SEEDS=$INSTANCE1,$INSTANCE2 cassandra:3.7
14c534422c194fbd8021c631a11f932259d36a4144bb9a77e0efbf567c4672f2
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ INSTANCE3=$(docker inspect --format="{{ .NetworkSettings.IPAddress }}" cassandra-3)
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ echo "Instance 3: ${INSTANCE3}"
Instance 3: 172.17.0.4
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$
```

Figure 19: Creating nodes for cassandra in terminal

A cassandra utility tool called nodetool was run on one cassandra instance of the docker terminal to see if all the 3 nodes have been created. UN stands for Up and Normal which means that the created nodes are healthy. Addresses for the three nodes can be seen from the fig 20.

```
(base) Shilpas-MacBook-Air:~ shilpashivarudraiah$ docker exec cassandra-3 nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns (effective)  Host ID                               Rack
UN  172.17.0.3  102.51 KiB  256          69.3%             f89a49b5-0728-4a4f-9c51-dc119e3e17c0  rack1
UN  172.17.0.2  107.96 KiB  256          68.6%             c1d503e7-b34d-427d-b536-630989b63969  rack1
UN  172.17.0.4  88.72 KiB  256          62.1%             2320e6b7-0343-43bc-becb-16752c3c6851  rack1
```

Figure 20: Checking status of created nodes

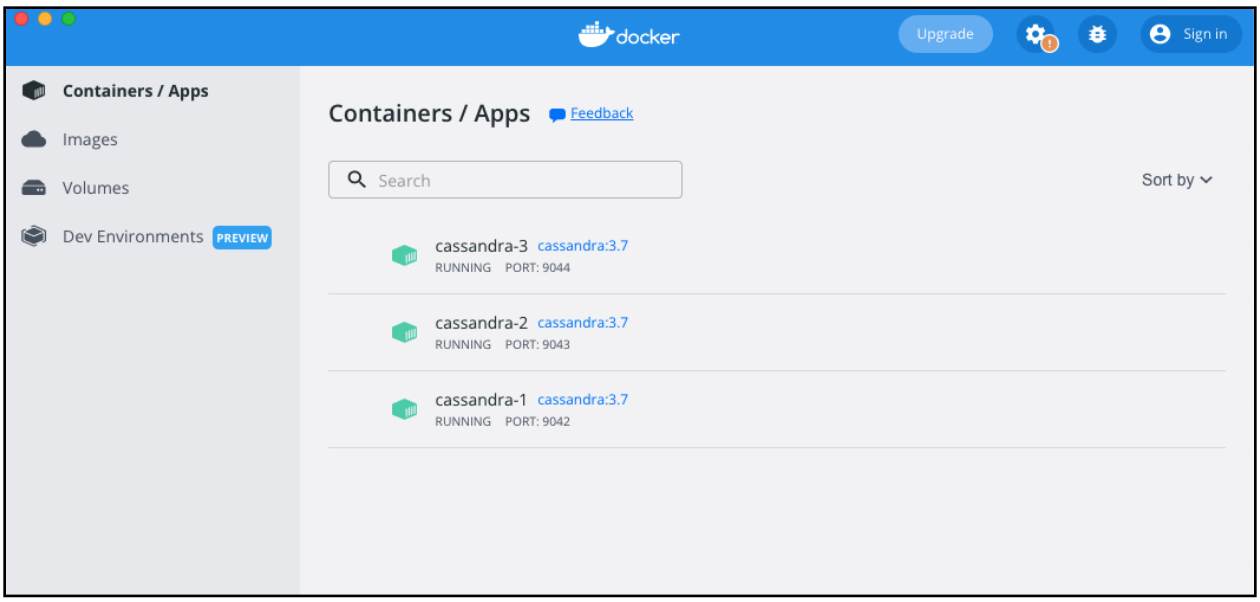


Figure 21: Viewing 3 nodes created on the docker

We then connected to one of the nodes (cassandra-1) using the cqlsh in the docker terminal (CLI) as shown in fig 22.

Figure 22: Connecting to node 1

MongoDB

Steps to create nodes in MongoDB are as shown in table 5.

Steps:	Command used
Running the mongo shell on terminal	Mongo d
Created data directories for MongoDB instances	mkdir -p /srv/MongoDB/rs0-0 /srv/MongoDB/rs0-1 /srv/MongoDB/rs0-2
Created the Replica Set with port : 27018, 27019, 27020	sudo mongod --port 27018 --dbpath /data/db/rs0-0 --replSet rs0 --bind_ip localhost sudo mongod --port 27019 --dbpath /srv/MongoDB/rs0-0 --replSet rs0 --bind_ip localhost sudo mongod --port 27020 --dbpath /srv/MongoDB/rs0-2 --replSet rs0 --bind_ip localhost
Connecting to the primary node	mongo – port 27018
Use the rs.initiate() command to start the replication process	rs.initiate({_id: "rs0",members: [{ _id: 0, host : "127.0.0.1:27018" },{ _id: 1, host : "127.0.0.1:27019" },{ _id: 2, host : "127.0.0.1:27020" }]})
Check the status of nodes	rs.status( )

Table 5: Steps to create nodes in MongoDB

Figure 23: Connecting to primary node (Port 27018)

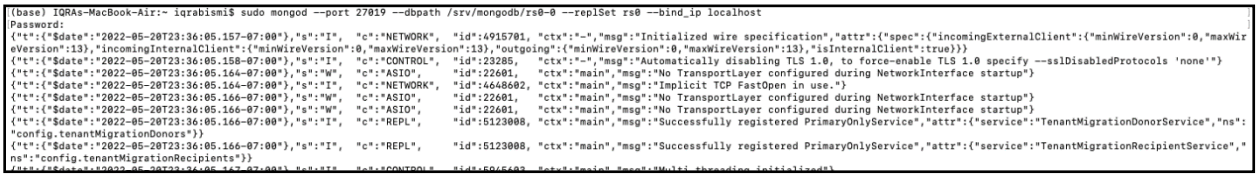


Figure 24: Connecting to one of the replica nodes for MongoDB

```
rs0:PRIMARY> rs.initiate(
...     {
...         _id: "rs0",
...         members: [
...             { _id: 0, host : "127.0.0.1:27018" },
...             { _id: 1, host : "127.0.0.1:27019" },
...             { _id: 2, host : "127.0.0.1:27020" }
...         ]
...     }
... )
{
        "ok" : 0,
        "errmsg" : "already initialized",
        "code" : 23,
        "codeName" : "AlreadyInitialized",
        "$clusterTime" : {
                "clusterTime" : Timestamp(1653115398, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        },
        "operationTime" : Timestamp(1653115398, 1)
}
```

Figure 25: Replication process for MongoDB

Figure 26: Status of nodes for MongoDB

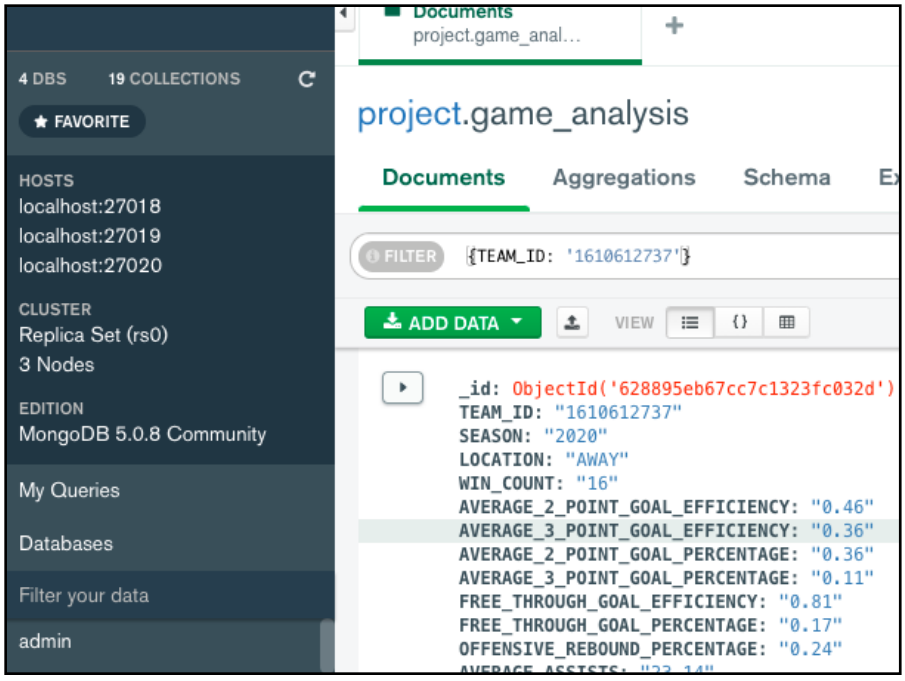


Figure 27:MongoDB connected to three nodes

5.1.3 CRUD Operations

This section contains all the queries that were used for performing CRUD operations fro Cassandra as well as MongoDB

Cassandra (CQL queries)

1. **Read**

To read one record:

```
SELECT * FROM team_analysis WHERE TEAM_ID = 1610612737
```

To read multiple records:

```
SELECT * FROM game_analysis WHERE TEAM_ID = 1610612737 ALLOW FILTERING;
```

2. **Update**

To update a single record:

```
UPDATE game_analysis SET TEAM_SLUG = 'BTL' WHERE id = 3;
```

To update multiple records:

```
UPDATE game_analysis SET TEAM_SLUG = 'FFF' WHERE ID IN (1,2,3,4,5,6,7,8,9,10)
```

3. **Delete**

To delete single record:

```
DELETE FROM game_analysis WHERE id = 3;
```

To delete entire column family(table):

```
DROP COLUMNFAMILY player_analysis;
```

4. **Insert**

Insert values into team_analysis table

```
INSERT INTO basketball.team_analysis
(TEAM_ID,TEAM_NAME,TEAM_CITY,TEAM_SLUG,YEAR_FOUNDED,SALARY_2020_
2021,SALARY_2021_2022,SALARY_2022_2023,SALARY_2023_2024,SALARY_2024_202
5,SALARY_2025_2026,HOME_WIN_PERCENT ,HOME_LOSS_PERCENT
,AWAY_WIN_PERCENT,AWAY_LOSS_PERCENT,TOTAL_WIN_PERCENT,TOTAL_LOS
S_PERCENT) VALUES (1610612778, 'Charlotte Hornets', 'Charlotte', 'CAA',
1988,108219000,8125900,44614400,31500000,0,0,0.54,0.46,0.3,0.7,0.45,0.56)
```

Similar queries were done for each CRUD operations for all the tables in the basketball keyspace.

MongoDB

Queries were implemented in MongoDB compass and by default db on mongosh is "test". Then command "use database db_project" was given to change to the relevant database.

Following are the queries for CRUD:

1. **Read:**

```
db.game_analysis.find({Team_ID: "1610612737"})
```

2. **Update:**

We used the query below to update TEAM_SLUG to “BTL”.

To update a single record

```
db.game_analysis.updateOne({TEAM_ID:"1610612738"},{$set:{TEAM_SLUG:"BTL"}
,{upsert: true})
```

To update multiple records

```
db.game_analysis.updateMany({TEAM_ID:"1610612738"},{$set:{TEAM_SLUG:"BTL
"})
```

3. **Delete:**

To remove multiple records from collection

```
db.game_analysis.remove({ })
```

To remove single record from collection

```
db.game_analysis.remove({Team_ID: “1610612738”}, { justOne:true})
```

4. **Insert Operation:**

To insert a single record in collection

```
db.game_analysis.insertOne({id:272222,Team_ID:1610612699,SEASON:2022,LOCATI
ON:"Away"})
```

To insert many records in collection

```
db.game_analysis.insertMany([{id:272222,Team_ID:1610612699,SEASON:2022,LOCATION:"Away"},{id:272222,Team_ID:1610612700,SEASON:2022,LOCATION:"Away"}])
```

5.1.4 CRUD Operations for different consistency levels (CAP)

Cassandra

1. Read

Consistency ONE: Only one of the three Cassandra nodes in the data center must respond to a read operation for the operation to succeed

```
cqlsh:basketball> CONSISTENCY ONE
Consistency level set to ONE.
cqlsh:basketball> SELECT * FROM game_analysis WHERE TEAM_ID = 1610612738 ALLOW FILTERING;
```

Figure 28: Query for READ consistency ONE Cassandra

Consistency TWO : Returns the response from two of the nodes

```
cqlsh:basketball> CONSISTENCY TWO
Consistency level set to TWO.
cqlsh:basketball> SELECT * FROM game_analysis WHERE TEAM_ID = 1610612738 ALLOW FILTERING;
```

Figure 29: Query for READ consistency Quorum Cassandra

Consistency ALL: All the Cassandra nodes in the data center must respond to a read operation for the operation to succeed

```
cqlsh:basketball> CONSISTENCY ALL
Consistency level set to ALL.
cqlsh:basketball> SELECT * FROM game_analysis WHERE TEAM_ID = 1610612738 ALLOW FILTERING;
```

Figure 30: Query for READ consistency ALL Cassandra

2. Write

Update:

Consistency ONE: Only one of the three Cassandra nodes in the data center must respond to an update operation for the operation to succeed

```
cqlsh:basketball> CONSISTENCY ONE
Consistency level set to ONE.
cqlsh:basketball> UPDATE game_analysis SET TEAM_SLUG = 'BTL' WHERE Id = 1;
```

Figure 31: Query for write Consistency ONE Cassandra

Consistency QUORUM : Majority of the cassandra nodes must respond to an update operation for the operation to succeed

```
cqlsh:basketball> CONSISTENCY QUORUM
Consistency level set to QUORUM.
cqlsh:basketball> UPDATE game_analysis SET TEAM_SLUG = 'BTL' WHERE Id = 1;
```

Figure 32: Quesry for write Consistency Quorum for Cassandra

Consistency ALL : All the Cassandra nodes in the data center must respond to an update operation for the operation to succeed

```
cqlsh:basketball> CONSISTENCY ALL
Consistency level set to ALL.
cqlsh:basketball> UPDATE game_analysis SET TEAM_SLUG = 'BTL' WHERE Id = 1;
```

Figure 33: Query for write consistency ALL Cassandra

Insert

Consistency One: Only one of the Cassandra nodes in the data center must respond to an insert operation for the operation to succeed

```
CONSISTENCY ONE
Set to QUORUM
INSERT INTO game_analysis(id, TEAM_ID, SEASON, LOCATION) VALUES   (2725,1610612699,2022,'HOME');
```

Figure 34: Query for INSERT consistency ONE Cassandra

Consistency Quorum: Majority of the Cassandra nodes in the data center must respond to an insert operation for the operation to succeed

```
cqlsh:basketball> CONSISTENCY QUORUM
 Consistency level set to QUORUM.
cqlsh:basketball> INSERT INTO game_analysis(id, TEAM_ID, SEASON, LOCATION) VALUES (2725,1610612699,2022, 'HOME');
```

Figure 35: Query for Insert Consistency Quorum Cassandra

Consistency All: All the Cassandra nodes in the data center must respond to an insert operation for the operation to succeed

```
cqlsh:basketball> CONSISTENCY ALL
Consistency level set to ALL.
cqlsh:basketball> INSERT INTO game_analysis(id, TEAM_ID, SEASON, LOCATION) VALUES (2726,1610612699,2022, 'HOME');
```

Figure 36: Query for Insert Consistency ALL Cassandra

MongoDB

1. **Read Consistency Levels**

Read Consistency Level: "local":

"Returns data from the instance with no guarantee that the data has been written to a majority of the replica set members"

```
db.basketball_analysis.find({Team_ID: "1610612737"}). readConcern("local")
```

Read Consistency Level: "majority":

"Returns the data that has been acknowledged by a majority of the replica set members"

```
db.basketball_analysis.find({Team_ID: "1610612737"}). readConcern("majority")
```

Read Consistency Level: "available":

"Returns data from the instance with no guarantee that the data has been written to a majority of the replica set members.

```
db.basketball_analysis.find({Team_ID: "1610612737"}). readConcern("available")
```

2. **Write Consistency Levels**

For Updates

Write Consistency Level: 1:

"Requests acknowledgement that the write operation has propagated to the standalone mongod or the primary in a replica set"

```
db.basketball_analysis.updateMany({TEAM_ID:"1610612737"},{$set:{TEAM_SLUG:
"BTL"}},{writeConcent:{w:1}})
```

Write Consistency Level: "majority":

"Requests acknowledgment that write operations have propagated to the calculated majority of the nodes (i.e. primary and secondaries replica nodes)"

```
db.basketball_analysis.updateMany({TEAM_ID:"1610612737"},{$set:{TEAM_SLUG:
"BTL"}},{writeConcent:{w: "majority"}})
```

For Insert

Write Consistency Level: 1:

```
db.game_analysis.insertOne({id:272222,Team_ID:1610612699,SEASON:2022,LOCATI
ON:"Away"},{writeConcent:{w:1}})
```

Write Consistency Level: "majority":

```
db.game_analysis.insertOne({id:272222,Team_ID:1610612699,SEASON:2022,LOCATI
ON:"Away"},{writeConcent:{w: "majority"}})
```

5.1.5 Queries to Measure Time in MongoDB & Cassandra

Cassandra

'TRACING ON' query was used before executing the queries to get the time required for executing each CRUD operation.

MongoDB

To measure query time in MongoDB, profiling level was set before running the query and after the query system.profile.find() command was used to measure the query time.

```
{
db.setProfilingLevel(2)
"Query"
db.system.profile.find().limit(1).sort({ts:-1}
)
```

5.1.6 Queries to Measure Latency

Cassandra

We used the below command to measure the read and write latency

"nodetool cfstats"

MongoDB

Query to measure latency for varying read and write operations for different collections

"db.player_analysis.latencyStats({ histograms: true })"

5.2 Design document and flowchart

Flowchart for Cassandra and MongoDB has been displayed in fig 37 and fig 38 respectively.

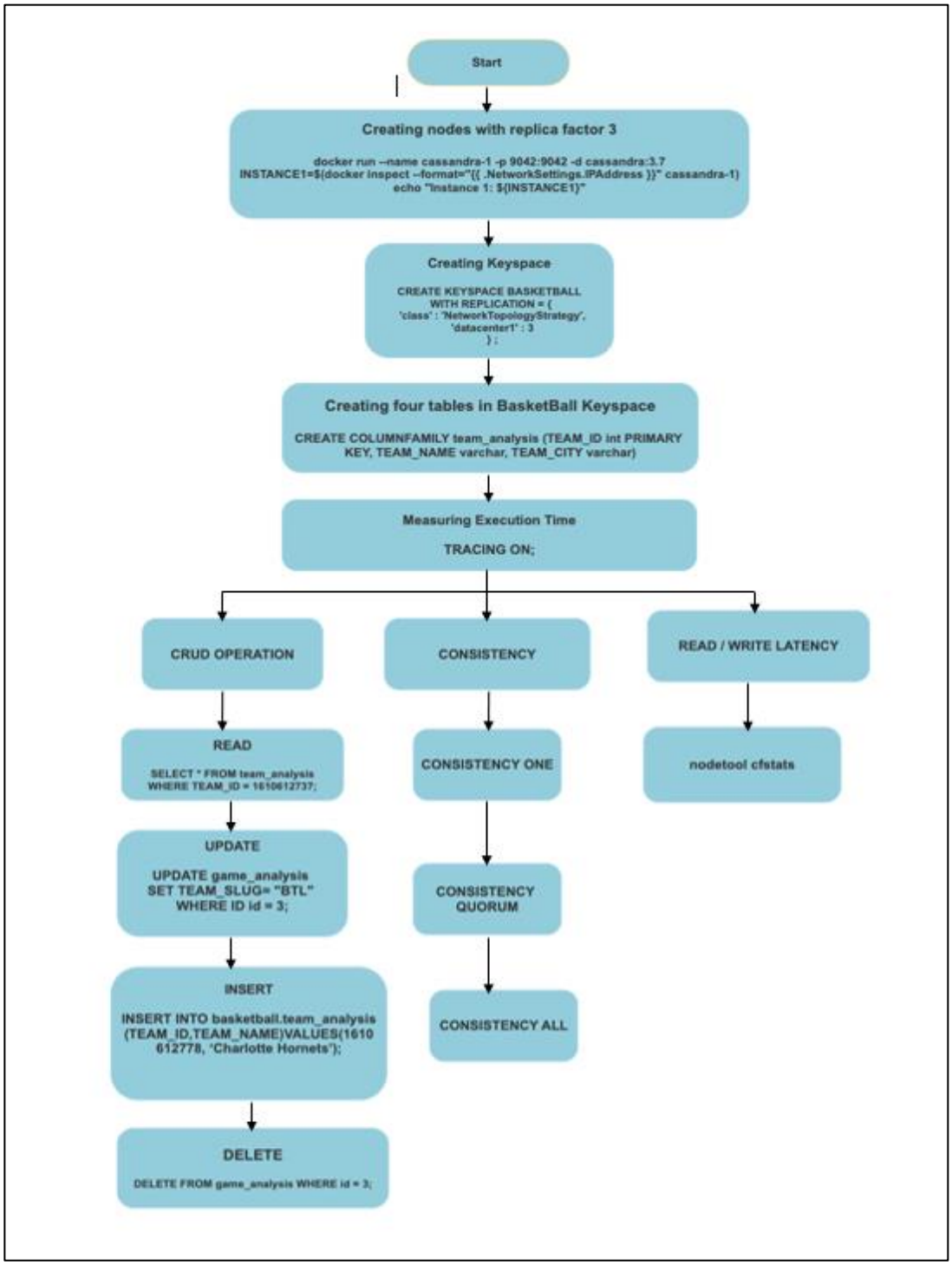


Figure 37: Flowchart for Cassandra

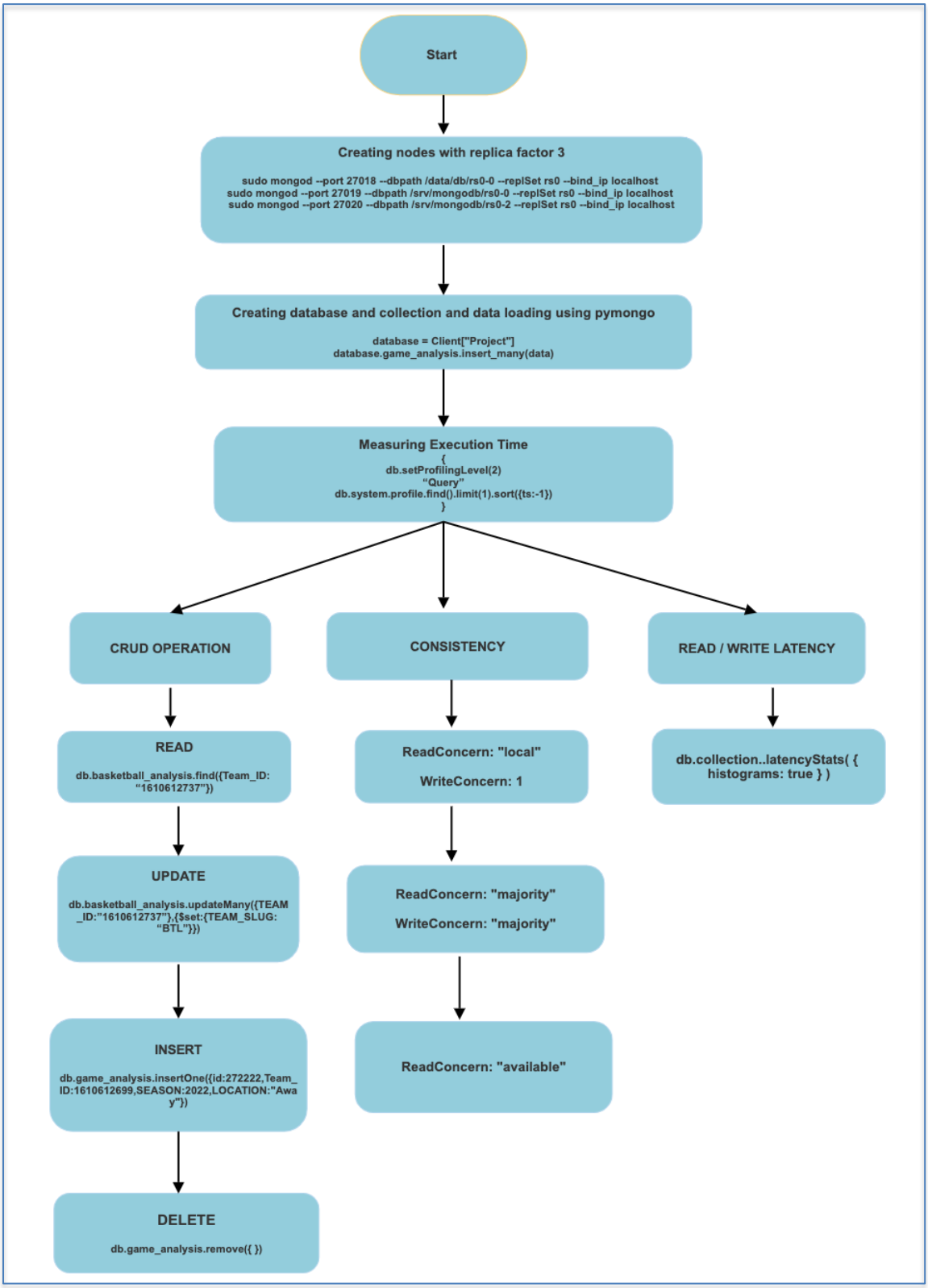


Figure 38: Flowchart for MongoDB

6 Data analysis and Discussion

6.1 Output generation

We are comparing the execution time required for CRUD operations and also measuring the performance by varying Consistency levels for Cassandra and MongoDB.

6.1.1 Data Loading

Cassandra

We are measuring the time required to copy the csv files into the created column families. It can be seen from the fig 39 that bb_analysis which contained 34566 rows and 79 columns took around 1 minute 43.549 seconds (103549 ms) Player_analysis.csv (fig 41) which contained 393 rows and 24 columns took 0.659 seconds for loading the data in cassandra. The average rate at which data is loaded was 596 rows/s. Team_analysis.csv which contained 29 rows and 17 columns took 0.398 seconds and avg rate of 73 rows/s. whereas the Game_analysis.csv (fig 42) took around 4.25 seconds.

```
Starting copy of basketball.bb_analysis with columns [id, team_id, season, location, win_count, average_2_point_goal_efficiency, average_3_point_goa
l_efficiency, average_2_point_goal_percentage, average_3_point_goal_percentage, free_through_goal_efficiency, free_through_goal_percentage, offensiv
e_rebound_percentage, average_assists, average_paint_points, average_2nd_chance_points, defensive_rebound_percentage, average_number_of_steals, aver
age_number_of_blocks, average_points_after_turnover_percentage, avergae_fouls, average_turnover, game_count, total_win_count, total_average_2_point_
goal_efficiency, total_average_3_point_goal_efficiency, total_average_2_point_goal_percentage, total_average_3_point_goal_percentage, total_free_thr
ough_goal_efficiency, total_free_through_goal_percentage, total_offensive_rebound_percentage, total_average_assists, total_average_paint_points, tot
al_average_2nd_chance_points, total_defensive_rebound_percentage, total_average_number_of_steals, total_average_number_of_blocks, total_average_poin
ts_after_turnover_percentage, total_avergae_fouls, total_average_turnover, total_game_count, team_name, team_slug, team_city, year_founded, salary_2
020_2021, salary_2021_2022, salary_2022_2023, salary_2023_2024, salary_2024_2025, salary_2025_2026, home_win_percent, home_loss_percent, away_win_pe
rcent, away_loss_percent, total_win_percent, total_loss_percent, player_id, player_name, school, country, birthdate, player_current_age, height, wei
ght, season_exp, position, rosterstatus, pts, reb, ast, player_nm, contract_type_2020_2021, player_salary_2020_2021, contract_type_2021_2022, player
_salary_2021_2022, contract_type_2022_2023, player_salary_2022_2023, contract_type_2023_2024, player_salary_2023_2024].

Processed: 34566 rows; Rate:     188 rows/s; Avg. rate:     334 rows/s
34566 rows imported from 1 files in 1 minute and 43.549 seconds (0 skipped).
```

Figure 39: Data loading time for bb_analysis Cassandra

```
cqlsh:basketball> COPY basketball.team_analysis (TEAM_ID, TEAM_NAME, TEAM_CITY, TEAM_SLUG, YEAR_FOUNDED, SALARY_2020_2
021, SALARY_2021_2022, SALARY_2022_2023, SALARY_2023_2024, SALARY_2024_2025, SALARY_2025_2026, HOME_WIN_PERCENT, HOME_
LOSS_PERCENT, AWAY_WIN_PERCENT, AWAY_LOSS_PERCENT, TOTAL_WIN_PERCENT, TOTAL_LOSS_PERCENT) FROM 'Team_Analysis.csv ' WI
TH DELIMITER=',' AND HEADER=TRUE;
Using 1 child processes

Starting copy of basketball.team_analysis with columns [team_id, team_name, team_city, team_slug, year_founded, salary
_2020_2021, salary_2021_2022, salary_2022_2023, salary_2023_2024, salary_2024_2025, salary_2025_2026, home_win_percent
, home_loss_percent, away_win_percent, away_loss_percent, total_win_percent, total_loss_percent].
Processed: 29 rows; Rate:      49 rows/s; Avg. rate:      73 rows/s
29 rows imported from 1 files in 0.398 seconds (0 skipped).
cqlsh:basketball>
```

Figure 40: Data loading time for team_analysis Cassandra

```
(29 rows)
cqlsh:basketball> COPY basketball.player_analysis (PLAYER_ID, PLAYER_NAME, SCHOOL, COUNTRY, BIRTHDATE, PLAYER_CURRENT_AGE, HEIGHT, WEIGHT, SEASON_EXP, POSITION, ROSTERSTATUS, TEAM
_ID, PTS, REB, AST, PLAYER_NM, CONTRACT_TYPE_2020_2021, SALARY_2020_2021, CONTRACT_TYPE_2021_2022, SALARY_2021_2022, CONTRACT_TYPE_2022_2023, SALARY_2022_2023, CONTRACT_TYPE_2023_
2024, SALARY_2023_2024) FROM 'Player_Analysis.csv' WITH NULL='NULL' AND DELIMITER=',' AND HEADER=TRUE;
Using 1 child processes

Starting copy of basketball.player_analysis with columns [player_id, player_name, school, country, birthdate, player_current_age, height, weight, season_exp, position, rosterstatu
s, team_id, pts, reb, ast, player_nm, contract_type_2020_2021, salary_2020_2021, contract_type_2021_2022, salary_2021_2022, contract_type_2022_2023, salary_2022_2023, contract_typ
e_2023_2024, salary_2023_2024].
Processed: 393 rows; Rate:      352 rows/s; Avg. rate:      596 rows/s
393 rows imported from 1 files in 0.659 seconds (0 skipped).
cqlsh:basketball>
```

Figure 41: Data loading time for player_analysis Cassandra

```
cqlsh:basketball> COPY basketball.game_analysis
( AND DELIMITER = ',' AND HEADER=TRUE;) FROM 'Game_Analysis.csv' WITH NULL='NULL'
Using 1 child processes

Starting copy of basketball.game_analysis with columns [team_id, season, location, win_count, average_2_point_goal_efficiency, average_3_point_goal_efficiency, average_2_point
_goal_percentage, average_3_point_goal_percentage, free_through_goal_efficiency, free_through_goal_percentage, offensive_rebound_percentage, average_assists, average_paint_poi
nts, average_2nd_chance_points, defensive_rebound_percentage, average_number_of_steals, average_number_of_blocks, average_points_after_turnover_percentage, avergae_fouls, aver
age_turnover, game_count, total_win_count, total_average_2_point_goal_efficiency, total_average_3_point_goal_efficiency, total_average_2_point_goal_percentage, total_average_3
_point_goal_percentage, total_free_through_goal_efficiency, total_free_through_goal_percentage, total_offensive_rebound_percentage, total_average_assists, total_average_paint_
points, total_average_2nd_chance_points, total_defensive_rebound_percentage, total_average_number_of_steals, total_average_number_of_blocks, total_average_points_after_turnove
r_percentage, total_avergae_fouls, total_average_turnover, total_game_count, team_name, team_slug].
Processed: 2720 rows; Rate:      327 rows/s; Avg. rate:      636 rows/s
2720 rows imported from 1 files in 4.275 seconds (0 skipped).
cqlsh:basketball>
```

Figure 42: Data loading time for game_analysis Cassandra

MongoDB

The time required for loading 29 documents of Team_analysis (fig 43) took only 1 ms. Game_Analysis (fig 44) which contains around 2700 documents took 4 ms whereas bb_analysis (Fig 46) which has 34.6K documents took 36 ms respectively.

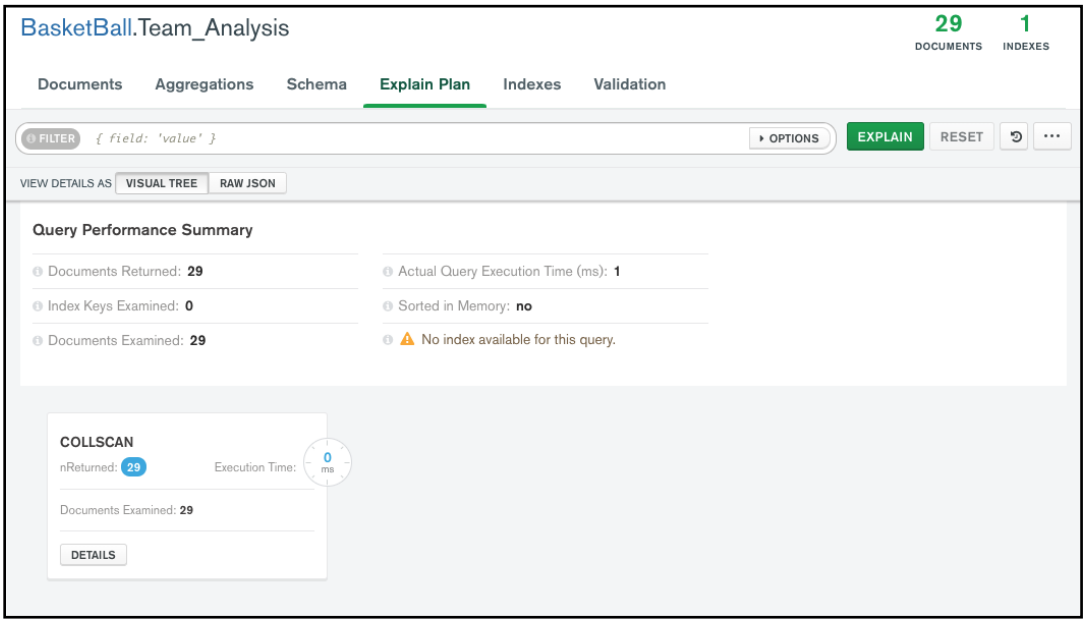


Figure 43: Data loading time for Team_analysis MongoDB

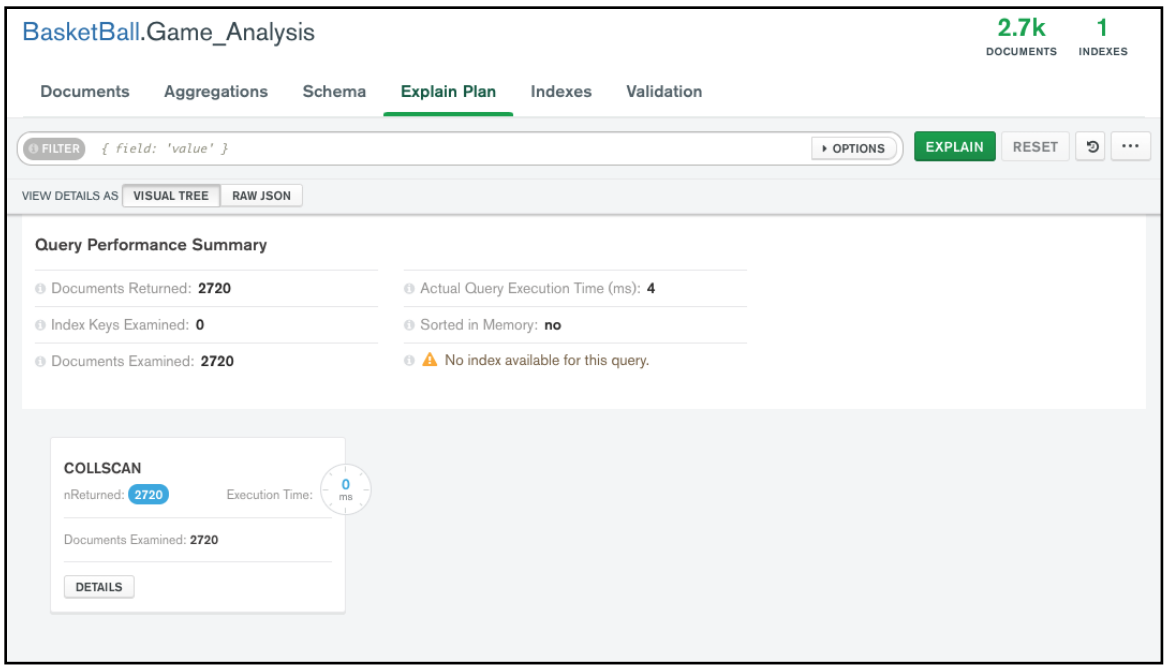


Figure 44: Data loading time for Game_analysis MongoDB

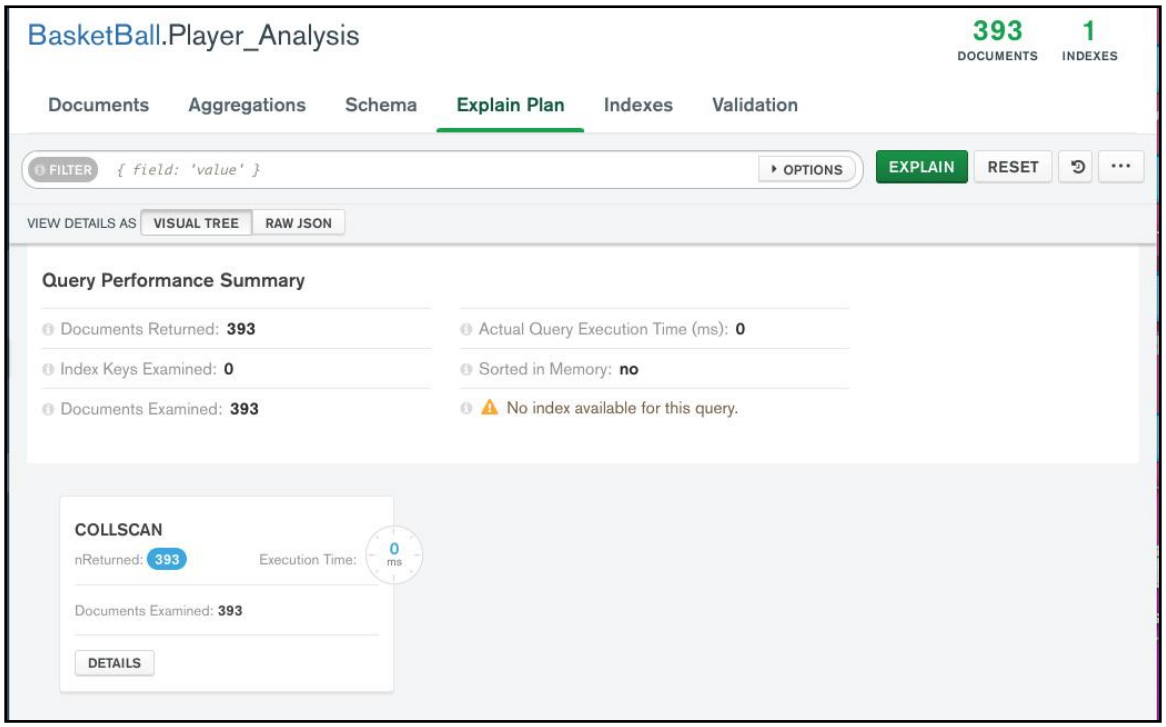


Figure 45:Data loading time for Player_analysis MongoDB

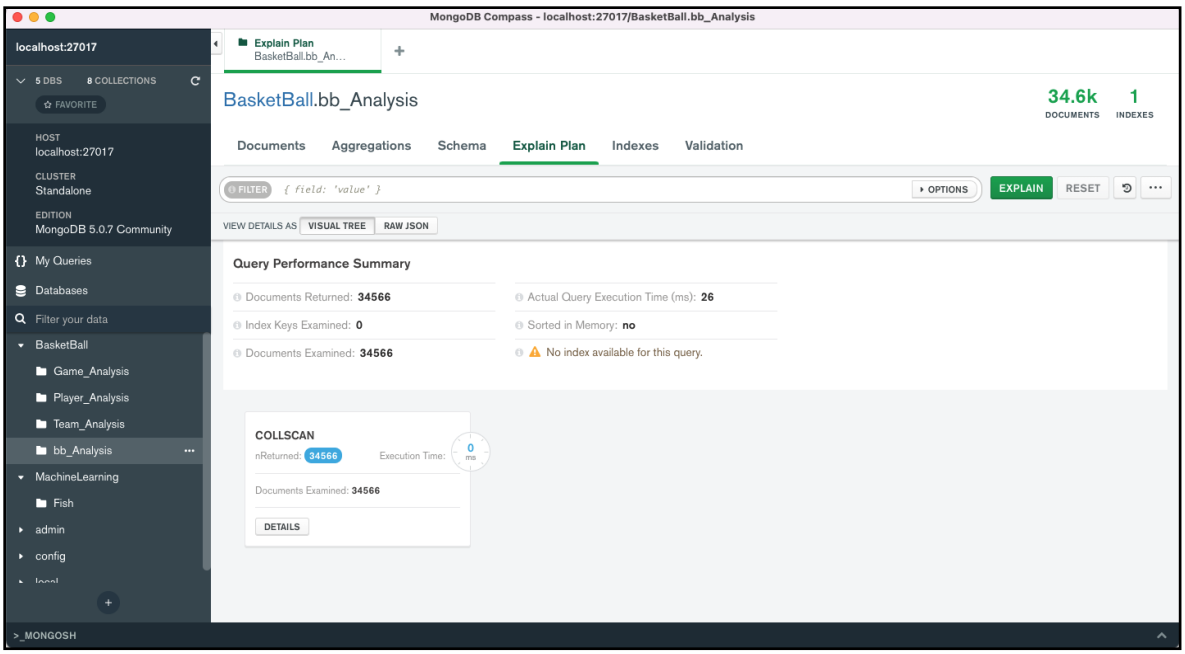


Figure 46: Data loading time for bb_Analysis MongoDB

6.1.2 CRUD Operations

1. READ

Cassandra

The time required for reading a record is around 13.887 ms

```
cqlsh:basketball> SELECT * FROM team_analysis WHERE team_id = 1610612777;

 team_id    | away_loss_percent | away_win_percent | home_loss_percent | home_win_percent | salary_2020_2021 | salary_2021_2022 | salary_2022_2023 | salary_2023_2024 | salary_2024_2025 | salar
y_2025_2026 | team_city | team_name          | team_slug | total_loss_percent | total_win_percent | year_founded
-------------+-------------------+------------------+-------------------+------------------+------------------+------------------+------------------+------------------+------------------+------
-------------+-----------+--------------------+-----------+--------------------+-------------------+--------------
 1610612777 |               0.7 |              0.3 |              0.46 |             0.54 |        108219000 |         83125900 |         44614400 |         31500000 |                0 |
           0 | Charlotte | Charlotte Hornets |       CAA |               0.56 |              0.45 |             1988

(1 rows)

Tracing session: 28ed2dd0-c35d-11ec-820e-8309326e55a4

 activity                                                                                                | timestamp                  | source      | source_elapsed | client
---------------------------------------------------------------------------------------------------------+----------------------------+-------------+----------------+-----------
                                                                                      Execute CQL3 query | 2022-04-23 23:29:02.765000 | 172.17.0.2 |              0 | 127.0.0.1
 Parsing SELECT * FROM team_analysis WHERE team_id = 1610612777; [Native-Transport-Requests-1] | 2022-04-23 23:29:02.765000 | 172.17.0.2 |            391 | 127.0.0.1
                                                  Preparing statement [Native-Transport-Requests-1] | 2022-04-23 23:29:02.766000 | 172.17.0.2 |            636 | 127.0.0.1
                                       Executing single-partition query on team_analysis [ReadStage-3] | 2022-04-23 23:29:02.770000 | 172.17.0.2 |           4509 | 127.0.0.1
                                                    Acquiring sstable references [ReadStage-3] | 2022-04-23 23:29:02.770000 | 172.17.0.2 |           4791 | 127.0.0.1
                                                     Merging memtable contents [ReadStage-3] | 2022-04-23 23:29:02.770000 | 172.17.0.2 |           4863 | 127.0.0.1
                                   Read 1 live rows and 0 tombstone cells [ReadStage-3] | 2022-04-23 23:29:02.776000 | 172.17.0.2 |          11352 | 127.0.0.1
                                                                        Request complete | 2022-04-23 23:29:02.778887 | 172.17.0.2 |          13887 | 127.0.0.1

cqlsh:basketball>
```

Figure 47: Execution time for READ Cassandra

MongoDB

Reading a document from the collection “game_analysis” for specific record “Team_ID” is 7 ms.

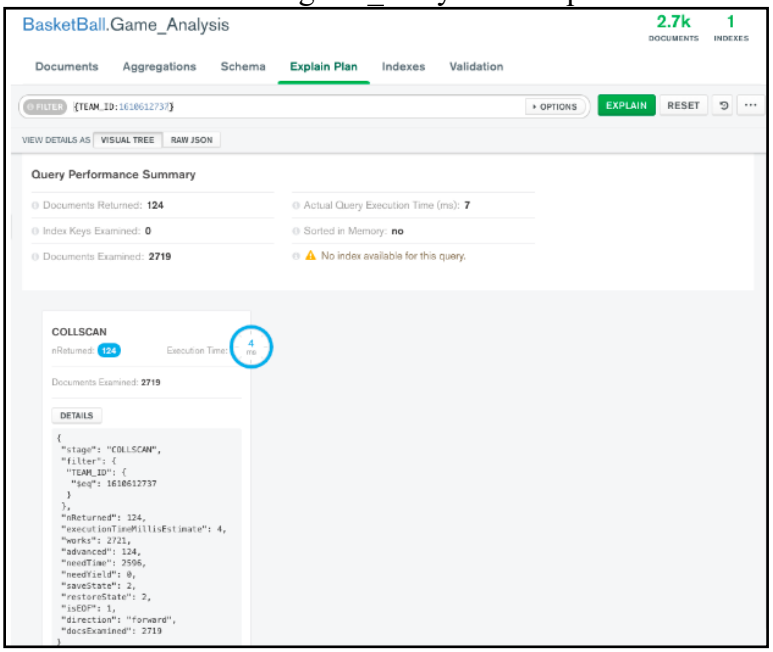


Figure 48:Execution time for READ MongoDB

2. UPDATE

Cassandra

Updating the value of one of the columns (Team_slug) for a specific record (where id= 3) takes around 54.020 ms.

Figure 49:Execution time for UPDATE Cassandra

MongoDB

Updating the value of one of the field (Team_slug) for specific record took around 63 ms.

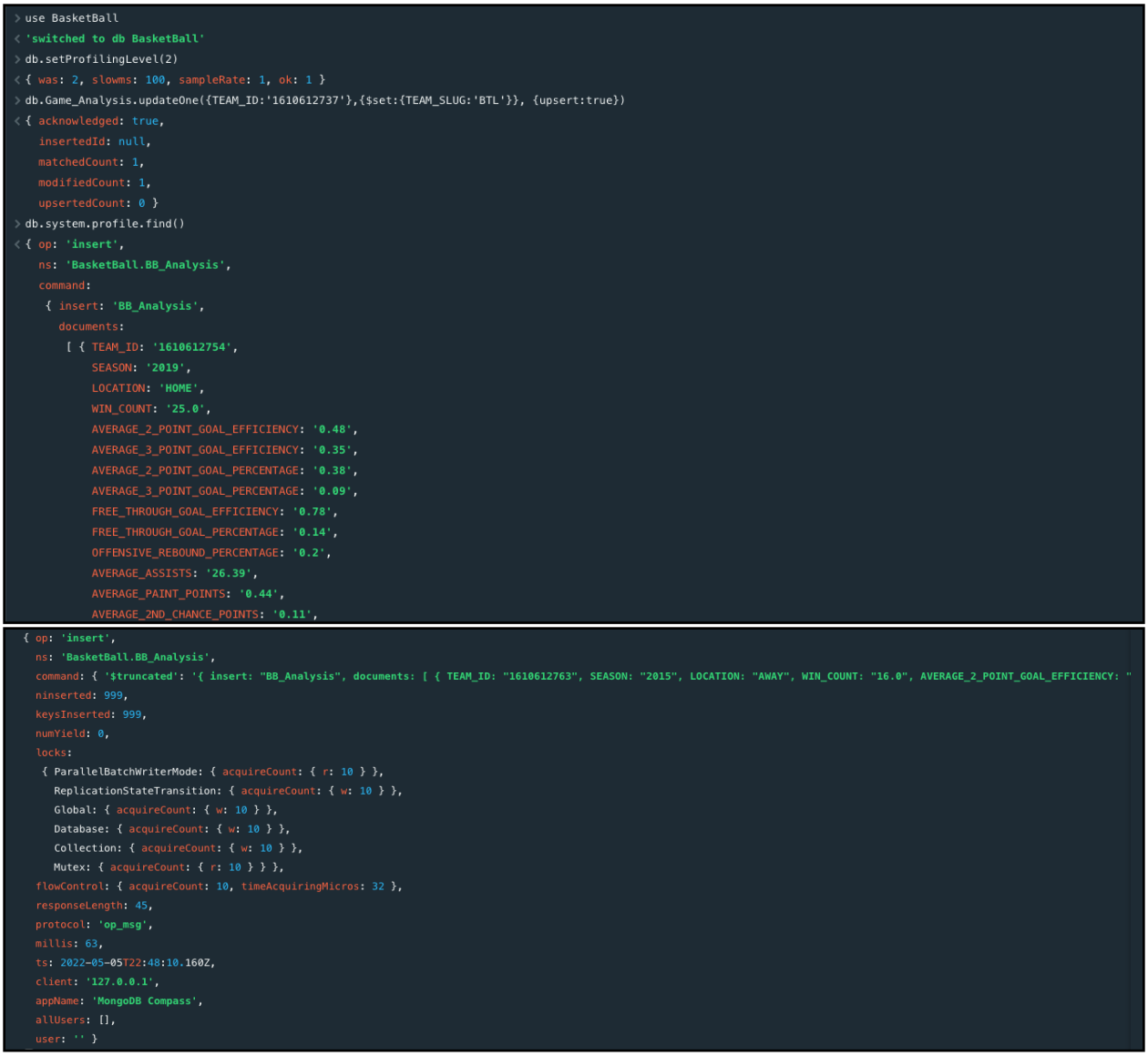


Figure 50: Recording Exceution time for UPDATE MongoDB

3. DELETE

Cassandra

Deleting a particular record from the game_analysis table using id as filter condition. Time required is 33.122 ms

```
cqlsh:basketball> DELETE FROM game_analysis WHERE id = 3;

Tracing session: aacc93f0-ccdf-11ec-8730-6d2c86545d91

 activity                                                                                                  | timestamp                  | source      | source_elapsed | client
-----------------------------------------------------------------------------------------------------------+----------------------------+-------------+----------------+-----------
                                                                                        Execute CQL3 query | 2022-05-06 01:55:55.696000 | 172.17.0.2  |              0 | 127.0.0.1
                                     Parsing DELETE FROM game_analysis WHERE id = 3; [SharedPool-Worker-1] | 2022-05-06 01:55:55.697000 | 172.17.0.2  |           1421 | 127.0.0.1
                                                  Preparing statement [SharedPool-Worker-1] | 2022-05-06 01:55:55.708000 | 172.17.0.2  |          12716 | 127.0.0.1
                                        Determining replicas for mutation [SharedPool-Worker-1] | 2022-05-06 01:55:55.709000 | 172.17.0.2  |          14041 | 127.0.0.1
                    Sending MUTATION message to /172.17.0.3 [MessagingService-Outgoing-/172.17.0.3] | 2022-05-06 01:55:55.711000 | 172.17.0.2  |          15703 | 127.0.0.1
                 MUTATION message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-06 01:55:55.712000 | 172.17.0.3  |             21 | 127.0.0.1
                                                  Appending to commitlog [SharedPool-Worker-3] | 2022-05-06 01:55:55.714000 | 172.17.0.3  |           2595 | 127.0.0.1
                                     Adding to game_analysis memtable [SharedPool-Worker-3] | 2022-05-06 01:55:55.714000 | 172.17.0.3  |           2840 | 127.0.0.1
                                   Enqueuing response to /172.17.0.2 [SharedPool-Worker-3] | 2022-05-06 01:55:55.715000 | 172.17.0.3  |           3016 | 127.0.0.1
            Sending REQUEST_RESPONSE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-06 01:55:55.715000 | 172.17.0.3  |           3707 | 127.0.0.1
         REQUEST_RESPONSE message received from /172.17.0.3 [MessagingService-Incoming-/172.17.0.3] | 2022-05-06 01:55:55.716000 | 172.17.0.2  |          20881 | 127.0.0.1
                    Sending MUTATION message to /172.17.0.4 [MessagingService-Outgoing-/172.17.0.4] | 2022-05-06 01:55:55.718000 | 172.17.0.2  |          23063 | 127.0.0.1
                 MUTATION message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-06 01:55:55.719000 | 172.17.0.4  |             55 | 127.0.0.1
                                                  Appending to commitlog [SharedPool-Worker-1] | 2022-05-06 01:55:55.720000 | 172.17.0.4  |            718 | 127.0.0.1
                                     Adding to game_analysis memtable [SharedPool-Worker-1] | 2022-05-06 01:55:55.720000 | 172.17.0.4  |           1174 | 127.0.0.1
                                   Enqueuing response to /172.17.0.2 [SharedPool-Worker-1] | 2022-05-06 01:55:55.721000 | 172.17.0.4  |           2243 | 127.0.0.1
            Sending REQUEST_RESPONSE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-06 01:55:55.722000 | 172.17.0.4  |           3369 | 127.0.0.1
         REQUEST_RESPONSE message received from /172.17.0.4 [MessagingService-Incoming-/172.17.0.4] | 2022-05-06 01:55:55.725000 | 172.17.0.2  |          29473 | 127.0.0.1
                                  Processing response from /172.17.0.4 [SharedPool-Worker-5] | 2022-05-06 01:55:55.726000 | 172.17.0.2  |          30865 | 127.0.0.1
                                                  Appending to commitlog [SharedPool-Worker-3] | 2022-05-06 01:55:55.727000 | 172.17.0.2  |          28604 | 127.0.0.1
                               Processing response from /172.17.0.3 [SharedPool-Worker-10] | 2022-05-06 01:55:55.728000 | 172.17.0.2  |          29904 | 127.0.0.1
                                     Adding to game_analysis memtable [SharedPool-Worker-3] | 2022-05-06 01:55:55.740000 | 172.17.0.2  |          44216 | 127.0.0.1
                                                                                Request complete | 2022-05-06 01:55:55.729122 | 172.17.0.2  |          33122 | 127.0.0.1
```

Figure 51:Execution time for DELETE Cassandra

MongoDB

Removing a specific document from game_analysis took around 5 ms.

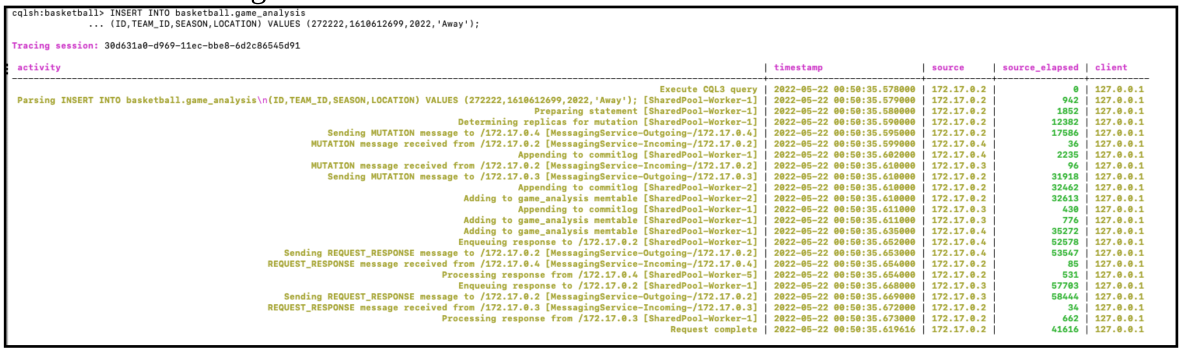
```
> db.game_analysis.remove({Team_ID: "1610612738"}, { justOne:true})
< { acknowledged: true, deletedCount: 0 }
> db.system.profile.find().limit(1).sort({ts:-1})
< { op: 'remove',
    ns: 'project.game_analysis',
    command: { q: { Team_ID: '1610612738' }, limit: 1 },
    keysExamined: 0,
    docsExamined: 2721,
    ndeleted: 0,
    numYield: 2,
    queryHash: '35C8D5FA',
    planCacheKey: '983420F7',
    locks:
     { ParallelBatchWriterMode: { acquireCount: { r: 4 } },
       ReplicationStateTransition: { acquireCount: { w: 5 } },
       Global: { acquireCount: { r: 2, w: 3 } },
       Database: { acquireCount: { w: 3 } },
       Collection: { acquireCount: { w: 3 } },
       Mutex: { acquireCount: { r: 2 } } },
    flowControl: { acquireCount: 3, timeAcquiringMicros: 4 },
    readConcern: { provenance: 'implicitDefault' },
    millis: 5,
    planSummary: 'COLLSCAN',
```

Figure 52: Exceution time for DELETE MongoDB

4. INSERT

Cassandra

Execution time for inserting a record took around 41.616 ms



```
cqlsh:basketball> INSERT INTO basketball.game_analysis
              ... (ID,TEAM_ID,SEASON,LOCATION) VALUES (272222,1610612699,2022,'Away');

Tracing session: 30d631a0-d969-11ec-bbe8-6d2c86545d91

 activity | timestamp | source | source_elapsed | client
------------------------------------------------------
 Execute CQL3 query | 2022-05-22 00:50:35.578000 | 172.17.0.2 | 0 | 127.0.0.1
 Parsing INSERT INTO basketball.game_analysis\n(ID,TEAM_ID,SEASON,LOCATION) VALUES (272222,1610612699,2022,'Away'); [SharedPool-Worker-1] | 2022-05-22 00:50:35.579000 | 172.17.0.2 | 942 | 127.0.0.1
 Preparing statement [SharedPool-Worker-1] | 2022-05-22 00:50:35.580000 | 172.17.0.2 | 1852 | 127.0.0.1
 Determining replicas for mutation [SharedPool-Worker-1] | 2022-05-22 00:50:35.590000 | 172.17.0.2 | 12382 | 127.0.0.1
 Sending MUTATION message to /172.17.0.4 [MessagingService-Outgoing-/172.17.0.4] | 2022-05-22 00:50:35.595000 | 172.17.0.2 | 17586 | 127.0.0.1
 MUTATION message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 00:50:35.599000 | 172.17.0.4 | 36 | 127.0.0.1
 Appending to commitlog [SharedPool-Worker-1] | 2022-05-22 00:50:35.602000 | 172.17.0.4 | 2235 | 127.0.0.1
 MUTATION message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 00:50:35.610000 | 172.17.0.3 | 96 | 127.0.0.1
 Sending MUTATION message to /172.17.0.3 [MessagingService-Outgoing-/172.17.0.3] | 2022-05-22 00:50:35.610000 | 172.17.0.2 | 31918 | 127.0.0.1
 Appending to commitlog [SharedPool-Worker-2] | 2022-05-22 00:50:35.610000 | 172.17.0.2 | 32462 | 127.0.0.1
 Adding to game_analysis memtable [SharedPool-Worker-2] | 2022-05-22 00:50:35.610000 | 172.17.0.2 | 32613 | 127.0.0.1
 Appending to commitlog [SharedPool-Worker-1] | 2022-05-22 00:50:35.611000 | 172.17.0.3 | 430 | 127.0.0.1
 Adding to game_analysis memtable [SharedPool-Worker-1] | 2022-05-22 00:50:35.611000 | 172.17.0.3 | 776 | 127.0.0.1
 Adding to game_analysis memtable [SharedPool-Worker-1] | 2022-05-22 00:50:35.635000 | 172.17.0.4 | 35272 | 127.0.0.1
 Enqueuing response to /172.17.0.2 [SharedPool-Worker-1] | 2022-05-22 00:50:35.652000 | 172.17.0.4 | 52578 | 127.0.0.1
 Sending REQUEST_RESPONSE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-22 00:50:35.653000 | 172.17.0.4 | 53547 | 127.0.0.1
 REQUEST_RESPONSE message received from /172.17.0.4 [MessagingService-Incoming-/172.17.0.4] | 2022-05-22 00:50:35.654000 | 172.17.0.2 | 85 | 127.0.0.1
 Processing response from /172.17.0.4 [SharedPool-Worker-5] | 2022-05-22 00:50:35.654000 | 172.17.0.2 | 531 | 127.0.0.1
 Enqueuing response to /172.17.0.2 [SharedPool-Worker-1] | 2022-05-22 00:50:35.668000 | 172.17.0.3 | 57703 | 127.0.0.1
 Sending REQUEST_RESPONSE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-22 00:50:35.669000 | 172.17.0.3 | 58444 | 127.0.0.1
 REQUEST_RESPONSE message received from /172.17.0.3 [MessagingService-Incoming-/172.17.0.3] | 2022-05-22 00:50:35.672000 | 172.17.0.2 | 34 | 127.0.0.1
 Processing response from /172.17.0.3 [SharedPool-Worker-1] | 2022-05-22 00:50:35.673000 | 172.17.0.2 | 662 | 127.0.0.1
 Request complete | 2022-05-22 00:50:35.619616 | 172.17.0.2 | 41616 | 127.0.0.1
```

Figure 53:Execution time for INSERT Cassandra

MongoDB

Inserting the record (a row) in collection took around 42 ms.

```
>_MONGOSH
> use BasketBall
< 'switched to db BasketBall'
> db.setProfilingLevel(2)
< { was: 0, slowms: 100, sampleRate: 1, ok: 1 }
> db.Game_Analysis.insertOne({id:272222,Team_ID:1610612699,SEASON:2022,LOCATION:"Away"})
< { acknowledged: true,
    insertedId: ObjectId("6289872440ca79dd3aa0f929") }
> db.system.profile.find().limit(1).sort({ts:-1})
< { op: 'insert',
    ns: 'BasketBall.Game_Analysis',
    command:
     { insert: 'Game_Analysis',
       documents:
        [ { id: 272222,
            Team_ID: 1610612699,
            SEASON: 2022,
            LOCATION: 'Away',
            _id: ObjectId("6289872440ca79dd3aa0f929") } ],
       ordered: true,
       lsid: { id: UUID("f6ffbd81-b130-479a-a5b5-4ad89b5de890") },
       '$db': 'BasketBall' },
```

```
    ninserted: 1,
    keysInserted: 1,
    numYield: 0,
    locks:
     { ParallelBatchWriterMode: { acquireCount: { r: 1 } },
       ReplicationStateTransition: { acquireCount: { w: 1 } },
       Global: { acquireCount: { w: 1 } },
       Database: { acquireCount: { w: 1 } },
       Collection: { acquireCount: { w: 1 } },
       Mutex: { acquireCount: { r: 1 } } },
    flowControl: { acquireCount: 1, timeAcquiringMicros: 1 },
    responseLength: 45,
    protocol: 'op_msg',
    millis: 42,
    ts: 2022-05-22T00:43:16.718Z,
    client: '127.0.0.1',
    appName: 'MongoDB Compass',
    allUsers: [],
    user: '' }
BasketBall>
```

Figure 54: Execution time for INSERT MongoDB

6.1.3 Consistency levels

1. READ

<u>Cassandra</u>

<u>Consistency Level ONE</u>

After setting the consistency to One, we read multiple records from the bb_analysis. The time required for reading the data using Consistency one was 225.147 ms

```
cqlsh> USe basketball;
cqlsh:basketball> TRACING ON
Now Tracing is enabled
cqlsh:basketball> CONSISTENCY ONE
Consistency level set to ONE.
cqlsh:basketball> SELECT * FROM bb_analysis WHERE TEAM_ID = 1610612737 ALLOW FILTERING;

 id    | ast | average_2_point_goal_efficiency | average_2_point_goal_percentage | average_2nd_chance_points | average_3_point_goal_efficiency | average_3_point_goal_percentage | average_assists | average_number_of_blocks | average_number_of_steals | average_paint_points | average_points_after_turnover_percentage | average_turnover | avergae_fouls | away_loss_percent | away_win_percent | birthdate | contract_type_2020_2021 | contract_type_2021_2022 | contract_type_2022_2023 | contract_type_2023_2024 | country      | defensive_rebound_percentage | free_through_goal_efficiency | free_through_goal_percentage | game_count | height | home_loss_percent | home_win_percent | location | offensive_rebound_percentage | player_current_age | player_id | player_name        | player_nm          | player_salary_2020_2021 | player_salary_2021_2022 | player_salary_2022_2023 | player_salary_2023_2024 | position       | pts   | reb | rosterstatus | salary_2020_2021 | salary_2021_2022 | salary_2022_2023 | salary_2023_2024 | salary_2024_2025 | salary_2025_2026 | school                | season | season_exp | team_city | team_id    | team_name      | team_slug | total_average_2_point_goal_efficiency | total_average_2_point_goal_percentage | total_average_2nd_chance_points | total_average_3_point_goal_efficiency | total_average_3_point_goal_percentage | total_average_assists | total_average_number_of_blocks | total_average_number_of_steals | total_average_paint_points | total_average_points_after_turnover_percentage | total_average_turnover | total_avergae_fouls | total_defensive_rebound_percentage | total_free_through_goal_efficiency | total_free_through_goal_percentage | total_game_count | total_loss_percent | total_offensive_rebound_percentage | total_win_count | total_win_percent | weight | win_count | year_founded
```

```
Tracing session: 95a416d0-d976-11ec-bbe8-6d2c86545d91

 activity                                                                                                                               | timestamp                  | source     | source_elapsed | client
----------------------------------------------------------------------------------------------------------------------------------------+----------------------------+------------+----------------+-----------
                                                                                                                     Execute CQL3 query | 2022-05-22 02:26:28.157000 | 172.17.0.2 |              0 | 127.0.0.1
                         Parsing SELECT * FROM bb_analysis WHERE TEAM_ID = 1610612737 ALLOW FILTERING; [SharedPool-Worker-1] | 2022-05-22 02:26:28.157000 | 172.17.0.2 |            199 | 127.0.0.1
                                                                                      Preparing statement [SharedPool-Worker-1] | 2022-05-22 02:26:28.157000 | 172.17.0.2 |            475 | 127.0.0.1
                                                                                  Computing ranges to query [SharedPool-Worker-1] | 2022-05-22 02:26:28.163000 | 172.17.0.2 |           6358 | 127.0.0.1
 Submitting range requests on 31 ranges with a concurrency of 3 (44.488464 rows per range expected) [SharedPool-Worker-1] | 2022-05-22 02:26:28.163000 | 172.17.0.2 |           6510 | 127.0.0.1
                                                                      Submitted 1 concurrent range requests [SharedPool-Worker-1] | 2022-05-22 02:26:28.163000 | 172.17.0.2 |           6631 | 127.0.0.1
                              Executing seq scan across 2 sstables for (70, min(-9223372036854775808)] [SharedPool-Worker-2] | 2022-05-22 02:26:28.164000 | 172.17.0.2 |           6792 | 127.0.0.1
                                                                     Read 1549 live and 0 tombstone cells [SharedPool-Worker-2] | 2022-05-22 02:26:28.377000 | 172.17.0.2 |         220575 | 127.0.0.1
                                                                                                                   Request complete | 2022-05-22 02:26:28.382147 | 172.17.0.2 |         225147 | 127.0.0.1
```

Figure 55: Consistency ONE for READ Cassandra

Consistency Level TWO :

The time required for reading records after setting the consistency level to 2 is 288.763 ms

```
cqlsh:basketball> CONSISTENCY QUORUM
Consistency level set to QUORUM.
cqlsh:basketball> SELECT * FROM bb_analysis WHERE TEAM_ID = 1610612737 ALLOW FILTERING;

 id   | ast | average_2_point_goal_efficiency | average_2_point_goal_percentage | average_2nd_chance_points | average_3_point_goal_efficiency | average_3_point_goal_percentage | average_assists | average_number_of_blocks | average_number_of_steals | aver
age_paint_points | average_points_after_turnover_percentage | average_turnover | avergae_fouls | away_loss_percent | away_win_percent | birthdate | contract_type_2020_2021 | contract_type_2021_2022 | contract_type_2022_2023 | contract_type_2023_2024 | co
untry     | defensive_rebound_percentage | free_through_goal_efficiency | free_through_goal_percentage | game_count | height | home_loss_percent | home_win_percent | location | offensive_rebound_percentage | player_current_age | player_id | player_name
    | player_nm        | player_salary_2020_2021 | player_salary_2021_2022 | player_salary_2022_2023 | player_salary_2023_2024 | position      | pts  | reb  | rosterstatus | salary_2020_2021 | salary_2021_2022 | salary_2022_2023 | salary_2023_2024 | s
alary_2024_2025 | salary_2025_2026 | school               | season | season_exp | team_city | team_id     | team_name     | team_slug | total_average_2_point_goal_efficiency | total_average_2_point_goal_percentage | total_average_2nd_chance_points | total_a
verage_3_point_goal_efficiency | total_average_3_point_goal_percentage | total_average_assists | total_average_number_of_blocks | total_average_number_of_steals | total_average_paint_points | total_average_points_after_turnover_percentage | total_average
_turnover | total_avergae_fouls | total_defensive_rebound_percentage | total_free_through_goal_efficiency | total_free_through_goal_percentage | total_game_count | total_loss_percent | total_offensive_rebound_percentage | total_win_count | total_win_perc
ent | weight | win_count | year_founded
```

```
Tracing session: 216fbf70-d977-11ec-bbe8-6d2c86545d91

 activity                                                                                                      | timestamp                  | source     | source_elapsed | client
---------------------------------------------------------------------------------------------------------------+----------------------------+------------+----------------+-----------
                                                                                            Execute CQL3 query | 2022-05-22 02:30:22.695000 | 172.17.0.2 |              0 | 127.0.0.1
                        Parsing SELECT * FROM bb_analysis WHERE TEAM_ID = 1610612737 ALLOW FILTERING; [SharedPool-Worker-1] | 2022-05-22 02:30:22.695000 | 172.17.0.2 |            455 | 127.0.0.1
                                                                           Preparing statement [SharedPool-Worker-1] | 2022-05-22 02:30:22.696000 | 172.17.0.2 |            611 | 127.0.0.1
                                                                       Computing ranges to query [SharedPool-Worker-1] | 2022-05-22 02:30:22.697000 | 172.17.0.2 |           1774 | 127.0.0.1
 Submitting range requests on 31 ranges with a concurrency of 3 (44.488464 rows per range expected) [SharedPool-Worker-1] | 2022-05-22 02:30:22.697000 | 172.17.0.2 |           1898 | 127.0.0.1
                                                                Enqueuing request to /172.17.0.2 [SharedPool-Worker-1] | 2022-05-22 02:30:22.697000 | 172.17.0.2 |           2217 | 127.0.0.1
                                   Sending RANGE_SLICE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-22 02:30:22.698000 | 172.17.0.2 |           3068 | 127.0.0.1
                                 RANGE_SLICE message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 02:30:22.699000 | 172.17.0.2 |           3836 | 127.0.0.1
                         Executing seq scan across 2 sstables for (70, min(-9223372036854775808)] [SharedPool-Worker-3] | 2022-05-22 02:30:22.699000 | 172.17.0.2 |           4541 | 127.0.0.1
                                                                Enqueuing request to /172.17.0.3 [SharedPool-Worker-1] | 2022-05-22 02:30:22.703000 | 172.17.0.2 |           2354 | 127.0.0.1
                                                             Submitted 1 concurrent range requests [SharedPool-Worker-1] | 2022-05-22 02:30:22.703000 | 172.17.0.2 |           7863 | 127.0.0.1
                                   Sending RANGE_SLICE message to /172.17.0.3 [MessagingService-Outgoing-/172.17.0.3] | 2022-05-22 02:30:22.703000 | 172.17.0.2 |           7956 | 127.0.0.1
                                 RANGE_SLICE message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 02:30:22.705000 | 172.17.0.3 |             30 | 127.0.0.1
                         Executing seq scan across 2 sstables for (70, min(-9223372036854775808)] [SharedPool-Worker-2] | 2022-05-22 02:30:22.707000 | 172.17.0.3 |           1749 | 127.0.0.1
                                                          Read 1549 live and 0 tombstone cells [SharedPool-Worker-2] | 2022-05-22 02:30:22.845000 | 172.17.0.3 |         139470 | 127.0.0.1
                                                               Enqueuing response to /172.17.0.2 [SharedPool-Worker-2] | 2022-05-22 02:30:22.845000 | 172.17.0.3 |         139915 | 127.0.0.1
                              Sending REQUEST_RESPONSE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-22 02:30:22.845000 | 172.17.0.3 |         140072 | 127.0.0.1
                            REQUEST_RESPONSE message received from /172.17.0.3 [MessagingService-Incoming-/172.17.0.3] | 2022-05-22 02:30:22.847000 | 172.17.0.2 |         151892 | 127.0.0.1
                                                          Processing response from /172.17.0.3 [SharedPool-Worker-5] | 2022-05-22 02:30:22.849000 | 172.17.0.2 |         154293 | 127.0.0.1
                                                          Read 1549 live and 0 tombstone cells [SharedPool-Worker-3] | 2022-05-22 02:30:22.957000 | 172.17.0.2 |         261755 | 127.0.0.1
                                                               Enqueuing response to /172.17.0.2 [SharedPool-Worker-3] | 2022-05-22 02:30:22.957000 | 172.17.0.2 |         261976 | 127.0.0.1
                              Sending REQUEST_RESPONSE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-22 02:30:22.958000 | 172.17.0.2 |         262647 | 127.0.0.1
                            REQUEST_RESPONSE message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 02:30:22.959000 | 172.17.0.2 |         263651 | 127.0.0.1
                                                          Processing response from /172.17.0.2 [SharedPool-Worker-3] | 2022-05-22 02:30:22.959000 | 172.17.0.2 |         263804 | 127.0.0.1
                                                                                             Request complete | 2022-05-22 02:30:22.983763 | 172.17.0.2 |         288763 | 127.0.0.1


cqlsh:basketball>
cqlsh:basketball>
```

Figure 56:Consistency Quorum (TWO) for READ Cassandra

Consistency ALL

When the consistency level is set to all , we get the time required to read to be 362.315 ms.

```
cqlsh:basketball>
cqlsh:basketball> CONSISTENCY ALL
Consistency level set to ALL.
cqlsh:basketball> SELECT * FROM bb_analysis WHERE TEAM_ID = 1610612737 ALLOW FILTERING;
```

```
Tracing session: 8220b540-d977-11ec-bbe8-6d2c86545d91

 activity                                                                                                      | timestamp                  | source     | source_elapsed | client
---------------------------------------------------------------------------------------------------------------+----------------------------+------------+----------------+-----------
                                                                                            Execute CQL3 query | 2022-05-22 02:33:04.916000 | 172.17.0.2 |              0 | 127.0.0.1
                        Parsing SELECT * FROM bb_analysis WHERE TEAM_ID = 1610612737 ALLOW FILTERING; [SharedPool-Worker-1] | 2022-05-22 02:33:04.916000 | 172.17.0.2 |            188 | 127.0.0.1
                                                                           Preparing statement [SharedPool-Worker-1] | 2022-05-22 02:33:04.916000 | 172.17.0.2 |            302 | 127.0.0.1
                                                                       Computing ranges to query [SharedPool-Worker-1] | 2022-05-22 02:33:04.916000 | 172.17.0.2 |            683 | 127.0.0.1
 Submitting range requests on 31 ranges with a concurrency of 3 (44.488464 rows per range expected) [SharedPool-Worker-1] | 2022-05-22 02:33:04.917000 | 172.17.0.2 |           1600 | 127.0.0.1
                                 RANGE_SLICE message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 02:33:04.918000 | 172.17.0.3 |             22 | 127.0.0.1
                                                                Enqueuing request to /172.17.0.3 [SharedPool-Worker-1] | 2022-05-22 02:33:04.918000 | 172.17.0.2 |           1836 | 127.0.0.1
                                                                Enqueuing request to /172.17.0.4 [SharedPool-Worker-1] | 2022-05-22 02:33:04.918000 | 172.17.0.2 |           2154 | 127.0.0.1
                                                                Enqueuing request to /172.17.0.2 [SharedPool-Worker-1] | 2022-05-22 02:33:04.918000 | 172.17.0.2 |           2226 | 127.0.0.1
                                                             Submitted 1 concurrent range requests [SharedPool-Worker-1] | 2022-05-22 02:33:04.918000 | 172.17.0.2 |           2274 | 127.0.0.1
                                   Sending RANGE_SLICE message to /172.17.0.3 [MessagingService-Outgoing-/172.17.0.3] | 2022-05-22 02:33:04.918000 | 172.17.0.2 |           2298 | 127.0.0.1
                                   Sending RANGE_SLICE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-22 02:33:04.918000 | 172.17.0.2 |           2333 | 127.0.0.1
                                   Sending RANGE_SLICE message to /172.17.0.4 [MessagingService-Outgoing-/172.17.0.4] | 2022-05-22 02:33:04.918001 | 172.17.0.2 |           2675 | 127.0.0.1
                         Executing seq scan across 2 sstables for (70, min(-9223372036854775808)] [SharedPool-Worker-2] | 2022-05-22 02:33:04.919000 | 172.17.0.3 |            763 | 127.0.0.1
                                 RANGE_SLICE message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 02:33:04.919000 | 172.17.0.4 |             26 | 127.0.0.1
                         Executing seq scan across 2 sstables for (70, min(-9223372036854775808)] [SharedPool-Worker-2] | 2022-05-22 02:33:04.923000 | 172.17.0.4 |           4441 | 127.0.0.1
                                 RANGE_SLICE message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 02:33:04.933000 | 172.17.0.2 |          17771 | 127.0.0.1
                         Executing seq scan across 2 sstables for (70, min(-9223372036854775808)] [SharedPool-Worker-2] | 2022-05-22 02:33:04.934000 | 172.17.0.2 |          18049 | 127.0.0.1
                                                          Read 1549 live and 0 tombstone cells [SharedPool-Worker-2] | 2022-05-22 02:33:05.119000 | 172.17.0.3 |         200770 | 127.0.0.1
                                                               Enqueuing response to /172.17.0.2 [SharedPool-Worker-2] | 2022-05-22 02:33:05.124000 | 172.17.0.3 |         205600 | 127.0.0.1
                              Sending REQUEST_RESPONSE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-22 02:33:05.125000 | 172.17.0.3 |         206699 | 127.0.0.1
                            REQUEST_RESPONSE message received from /172.17.0.3 [MessagingService-Incoming-/172.17.0.3] | 2022-05-22 02:33:05.127000 | 172.17.0.2 |         210851 | 127.0.0.1
                                                          Processing response from /172.17.0.3 [SharedPool-Worker-3] | 2022-05-22 02:33:05.128000 | 172.17.0.2 |         212719 | 127.0.0.1
                                                          Read 1549 live and 0 tombstone cells [SharedPool-Worker-2] | 2022-05-22 02:33:05.180000 | 172.17.0.4 |         261073 | 127.0.0.1
                                                               Enqueuing response to /172.17.0.2 [SharedPool-Worker-2] | 2022-05-22 02:33:05.180000 | 172.17.0.4 |         261230 | 127.0.0.1
                              Sending REQUEST_RESPONSE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-22 02:33:05.181000 | 172.17.0.4 |         261792 | 127.0.0.1
                            REQUEST_RESPONSE message received from /172.17.0.4 [MessagingService-Incoming-/172.17.0.4] | 2022-05-22 02:33:05.195000 | 172.17.0.2 |         279569 | 127.0.0.1
                                                          Processing response from /172.17.0.4 [SharedPool-Worker-3] | 2022-05-22 02:33:05.196000 | 172.17.0.2 |         280622 | 127.0.0.1
                                                          Read 1549 live and 0 tombstone cells [SharedPool-Worker-2] | 2022-05-22 02:33:05.224000 | 172.17.0.2 |         308399 | 127.0.0.1
                                                               Enqueuing response to /172.17.0.2 [SharedPool-Worker-2] | 2022-05-22 02:33:05.224000 | 172.17.0.2 |         308690 | 127.0.0.1
                              Sending REQUEST_RESPONSE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-22 02:33:05.224000 | 172.17.0.2 |         308769 | 127.0.0.1
                            REQUEST_RESPONSE message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 02:33:05.225000 | 172.17.0.2 |         308924 | 127.0.0.1
                                                          Processing response from /172.17.0.2 [SharedPool-Worker-2] | 2022-05-22 02:33:05.225000 | 172.17.0.2 |         309009 | 127.0.0.1
                                                                                             Request complete | 2022-05-22 02:33:05.268315 | 172.17.0.2 |         352315 | 127.0.0.1
```

Figure 57 : Consistency ALL for READ Cassandra

MongoDB

readConcern : "local":

Reading multiple records for TEAM_ID = 1610612737 took only 1 ms.

```
> db.basketball_analysis.find({TEAM_ID: "1610612737"}).readConcern("local")
< { _id: ObjectId("628921f7494392970d0a9387"),
    TEAM_ID: '1610612737',
    SEASON: '2020',
    LOCATION: 'AWAY',
    WIN_COUNT: '16.0',
    AVERAGE_2_POINT_GOAL_EFFICIENCY: '0.46',
    AVERAGE_3_POINT_GOAL_EFFICIENCY: '0.36',
    AVERAGE_2_POINT_GOAL_PERCENTAGE: '0.36',
    AVERAGE_3_POINT_GOAL_PERCENTAGE: '0.11',
    FREE_THROUGH_GOAL_EFFICIENCY: '0.81',
    FREE_THROUGH_GOAL_PERCENTAGE: '0.17',
    OFFENSIVE_REBOUND_PERCENTAGE: '0.24',
    AVERAGE_ASSISTS: '23.14',
    AVERAGE_PAINT_POINTS: '0.43',
    AVERAGE_2ND_CHANCE_POINTS: '0.14',
    DEFENSIVE_REBOUND_PERCENTAGE: '0.76',
    AVERAGE_NUMBER_OF_STEALS: '7.11',
    AVERAGE_NUMBER_OF_BLOCKS: '4.67',
    AVERAGE_POINTS_AFTER_TURNOVER_PERCENTAGE: '0.15',
    AVERGAE_FOULS: '19.36',
    AVERAGE_TURNOVER: '13.03',
    GAME_COUNT: '36',
    TOTAL_WIN_COUNT: '41.0',
```

```
  keysExamined: 0,
  docsExamined: 226,
  numYield: 0,
  nreturned: 101,
  queryHash: '782CE9E0',
  planCacheKey: 'AD9254A5',
  locks:
   { Global: { acquireCount: { r: 1 } },
     Mutex: { acquireCount: { r: 1 } } },
  flowControl: {},
  readConcern: { level: 'available', provenance: 'clientSupplied' },
  responseLength: 151721,
  protocol: 'op_msg',
  millis: 1,
  planSummary: 'COLLSCAN',
  execStats:
   { stage: 'COLLSCAN',
     filter: { TEAM_ID: { '$eq': '1610612738' } },
     nReturned: 101,
```

Figure 58: Consistency local for READ MongoDB

readConcern: "majority"

Time taken to read multiple records when consistency is set to majority is 5 ms.

Figure 59: Consistency Majority for READ MongoDB

readConcern: "available"

```
> db.basketball_analysis.find({TEAM_ID: "1610612737"}).readConcern("available")
< { _id: ObjectId("6289a20b494392970d0b2537"),
    TEAM_ID: '1610612737',
    SEASON: '2020',
    LOCATION: 'AWAY',
    WIN_COUNT: '16.0',
    AVERAGE_2_POINT_GOAL_EFFICIENCY: '0.46',
    AVERAGE_3_POINT_GOAL_EFFICIENCY: '0.36',
    AVERAGE_2_POINT_GOAL_PERCENTAGE: '0.36',
    AVERAGE_3_POINT_GOAL_PERCENTAGE: '0.11',
    FREE_THROUGH_GOAL_EFFICIENCY: '0.81',
    FREE_THROUGH_GOAL_PERCENTAGE: '0.17',
    OFFENSIVE_REBOUND_PERCENTAGE: '0.24',
    AVERAGE_ASSISTS: '23.14',
    AVERAGE_PAINT_POINTS: '0.43',
    AVERAGE_2ND_CHANCE_POINTS: '0.14',
    DEFENSIVE_REBOUND_PERCENTAGE: '0.76',
    AVERAGE_NUMBER_OF_STEALS: '7.11',
    AVERAGE_NUMBER_OF_BLOCKS: '4.67',
    AVERAGE_POINTS_AFTER_TURNOVER_PERCENTAGE: '0.15',
    AVERGAE_FOULS: '19.36',
    AVERAGE_TURNOVER: '13.03',
    GAME_COUNT: '36',
```

```
  keysExamined: 0,
  docsExamined: 226,
  numYield: 0,
  nreturned: 101,
  queryHash: '782CE9E0',
  planCacheKey: 'AD9254A5',
  locks:
   { Global: { acquireCount: { r: 1 } },
     Mutex: { acquireCount: { r: 1 } } },
  flowControl: {},
  readConcern: { level: 'available', provenance: 'clientSupplied' },
  responseLength: 151721,
  protocol: 'op_msg',
  millis: 1,
  planSummary: 'COLLSCAN',
  execStats:
   { stage: 'COLLSCAN',
     filter: { TEAM_ID: { '$eq': '1610612738' } },
     nReturned: 101,
```

Figure 60: Consistency available for READ MongoDB

2. WRITE

Update

Cassandra

Consistency One:

The time required to Update records when consistency is set to ONE is 108.951 ms.

```
cqlsh:basketball> CONSISTENCY ONE
Consistency level set to ONE.
cqlsh:basketball> UPDATE bb_analysis SET TEAM_SLUG = 'GGG' WHERE ID in (1,2,3,4,5,6,7,8,9,10);

Tracing session: 95118c10-d97c-11ec-bbe8-6d2c86545d91

 activity                                                                                  | timestamp                  | source     | source_elapsed | client
-------------------------------------------------------------------------------------------+----------------------------+------------+----------------+-----------
                                                                        Execute CQL3 query | 2022-05-22 03:09:24.177000 | 172.17.0.2 |              0 | 127.0.0.1
 Parsing UPDATE bb_analysis SET TEAM_SLUG = 'GGG' WHERE ID in (1,2,3,4,5,6,7,8,9,10); [SharedPool-Worker-1] | 2022-05-22 03:09:24.177000 | 172.17.0.2 |            122 | 127.0.0.1
                                                 Preparing statement [SharedPool-Worker-1] | 2022-05-22 03:09:24.178000 | 172.17.0.2 |            509 | 127.0.0.1
                                    Determining replicas for mutation [SharedPool-Worker-1] | 2022-05-22 03:09:24.178000 | 172.17.0.2 |           1012 | 127.0.0.1
                                               Appending to commitlog [SharedPool-Worker-1] | 2022-05-22 03:09:24.181000 | 172.17.0.2 |           3970 | 127.0.0.1
                                        Adding to bb_analysis memtable [SharedPool-Worker-1] | 2022-05-22 03:09:24.182000 | 172.17.0.2 |           4283 | 127.0.0.1
                                               Appending to commitlog [SharedPool-Worker-1] | 2022-05-22 03:09:24.182000 | 172.17.0.2 |           4557 | 127.0.0.1
                                        Adding to bb_analysis memtable [SharedPool-Worker-1] | 2022-05-22 03:09:24.182000 | 172.17.0.2 |           4682 | 127.0.0.1
                                               Appending to commitlog [SharedPool-Worker-1] | 2022-05-22 03:09:24.182000 | 172.17.0.2 |           4816 | 127.0.0.1
```

```
                      Processing response from /172.17.0.4 [SharedPool-Worker-2] | 2022-05-22 03:09:24.252000 | 172.17.0.2 |          74888 | 127.0.0.1
                      Processing response from /172.17.0.4 [SharedPool-Worker-2] | 2022-05-22 03:09:24.252000 | 172.17.0.2 |          74926 | 127.0.0.1
                      Processing response from /172.17.0.4 [SharedPool-Worker-2] | 2022-05-22 03:09:24.252000 | 172.17.0.2 |          74962 | 127.0.0.1
                      Processing response from /172.17.0.3 [SharedPool-Worker-2] | 2022-05-22 03:09:24.252001 | 172.17.0.2 |          75001 | 127.0.0.1
                            Adding to bb_analysis memtable [SharedPool-Worker-1] | 2022-05-22 03:09:24.283000 | 172.17.0.2 |          53448 | 127.0.0.1
                                   Appending to commitlog [SharedPool-Worker-1] | 2022-05-22 03:09:24.285000 | 172.17.0.2 |         107727 | 127.0.0.1
                            Adding to bb_analysis memtable [SharedPool-Worker-1] | 2022-05-22 03:09:24.285000 | 172.17.0.2 |         107876 | 127.0.0.1
 MUTATION message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 03:09:24.297000 | 172.17.0.3 |             17 | 127.0.0.1
                                   Appending to commitlog [SharedPool-Worker-4] | 2022-05-22 03:09:24.298000 | 172.17.0.3 |           1132 | 127.0.0.1
                            Adding to bb_analysis memtable [SharedPool-Worker-4] | 2022-05-22 03:09:24.298000 | 172.17.0.3 |           1267 | 127.0.0.1
                         Enqueuing response to /172.17.0.2 [SharedPool-Worker-4] | 2022-05-22 03:09:24.299000 | 172.17.0.3 |           1376 | 127.0.0.1
 Sending REQUEST_RESPONSE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-22 03:09:24.299000 | 172.17.0.3 |           1947 | 127.0.0.1
 MUTATION message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 03:09:24.313000 | 172.17.0.4 |             60 | 127.0.0.1
                                   Appending to commitlog [SharedPool-Worker-2] | 2022-05-22 03:09:24.313000 | 172.17.0.4 |            338 | 127.0.0.1
                            Adding to bb_analysis memtable [SharedPool-Worker-2] | 2022-05-22 03:09:24.315000 | 172.17.0.4 |           1235 | 127.0.0.1
 REQUEST_RESPONSE message received from /172.17.0.3 [MessagingService-Incoming-/172.17.0.3] | 2022-05-22 03:09:24.324000 | 172.17.0.2 |             45 | 127.0.0.1
                      Processing response from /172.17.0.3 [SharedPool-Worker-5] | 2022-05-22 03:09:24.325000 | 172.17.0.2 |            317 | 127.0.0.1
                         Enqueuing response to /172.17.0.2 [SharedPool-Worker-2] | 2022-05-22 03:09:24.339000 | 172.17.0.4 |          26181 | 127.0.0.1
 Sending REQUEST_RESPONSE message to /172.17.0.2 [MessagingService-Outgoing-/172.17.0.2] | 2022-05-22 03:09:24.340000 | 172.17.0.4 |          27042 | 127.0.0.1
 REQUEST_RESPONSE message received from /172.17.0.4 [MessagingService-Incoming-/172.17.0.4] | 2022-05-22 03:09:24.343000 | 172.17.0.2 |             22 | 127.0.0.1
                      Processing response from /172.17.0.4 [SharedPool-Worker-1] | 2022-05-22 03:09:24.344000 | 172.17.0.2 |           1011 | 127.0.0.1
 Sending MUTATION message to /172.17.0.3 [MessagingService-Outgoing-/172.17.0.3] | 2022-05-22 03:09:24.493000 | 172.17.0.2 |             -- | 127.0.0.1
 Sending MUTATION message to /172.17.0.4 [MessagingService-Outgoing-/172.17.0.4] | 2022-05-22 03:09:24.494000 | 172.17.0.2 |             -- | 127.0.0.1
                                                               Request complete | 2022-05-22 03:09:24.285951 | 172.17.0.2 |         108951 | 127.0.0.1
```

Figure 61: Consistency ONE for UPDATE Cassandra

Consistency Quorum

When the Quorum consistency is selected, the time required for UPDATE operation is higher, which is around 162.396 ms.

```
cqlsh:basketball> CONSISTENCY QUORUM
Consistency level set to QUORUM.
cqlsh:basketball> UPDATE bb_analysis SET TEAM_SLUG = 'FFF' WHERE ID in (1,2,3,4,5,6,7,8,9,10);

Tracing session: 7a206c00-d97c-11ec-bbe8-6d2c86545d91

 activity                                                                                  | timestamp                  | source     | source_elapsed | client
-------------------------------------------------------------------------------------------+----------------------------+------------+----------------+-----------
                                                                        Execute CQL3 query | 2022-05-22 03:08:38.976000 | 172.17.0.2 |              0 | 127.0.0.1
 Parsing UPDATE bb_analysis SET TEAM_SLUG = 'FFF' WHERE ID in (1,2,3,4,5,6,7,8,9,10); [SharedPool-Worker-1] | 2022-05-22 03:08:38.977000 | 172.17.0.2 |            363 | 127.0.0.1
                                                 Preparing statement [SharedPool-Worker-1] | 2022-05-22 03:08:38.989000 | 172.17.0.2 |          13029 | 127.0.0.1
                                    Determining replicas for mutation [SharedPool-Worker-1] | 2022-05-22 03:08:38.995000 | 172.17.0.2 |          19286 | 127.0.0.1
                                               Appending to commitlog [SharedPool-Worker-3] | 2022-05-22 03:08:38.997000 | 172.17.0.2 |          20723 | 127.0.0.1
                                        Adding to bb_analysis memtable [SharedPool-Worker-3] | 2022-05-22 03:08:38.997000 | 172.17.0.2 |          20856 | 127.0.0.1
           Sending MUTATION message to /172.17.0.3 [MessagingService-Outgoing-/172.17.0.3] | 2022-05-22 03:08:38.997000 | 172.17.0.2 |          20723 | 127.0.0.1
           Sending MUTATION message to /172.17.0.4 [MessagingService-Outgoing-/172.17.0.4] | 2022-05-22 03:08:38.997000 | 172.17.0.2 |          21399 | 127.0.0.1
        MUTATION message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 03:08:38.998000 | 172.17.0.3 |             24 | 127.0.0.1
        MUTATION message received from /172.17.0.2 [MessagingService-Incoming-/172.17.0.2] | 2022-05-22 03:08:38.999000 | 172.17.0.4 |             62 | 127.0.0.1
                                               Appending to commitlog [SharedPool-Worker-1] |        2022-05-22 03:08:39 | 172.17.0.4 |           1953 | 127.0.0.1
                                               Appending to commitlog [SharedPool-Worker-1] | 2022-05-22 03:08:39.001000 | 172.17.0.2 |          25416 | 127.0.0.1
                                        Adding to bb_analysis memtable [SharedPool-Worker-1] | 2022-05-22 03:08:39.002000 | 172.17.0.2 |          25515 | 127.0.0.1
                                               Appending to commitlog [SharedPool-Worker-1] | 2022-05-22 03:08:39.002000 | 172.17.0.2 |          25845 | 127.0.0.1
                                        Adding to bb_analysis memtable [SharedPool-Worker-1] | 2022-05-22 03:08:39.002000 | 172.17.0.2 |          25976 | 127.0.0.1
                                               Appending to commitlog [SharedPool-Worker-1] | 2022-05-22 03:08:39.002000 | 172.17.0.2 |          26298 | 127.0.0.1
                                        Adding to bb_analysis memtable [SharedPool-Worker-1] | 2022-05-22 03:08:39.004000 | 172.17.0.4 |           5901 | 127.0.0.1
```

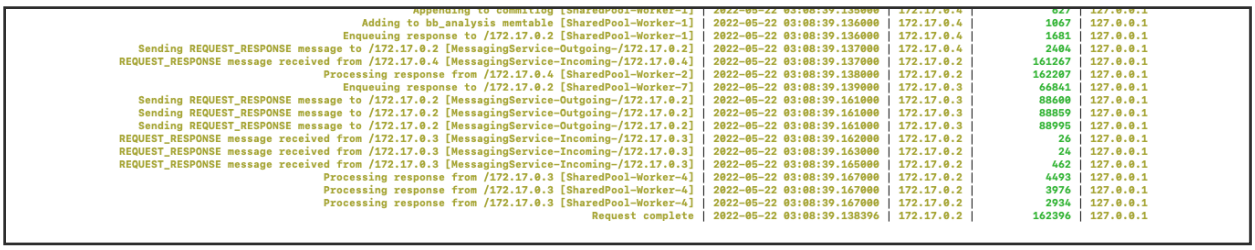


Figure 62: Consistency Quorum for READ MongoDB

MongoDB

WriteConcern: "majority"

The query to set the write concern to majority is as given below. It can be seen from the highlighted part that the time required to execute update operation when consistency is set to majority is 566 ms.

```
rs0:PRIMARY>
db.basketball_analysis.updateMany({TEAM_ID:"1610612737"},{$set:{TEAM_SLUG:"B.T.L"}},{writeConcern:{w:"majority"}
})

{ "acknowledged" : true, "matchedCount" : 1875, "modifiedCount" : 1875 }
rs0:PRIMARY> db.system.profile.find().limit(1).sort({ts:-1})
{ "op" : "update", "ns" : "project.basketball_analysis", "command" : { "q" : { "TEAM_ID" : "1610612737" }, "u" : { "$set" :
{ "TEAM_SLUG" : "B.T.L" } }, "multi" : true, "upsert" : false }, "keysExamined" : 0, "docsExamined" : 34566, "nMatched" :
1875, "nModified" : 1875, "nUpserted" : 0, "numYield" : 69, "queryHash" : "782CE9E0", "planCacheKey" : "AD9254A5",
"locks" : { "ParallelBatchWriterMode" : { "acquireCount" : { "r" : NumberLong(70) } }, "ReplicationStateTransition" : {
"acquireCount" : { "w" : NumberLong(71) } }, "Global" : { "acquireCount" : { "r" : NumberLong(1), "w" : NumberLong(70) }
}, "Database" : { "acquireCount" : { "w" : NumberLong(70) } }, "Collection" : { "acquireCount" : { "w" : NumberLong(70) } },
"Mutex" : { "acquireCount" : { "r" : NumberLong(1) } } }, "flowControl" : { "acquireCount" : NumberLong(70),
"timeAcquiringMicros" : NumberLong(114) }, "readConcern" : { "level" : "local", "provenance" : "implicitDefault" },
"storage" : {  }, "millis" : 566, "planSummary" : "COLLSCAN", "execStats" : { "stage" : "UPDATE", "nReturned" : 0,
"executionTimeMillisEstimate" : 520, "works" : 34568, "advanced" : 0, "needTime" : 34567, "needYield" : 0, "saveState" : 69,
"restoreState" : 69, "isEOF" : 1, "nMatched" : 1875, "nWouldModify" : 1875, "nWouldUpsert" : 0, "inputStage" : { "stage" :
"COLLSCAN", "filter" : { "TEAM_ID" : { "$eq" : "1610612737" } }, "nReturned" : 1875, "executionTimeMillisEstimate" :
67, "works" : 34568, "advanced" : 1875, "needTime" : 32692, "needYield" : 0, "saveState" : 1944, "restoreState" : 1944,
"isEOF" : 1, "direction" : "forward", "docsExamined" : 34566 } }, "ts" : ISODate("2022-05-21T18:01:41.210Z"), "client" :
"127.0.0.1", "appName" : "MongoDB Shell", "allUsers" : [ ], "user" : "" }
```

Writeconcern: 1

Changing the consistency level to 1, we get the time required for WRITE operation which is 70 ms as highlighted in the query output below.

```
rs0:PRIMARY>
db.basketball_analysis.updateMany({TEAM_ID:"1610612737"},{$set:{TEAM_SLUG:"BTL"}},{writeConcern:{w:1}})
{ "acknowledged" : true, "matchedCount" : 1875, "modifiedCount" : 0 }
rs0:PRIMARY> db.system.profile.find().limit(1).sort({ts:-1})
{ "op" : "update", "ns" : "project.basketball_analysis", "command" : { "q" : { "TEAM_ID" : "1610612737" }, "u" : { "$set" :
{ "TEAM_SLUG" : "BTL" } }, "multi" : true, "upsert" : false }, "keysExamined" : 0, "docsExamined" : 34566, "nMatched" :
1875, "nModified" : 0, "nUpserted" : 0, "numYield" : 34, "queryHash" : "782CE9E0", "planCacheKey" : "AD9254A5",
"locks" : { "ParallelBatchWriterMode" : { "acquireCount" : { "r" : NumberLong(35) } }, "ReplicationStateTransition" : {
"acquireCount" : { "w" : NumberLong(35) } }, "Global" : { "acquireCount" : { "w" : NumberLong(35) } }, "Database" : {
"acquireCount" : { "w" : NumberLong(35) } }, "Collection" : { "acquireCount" : { "w" : NumberLong(35) } }, "Mutex" : {
"acquireCount" : { "r" : NumberLong(1) } } }, "flowControl" : { "acquireCount" : NumberLong(35), "timeAcquiringMicros" :
NumberLong(154) }, "readConcern" : { "level" : "local", "provenance" : "implicitDefault" }, "millis" : 70, "planSummary" :
"COLLSCAN", "execStats" : { "stage" : "UPDATE", "nReturned" : 0, "executionTimeMillisEstimate" : 32, "works" : 34568,
"advanced" : 0, "needTime" : 34567, "needYield" : 0, "saveState" : 34, "restoreState" : 34, "isEOF" : 1, "nMatched" : 1875,
"nWouldModify" : 0, "nWouldUpsert" : 0, "inputStage" : { "stage" : "COLLSCAN", "filter" : { "TEAM_ID" : { "$eq" :
"1610612737" } }, "nReturned" : 1875, "executionTimeMillisEstimate" : 7, "works" : 34568, "advanced" : 1875, "needTime" :
32692, "needYield" : 0, "saveState" : 1909, "restoreState" : 1909, "isEOF" : 1, "direction" : "forward", "docsExamined" :
34566 } }, "ts" : ISODate("2022-05-21T18:00:21.119Z"), "client" : "127.0.0.1", "appName" : "MongoDB Shell", "allUsers" : [ ],
"user" : "" }
```

INSERT

Cassandra

Consistency One

After setting the consistency to ONE, we get the time required for executing INSERT operation to be 85.303 ms.

Figure 63: Consistency ONE for INSERT Cassandra

Consistency Quorum

When the consistency level is changed to Quorum, the time required for complete execution of INSERT operation takes 111.234 ms.

Figure 64: Consistency Quorum for READ Cassandra

MongoDB

writeConcern: 1

Time required for inserting records by setting writeConcern to 1 is around 2 ms.

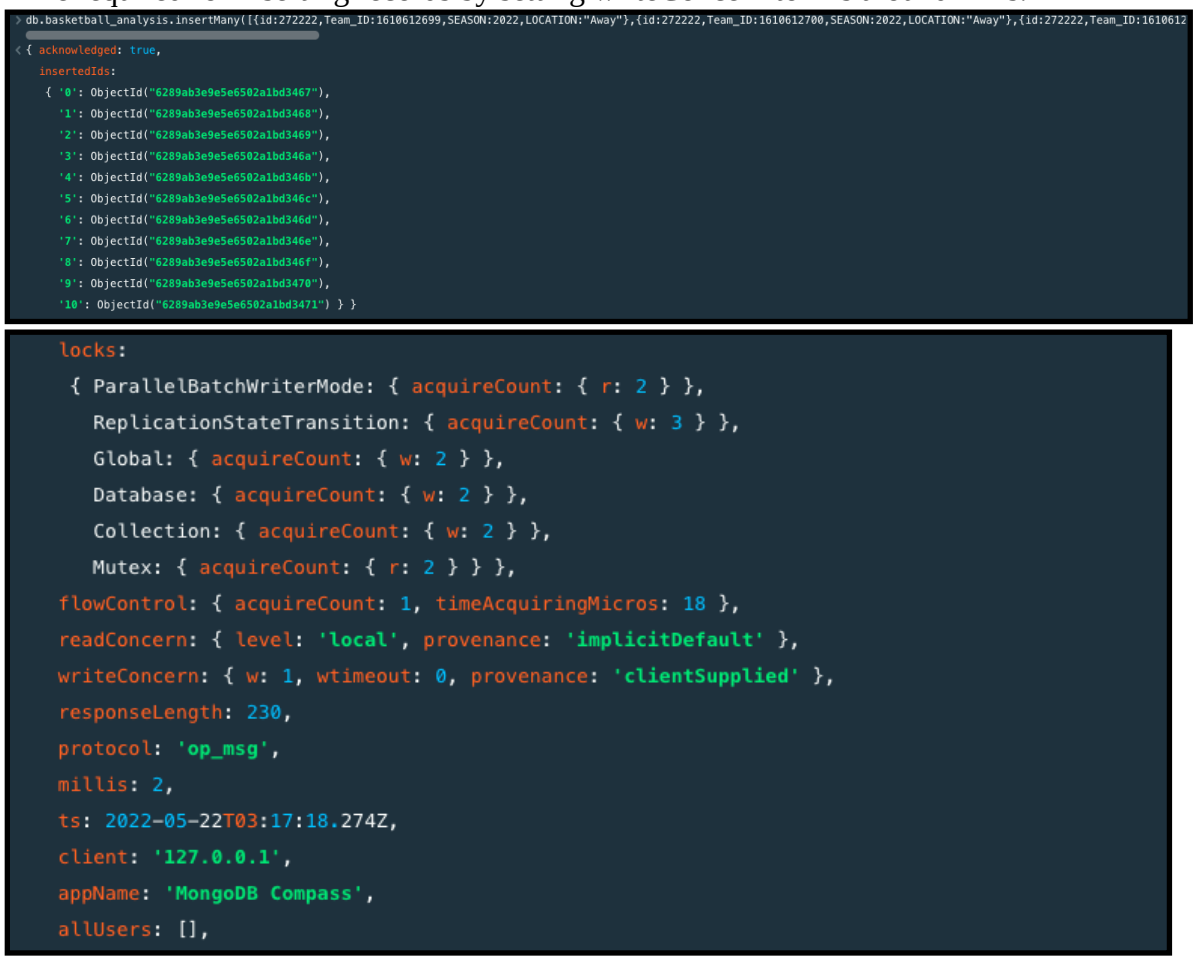


Figure 65: Consistency ONE for INSERT MongoDB

writeConcern: "majority"

Whereas after changing the consistency level to majority, the execution time increases to 36 ms.

```
> db.basketball_analysis.insertMany([{id:272222,Team_ID:1610612699,SEASON:2022,LOCATION:"Away"},{id:272222,Team_ID:1610612700,SEASON:2022,LOCATION:"Away"},{id:272222,Team_ID:1610612
< { acknowledged: true,
    insertedIds:
     { '0': ObjectId("6289ab6e9e5e6502a1bd3472"),
       '1': ObjectId("6289ab6e9e5e6502a1bd3473"),
       '2': ObjectId("6289ab6e9e5e6502a1bd3474"),
       '3': ObjectId("6289ab6e9e5e6502a1bd3475"),
       '4': ObjectId("6289ab6e9e5e6502a1bd3476"),
       '5': ObjectId("6289ab6e9e5e6502a1bd3477"),
       '6': ObjectId("6289ab6e9e5e6502a1bd3478"),
       '7': ObjectId("6289ab6e9e5e6502a1bd3479"),
       '8': ObjectId("6289ab6e9e5e6502a1bd347a"),
       '9': ObjectId("6289ab6e9e5e6502a1bd347b"),
       '10': ObjectId("6289ab6e9e5e6502a1bd347c") } }
```

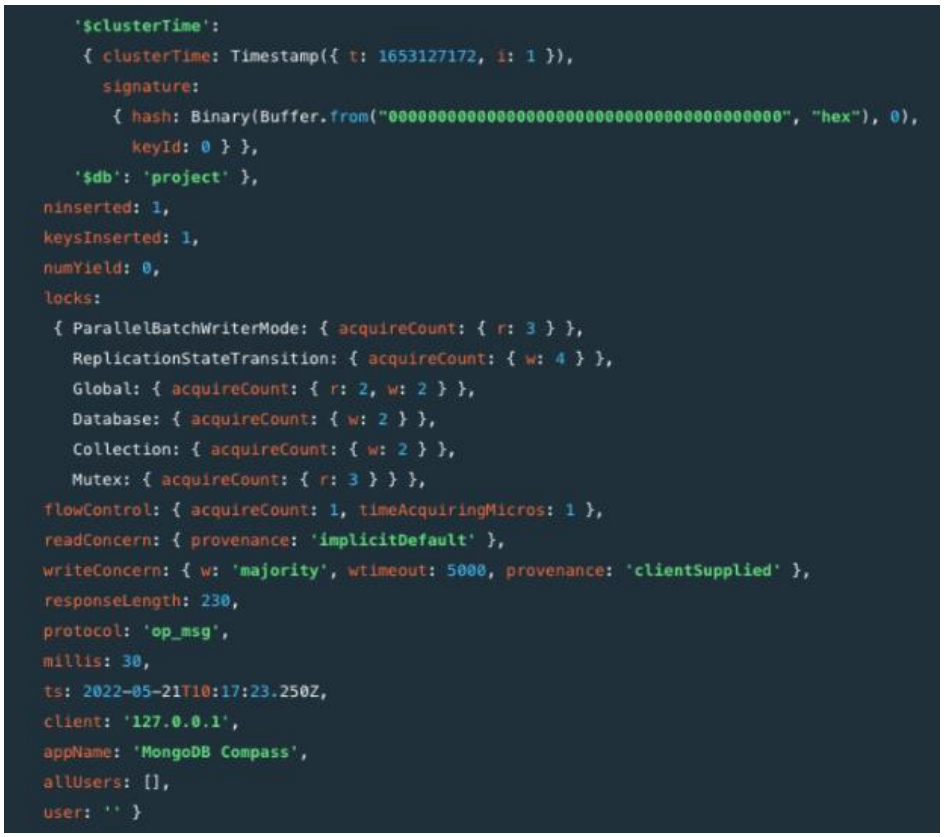


Figure 66: Consistency majority for INSERT MongoDB

6.1.4 Latency Measurement

Cassandra

Using nodestats command we get read and write latency. Fig 66, 67 and 68 shows the read and write latency for all the column families and for various read and write counts**.**

```
# nodetool cfstats basketball
Keyspace : basketball
        Read Count: 14
        Read Latency: 10.311785714285714 ms.
        Write Count: 36052
        Write Latency: 0.8415192777099745 ms.
        Pending Flushes: 0
                Table: bb_analysis
                Space used (live): 7278455
                Space used (total): 7278455
                Space used by snapshots (total): 0
                Off heap memory used (total): 48944
                SSTable Compression Ratio: 0.4104419601541656
                Number of keys (estimate): 34956
                Memtable cell count: 1136
                Memtable data size: 1674572
                Memtable off heap memory used: 0
                Memtable switch count: 2
                Local read count: 9
                Local read latency: 11.994 ms
                Local write count: 36047
                Local write latency: 0.931 ms
                Pending flushes: 0
                Bloom filter false positives: 0
                Bloom filter false ratio: 0.00000
                Bloom filter space used: 42648
                Bloom filter off heap memory used: 42632
                Index summary off heap memory used: 4272
                Compression metadata off heap memory used: 2040
                Compacted partition minimum bytes: 447
                Compacted partition maximum bytes: 535
                Compacted partition mean bytes: 535
                Average live cells per slice (last five minutes): 1783.858585858586
                Maximum live cells per slice (last five minutes): 2299
                Average tombstones per slice (last five minutes): 1.0
                Maximum tombstones per slice (last five minutes): 1
                Dropped Mutations: 151
```

Figure 67: Latency measurement for bb_analysis Cassandra

```
        Table: game_analysis
        Space used (live): 341652
        Space used (total): 341652
        Space used by snapshots (total): 0
        Off heap memory used (total): 3848
        SSTable Compression Ratio: 0.48665504258784203
        Number of keys (estimate): 2729
        Memtable cell count: 3
        Memtable data size: 137
        Memtable off heap memory used: 0
        Memtable switch count: 0
        Local read count: 5
        Local read latency: 10.532 ms
        Local write count: 3
        Local write latency: 2.392 ms
        Pending flushes: 0
        Bloom filter false positives: 0
        Bloom filter false ratio: 0.00000
        Bloom filter space used: 3424
        Bloom filter off heap memory used: 3416
        Index summary off heap memory used: 352
        Compression metadata off heap memory used: 80
        Compacted partition minimum bytes: 18
        Compacted partition maximum bytes: 258
        Compacted partition mean bytes: 239
        Average live cells per slice (last five minutes): 83.6470588235294
        Maximum live cells per slice (last five minutes): 103
        Average tombstones per slice (last five minutes): 1.0
        Maximum tombstones per slice (last five minutes): 1
        Dropped Mutations: 0
```

Figure 68: : Latency measurement for game_analysis Cassandra

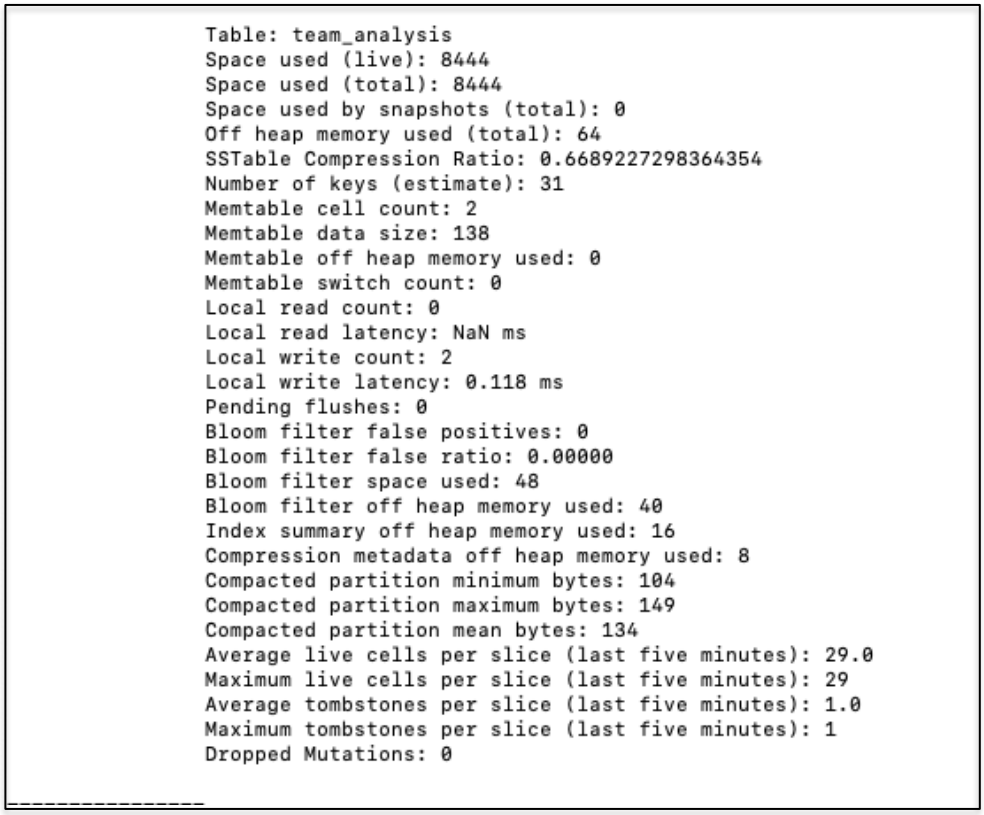


Figure 69: : Latency measurement for team_analysis Cassandra

.

MongoDB

Attached below are the stats for latency wrt read and write count obtained by running the following query for each collection.

rs0:PRIMARY> db.basketball_analysis.latencyStats({ histograms: true })

{ "ns" : "project.basketball_analysis", "host" : "IQRAs-MacBook-Air.local:27018", "localTime" : ISODate("2022-05-22T06:56:27.014Z"), "latencyStats" : { "reads" : { "histogram" : [{ "micros" : NumberLong(64), "count" : NumberLong(7) }, { "micros" : NumberLong(128), "count" : NumberLong(8) }, { "micros" : NumberLong(256), "count" : NumberLong(4) }, { "micros" : NumberLong(512), "count" : NumberLong(1) }, { "micros" : NumberLong(1024), "count" : NumberLong(1) }], "latency" : NumberLong(5434), "ops" : NumberLong(21) }, "writes" : { "histogram" : [{ "micros" : NumberLong(256), "count" : NumberLong(3) }, { "micros" : NumberLong(512), "count" : NumberLong(22) }, { "micros" : NumberLong(1024), "count" : NumberLong(9) }, { "micros" : NumberLong(2048), "count" : NumberLong(1) }, { "micros" : NumberLong(3072), "count" : NumberLong(1) }, { "micros" : NumberLong(49152), "count" : NumberLong(5) }, { "micros" : NumberLong(65536), "count" : NumberLong(15) }, { "micros" : NumberLong(98304), "count" : NumberLong(10) }, { "micros" : NumberLong(131072), "count" : NumberLong(3) }, { "micros" : NumberLong(196608), "count" : NumberLong(1) }, { "micros" : NumberLong(393216), "count" : NumberLong(2) }, { "micros" : NumberLong(786432), "count" : NumberLong(1) }], "latency" : NumberLong(5031187), "ops" : NumberLong(73) }, "commands" : { "histogram" : [{ "micros" : NumberLong(32), "count" : NumberLong(1) }, { "micros" : NumberLong(2048), "count" : NumberLong(1) }, { "micros" : NumberLong(3072), "count" : NumberLong(2) }, { "micros" : NumberLong(65536), "count" : NumberLong(1) }], "latency" : NumberLong(93917), "ops" : NumberLong(5) }, "transactions" : { "histogram" : [ ], "latency" : NumberLong(0), "ops" : NumberLong(0) } }

```
rs0:PRIMARY> db.game_analysis.latencyStats( { histograms: true } )

{ "ns" : "project.game_analysis", "host" : "IQRAs-MacBook-Air.local:27018", "localTime" : ISODate("2022-05-
22T06:57:06.546Z"), "latencyStats" : { "reads" : { "histogram" : [ { "micros" : NumberLong(64), "count" : NumberLong(2) }, {
"micros" : NumberLong(128), "count" : NumberLong(15) }, { "micros" : NumberLong(256), "count" : NumberLong(14) }, {
"micros" : NumberLong(512), "count" : NumberLong(9) }, { "micros" : NumberLong(1024), "count" : NumberLong(7) }, {
"micros" : NumberLong(2048), "count" : NumberLong(3) }, { "micros" : NumberLong(3072), "count" : NumberLong(2) }, {
"micros" : NumberLong(4096), "count" : NumberLong(1) }, { "micros" : NumberLong(6144), "count" : NumberLong(1) }, {
"micros" : NumberLong(8192), "count" : NumberLong(3) }, { "micros" : NumberLong(12288), "count" : NumberLong(1) } ],
"latency" : NumberLong(91156), "ops" : NumberLong(58) }, "writes" : { "histogram" : [ { "micros" : NumberLong(128),
"count" : NumberLong(1) }, { "micros" : NumberLong(256), "count" : NumberLong(2) }, { "micros" : NumberLong(512),
"count" : NumberLong(2) }, { "micros" : NumberLong(1024), "count" : NumberLong(2) }, { "micros" : NumberLong(6144),
"count" : NumberLong(2) }, { "micros" : NumberLong(24576), "count" : NumberLong(1) }, { "micros" : NumberLong(49152),
"count" : NumberLong(2) } ], "latency" : NumberLong(166644), "ops" : NumberLong(12) }, "commands" : { "histogram" : [ {
"micros" : NumberLong(32), "count" : NumberLong(5) }, { "micros" : NumberLong(64), "count" : NumberLong(2) }, {
"micros" : NumberLong(2048), "count" : NumberLong(4) }, { "micros" : NumberLong(3072), "count" : NumberLong(2) }, {
"micros" : NumberLong(8192), "count" : NumberLong(2) }, { "micros" : NumberLong(98304), "count" : NumberLong(1) } ],
"latency" : NumberLong(148936), "ops" : NumberLong(16) }, "transactions" : { "histogram" : [ ], "latency" : NumberLong(0),
"ops" : NumberLong(0) } } }
```

```
rs0:PRIMARY> db.player_analysis.latencyStats( { histograms: true } )

{ "ns" : "project.player_analysis", "host" : "IQRAs-MacBook-Air.local:27018", "localTime" : ISODate("2022-05-
22T07:14:00.740Z"), "latencyStats" : { "reads" : { "histogram" : [ { "micros" : NumberLong(64), "count" : NumberLong(3) }, {
"micros" : NumberLong(128), "count" : NumberLong(6) }, { "micros" : NumberLong(256), "count" : NumberLong(4) } ],
"latency" : NumberLong(2797), "ops" : NumberLong(13) }, "writes" : { "histogram" : [ { "micros" : NumberLong(24576),
"count" : NumberLong(1) } ], "latency" : NumberLong(30222), "ops" : NumberLong(1) }, "commands" : { "histogram" : [ {
"micros" : NumberLong(128), "count" : NumberLong(1) }, { "micros" : NumberLong(2048), "count" : NumberLong(2) }, {
"micros" : NumberLong(8192), "count" : NumberLong(1) }, { "micros" : NumberLong(65536), "count" : NumberLong(1) } ],
"latency" : NumberLong(90138), "ops" : NumberLong(5) }, "transactions" : { "histogram" : [ ], "latency" : NumberLong(0),
"ops" : NumberLong(0) } } }
```

```
rs0:PRIMARY> db.team_analysis.latencyStats( { histograms: true } )

{ "ns" : "project.team_analysis", "host" : "IQRAs-MacBook-Air.local:27018", "localTime" : ISODate("2022-05-
22T07:15:32.025Z"), "latencyStats" : { "reads" : { "histogram" : [ { "micros" : NumberLong(64), "count" : NumberLong(4) }, {
"micros" : NumberLong(128), "count" : NumberLong(7) }, { "micros" : NumberLong(256), "count" : NumberLong(2) } ],
"latency" : NumberLong(2324), "ops" : NumberLong(13) }, "writes" : { "histogram" : [ { "micros" : NumberLong(1024),
"count" : NumberLong(1) } ], "latency" : NumberLong(1576), "ops" : NumberLong(1) }, "commands" : { "histogram" : [ {
"micros" : NumberLong(64), "count" : NumberLong(1) }, { "micros" : NumberLong(2048), "count" : NumberLong(2) }, {
"micros" : NumberLong(3072), "count" : NumberLong(1) }, { "micros" : NumberLong(65536), "count" : NumberLong(1) } ],
"latency" : NumberLong(99903), "ops" : NumberLong(5) }, "transactions" : { "histogram" : [ ], "latency" : NumberLong(0),
"ops" : NumberLong(0) } }
```

6.2 Output Analysis

6.2.1 Comparison for data loading between Cassandra and MongoDB

The execution time required for loading data for both the databases has been summarized in the table 6.

Data loading	Cassandra (ms)	MongoDB (ms)
Team_analysis	398	1
Player_analysis	659	0
Game_analysis	4275	4
Basketball_analysis	103549	26

Table 6: Comparison for Data loading between Cassandra and MongoDB

We plotted a graph for comparing the time required for loading data in Cassandra and MongoDB. It can be seen that loading time increases as the number of records increases for cassandra. For MongoDB, there is not a significant increase in the loading time as compared to Cassandra. Hence, it can be said that time required for loading data is less in MongoDB as compared to cassandra.

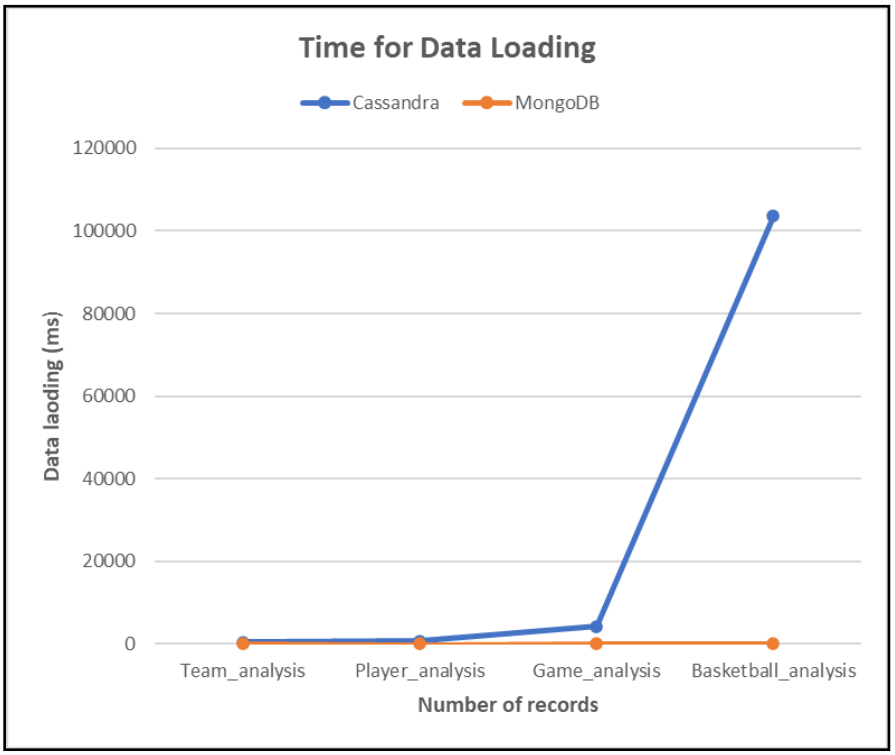


Figure 70: Time For data loading for Cassandra and MongoDB

6.2.2 Comparison between execution time for various CRUD operations for Cassandra and MongoDB

The table below shows a summarized view of the time taken for various CRUD operations for both the databases.

CRUD Operations	Cassandra (ms)	MongoDB (ms)
Insert	41.616	42
Read	15.403	7
Update	54.020	63
Delete	33.122	5

Table 7: Comparison for CRUD operations between Cassandra and MongoDB

The graph below (Fig 71) shows a comparison for execution time for each CRUD operation for Cassandra and MongoDB. It can be seen that MongoDB takes higher time to update data as compared to cassandra. Time required for inserting a record is almost similar (slightly higher for cassandra) whereas for read as well as delete operations, cassandra takes higher execution time.

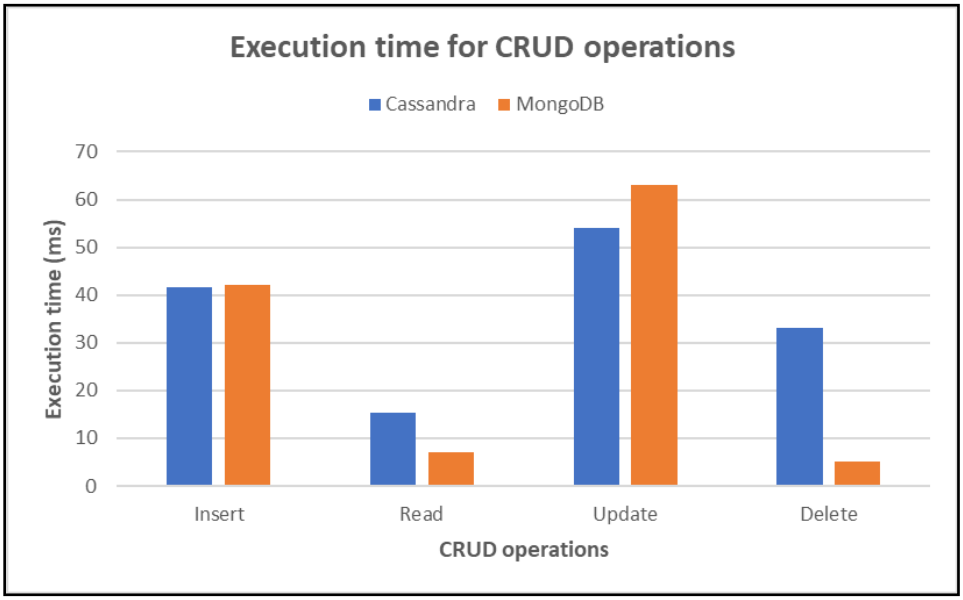


Figure 71: Execution time for CRUD operations for Cassandra and MongoDB

Comparison for execution time for different consistency levels

The time required for READ and WRITE operations for different consistency levels is summarized in Table 8

CRUD	Consistency levels	Cassandra (ms)	MongoDB (ms)
Read	One	225.147	1
	Quorum	288.763	5
Update	One	108.951	70
	Quorum	162.396	566
Insert	One	85.303	2
	Quorum	111.234	36

Table 8: Comparison for different consistency levels for CRUD operations

For Cassandra, CAP theorem is AP i.e. Availability and partition tolerance. By default, Cassandra is highly available i.e. it takes much less time responding to the queries. So for normal CRUD operations, time required is less as compared to the time required for these operations for different consistency levels. When the consistency is set to one, that means only one node needs to respond. As only one replica node has to acknowledge, the time required for read and write operations for consistency one is less as compared to the scenario when the consistency is set to quorum.

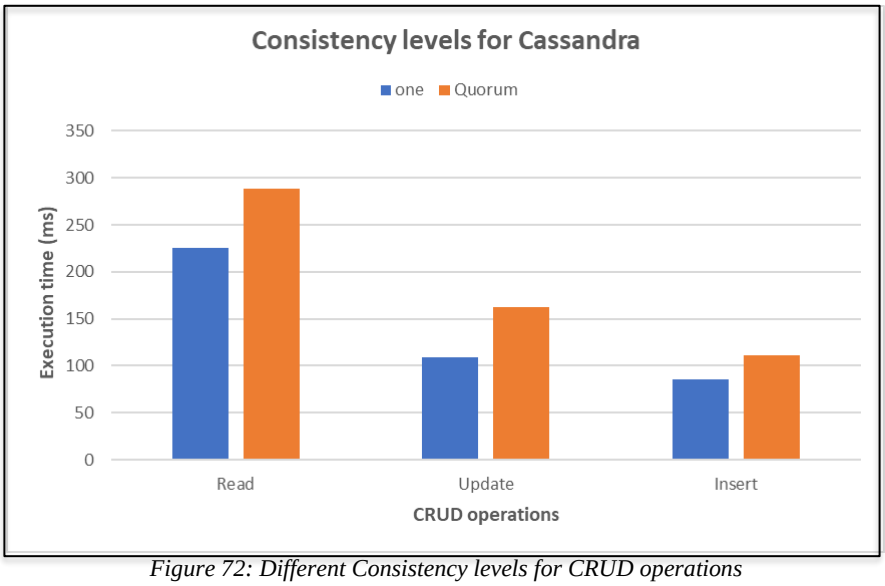


Figure 72: Different Consistency levels for CRUD operations

For MongoDB, CAP theorem is CP ie. Consistent and partition tolerance. By default, MongoDB is highly consistent i.e it takes much more time in responding to the queries.So for normal CRUD operations, time required is comparatively more as compared to the time required for CRUD operations at different consistency levels.
With reference to the table above, we can see that when the consistency is set to one, it returns data from the instance (ie. without no guarantee that the data has been returned to a majority of the replica nodes), so time taken is less approximately 1ms and 2 ms for read and update operation respectively.
However, when the consistency level is set to quorum(ie. the data is more consistent), we can see that time taken was more for read and insert operations ie. around 5 ms and 30 ms respectively .

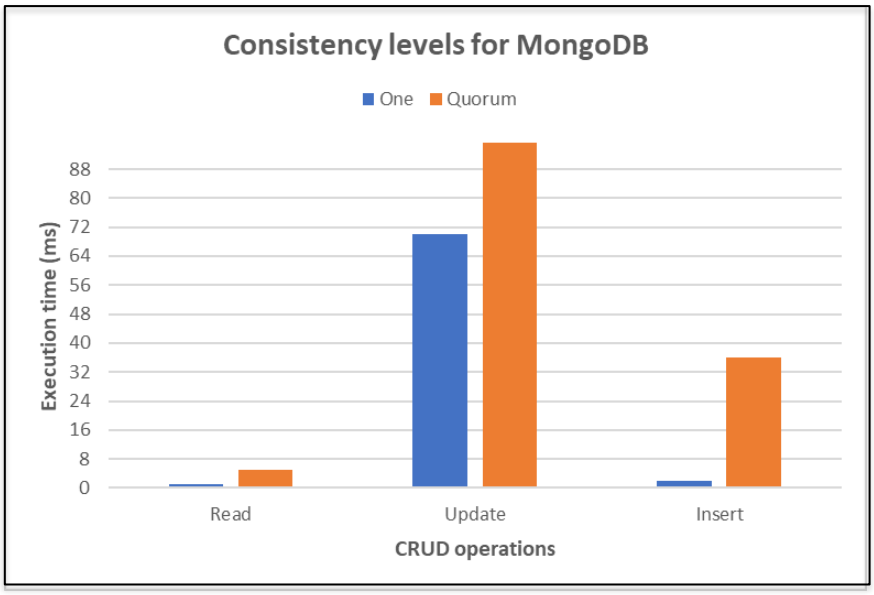


Figure 73: Different consistency levels for CRUD operations MongoDB

6.2.3 Latency measurements

Cassandra

Table 9 shows the summarised table for read and write latencies.

Read Operation		Write Operation	
Count	**Time Taken (ms)**	**Count**	**Time Taken(ms)**
9	11.994	2	0.118
5	10.532	3	0.239
14	10.311	36052	0.8415

Table 9: Latency measurements for Cassandra

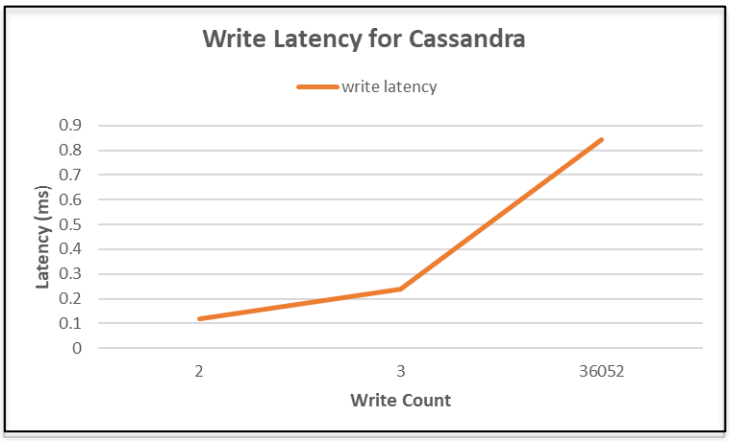


Figure 74: Write latency Cassandra

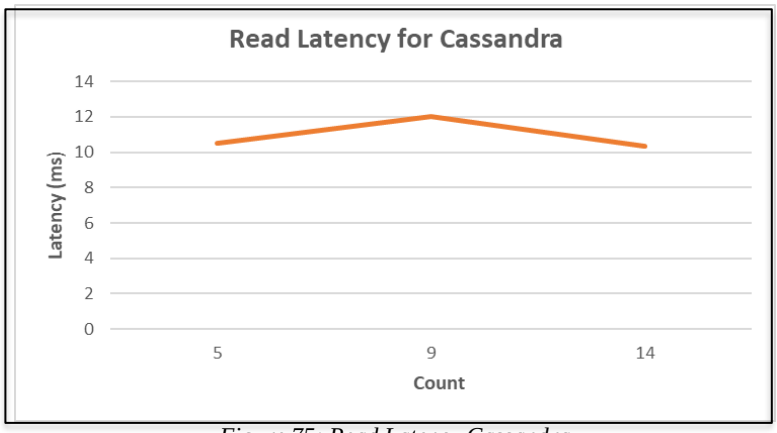


Figure 75: Read Latency Cassandra

It can be seen from the above graphs that as the number of write counts increases, the write latency also increases from 0.1 to around 0.8 ms. However for READ operation, as the number of count

increases from 5 to 14, latency is in the range of 10-12 ms i.e we observe very slight increase in the read latency for increase in the read count**.**

Table 10 summarizes the read and write latencies for increasing number of read and write counts for MongoDB

Read Operation		**Write Operation**	
Count	**Time Taken (ms)**	**Count**	**Time Taken (ms)**
2	0.64	2	2.048
4	1.28	4	2.56
6	1.28	6	49.152
8	1.024	10	98.304
10	5.12	12	166.644
20	5.434	15	148.936

Table 10:Latency measurements for MongoDB

For MongoDB, we can observe from fig 76 that as the number of records increases, the update latency keeps on increasing whereas the read latency is almost stable with very slight increase with the increase in the number of read counts.

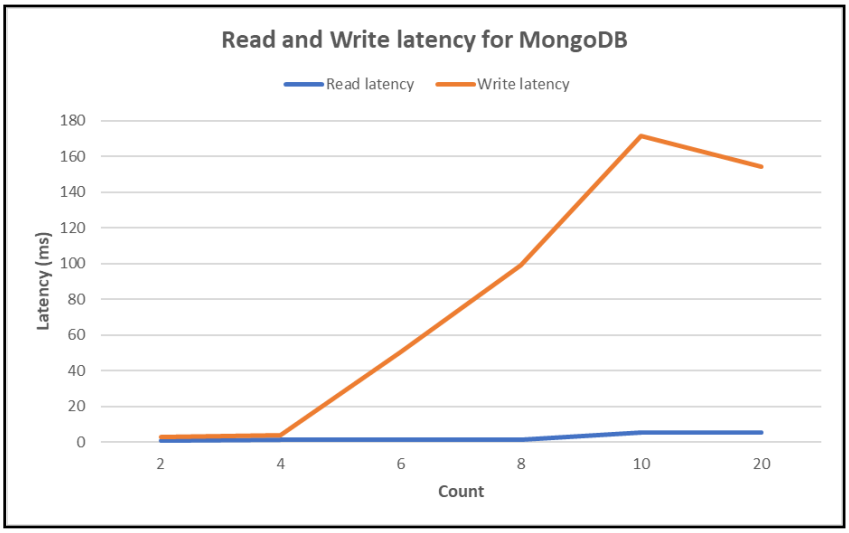


Figure 76: read and Write Latency for MongoDB

6.3 Compare output Against Hypothesis

Hypothesis 1: To test and verify if MongoDB performs better in CRUD operations and Cassandra performs better in write operations.

As per our analysis, we observe that MongoDB is better for READ and DELETE operation. Execution time for INSERT operation is almost similar for both the databases whereas Cassandra performs better for UPDATE operation. The results obtained from our analysis proves our initial hypothesis.

Hypothesis 2: To test and verify if Cassandra has low update and read latency as compared to MongoDB, when the load increases.

There is not a significant increase in read and write latency for Cassandra as the number of counts increases. However for MongoDB, the write latency increases significantly with the number of write counts whereas the read latency is increasing slightly. Similar results were observed from the research papers. Hence, it can be stated that our initial hypothesis are verified.

6.4 Data Visualization

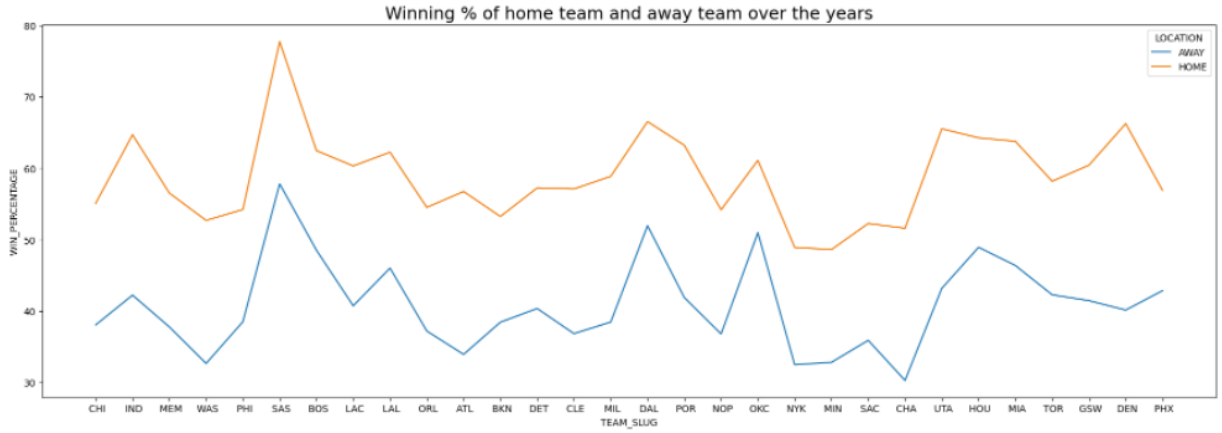


Figure. 77: Winning percentage of teams

Figure. 77. shows the winning percentage of the team when they have played as a home team and as an away team, aggregated from the year 2000 to the present. It is intriguing to see that for every team, their winning the percentage is higher when they have played in their home location. Although the factors leading to this huge difference could be psychometric or unknown, in this project, we investigate this from the attributes available in our data. Two such factors that would have contributed are the number of team fouls and free throw percentage. In the following analysis, we will deep dive into the influence and significance of these factors on the winning percentage.

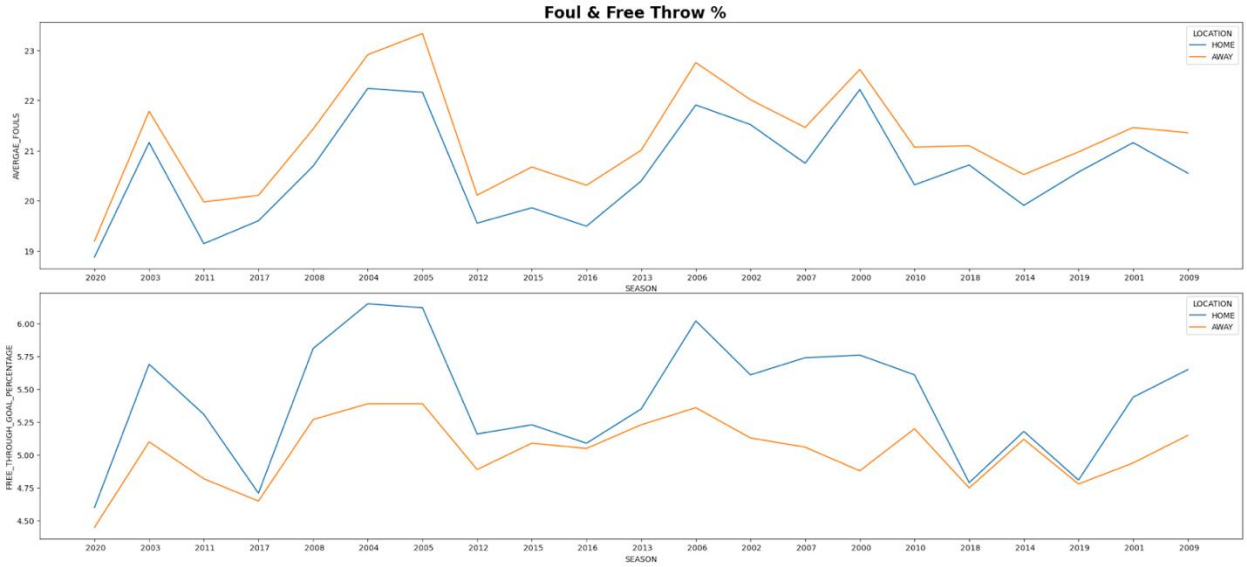


Figure. 78: Percentage of Foul and free throw.

Figure 78, clearly shows that the average fouls given to the home teams are always less compared to the away teams. In fact, the difference is significant for many seasons. This finding is in line with the increase in free-throw goal percentage of home teams compared to the away teams as shown in the fig. More fouls are given to the away teams, which led the home team to get an increased number of free throws, and thus they have a higher percentage of free throws compared to the away teams. This could also explain the significant difference between the winning percentage of the home teams and the away teams.

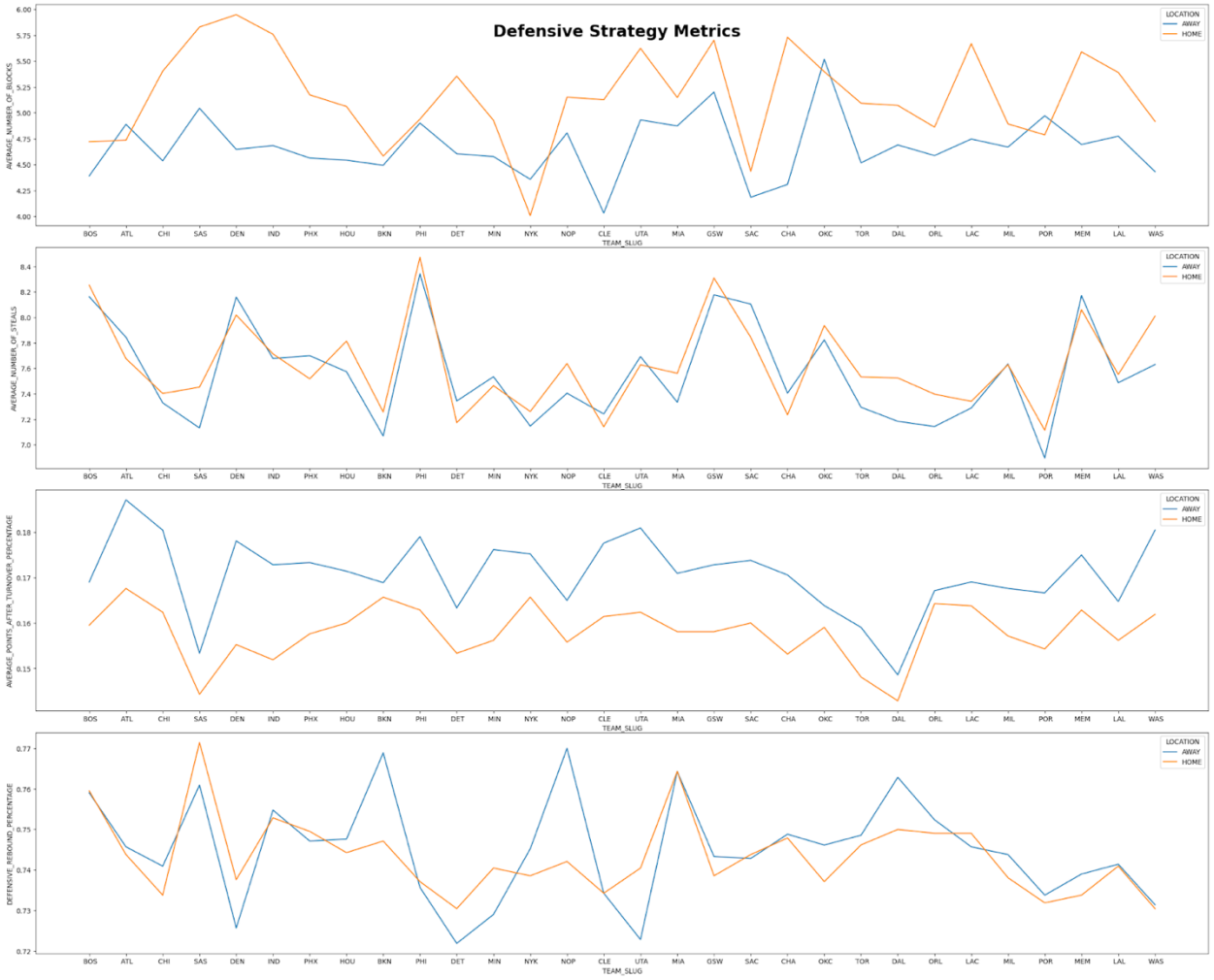


Figure. 79: Defensive strategy metrics

Figure 79, shows the defense metrics of all the teams when they played as a home team and as an away team. We can see that all the teams except a few are good at blocking the opponent when they play as a home team. This could be a factor that contributes to the home team win as they will be blocking their opponents and preventing them from scoring. But surprisingly the points scored after turnover is less for the home team compared to the away teams. The other defense metrics such as defensive rebound and number of steals don't show much difference between the home and away games.

Figure. 80: Offensive Strategy metrics

Figure 80, The offense metrics of all the teams for their home games and away games are shown in. We can see that the average number of assists per game is relatively higher for most of the

teams when they play in their home location. The teams DEN and GSW have a higher number of assists compared to other teams.The other metrics such as offense rebound, points scored in the paint region and 2nd chase points have no significant difference for the team's home and away games. Also, we can see that the teams that have higher percentage of offensive rebound have good 2nd chance point score percentage. This is in line with the fact that the offensive rebound is important for a team to score points by 2nd chance, thus increasing the overall score.

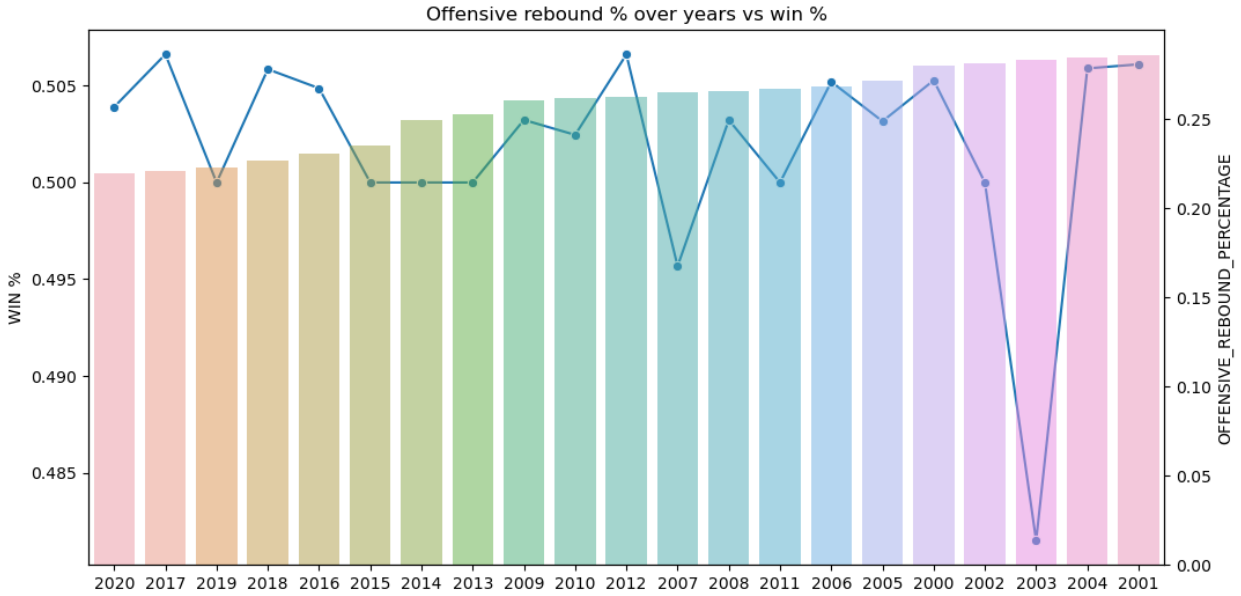


Figure. 81: Offensive rebound percentage

Figure 81, shows the percentage of offensive rebound of all the teams and the yellow line shows the winning percentage of the teams. It is surprising to see that there is a significant decrease in the average rebound for all the teams over the years. This could be an important fact for a team to work on, to increase their winning percentage.

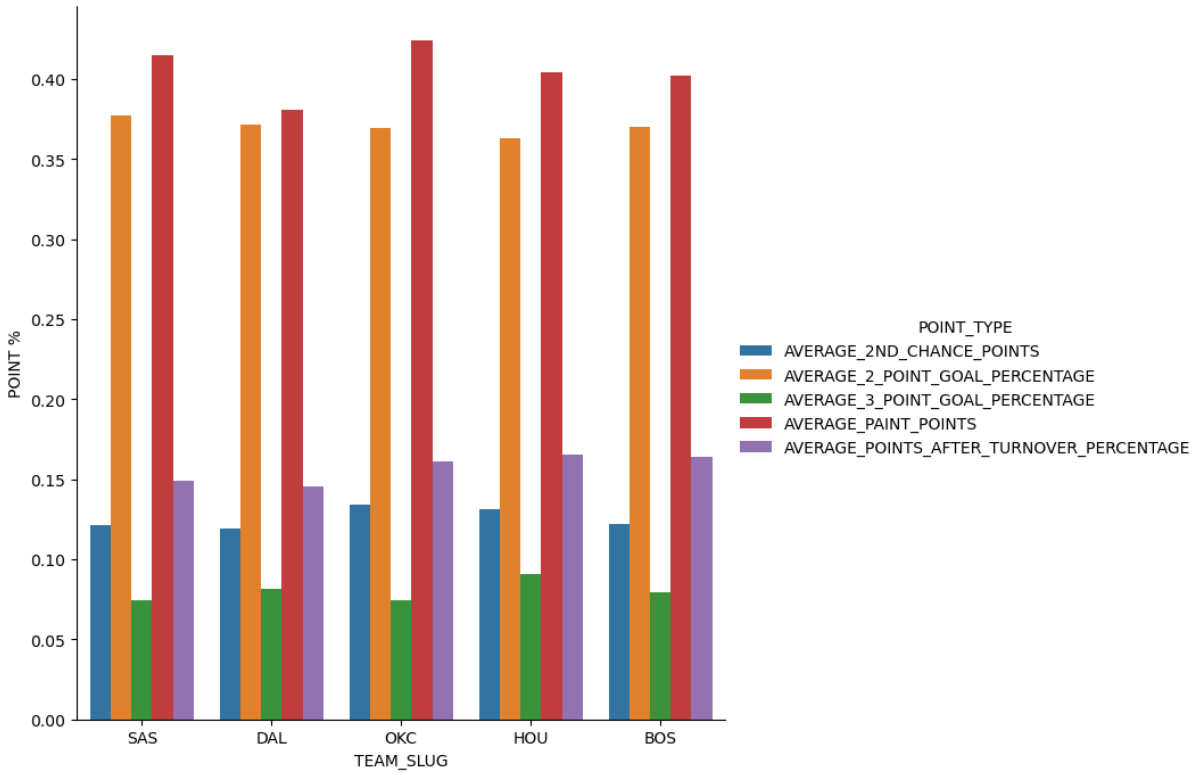


Figure. 82: Point percentage for various teams

Figure 82, show the distribution of overall points as percentage scored as two points, three points, one point (free throw) and 2nd chance points. The distribution of points in percentage shows that the points scored in the paint region and within the three point circle hold the highest percentages compared to the others. The points that are scored outside the three-point circle have the least percentage. Even though the three point goal has more weightage to the overall score, it is scored less regardless of the team. This shows that the three points is the most difficult score to make compared to others.

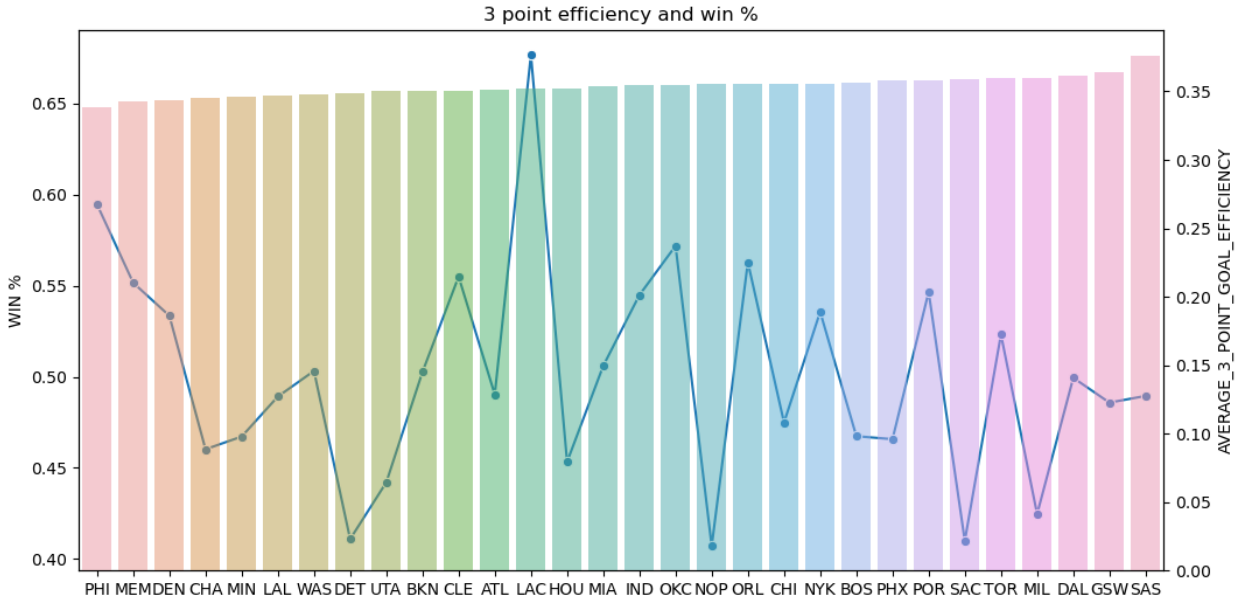


Figure. 83: 3 point efficiency and win %

Figure 83, depicts the percentage of three points efficiency for all the teams shown by bars along with the average winning percentage of each team shown by the yellow line graph. We can observe that most of the teams whose winning percentage is higher than the average winning percentage are efficient in their three point goals. This indicates that the three points score efficiency is an important factor for a team to 38 win. The team SAS shows profound difference compared to others, having the highest three point goal efficiency and also holding the highest winning rate.

MongoDB charts

Graphs were also plotted directly using Mongo charts to study the performance of players and teams which are attached in this section. Fig 77 shows the win count percentage of the teams when they have played on home ground or elsewhere. It can be observed that the chances of winning of a team when they play on their home ground is higher as compared to the game played elsewhere.

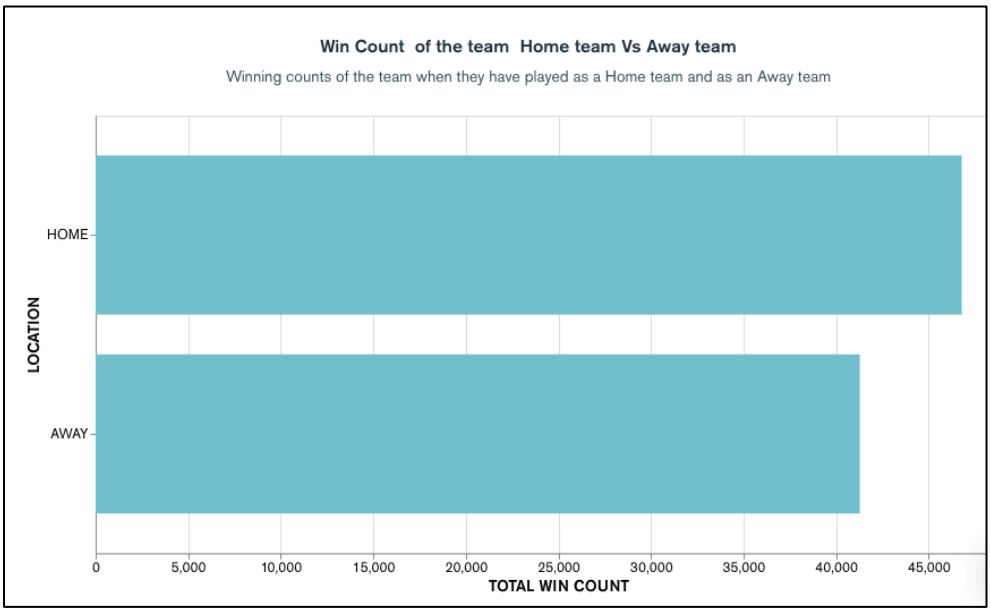


Figure 77: Win count for teams (Home team and Away team)

Fig 78 shows the Free throw percentage at home ground or other locations over the years. It can be observed throughout the various seasons that the percentage of free throw when game is played on home ground is higher than the game played on other location. Similar trend is also observed for the average number of assists for various seasons which is shown in fig. 79.

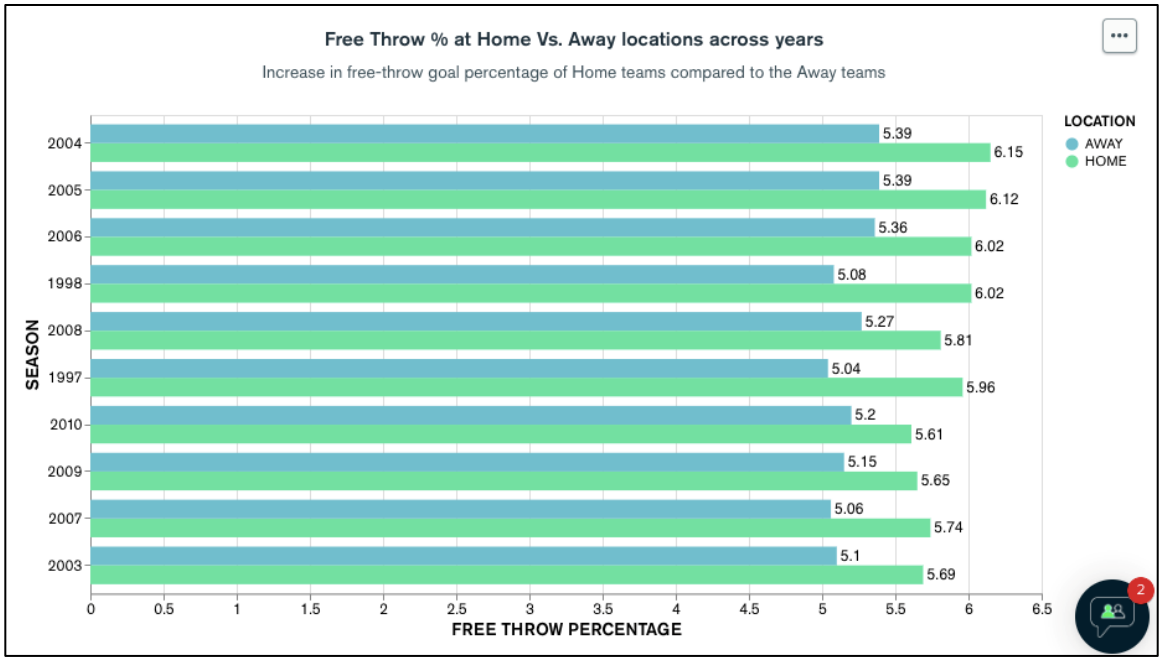


Figure 78: Free throw %

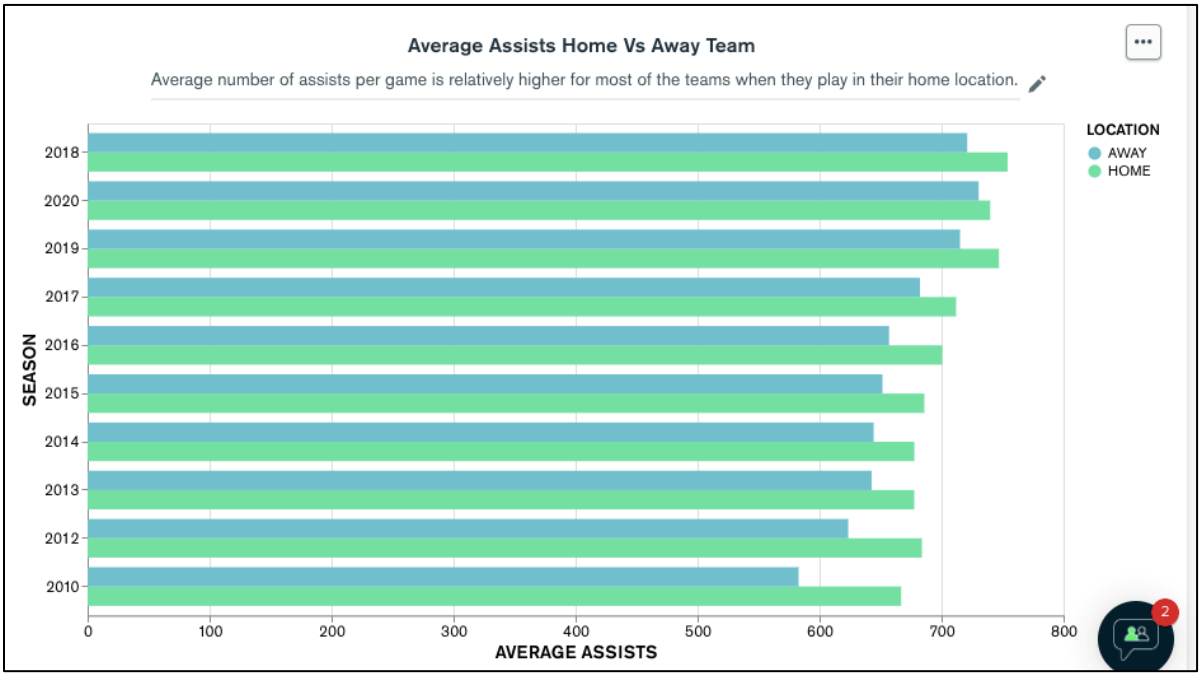


Figure 79: Average Assists home Vs away team

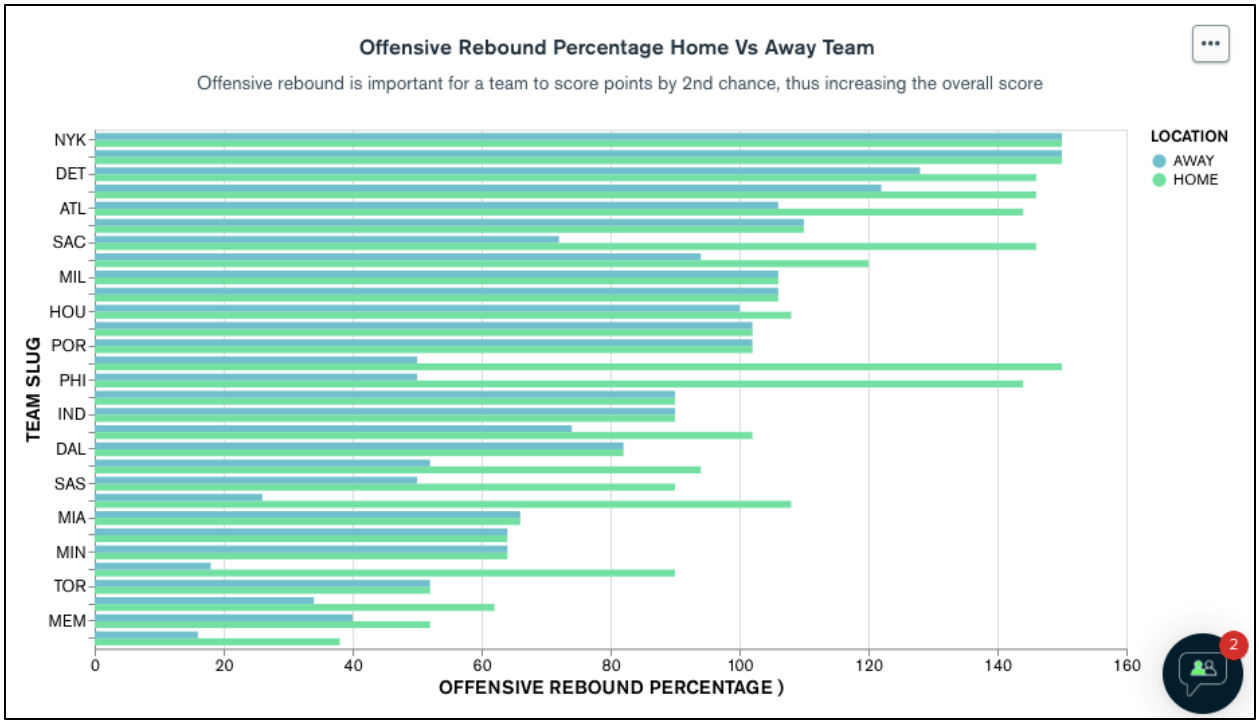


Figure 80: Offensive Rebound Percentage Home Vs Away Team

7 Conclusions and Recommendations

7.1 Summary and Conclusions

We have analyzed the performance for MongoDB and Cassandra by measuring the execution time required for CRUD operations, its read and write latency based upon the number of read and write counts as well as by looking at the consistency of each database. We also analyzed the efficiency of NBA players using MongoDB charts. This project was mainly carried out to get an idea on the positive and negative aspects of each database and help in making decisions to choose the best database as per the application.

We did our analysis in four parts:

- Firstly, we measured the time required to load data into both the databases. It was pretty evident from the graphs that MongoDB loads the data faster in comparison to Cassandra. We then performed various CRUD operations on the dataset. It was observed that except for UPDATE operation, MongoDB was much more efficient than Cassandra
- Secondly, we then varied the consistency levels for Cassandra and MongoDB in-order to have an understanding about their performance based on the consistency levels

  For Cassandra, It was observed that when the consistency is set to one, the time required for each CRUD operation to be executed was less as compared to the time required for Quorum consistency.

  Whereas for MongoDB, at consistency set to 1, time required for read/write operation was less as compared to higher consistency levels. This is mainly due to CAP theorem, which is CP for MongoDB i.e. DB is highly consistent

- Thirdly, we measured the latency for each database in-order to have an understanding of the latency trends with respect to the number of read and write counts. It was observed that Cassandra performed much better in terms of latency as the number of counts increased. Latency for MongoDB rose significantly when write count was more than 5. However, on the other hand, Cassandra was still able to maintain latency for higher counts at a steady rate
- Lastly, we tried to evaluate the performance of the NBA players using the MONGO charts in Atlas as well as using a Jupyter notebook by connecting Atlas to python. It was observed that doing data analysis using mongo charts is much more efficient as data is always up-to-date and one can get accurate statistics for better data analysis.

7.2 Recommendations for future studies

- Large amount of read and write operations can be performed to better understand the latency patterns as well as the CRUD operations of Cassandra and MongoDB
- Analysis can be performed for more records and on cloud which shall help to get a more concrete idea about performance of both the databases
- Comparison between Cassandra and MongoDB can be done on real time data for on-field game analytics to get better insights of their performance metrics with respect to real time
- Doing data analysis using Mongo charts had an additional advantage that the dashboard keeps updating after a couple of hours. So this is useful in order to do analysis on current data rather than stale data. Hence, using Mongo Charts for analysis would be fruitful in sports analytics as dashboard will be always in latest mode
- Horizontal scalability can also be measured by increasing the number of nodes.

8 Bibliography

[1] Hendawi, A., Gupta, J., Jiayi, L., Teredesai, A., Naveen, R., Mohak, S., & Ali, M. (2019). Distributed NoSQL Data Stores: Performance Analysis and a Case Study. *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*, 1937–1944. https://doi.org/10.1109/BigData.2018.8622544

[2] Seghier, N. ben, & Kazar, O. (2021). *Performance Benchmarking and Comparison of NoSQL Databases: Redis vs MongoDB vs Cassandra Using YCSB Tool*. 1–6. https://doi.org/10.1109/icrami52622.2021.9585956

[3] Anusha, K., Rajesh, N., Kavitha, M., & Ravinder, N. (2021). Comparative Study of MongoDB vs Cassandra in big data analytics. *Proceedings - 5th International Conference on Computing Methodologies and Communication, ICCMC 2021*, 1831–1835. https://doi.org/10.1109/ICCMC51019.2021.9418441

[4] Dharavath Ramesh, Anand Kumar (2018) Query Driven implementation of Twitter base using Cassandra. *Proceeding of 2018 IEEE International Conference on Current Trends toward Converging Technologies, Coimbatore, India*

[5] Ferreira, L., de Oliveira, F. S., da Rocha, O. P., Holanda, M., de Carvalho Victorino, M., & Ribeiro, E. (2021, June 23). MongoDB: Analysis of Performance with Data from the National High School Exam (Enem). *Iberian Conference on Information Systems and Technologies, CISTI*. https://doi.org/10.23919/CISTI52073.2021.9476248

[6] Araujo, J. M. A., de Moura, A. C. E., da Silva, S. L. B., Holanda, M., Ribeiro, E. D. O., & da Silva, G. L. (2021, June 23). Comparative Performance Analysis of NoSQL Cassandra and MongoDB Databases. *Iberian Conference on Information Systems and Technologies, CISTI*. https://doi.org/10.23919/CISTI52073.2021.9476319

[7] Sarlis, V., & Tjortjis, C. (2020). Sports analytics — Evaluation of basketball players and team performance. *Information Systems, 93*. https://doi.org/10.1016/j.is.2020.101562

9 Appendices

9.1 Program source code with documentation

Source code is submitted online in the mail

9.2 Input/ Output listing

The complete process flow diagram has been given in section 5.2

Figure Number	**Figure Title**
Figure 37	Flowchart for Cassandra
Figure 38	Flowchart for MongoDB