1. Bash for (Big) Data Analysis

1. **How many lines of content (no header) is there in the file? (tail, wc)**
   File used: 1_lines_in_file.sh
   Answer: 40
   Used 'tail +2' to exclude the header, and then wc -l to count lines.

2. **create a bash command (using, among other things, sed and wc) to count the number of columns**
   File used: 2_number_of_columns.sh
   Answer: 11
   Used sed to select the first row, piped to 'tr' to replace commas (',') with \n, and then piped to wc -l to count lines.

3. **For a given city (given as a column number, e.g., 10=Sydney), what is the type of crime on top of the crime list (cat, cut, sort, head)?**
   File name: 3_crime_top_sh
   Answer: Drugs ? Imported
   A column number is passed as argument, which is used to sort the file. To get the crime name, we use cut to only get the first column, and use head -1 to get the top crime.

4. **Find the number of crimes for a given city (given as a column number, e.g., 10=Sydney): create a bash script that reads all the rows (see previous question) and sums up the crime values**
   File name: 4_total_crime_city.sh
   Answer: 472
   Initialised a sum variable to 0. Used cut to select only the particular column passed, and tail to exclude header - this is passed to the while loop to calculate sum.

5. **Same question with the average – look at question 1 to get the number of rows & use tr to remove the empty white spaces and get the number.**
   File: 5_average_crime_city.sh
   Answer: 11.800
   Got the total number of rows using tail and wc -l. Used a while loop along with cut to get the sum of all rows to get total crime, like in q4. Used the bc calculator to get the average - sum/total.

6. **Get the city with the lowest average crime. Create a bash script that goes through all the cities, compute the average crime rate and keep only the city with the lowest value. Finally display the city and the average number of crimes.**
   File: 6_average_crime_across_cities.sh
   Answer: Minimum crime is in Hobart
   Minimum average crime rate: .525

   Got the total number of columns by using sed and tr - similar to question 2. Got the total number of rows similar to question 1. We first set the minimum average and city to the average crime of the first city - row 3, and then looped through the rest of the columns, updating the minimum average and city every time an average value less than the current minimum is found.
   At the end, the city with the minimum average and the min. avg is echoed.

2. Data Management

Datasets:

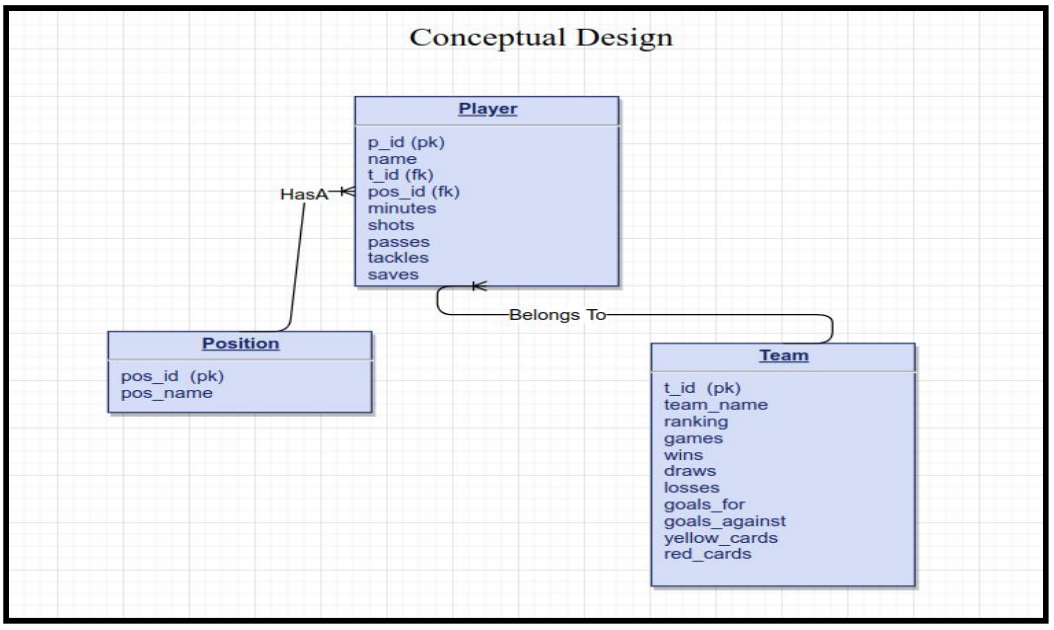
1. **Describe in a short paragraph the two datasets.**
   The two datasets are Players.csv and Teams.csv

   Players.csv:
   This file contains the information about players across football teams. Each row of the data is a player, with a total of 595 players, having features:
   The surname of the player,
   The team they play for,
   Their position on the team,
   The number of minutes the player has played
   The number of shots made by the player
   The number of passes made.
   Number of tackles made, and
   The number of saves made.

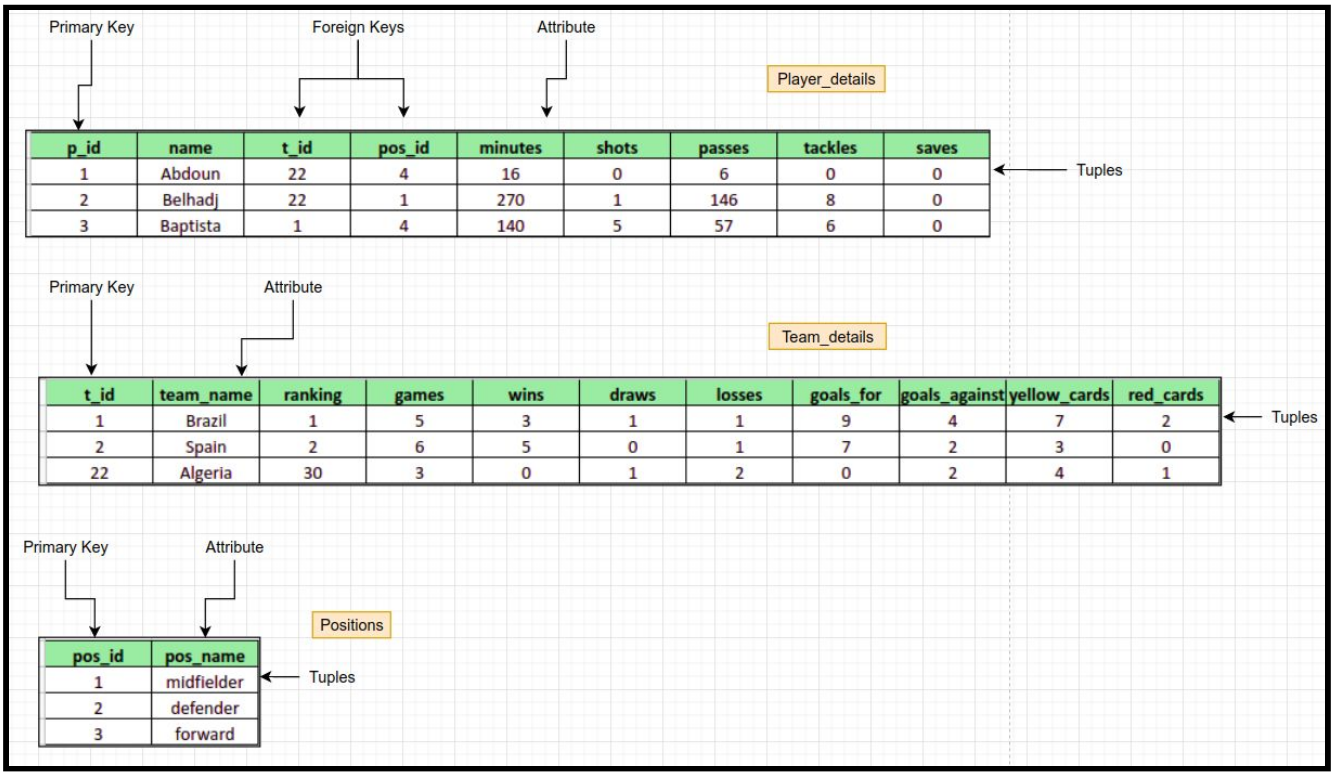
   Teams.csv:
   This dataset contains information about different football teams, Each row of data is a team, with a total of 32 teams, having features:
   The team name,
   Ranking of the team,
   Number of games played,
   Number of wins,
   Number of draws,
   Number or losses,
   Number of goals scored by team,
   Number of goals scored against team,
   Number of Yellow and Red cards incurred by the team.

2. **Create a database in MySQL and a few tables to represent the dataset. Give the conceptual design and the relational model associated with your database.**
   Three tables are created - Player_details, Team_details and Positions.
   The Player_details table is normalized, and positions are stored separately in the Positions table, with pos_id acting as the foreign key. Similarly, t_id acts as a foreign key to the Team_details table.

Conceptual Design



Relational Model



p_id	name	t_id	pos_id	minutes	shots	passes	tackles	saves
1	Abdoun	22	4	16	0	6	0	0
2	Belhadj	22	1	270	1	146	8	0
3	Baptista	1	4	140	5	57	6	0

t_id	team_name	ranking	games	wins	draws	losses	goals_for	goals_against	yellow_cards	red_cards
1	Brazil	1	5	3	1	1	9	4	7	2
2	Spain	2	6	5	0	1	7	2	3	0
22	Algeria	30	3	0	1	2	0	2	4	1

pos_id	pos_name
1	midfielder
2	defender
3	forward

3. Populate the database using a bash script.

Bash script file - mysql_import.sh

mysql_import.sh: This file takes in two arguments - the mysql username, and the password.

First, a create statement is generated to create the Team_details table.

Then, we loop through the Teams.csv file, and create an SQL statement to load each team. A count variable is maintained which is used as the team_id primary key. The create and insert statement generated are then executed using the -e option of mysql.

A positions.csv file is created using the cut and sort -u commands, which give us unique positions of players from Players.csv.
A Positions table is created. We generate an SQL statement to load each position from the positions.csv file by looping through them, with the pos_id primary key being auto-incremented.

Finally, we load the player information. First, a create table statement is generated to create the Player_details table.
Looping through Players.csv, an sql statement is created to insert all player information into Player_details. Inside each insertion, we perform a select statement to get the team id and position id for every player - using the team name and position - and use these ids as foreign keys to the team and position tables.

4. **Create MongoDB collections with every line in the CSV files as an entry/document. Populate the collections with the content of the CSV files (using a script).**
Bash script file - mongo_import.sh
mongo_import.sh: This file takes in at least two arguments, the database name, and the csv files needed to populate the collections, separated by space.

We loop through the files, and create a collection for each csv file using the mongoimport command. The -d option specifies a database, which is created if not already present. The cut command is used to get the name of the collections using the name of the csv file.
Two collections are created - one for Teams and another from Players.

Queries:

1. **What player on a team with "ia" in the team name played less than 200 minutes and made more than 100 passes? Return the player surname.**
mysql> select name as surname from Player_details p left join Team_details t on p.t_id=t.t_id where minutes < 200 and passes > 100 and team_name like '%ia%';
We use a left join to match Player_details and Team_details on team id, and use the where clause to find the player who played less than 200 minutes and made more than 100 passes, and used 'like' to find team with "ia" in the team name.

mongo> db.Players.find({$and : [{"team": /.*ia.*/}, {"minutes":{$lt:200}}, {"passes":{$gt:100}}]},{surname:1})
Used the find method to match the team name where "ia" is present using regex, and the $lt and $gt operators to find players who played less than 200 minutes, made more than 100 passes. The surname field is projected.
The answer is Kuzmanovic.

```
+------------+
| surname    |
+------------+
| Kuzmanovic |
+------------+
1 row in set (0.00 sec)
```

2. **Find all players who made more than 20 shots. Return all player information in descending order of shots made.**
mysql> select * from Player_details where shots > 20 order by shots desc;

Selected all details from Player_details, used where clause to select players with more than 20 shots, and 'order by desc' to sort in descending order.

mongo> db.Players.find({"shots":{$gt:20}}).sort({shots: -1})

Used the find method to find players with more than 20 shots, and then used the sort method to sort the players descending order of shots.

We get three results - Gyan, Villa, Messi.

p_id	name	t_id	pos_id	minutes	shots	passes	tackles	saves
201	Gyan	24	2	501	27	151	1	0
536	Villa	2	2	529	22	169	2	0
29	Messi	7	2	450	21	321	10	0

3 rows in set (0.00 sec)

3. **Find the goalkeepers of teams that played more than four games. List the surname of the goalkeeper, the team, and the number of minutes the goalkeeper played.**

mysql> select name as surname, team_name, minutes from Player_details p left join Team_details t on p.t_id = t.t_id left join Positions pos on p.pos_id = pos.pos_id where games > 4 and pos_name = 'goalkeeper';

Used a left join to join Player_details and Team_details on team id, and another left join to join Positions on position id. Where clause to select goalkeepers who played more than 4 games.

```
mongo> db.Players.aggregate([{
        $lookup: {from:"Teams", localField: "team", foreignField: "team", as: "Teams"}}, {$match:{$and:
[{"position":"goalkeeper"}, {"Teams.games":{$gt:4}}]}},
{$project:{"surname":1, "team":1, "minutes":1}}])
```

Used the $lookup aggregate to match the Players collection to Teams collection using the localField and foreignField attributes. It is the equivalent of a left outer join.

The $match aggregation is used to filter only the players who have the position 'goalkeeper', and whose teams played more than 4 games. The surname, team name and minutes are then projected.

surname	team_name	minutes
Julio Cesar	Brazil	450
Casillas	Spain	540
Stekelenburg	Netherlands	540
Neuer	Germany	540
Romero	Argentina	450
Muslera	Uruguay	570
Villar	Paraguay	480
Kingson	Ghana	510

8 rows in set (0.00 sec)

4. **How many players who play on a team with ranking <10 played more than 350 minutes? Return one number in a column named 'superstar'.**

mysql> select count(*) as 'superstar' from Player_details p left join Team_details t on p.t_id = t.t_id where ranking < 10 and minutes > 350;

Used a left join to join Player_details and Team_details, and where clause to filter out teams. Then used count(*) to count all the rows, which is returned as 'superstar'

mongo> db.Players.aggregate([{$lookup: {from:"Teams", localField: "team", foreignField: "team", as: "Teams"}}, {$match:{$and: [{"Teams.ranking":{$lt:10}}, {"minutes":{$gt:350}}]}}, {$count:"superstar"}])

Used the $lookup aggregate to match the Players collection to Teams collection using the localField and foreignField attributes.

The $match aggregation is used to filter only the players who have played more than 350 minutes, and whose team ranking is less than 4. The $count aggregation then returns the count of all such players as 'superstar'

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

5. **What is the average number of passes made by forwards? By midfielders? Write one query that gives both values with the corresponding position.**

mysql> select pos_name as "Position", avg(passes) as "Average Passes" from Player_details p left join Positions pos on p.pos_id = pos.pos_id where pos_name = 'midfielder' or pos_name = 'forward' group by pos_name;

Used left join to join Player_details with Positions based on position id, where clause to filter out players who are either midfielder or forwards, and grouped by their position name. The avg() aggregate function is used to calculate the average passes made by forwards and midfielders.

```
mongo> db.Players.aggregate([
 { $group: { _id: "$position", avg: { $avg: "$passes" } } },
 { $match: { $or: [{ _id: "midfielder" }, { _id: "forward" }] } },
 { $project: { _id: 0, Position: "$_id", "Average Passes": "$avg" } }
]);
```

Used the $group aggregate to group by position, and the $avg aggregate to take the average number of passes. The position is the _id field in the group. Then used the $match aggregate with $or to match only ids where _id is either midfielder or forward.

```
+------------+-----------------+
| Position   | Average Passes |
+------------+-----------------+
| forward    |         50.8252 |
| midfielder |         95.2719 |
+------------+-----------------+
2 rows in set (0.01 sec)
```

6. **Find all pairs of teams who have the same number of goalsFor as each other and the same number of goalsAgainst as each other. Return the teams and numbers of goalsFor and goalsAgainst. Make sure to return each pair only once.**

mysql> select t1.team_name as "Team 1", t2.team_name as "Team 2", t1.goals_for as "GoalsFor", t1.goals_against as "GoalsAgainst" from Team_details t1, Team_details t2 where t1.goals_for = t2.goals_for and t1.goals_against = t2.goals_against and t1.team_name < t2.team_name;

Here, we used a self-join to join the Team_details table with itself, with aliases t1 and t2. A where clause was used to find t1 and t2 for which both goalsFor and goalsAgainst match. The condition t1.team_name < t2.team_name ensures each pair is only returned once - the team name occurring first alphabetically will always be "Team 1."

```
mongo>
db.Teams.aggregate([
{ $lookup: { from: "Teams",
let: { id2: "$_id", team2: "$team", goalsFor2: "$goalsFor", goalsAgainst2: "$goalsAgainst"},
pipeline: [{$match: {$and: [{ $expr: { $lt: ["$_id", "$$id2"] } }, { $expr: { $ne: ["$team", "$$team2"] } }, {
$expr: { $eq: ["$goalsFor", "$$goalsFor2"] } }, { $expr: { $eq: ["$goalsAgainst", "$$goalsAgainst2"] } } ] } } ],
   as: "team2" } },
{ $unwind: "$team2" },
{ $addFields: { against_team: "$team2.team" } },
{ $project: { team: 1, goalsFor: 1, goalsAgainst: 1, against_team: 1 } }
]);
```

Here, a lookup operation is performed on Teams from Teams to match where goalsFor and goalsAgainst are the same.

However, since mongodb does not explicitly support joins and self joins in particular, a pipeline field has to be used. The pipeline field cannot directly access the documents in the collection, and for that we use let variables, where we select id, team name, goalsFor and goalsAgainst of Team 2.

These let variables are accessed by the pipeline, where, similar to the sql query, we specify id of document is lesser than the id of lookup document, the names are not the same, and goalsFor and goalsAgainst match between Team 1 and the let variables. This output of pipeline is forwarded as team2 and unwinded to get documents.

The team name of team2 is added to the original document, and team names along with goalsFor and goalsAgainst are projected.

```
+------------+---------------+-----------+---------------+
| Team 1     | Team 2        | GoalsFor  | GoalsAginst   |
+------------+---------------+-----------+---------------+
| Chile      | England       |         3 |             5 |
| Cameroon   | Greece        |         2 |             5 |
| Italy      | Mexico        |         4 |             5 |
| England    | Nigeria       |         3 |             5 |
| Chile      | Nigeria       |         3 |             5 |
| Australia  | Denmark       |         3 |             6 |
| England    | South Africa  |         3 |             5 |
| Chile      | South Africa  |         3 |             5 |
| Nigeria    | South Africa  |         3 |             5 |
+------------+---------------+-----------+---------------+
9 rows in set (0.00 sec)
```

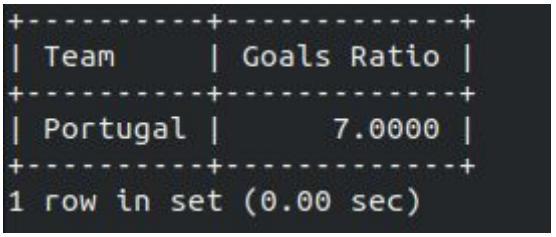
7. **Which team has the highest ratio of goalsFor to goalsAgainst?**

mysql> select team_name as "Team", (goals_for / goals_against) as "Goals Ratio" from Team_details where (goals_for / goals_against) = (select max(goals_for / goals_against) from Team_details);

We use a subquery to get the maximum ratio of goalsFor/goalsAgainst - using the max() aggregate in the subquery. This maximum ratio is used in the where clause of the original query to get the team with maximum goal ratio.

mongo> db.Teams.aggregate([{$project: { team : 1, ratio: { $divide: ["$goalsFor", "$goalsAgainst"] } } },{ $sort: { ratio: -1 } }, { $limit: 1}]);

We use $project to project the team name, and the $divide operator to get the ratio of goalsFor / goalsAgainst. This is then sorted according to the ratio, and the $limit operator is used to get the top result.



```
+----------+-------------+
| Team     | Goals Ratio |
+----------+-------------+
| Portugal |      7.0000 |
+----------+-------------+
1 row in set (0.00 sec)
```

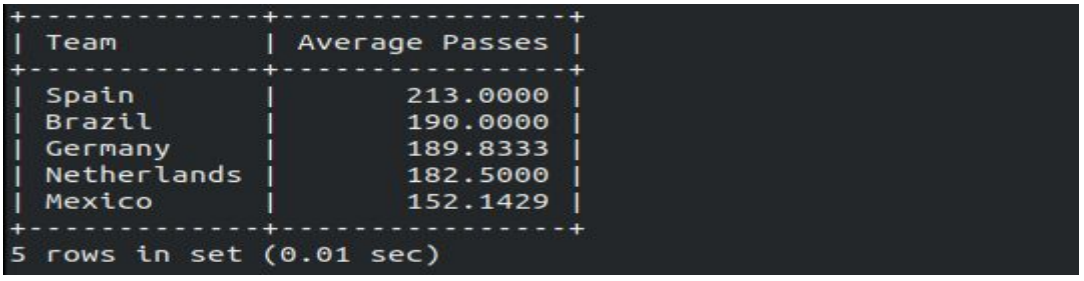
8. **Find all teams whose defenders averaged more than 150 passes. Return the team and average number of passes by defenders, in descending order of average passes.**

mysql> select team_name as "Team" ,avg(passes) as "Average Passes" from Team_details t inner join Player_details p on t.t_id = p.t_id inner join Positions pos on p.pos_id = pos.pos_id where pos_name = 'defender' group by team_name having avg(passes)>150 order by avg(passes) desc;

Used inner join to join Team_details table to Player_details oj team id, which is in turn inner joined to Positions table on position id. Used where clause to filter on defenders, group by based on team name, and having to only select teams with average passes greater than 150. These are then sorted in descending order based on average passes. The team name and average passes is returned.

```
mongo>db.Teams.aggregate([
 {$lookup: { from: "Players", localField: "team", foreignField: "team", as: "p" } },
 { $unwind: "$p" },{ $match: { "p.position": "defender" } },
 { $group: { _id: "$team", avg_a: { $avg: "$p.passes" } } },{ $match: { avg_a: { $gt: 150 } } },
 { $sort: { avg_a: -1 } },
{$project: {_id: 0, Team: "$_id", "Average Passes": "$avg_a" }} ]);
```

Used the $lookup aggregate to match the Players collection to Teams collection using the localField and foreignField attributes. On unwinding, we use $match to select only documents where position is defended. The $group aggregate is used to group on team name as id, and average of passes as the operation. Another $match operation is used to select only teams with average passes greater than 150. Finally, $sort is used to sort on the average number of passes. This is then projected to output.



```
+-------------+-----------------+
| Team        | Average Passes  |
+-------------+-----------------+
| Spain       |        213.0000 |
| Brazil      |        190.0000 |
| Germany     |        189.8333 |
| Netherlands |        182.5000 |
| Mexico      |        152.1429 |
+-------------+-----------------+
5 rows in set (0.01 sec)
```

3. Reflection

1: Compare relational and NoSQL database management models and in particular give your impressions on why one is better than the other and in which context.

RDBMS is an older and static model built for simple data structures, while NoSQL is recent-ish, built for unstructured and flexible structures. While RDBMS still has some use cases, modern data which is constantly changing requires a flexible model that NoSQL provides. RDBMS on the other hand is great for tabular data and where data follows a strictly structured schema which isn't updated regularly. Changing the schema for RDBMS is an expensive process.

Similarly, for large scale systems, NoSQL is better since scaling horizontally is much cheaper and powerful, compared to RDBMS which needs to be scaled vertically and remain on a single server. (NoSQL vs SQL (Relational Databases), 2020)

Performance evaluation and usability:

In terms of the data used in the previous section, I feel RDBMS was better suited.

RDBMS is more suited to fixed, structured data using normalization techniques for separating data and generally reducing redundancy, while NoSQL is suitable for unstructured, complex data which are not normalized. Thus, reads and writes and basically loading files into the database is faster with NoSQL, however, RDBMS is faster for complex queries which need self joins or other complex joins.

Based on the usability of both queries, MySQL is a simpler query language, while NoSQL requires various aggregation techniques and has a complex syntactic structure.

As explained above, since the data for RDBMS needed to be normalized, a complex script needed to be written to load data into the mysql database, while it was pretty simple to load documents into MongoDB directly from the csv files.

However, since the data was structured, it was easier to run queries on MySQL as they required simple joins and where clauses, or sub-queries. However, the lack of much support for joins led to the complexity of the Mongo query increasing, as $lookup and pipelines needed to be used for complex queries, such as in Q6. These complex queries are more expensive.

A simple (but naive, since it is not entirely accurate) example to calculate runtime was used for query 6, and found that the Mongo query (using Date from js) took 7 milliseconds to finish, while MySQL (using profiling) took 1.125 milliseconds to run on the local machine. (Zagorulkin, 2020)

Thus, for the given dataset, RDBMS is better suited than NoSQL.

2: Summary of *'Eventually Consistent'* by Werner Vogels.

<u>Motivation:</u>

Engineering systems that operate at a large, global scale processing trillions of requests like Amazon need to meet certain strict requirements with respect to security, scalability, availability, performance and cost-effectiveness. These systems are made up of large distributed systems under the hood, which creates further problems because at this scale, events with even extremely low probabilities are certain to occur, and the system needs to be designed to handle these. A number of replication techniques are used at Amazon to ensure consistency and high availability, however the consequences of this are such that we need to take into account the trade-offs between data consistency and high availability. This paper explores the nature of these trade-offs, and the approaches Amazon used for reliable global scale systems.

<u>Historical Work:</u>

In the 1970s, the focus was on achieving high consistency, and many techniques tried to achieve distribution transparency - i.e to the user of the system, there is only one system. With the growth of large scale systems, the notion that availability was more important started to take hold.

In 2000, Eric Brewer presented the CAP theorem, which stated that only two of the three properties (consistency, availability, partition tolerance) of shared data systems can be achieved. This was confirmed by Gilbert and Lynch in 2002.

A trade-off needs to be made here between data consistency and high availability, as in large distributed-scale systems network partitions are ubiquitous, and partition tolerance becomes quite important. It is upto the developer to ensure how to handle this trade-off, for eg in systems focusing on availability, the client developer needs to make sure that the system is ready for cases when the read does not reflect the latest updated value.

An overview of consistency models is given in the next section.

Consistency Models - Client and Server

Client side consistency deals with how and when clients or processes see the updates made to a data value in storage.

Strong consistency means once an update is done, any access will reflect the new value, while a weak consistency does not guarantee it immediately. Usually, a specific form of weak consistency is used - eventual consistency - which guarantees that eventually all accesses will reflect the new value if no new updates are done. The time between update and all observers being guaranteed the updated value is the inconsistency window.

Eventual consistency has a number of forms, the most important of which from a practical point of view are monotonic reads and read-your-writes consistency.

Monotonic reads maintain that an earlier value of data is never returned once a process has seen the updated value. Read-your-writes consistency maintains that when a process has updated a data value, it will always see the newer value. Many modern RDBMS models also make use of eventual consistency while reading from backups.

On the server side, to achieve strong consistency, W+R > N. Here,

W: Number of replicas that must acknowledge receiving the update before it completes.

N: Number of nodes which store replicas of data.

R: The number of replicas contacted when a read operation is performed on the data.

For N=2, W=2 and R=1, strong consistency is guaranteed since write and read set overlap. In case of R+W=N, it cannot be guaranteed.

In distributed systems which need high performance and availability, the number of replicas is greater than 2. For high fault tolerance, it may even be in hundreds, with R value being 1.

The configuration of N, W and R depends on what performance needs to be optimized. For R=1, N=W, read is optimized, while for W=1, R=N, writes are optimized.

Eventual consistency is possible when W+R<=N. For updating the values to the newest set, the write set is smaller than replica, and lazy updates are applied to the replica set. The time during which the update is being done is during the inconsistency window.

Conclusion:

Data inconsistency will almost always be present in large scale distributed systems, which must be tolerated. It will depend on the client application to determine if such inconsistencies are acceptable or not, and to what extent, which makes it the responsibility of the developers to take into account the various consistency models, and which model or guarantee is suitable for an application.

At Amazon, DynamoDB brings all such properties into the control of the application, where the application owner can configure these trade-offs between consistency, availability and performance. (Vogels, 2009)

My views on the Paper:

The goal of the paper was to describe the various consistency models in detail, and raise awareness of complexity of large scale systems. The paper is written in a pretty informal way in an easy to understand language, gave a background of early systems and how complex systems can deal with the problem of consistency and availability. The paper clearly explained and went through the various consistency models at both the client and server sides, and the need to understand them for building large-scale systems.

4. Hadoop

1: What is the average number of passengers per trip in general and per day of the week?

We use mapReduce to calculate the average passengers per trip in general and per day of the week.
In the map function, for every row 4 keys are generated - Total_Passenger - which is written to context with the value being the number of passengers.
Total_Trip - which is written to context with the value 1.
Similarly, this is repeated with the keys DAY_Passenger, and DAY_Trip, where we get DAY by parsing the datetime column of the row (column 1), and converting it to day of the week.

The combiner simply combines all the values for each key in the particular split. Here, we sum up each value at the particular splits.
Hadoop then internally combines each key into a <key,list of values> pair.

These keys are then passed onto the reducer.
In the reducer, we check if the key contains "Trip" or not. If it does, it is a trip key, otherwise it is a passenger key. If a passenger key is found, the total count of passengers is calculated and stored in a global variable. If a trip key is passed, we calculate the total number of trips by looping through the values, the average is calculated and written to output. This is done for every pair of passenger and trip keys.

Since the keys are sorted by hadoop, our passengerKey is always found first, followed by a trip key - (FRIDAY_Passenger will always precede FRIDAY_Trip and so on.)

```
FRIDAY_Passenger_Average        1.562183
MONDAY_Passenger_Average        1.5417894
SATURDAY_Passenger_Average      1.6362531
SUNDAY_Passenger_Average        1.6209805
THURSDAY_Passenger_Average      1.5373003
TUESDAY_Passenger_Average       1.5541238
Total_Passenger_Average 1.5670782
WEDNESDAY_Passenger_Average     1.5417327
```

2: What is the average trip distance in general and per day of the week?
Similar to question 1, only our passenger key is replaced by a distance key. Also, the trip distance for every row is passed instead of the number of passengers.

```
FRIDAY_TripDistance_Average     2.7777095
MONDAY_TripDistance_Average     2.8721333
SATURDAY_TripDistance_Average   2.6173317
SUNDAY_TripDistance_Average     2.9626951
THURSDAY_TripDistance_Average   2.755655
TUESDAY_TripDistance_Average    2.8661635
Total_TripDistance_Average      2.8010666
WEDNESDAY_TripDistance_Average  2.7936478
```

3: What are the most used payment types? Create an ordered list using Hadoop MapReduce or a Bash script.
We used a bash script to create the ordered list. We use tail +2 to remove the header from the file, and then cut based on the payment types field. This is then sorted and passed to uniq -c which counts unique values, which is then sorted.
Credit card (1) is the most common payment type, followed by Cash (2), and No charge (3). This is followed by cases where there are disputes (4). (TLC, 2020)
Command: tail +2 ${1} | cut -d, -f10 | sort | uniq -c | sort -nr
File used - most_payment_types.sh

```
kunal@kunal:~/Downloads/UCD/Sem 2/Big Data Programmin
./most_payment_types.sh yellow_tripdata_2019-01.csv
Count   Payment Type
5486027 1
2137415 2
  33186 3
  11164 4
```

4: Create a graph (using the output of a MapReduce job) showing the average number of passengers over the day (per hour). Create a version for work days and another for weekend days.

Similar to question 1, only we create keys based on HOUR_Passenger and HOUR_Trip.

Also, for weekdays, keys created are HOUR_WEEKDAY_Passenger and HOUR_WEEKDAY_Trip, while for weekends, keys are HOUR_WEEKEND_Passenger and HOUR_WEEKEND_Trip.

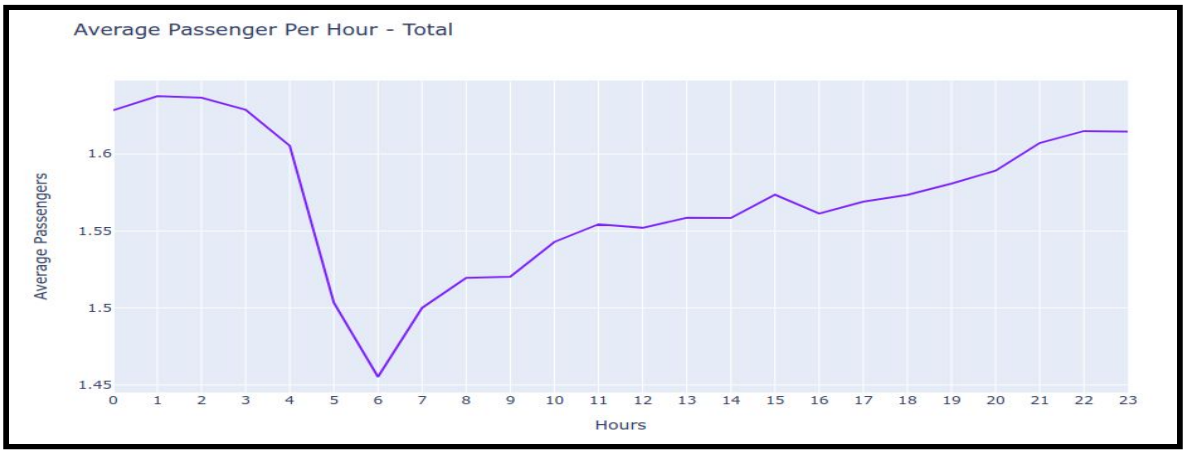
In all, 24 keys for each hour are generated for part 1, while 48 hours (24 hours for weekend and weekday) are generated in part 2.

A part of the output is shown below:

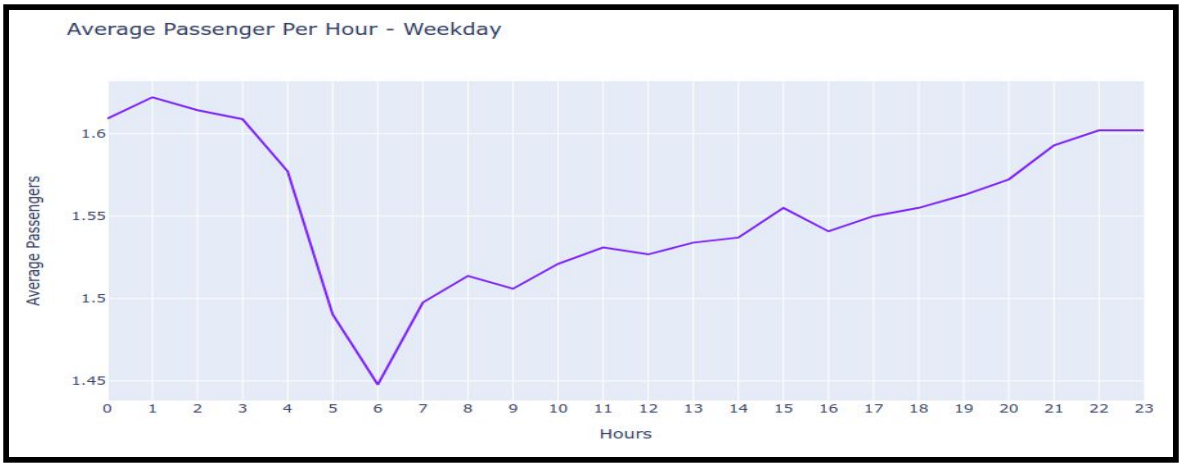
```
00_Passenger_Average    1.6284041164827863
01_Passenger_Average    1.6375902354668115
02_Passenger_Average    1.6365921889855921
03_Passenger_Average    1.628772055714304
04_Passenger_Average    1.6053531403890515
```

```
00_Weekday_Passenger_Average    1.6092550536668768
00_Weekend_Passenger_Average    1.6525488317400867
01_Weekday_Passenger_Average    1.622233166434102
01_Weekend_Passenger_Average    1.6523424666535251
02_Weekday_Passenger_Average    1.6143939393939395
02_Weekend_Passenger_Average    1.6544992071881606
```

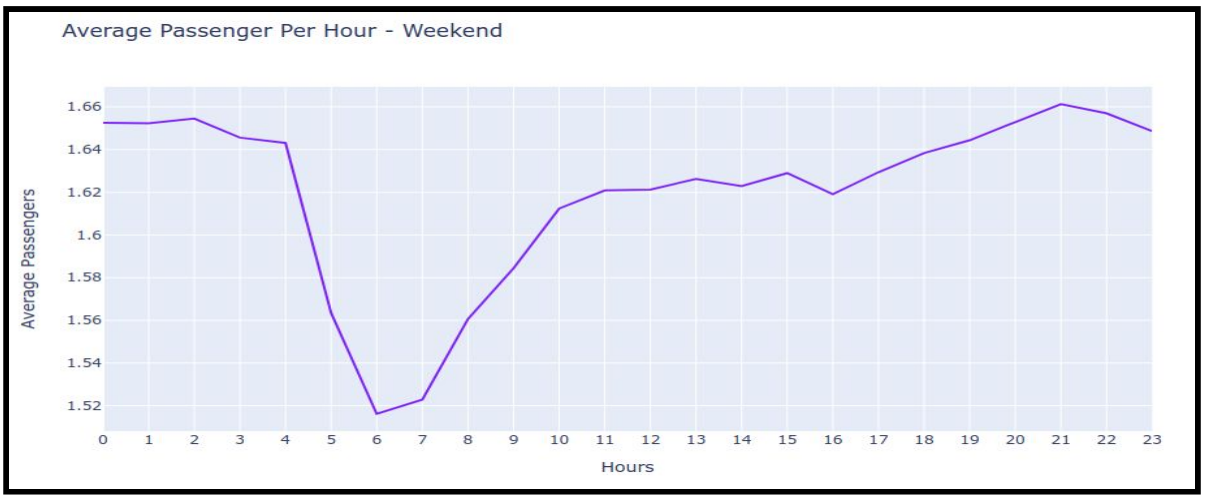
(Partial Outputs)



(Average Passengers vs Hours - Total)



(Average Passengers vs Hours - Weekday)



(Average Passengers vs Hours - Weekend)

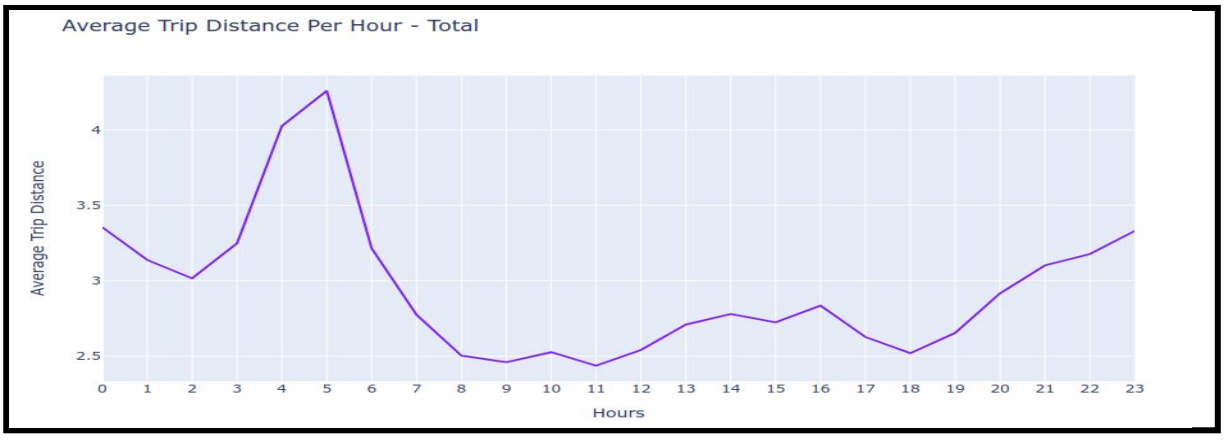
5: Create a graph showing the average trip distance over the day (per hour). Create a version for work days and another for weekend days.

Similar to question 4, only our passenger key is replaced by a distance key. Also, the trip distance for every row is passed instead of the number of passengers. 24 keys are generated in part 1, and 48 in part 2.

```
00_TripDistance_Average 3.3547843758578177
01_TripDistance_Average 3.137899231197171
02_TripDistance_Average 3.0164440868865676
03_TripDistance_Average 3.2481581476403387
04_TripDistance_Average 4.0254171097622775
```

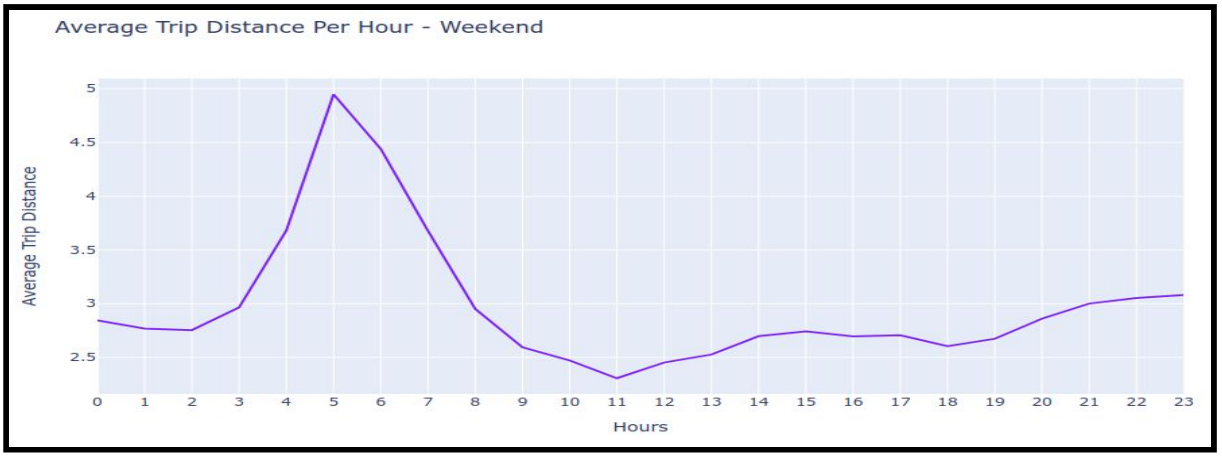
```
00_Weekday_TripDistance_Average 3.7596002832298643
00_Weekend_TripDistance_Average 2.8443591446551846
01_Weekday_TripDistance_Average 3.523715149525298
01_Weekend_TripDistance_Average 2.767278664826861
02_Weekday_TripDistance_Average 3.3428304668304687
02_Weekend_TripDistance_Average 2.753152748414379
```

(Partial Outputs)



(Average Trip Distance vs Hours - Total)

(Average Trip Distance vs Hours - Weekday)



(Average Trip Distance vs Hours - Weekend)

References:

MongoDB. 2020. *Nosql Vs SQL (Relational Databases)*. [online] Available at: <https://www.mongodb.com/scale/nosql-vs-relational-databases> [Accessed 29 March 2020].

Vogels, W., 2009. Eventually consistent. *Communications of the ACM*, [online] 52(1), p.40. Available at: <https://dl.acm.org/doi/10.1145/1435417.1435432>.

Zagorulkin, D., 2020. *Mongodb: How Can I See The Execution Time For The Aggregate Command?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/14021605/mongodb-how-can-i-see-the-execution-time-for-the-aggregate-command/14022199> [Accessed 29 March 2020].