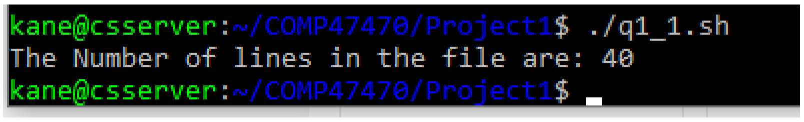
Rohit Jyotiba Chougule **Student Number- 19200240**

COMP47470 Big Data Programming- Project 1

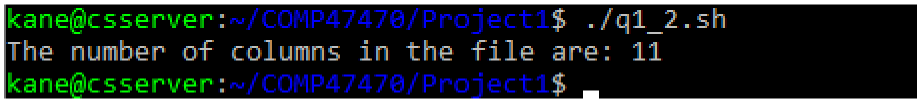
1. BASH for Big Data Analysis. (Refer Q1 section in README.txt)

1. Command used in script: Run the script *q1_1.sh*
   `tail crimedata-australia.csv -n +2| wc -l`
   Using tail with the csv files to exclude first line by specifying the parameter -n. Piping the output of tail to wc -l
   Output-



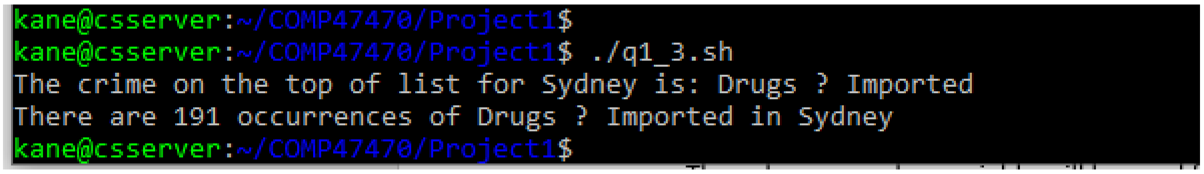
```
kane@csserver:~/COMP47470/Project1$ ./q1_1.sh
The Number of lines in the file are: 40
kane@csserver:~/COMP47470/Project1$
```

2. Command used in script: Run the script *q1_2.sh*
   `head -1 crimedata-australia.csv | sed 's/[^,]//g' | wc -c`
   Using the head command to get header of the file by passing the parameter 1, sed will help in identifying column names by comma separated values, wc -c results in word count in the output of sed.
   Output-



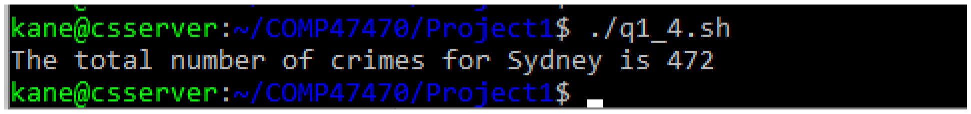
```
kane@csserver:~/COMP47470/Project1$ ./q1_2.sh
The number of columns in the file are: 11
kane@csserver:~/COMP47470/Project1$
```

3. Run the script *q1_3.sh*
   The column_number variable will be used for specifying the on which column the operation should performed. city_name variable gives the name of the city for which the number was provided. The variable top_crime will give the topmost crime in the specified city as the result is sorted. The city that needs to be checked can be changed by changing the variable column_number.
   Output-



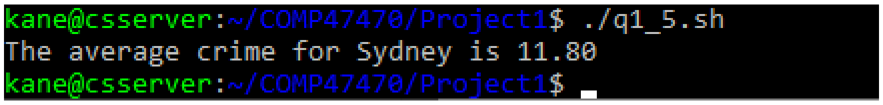
```
kane@csserver:~/COMP47470/Project1$
kane@csserver:~/COMP47470/Project1$ ./q1_3.sh
The crime on the top of list for Sydney is: Drugs ? Imported
There are 191 occurrences of Drugs ? Imported in Sydney
kane@csserver:~/COMP47470/Project1$
```

4. Run the script *q1_4.sh*
   The same logic from the question3 script is being used in this one, additionally, to compute the sum awk has been used, which will sum up all the rows from the piped output.
   Output-



```
kane@csserver:~/COMP47470/Project1$ ./q1_4.sh
The total number of crimes for Sydney is 472
kane@csserver:~/COMP47470/Project1$
```

5. Run the script *q1_5.sh*
   The same script from question 4 and question 1 can be used in combination to get the average number of crimes.
   Output-

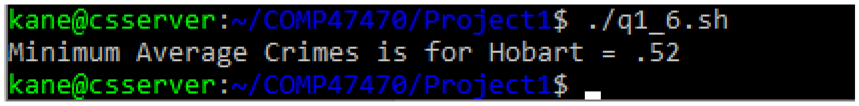


```
kane@csserver:~/COMP47470/Project1$ ./q1_5.sh
The average crime for Sydney is 11.80
kane@csserver:~/COMP47470/Project1$
```

6. Run the script *q1_6.sh*

   The script from question1, 5 is used in combination to compare the average of all the cities, by setting initial minimum average to a maximum possible number which helps in finding the city with minimum average crime.

   Output-



```
kane@csserver:~/COMP47470/Project1$ ./q1_6.sh
Minimum Average Crimes is for Hobart = .52
kane@csserver:~/COMP47470/Project1$ _
```

2. **Data Management:**

1. Short Description of the two datasets:

   The teams dataset has a list of worldwide teams playing football in a tournament, and the attributes - ranking: which suggest the standing of the team after the tournament, games: are the number of games the team played in the tournament, wins: The number of matches won by the team, draws: the matches that did not end with any result, losses: number of losses faced by the team, goalsFor: Are the number of goals scored by the team, goalsAgainst: The number of goals scored against the team, yellowCard: The number of yellow cards the players received during the tournament , redCard: The number of red cards the players received during the tournament.

   The Players dataset has details on the list of players, the team they play for, their position in the field, number of minutes they played, number of shots by the player, number of passes, tackles and saves by the player.

2. Creating the database in MySQL with tables to represent the dataset.

   From going through the structure of the csv files, it seems that the data stored in the csv is from a football tournament, where in one of the csv files has data related to the teams participated and their rankings along with other stats for the goals scored, wins, losses and the number of yellow & red cards the team got. Whereas the other csv consists of players individual records, which is related to the teams by the attribute team, where in we can use the team attribute as a foreign key in the players table from reference to the teams.

   The database and tables can be created using the BASH script file- *q2_2_sql_script.sh*

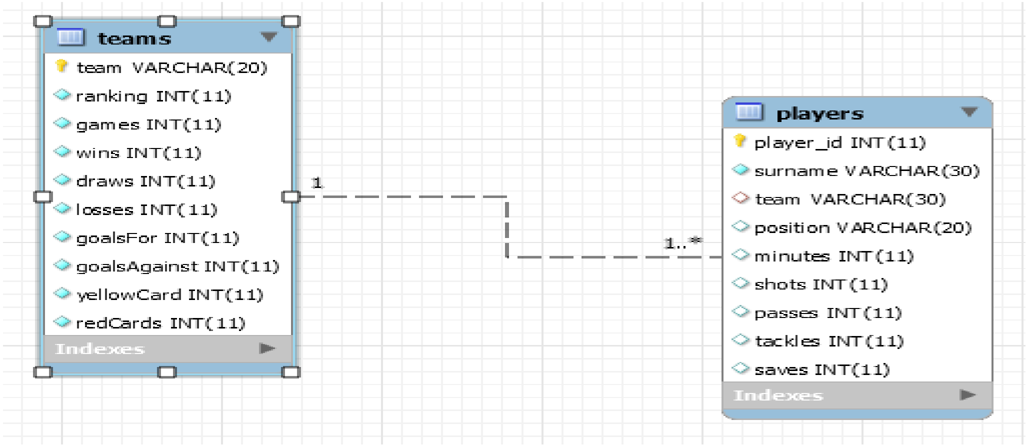
   The Queries for creating database are as follows-

   DROP DATABASE IF EXISTS football;

   CREATE DATABASE football;

   However, the Linux machine server allows to use only the database of our username, hence we use the same database:

   ER Diagram- (Database: football) (used- kane)

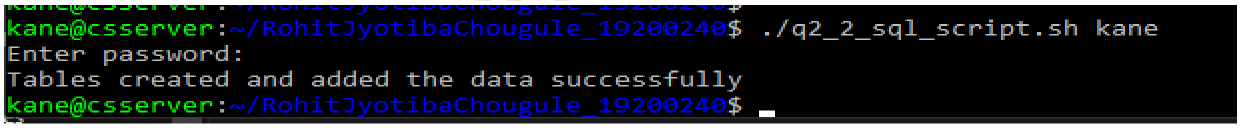


use kane; (username)

Table: *teams*
CREATE TABLE teams(team VARCHAR(20) PRIMARY KEY, ranking INT NOT NULL, games INT NOT NULL, wins INT NOT NULL, draws INT NOT NULL, losses INT NOT NULL, goalsFor INT NOT NULL, goalsAgainst INT NOT NULL, yellowCard INT NOT NULL, redCards INT NOT NULL);

Table: *players*
CREATE TABLE players(player_id INT NOT NULL AUTO_INCREMENT, surname VARCHAR(30) NOT NULL, team VARCHAR(20), position VARCHAR(20), minutes INT, shots INT, passes INT, tackles INT, saves INT, FOREIGN KEY (players.team) REFERENCES teams(team));

Alternatively, the script *q2_2_sql_script.sh* can be executed which creates tables and populates them. Execute the script as directed in the *README.txt: q2_2_sql_script.sh*

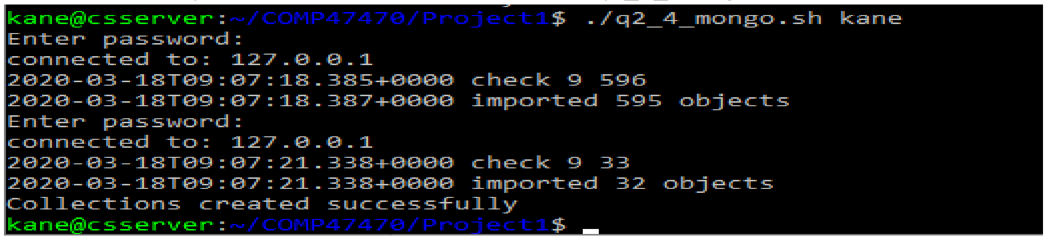

```
kane@csserver:~/RohitJyotibaChougule_19200240$ ./q2_2_sql_script.sh kane
Enter password:
Tables created and added the data successfully
kane@csserver:~/RohitJyotibaChougule_19200240$ _
```

3. If the script *q2_2_sql_script.sh* is already executed, no need to execute these commands again, commands in the shell script above are:

   load data local infile 'Teams.csv' into table teams fields terminated by ',' lines terminated by '\n';

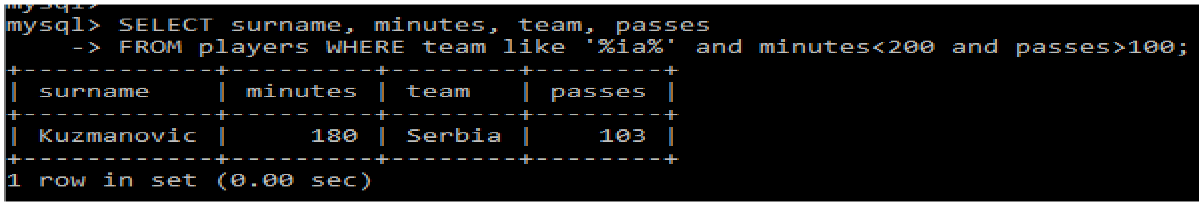
   load data local infile 'Players.csv' into table players fields terminated by ',' lines terminated by '\n' (surname, team, position, minutes, shots, passes, tackles, saves);

4. Execute the script as directed in the *README.txt*: *q2_4_mongo.sh*

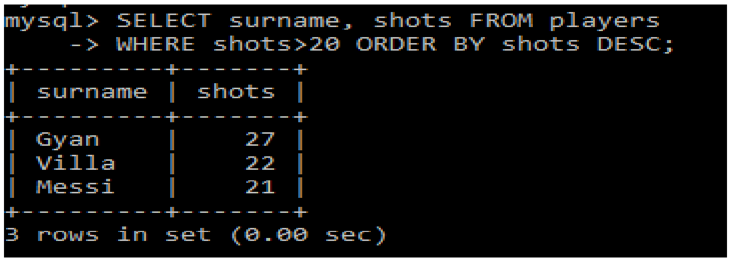

```
kane@csserver:~/COMP47470/Project1$ ./q2_4_mongo.sh kane
Enter password:
connected to: 127.0.0.1
2020-03-18T09:07:18.385+0000 check 9 596
2020-03-18T09:07:18.387+0000 imported 595 objects
Enter password:
connected to: 127.0.0.1
2020-03-18T09:07:21.338+0000 check 9 33
2020-03-18T09:07:21.338+0000 imported 32 objects
Collections created successfully
kane@csserver:~/COMP47470/Project1$ _
```

- Answers to the Questions using SQL queries and Mongo searches: **(All SQL queries saved in q2_sql_queries.sql)**

1. *SELECT surname, minutes, team, passes FROM players WHERE team like '%ia%' and minutes<200 and passes>100;*
   where clause with like operator for the regular expression retrieves teams with 'ia' and conditions for minutes and passes columns.
   Results:

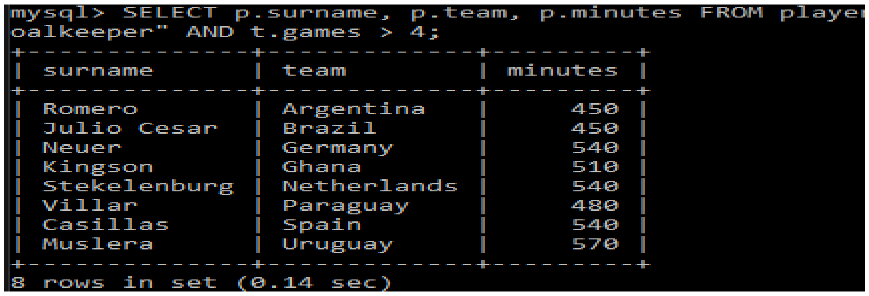

```
mysql> SELECT surname, minutes, team, passes
    -> FROM players WHERE team like '%ia%' and minutes<200 and passes>100;
+------------+---------+--------+--------+
| surname    | minutes | team   | passes |
+------------+---------+--------+--------+
| Kuzmanovic |     180 | Serbia |    103 |
+------------+---------+--------+--------+
1 row in set (0.00 sec)
```

2. *SELECT surname, shots FROM players WHERE shots>20 ORDER BY shots DESC;*
   Where clause to retrieve the results on condition for shots column, ORDERBY DESC sorts the results in descending.
   Results:



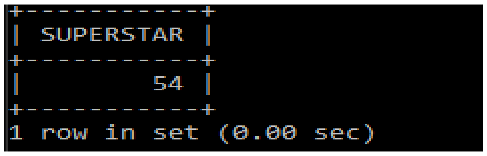
```
mysql> SELECT surname, shots FROM players
    -> WHERE shots>20 ORDER BY shots DESC;
+---------+-------+
| surname | shots |
+---------+-------+
| Gyan    |    27 |
| Villa   |    22 |
| Messi   |    21 |
+---------+-------+
3 rows in set (0.00 sec)
```

3. *SELECT p.surname, p.team, p.minutes FROM players p inner join teams t ON p.team = t.team WHERE p.position LIKE "goalkeeper" AND t.games > 4;*
   Joining on teams column in both tables, as it's a foreign key in Players table, and using LIKE for for regular expression.
   Results:



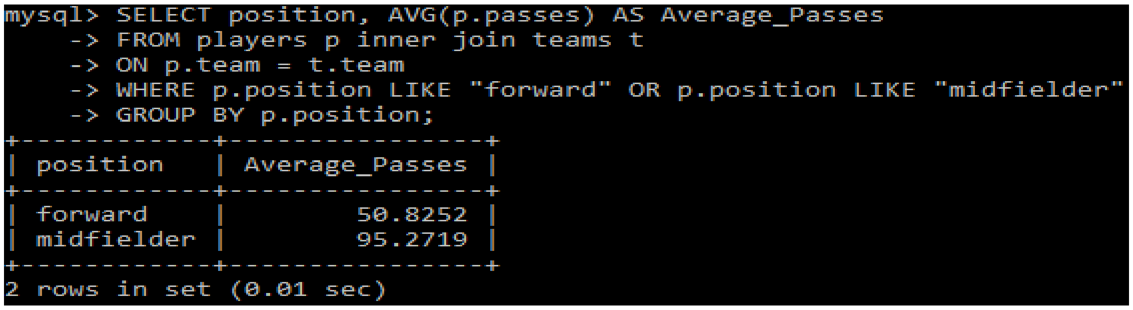
```
mysql> SELECT p.surname, p.team, p.minutes FROM playe
oalkeeper" AND t.games > 4;
+--------------+-------------+---------+
| surname      | team        | minutes |
+--------------+-------------+---------+
| Romero       | Argentina   |     450 |
| Julio Cesar  | Brazil      |     450 |
| Neuer        | Germany     |     540 |
| Kingson      | Ghana       |     510 |
| Stekelenburg | Netherlands |     540 |
| Villar       | Paraguay    |     480 |
| Casillas     | Spain       |     540 |
| Muslera      | Uruguay     |     570 |
+--------------+-------------+---------+
8 rows in set (0.14 sec)
```

4. *SELECT count(*) AS SUPERSTAR FROM players p inner join teams t ON p.team = t.team WHERE t.ranking<10 AND p.minutes>350;*
   Using the count for getting count of records from joining the tables on column team on rank & minutes condition.
   Results:



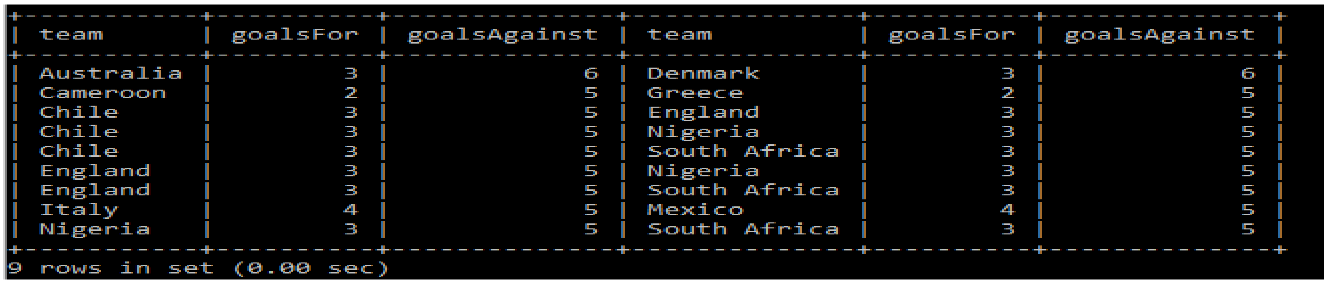
```
+-----------+
| SUPERSTAR |
+-----------+
|        54 |
+-----------+
1 row in set (0.00 sec)
```

5. *SELECT position, AVG(p.passes) AS Average_Passes FROM players p inner join teams t ON p.team = t.team WHERE p.position LIKE "forward" OR p.position LIKE "midfielder" GROUP BY p.position;*
   Using AVG function on the column passes, using aliases and joining tables on column team with conditions on LIKE operator and using GROUPBY to group the aggregated results.
   Results:



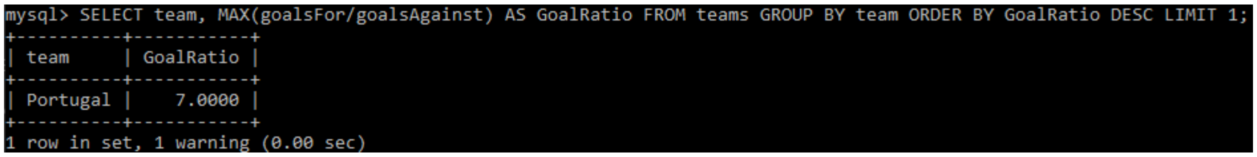
```
mysql> SELECT position, AVG(p.passes) AS Average_Passes
    -> FROM players p inner join teams t
    -> ON p.team = t.team
    -> WHERE p.position LIKE "forward" OR p.position LIKE "midfielder"
    -> GROUP BY p.position;
+------------+----------------+
| position   | Average_Passes |
+------------+----------------+
| forward    |        50.8252 |
| midfielder |        95.2719 |
+------------+----------------+
2 rows in set (0.01 sec)
```

6. *SELECT t1.team, t1.goalsFor, t1.goalsAgainst, t2.team, t2.goalsFor, t2.goalsAgainst FROM teams t1, teams t2 WHERE t1.goalsFor = t2.goalsFor AND t1.goalsAgainst = t2.goalsAgainst AND t1.team < t2.team;*
   Self joining teams table with itself, to compare number of goalsFor and goalsAgainst, t1.team < t2.team ensures that there are no repeated results retrieved for two teams.
   Results:



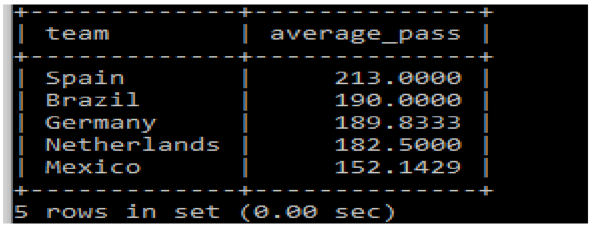
```
+-----------+----------+--------------+--------------+----------+--------------+
| team      | goalsFor | goalsAgainst | team         | goalsFor | goalsAgainst |
+-----------+----------+--------------+--------------+----------+--------------+
| Australia |        3 |            6 | Denmark      |        3 |            6 |
| Cameroon  |        2 |            5 | Greece       |        2 |            5 |
| Chile     |        3 |            5 | England      |        3 |            5 |
| Chile     |        3 |            5 | Nigeria      |        3 |            5 |
| Chile     |        3 |            5 | South Africa |        3 |            5 |
| England   |        3 |            5 | Nigeria      |        3 |            5 |
| England   |        3 |            5 | South Africa |        3 |            5 |
| Italy     |        4 |            5 | Mexico       |        4 |            5 |
| Nigeria   |        3 |            5 | South Africa |        3 |            5 |
+-----------+----------+--------------+--------------+----------+--------------+
9 rows in set (0.00 sec)
```

7. *SELECT team, MAX(goalsFor/goalsAgainst) AS GoalRatio FROM teams GROUP BY team ORDER BY GoalRatio DESC LIMIT 1;*
   MAX aggregation on ratio of (goalsFor/goalsAgainstused) used along with GROUP BY and ORDER BY by limiting results to 1st record.
   Results:



```
mysql> SELECT team, MAX(goalsFor/goalsAgainst) AS GoalRatio FROM teams GROUP BY team ORDER BY GoalRatio DESC LIMIT 1;
+----------+-----------+
| team     | GoalRatio |
+----------+-----------+
| Portugal |    7.0000 |
+----------+-----------+
1 row in set, 1 warning (0.00 sec)
```

8. *SELECT t.team, AVG(p.passes) AS average_pass FROM teams t INNER JOIN players p ON t.team = p.team WHERE p.position LIKE "defender" GROUP BY t.team HAVING average_pass > 150 ORDER BY average_pass DESC;*
   Using AVG aggregation by joining tables on team column and using GROUP BY to aggregate, further using HAVING clause to retrieve the grouped results.
   Results:



```
+-------------+--------------+
| team        | average_pass |
+-------------+--------------+
| Spain       |     213.0000 |
| Brazil      |     190.0000 |
| Germany     |     189.8333 |
| Netherlands |     182.5000 |
| Mexico      |     152.1429 |
+-------------+--------------+
5 rows in set (0.00 sec)
```

Mongo Searches for the questions (**All mongo queries and results attached with the search queries in *q2_mongo_searches.js, some outputs attached with results in the report***):

1. The regular expression */.*ia.*/* can be used to search for the team names & pretty to get a human readable output.
   Output: *{ "_id" : ObjectId("5e73cf9f1d7790aa3d3a0f0f"), "surname" : "Kuzmanovic" }*

2. sort() will give the player information in descending order, when the parameter passed to the field is -1.
   Output snippet:

{ "_id" : ObjectId("5e73cf9f1d7790aa3d3a0e28"), "surname" : "Gyan", "team" : "Ghana", "position" : "forward", "minutes" : 501, "shots" : 27, "passes" : 151, "tackles" : 1, "saves" : 0 }
Further output results saved with q2_2.js file, the output gives about 3 results

3. The first mongo query saves the results from Teams collection to a variable team. We can then iterate over each team which has played more than 4 games in Players collection to retrieve goalkeepers from Players collection.
   Output snippet: Julio Cesar Brazil 450
   Casillas Spain 540
   Further results for output saved with q2_2.js file as the output gives about 8 results

4. We save the results from first mongo query in teams_10 variable which has list of teams with rank less than 10. We iterate over each team object, to push the team name in to an array. Then aggregation to match the condition of minutes and team name in the list of previous teams, to further group by summing each occurrence, project is being used to suppress the _id field of the document and just return the count of superstars.
   Output: { "Superstar" : 54 }

5. Aggregation is used to match on the condition of player positions, and then further to group on the number of average passes.
   Output: { "_id" : "forward", "average_passes" : 50.82517482517483 }
   { "_id" : "midfielder", "average_passes" : 95.2719298245614 }

6. We use a cursor to get all teams, goalsFor, goalsAgainst from teams collection. Again iterate with the same cursor by specifying team less than collection.team to fetch distinct documents on same goalsFor and goalsAgainst.
   Output Snippet: England Chile 3 5
   Greece Cameroon 2 5
   Further Output results saved with q2_2.js file as the output gives about 9 results

7. We aggregate the results by following: - Project gives the list of teams along with their ratio of goalsFor and goalsAgainst, further sorting the results by ratio:-1 gives descending order of the ratios. Then the results can be limited by 1 to get the highest goalsFor to goalsAgainst ratio. We can project further to drop the id and just display team name and their ratio.
   Output: *{ "team" : "Portugal", "ratio" : 7 }*

8. *Aggregation* is first performed by grouping the teams with their average number of passes, further we use the calculated average to filter out averages more than 150 with *match*. Further, project has been used to display the output in readable format by removing the id from results to just display average passes and team name.
   Output Snippet: { "average_passes" : 169.5, "team" : "Netherlands" }
   { "average_passes" : 212.5, "team" : "Spain" }
   Further results saved with q2_2.js file as the output gives 5 results

3. Reflection

1. Comparison of SQL & NoSQL databases:
The purpose of any database management system is managing data in an efficient manner. The relational databases are those which were being used traditionally and are also being widely used in many domains. The relational databases store

data in a structured manner, in a tabular form, where there exists a relation between tables in a database. Up to certain limit of data the performance of such relational databases is great, however, as the volume of the data is increased, the performance of the relational databases is degraded. Hence, there was a need for a solution which could not only resolve the problem with Volume, but also the Variety and Velocity of the data being generated and stored by organizations these days. Due to which the term, NoSQL came up where in data is not needed to be stored in a tabular/structured form. The NoSQL databases are not supposed to be a replacement for the relational databases, rather there are few domains wherein the relational databases will be best suited. However, in certain domains the drawbacks of the relational databases can overcome with the NoSQL databases. There are different schema types for the NoSQL database storage, for ex: NoSQL databases can be document based, column oriented, graph-based or key-value paired.
There are many research papers where the authors have compared the relational and NoSQL databases, in various aspects like database insertion/update performance, usability, scalability etc. Also from the authors research and work in [1] and [2], they compared the performance of MongoDB and SQL Server, after which they came to conclusion that the performance of MongoDB was better than SQL Server in terms of insert, update and simple queries, however, SQL Server performed better in update and queries with non-key attributes and aggregation operations.

In terms of Scalability & performance SQL databases can be scaled vertically, whereas the NoSQL databases are horizontally scalable. [1] It means when the size of the data being stored is increased, the storage of the existing hardware can be expanded(i.e addition of new SSD, HDD, RAM). In such vertical scaling there is a certain risk associated with hardware failure which could lead to data loss. Whereas in vertical scaling the performance of the NoSQL database can be load balanced by simply adding new servers to the existing cluster of NoSQL environment. The horizontal scaling of databases is relatively cheap, and nodes can be added as and when required.

The main difference between the SQL and NoSQL databases can also be described by their properties which are ACID (Atomicity, Consistency, Isolation, Durability) and BASE (Basically Available, Soft state, and eventually consistent). SQL databases inherit the ACID properties, whereas the NoSQL database show BASE properties. Both the properties are adopted from CAP Theorem, which is-

- Consistency: All clients have the same view.
- Availability: Data must be always available, every result to non-failed node should give correct response.
- Partition Tolerance: The database system must work fine despite of working under arbitrary network partitions or failures.

According to the CAP theorem, it is impossible to meet all the three aspects at the same time. At a time, a database can at most meet two of the aspects in the CAP theorem.

In terms of flexibility, the NoSQL database are quite flexible as compared to the SQL databases. The schema of a SQL database is predefined. Changing the schema of a database which is in production and has data loaded in it is a tedious task and there is a huge problem when modifying such database. Whereas there is no need for predefining the structure of a NoSQL database, therefore they can easily adapt changes to its dynamic design. The query language for SQL databases is a standard which is maintained across most of the vendors and is powerful to handle complex queries. However, it is not the case with NoSQL database, the results to fetched needs to be queried by the vendor specific language or commands.

The same was the case when writing the queries for the Teams and Players tables, when using SQL, it was a bit easier to fetch complex queries, however, when writing complex queries with MongoDB it is a tough task. In this case, the Teams and Players data should ideally be in a SQL database as the size of the data is not that huge and there is a structured aspect to the data wherein the each of the record has those values associated with it. In such scenarios, using a SQL database would be beneficial over a NoSQL database. The performance of retrieving the results in Mongo was better than SQL, however, the difference in both the cases was not large, as the volume of the data was not that huge. If we consider complex queries, the SQL queries in question 5 & 6 required about 10 milliseconds to fetch results, whereas the Mongo queries gave the same results in about 3 milliseconds. The time required to fetch results would have been much more in case the dataset was much more large and further complexity was involved in the queries. Though, the performance of Mongo queries is much better, writing SQL queries is easier as it is easy to understand and is followed same across different database providers, whereas Mongo has its own style of writing search queries.

We can conclude that though each of the databases has its own pros and cons, the decision of choosing a database completely depends on the domain or the type of use for the database. With the Volume, variety and velocity of data being generated by the organisations these days, they should prefer NoSQL databases, however, if the data is in a structured form and the volume is not that huge, a SQL database would be better choice. Aspects like scalability, flexibility, availability, performance, querying language, security etc., should also be considered when selecting a database for an organisation.

References:

[1] Sahatqija, Kosovare & Ajdari, Jaumin & Zenuni, Xhemal & Raufi, Bujar & Ismaili, Florie. (2018). Comparison between relational and NOSQL databases. 0216-0221. 10.23919/MIPRO.2018.8400041.

[2] Z. Parker, S. Poe, S. V. Vrbsky, "Comparing NoSQL MongoDB to an SQL DB", In Proceedings of the 51st ACM Southeast Conference (ACMSE '13). ACM, New York, NY, USA, Article 5, 6 pages, April 4-6, 2013.

SQL vs. NoSQL: How Are They Different and What Are the Best SQL and NoSQL Database Systems?

2. Short report on: Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, Google, Inc.

The questions that the paper addresses include but is not limited to the list of challenges that Google faced:
- Consistent model for dealing with a variety of structured and semi-structured data.
URL contents, user specific data (profile info, preferences etc.), geographic information
- Scalable system for dealing with high volume and velocity of data.
Large number of items with different versions, concurrency for large user base, arbitrarily large dataset.
The paper describes the BigTable data model provided by Google and its implementation, that enables dynamic control over data layout and format. A large number of products at Google have adopted this distributed storage approach for structured data. The paper discusses how BigTable has achieved several goals like wide acceptability, scalability, high performance and high availability. The authors also discuss how BigTable resembles a database, by comparing how it shares implementation strategies with Parallel and main memory database, and how BigTable provides different interface than such systems. Bigtable is essentially a NoSQL database, large database with high throughput. It is one of the example of wide column based database along with key-value store. The design of BigTable is suitable even for 1000s of attributes, or even unknown number of future columns. Some of the other databases influenced from the design of the Google BigTable include Apache HBase, Apache Cassandra, accumulo as they are directly or indirectly inherited from BigTable. Some of the characteristics of a BigTable are: map, persistent and distributed storage, sparse, sorted, multidimensional and timestamp based. The BigTable can be indexed by the row-key, column-key, timestamp which lets us to do a fast lookup and also store multiple versions of the same data. The number of versions to be stored depends on the configuration set by the user.

The main components in the implementation of BigTable include:
- Client Library: It is a library that is linked to each client, it caches the tablet server location mappings.
- One Master server: It is used to assign tablets to servers, detect addition and deletion of tablet servers, garbage collection of GFS files.
- Multiple tablet servers: Handling of read and write requests into its tablets.

A BigTable stores a large number of tables and each table has range of tablets, which contain all data within a row. The BigTable is based on the Google File System (GFS). In addition to the three main components, Chubby is a service which is used to track tablet servers. It ensures there is a single master and it stores information on bootstrap location of BigTable. It is also used for tracking tablet servers. The BigTable utilizes three-level hierarchy similar to a B+ tree to store tablet location details. It provides API functions for creating and deleting column families for the client application. The applications can read/write values in a BigTable. The research in the implementation of BigTable required different algorithms and projects, one of which being the Paxos algorithm. A Paxos is a family of distributed algorithms which is used to reach consensus. Chubby uses the Paxos algorithm to for managing its replicas consistently. The authors also discuss the Boxwood project in the paper as it has some similar components as BigTable for distributed storage. However, the Boxwood projects focus is infrastructure for developing high level services like file systems, databases. Whereas BigTable emphasizes to support client applications that wish to store the data.

Regarding the evaluation of Performance, a BigTable cluster with N tablet servers was setup to evaluate the performance as N is changed. From going through the performance evaluation details, it seemed like as the number of tablet servers increased operations like random reads, random writes improved a lot. The performance of sequential reads, sequential writes and scans was improved as well, though the improvement was not as significant as random reads, writes. For most of the benchmarks, there was a significant throughput on increasing the number of tablet servers. We get to learn different experiences how the authors discussed large distributed systems that are vulnerable to many types of failures.

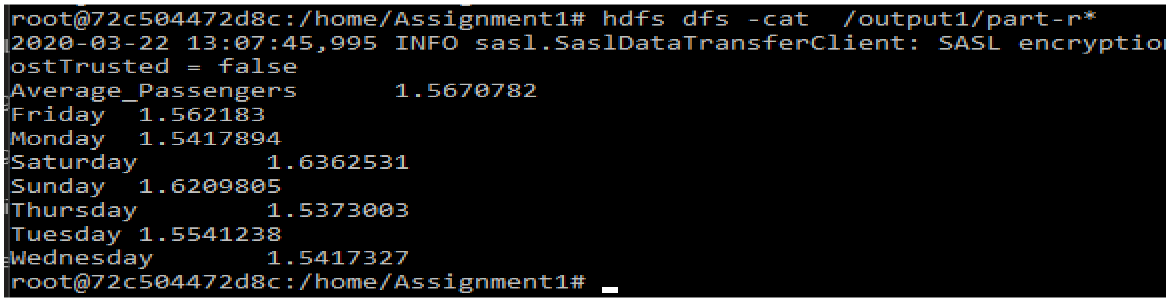
I believe the idea of storing data into column family store is efficient and having unique row key identifiers for the tablets. You can collectively store the relevant data or aggregate data into a single column family.

For example: If we want to store a profile information, it will be better to store multiple columns like- Address Line 1, Line 2, Zip code in one column family, other information like contact number, email could be in one column family and so on.

In the use-cases of such semi-structured data the BigTable will be very efficient and effective for implementation with a huge user base. With a large number of Google products already implementing the BigTable, it seems to be a very efficient solution for handling such large unstructured data with minimum failure.

4. Hadoop

Q1. Average number of passengers per trip and in general and per day of week.
Run the Question1.java which implements map-reduce by following the steps in the README.txt. We use trip-start column for retrieving day from the timestamp and passenger count column in the map function. The map sends key value pair of each day, and passenger count to get the average by day as well as in general in the reducer by iteration and counter.
Output:

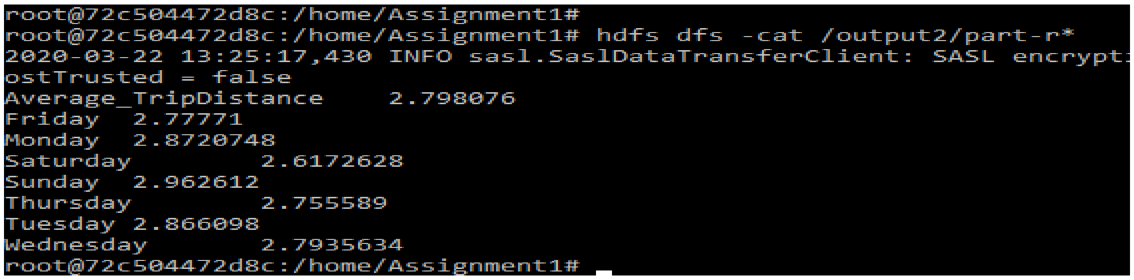


```
root@72c504472d8c:/home/Assignment1# hdfs dfs -cat  /output1/part-r*
2020-03-22 13:07:45,995 INFO sasl.SaslDataTransferClient: SASL encryptio
ostTrusted = false
Average_Passengers      1.5670782
Friday  1.562183
Monday  1.5417894
Saturday        1.6362531
Sunday  1.6209805
Thursday        1.5373003
Tuesday 1.5541238
Wednesday       1.5417327
root@72c504472d8c:/home/Assignment1# _
```

Q2. Average distance per trip and in general and per day of week.

Run the Question2.java which implements map-reduce by following the steps in the README.txt. We use trip-start column for retrieving day from the timestamp and trip distance column in the map function. The map sends key value pair of each day, and trip distance to get the average by day as well as in general in the reducer by iteration and counter.

Output:

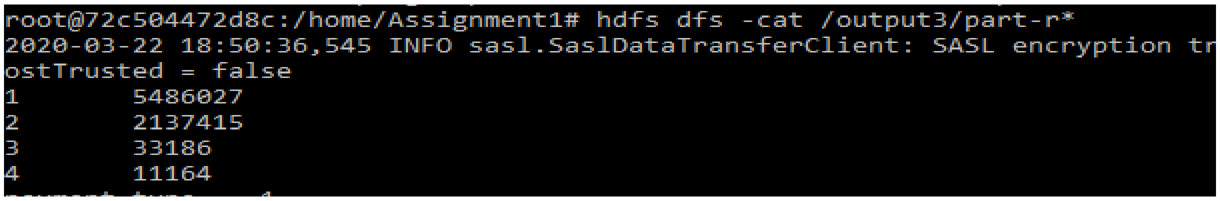


```
root@72c504472d8c:/home/Assignment1#
root@72c504472d8c:/home/Assignment1# hdfs dfs -cat /output2/part-r*
2020-03-22 13:25:17,430 INFO sasl.SaslDataTransferClient: SASL encrypt
ostTrusted = false
Average_TripDistance    2.798076
Friday  2.77771
Monday  2.8720748
Saturday        2.6172628
Sunday  2.962612
Thursday        2.755589
Tuesday 2.866098
Wednesday       2.7935634
root@72c504472d8c:/home/Assignment1# _
```

Q3. Most used payment types:

Run the Question3.java which implements map-reduce by following the steps in the README.txt. We use payment type column as key and write 1 for each in the map function. The map sends key value pair of each type, and its count to get the count by group (payment id) in the reducer.

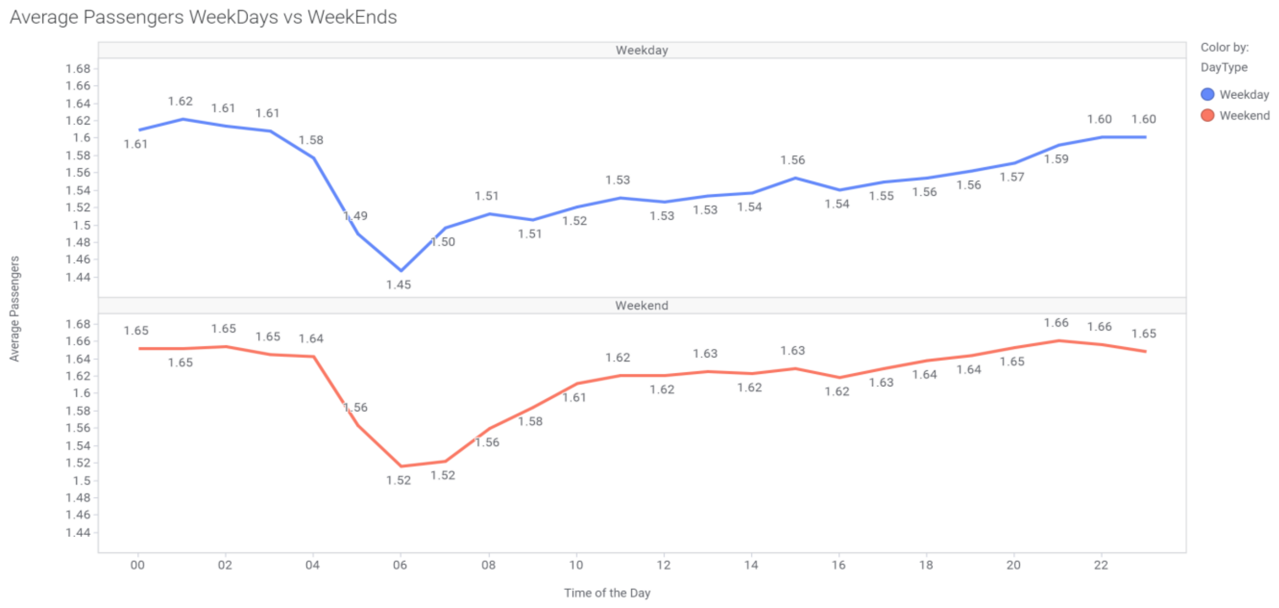
Output:



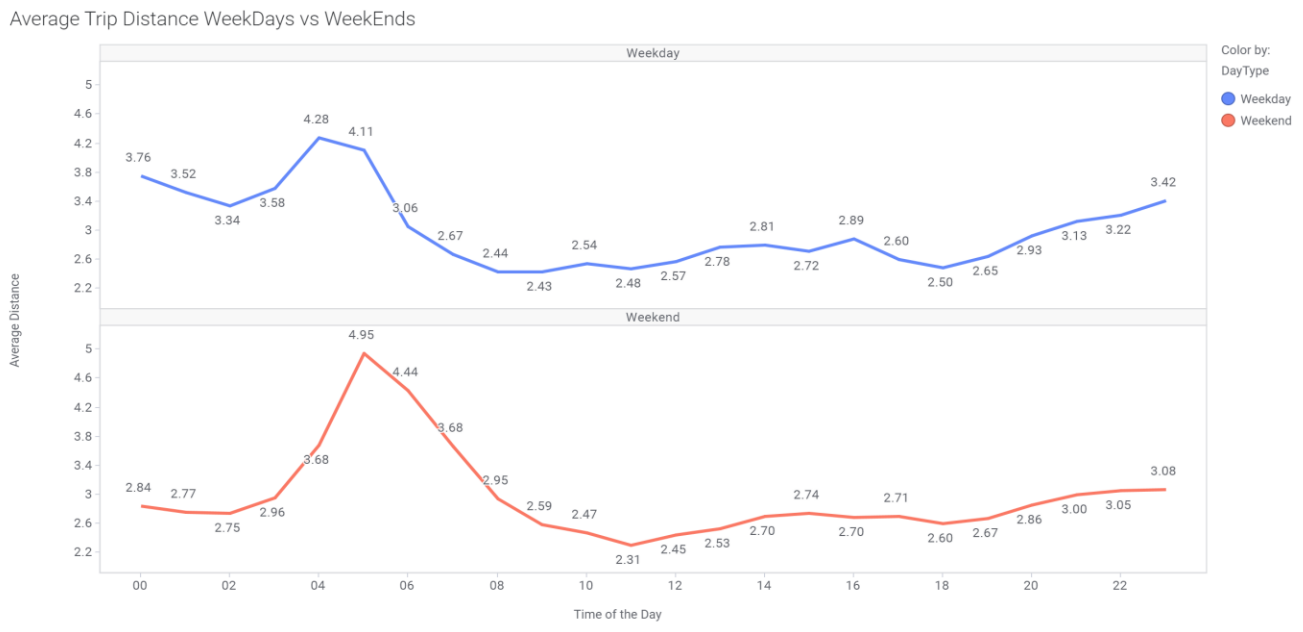
```
root@72c504472d8c:/home/Assignment1# hdfs dfs -cat /output3/part-r*
2020-03-22 18:50:36,545 INFO sasl.SaslDataTransferClient: SASL encryption tr
ostTrusted = false
1       5486027
2       2137415
3       33186
4       11164
```

Q4. Create a graph (using the output of a MapReduce job) showing average number of passengers over the day (per hour). Output also saved in hadoop_4.txt

Run the Question4.java which implements map-reduce by following the steps in the README.txt. The approach is similar as question 1, we specify weekday and weekend and provide the input to the map accordingly to get hourly passenger count for type of day. Further aggregate by average in the reducer by the type of day and passenger count sent by map. The below graph from the output is created using a Data Visualisation tool. (The image is also in the folder for detailed viewing- HadoopOutput4.jpg)



Q5. Create a graph showing the average trip distance over the day (per hour). Output also saved in hadoop_5.txt
Run the Question5.java which implements map-reduce by following the steps in the README.txt. The approach is similar as question 2, we specify weekday and weekend and provide the input to the map accordingly to get hourly trip distance for type of day. Further aggregate by average in the reducer by the type of day and trip distance sent by map.
The below graph from the output is created using a Data Visualisation tool. (The image is also in the folder for detailed viewing- HadoopOutput5.jpg)



Q6. The output of Q6 is same as the Question 4 which gives the aggregation of average passenger over the day every hour. Same logic can be used as mentioned in the Question4 of Hadoop section.