Final Project Report on Data Analysis of Mileage Savings via Ride Sharing

Group Members:
Edison Larco, Filmon Ghebremeskel, Daniel Aguilar, Mallika Patil

Github Link:
https://github.com/mallikampatil/Data-Analysis-of-Mileage-Savings-via-Ride-Sharing

The Database Querying Strategy

The process for this project in mySQL used a total of 4 window queries. After this, python was used to calculate the percentage of trips that were merged. We will be discussing each one of these window queries.

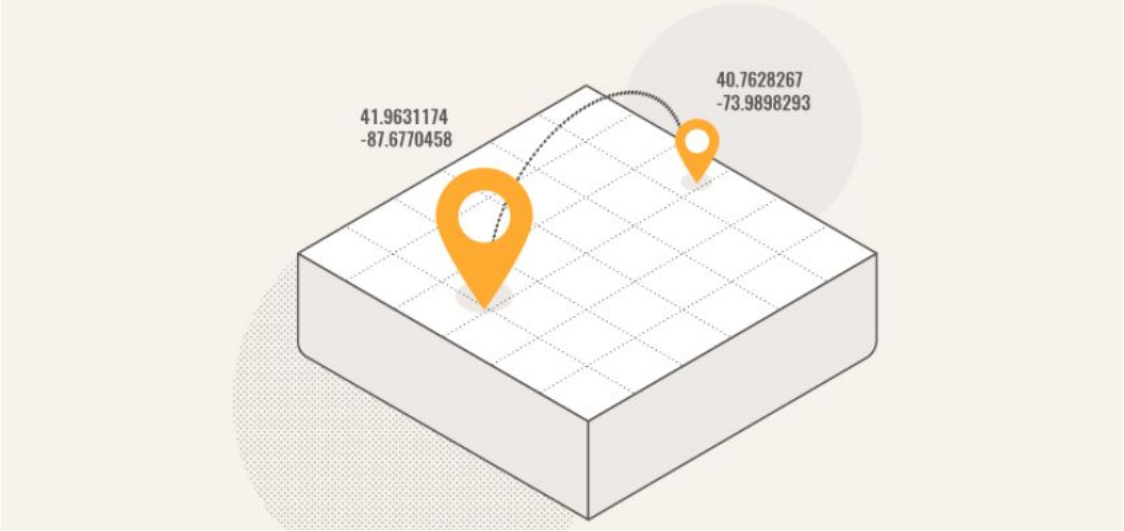
The first one that was designed was to get the average speed so that we could know the time it could take a Taxi C to go from the pick up location of Taxi B to the pickup location of Taxi A, as show below:

```
SELECT  g_id, RIGHT(sec_to_time((TIME_TO_SEC(g1.Lpep_dropoff_datetime) - TIME_TO_SEC(g1.Ipep_pickup_datetime))),5)  AS Trip_time_in_minut
        (g1.Trip_distance / ((TIME_TO_SEC(g1.Lpep_dropoff_datetime) - TIME_TO_SEC(g1.Ipep_pickup_datetime)) / 3600) ) AS Trip_Speed,
        g1.Passenger_count, g1.Trip_distance, ST_Distance_Sphere(
 point(Pickup_longitude, Pickup_latitude),
 point(Dropoff_longitude, Dropoff_latitude)
 ) * .000621371192 AS P2PDistance, g1.Fare_amount, g1.Total_amount, g1.Ipep_pickup_datetime, g1.Lpep_dropoff_datetime,
        g1.Pickup_longitude, g1.Pickup_latitude, g1.Dropoff_Longitude, g1.Dropoff_Latitude
FROM greentripdatajan g1
WHERE g1.Passenger_count < 3
```

The main idea was also to start filtering out the data so we could start seeing better results, adjusting to our goal. As shown above, we limited the query to records that had a number of passengers no greater than 2. This way, we could make sure that when we did a cross join, we could get possible shareable trips with no more than 3 people.

The next window query that was implemented in mySQL had to do with trying to find out how far each of the points in the given sequences were from each other. The Euclidean formula was used at the beginning but the results and the way that this formula calculates a distance in a straight line was not convincing to keep using it for the rest of the queries.

Instead, the native mySql function, ST_Distance_Sphere, was used. This function is used for calculating the results over a sphere surface as shown below:



The results obtained by using this function were much closer to the trip distance column given for both databases. We used this strategy for calculating the distances from every single point in the sequences, as shown below:

```
ST_Distance_Sphere(
point(A.Pickup_longitude, A.Pickup_latitude),
point(B.Dropoff_longitude, B.Dropoff_latitude)
) * .000621371192 AS o1_to_d2Dist,
ST_Distance_Sphere(
```

Here we can see the coordinates for the green database that were used to calculate a distance, from o1_to_d2Dist.

The strategy was, at this point, to do a cross join of records so we could start comparing, in the following queries, which trips could be merged or not.

During this process we mainly worked by using the green database. This means that we did a cross join of green with green. And later, we did yellow with yellow.

However, by taking into consideration the feedback that was given during the video presentation, the green and yellow databases were merged by doing the cross join as show below:

```
ST_Distance_Sphere(
point(B.Dropoff_longitude, B.Dropoff_latitude),
point(A.dropoff_Longitude, A.dropoff_Latitude)
) * .000621371192 AS d2_to_d1Dist
FROM gdataspeedtimejan AS B
    CROSS JOIN
    ydataspeedtimejan AS A
WHERE B.Passenger_count + A.passenger_count < 4
```

Here we can see that the green database from January is being cross-joined with the yellow database from January. At the same time, we are specifying that the results should contain a total of passengers less than 4 for every single record. These are some of the results:

trip_distance	Second_trip_distance	Trip_Speed	Second_speed	passenger_count	Second_trip_passenger
1.59	10.80	5.285319	16.889661	1	1
1.59	0.90	5.285319	10.418007	1	2
1.59	2.50	5.285319	8.100810	1	1
1.59	10.20	5.285319	16.488550	1	2
1.59	0.22	5.285319	17.600000	1	1
1.59	0.70	5.285319	10.080000	1	1

Next, we have the results for the computation for the distances between two points for each sequence:

o1_to_d2Dist	o1_to_d1Dist	o1_to_o2Dist	o2_to_o1Dist	o2_to_d1Dist	o2_to_d2Dist	d1_to_d2Dist	d2_to_d1Dist
9.824025949021694	1.0009603149896031	4.011983661000441	4.011983661000441	3.059301020750074	7.354114877018728	9.325841136562849	9.325841136562849
4.011572704768918	1.0009603149896031	4.455026955685995	4.455026955685995	4.045306891039269	0.5266115976131186	3.549558925974821	3.549558925974821
5.5444923691794665	1.0009603149896031	4.933165436013327	4.933165436013327	4.863136321058425	1.1767084643438526	5.654557263039728	5.654557263039728
5.169375808375396	1.0009603149896031	5.448414123942195	5.448414123942195	5.33074981879373	6.048865571749442	4.182304953358411	4.182304953358411
8.743341143031317	1.0009603149896031	8.946001796015999	8.946001796015999	8.556079717292148	0.2282592407067096	8.365785311363645	8.365785311363645
4.282197128754194	1.0009603149896031	4.068273486465262	4.068273486465262	3.5864119705535438	0.582847413458546	3.6790360957541366	3.679036095754136

Then, the next window query was in charge of finding out the total time it could take from going from one point to another. This is very crucial information since with these time results more data was going to be filtered out to allow to see if a trip was a valid candidate for a shareable ride, as show below:

```
SELECT *, (o1_to_d2Dist / Average_speed) * 60 AS Time_o1_to_d2,
     (o1_to_d1Dist / Average_speed) * 60 AS Time_o1_to_d1,
     (o2_to_o1Dist / Average_speed) * 60 AS Time_o2_to_o1,
     (o2_to_d1Dist / Average_speed) * 60 AS Time_o2_to_d1,
     (o2_to_d2Dist / Average_speed) * 60 AS Time_o2_to_d2,
     (d1_to_d2Dist / Average_speed) * 60 AS Time_d1_to_d2,
     (d2_to_d1Dist / Average_speed) * 60 AS Time_d2_to_d1,
     (o1_to_o2Dist+o2_to_d1Dist+d1_to_d2Dist) AS Sequence1Dist,
     (o1_to_o2Dist+ o2_to_d2Dist + d2_to_d1Dist) AS Sequence2Dist,
     (o2_to_o1Dist + o1_to_d1Dist + d1_to_d2Dist) AS Sequence3Dist,
     (o2_to_o1Dist + o1_to_d2Dist + d2_to_d1Dist) AS Sequence4Dist,
     (o1_to_d1Dist + o2_to_d2Dist) AS totalDistanceP2P
```

This query produced the following results:

Time_o1_to_d2	Time_o1_to_d1	Time_o2_to_o1	Time_o2_to_d1	Time_o2_to_d2	Time_d1_to_d2	Time_d2_to_d1
27.455124367535582	6.341717188503636	23.724286730349608	17.39207143377616	5.655189407671911	21.299965332909956	21.299965332909956
38.10785637876614	6.026066552012906	32.79045220814722	31.11173334649841	11.9650996110623	38.178105541755656	38.178105541755656
47.43963022745986	5.819319947787181	47.590356050601244	42.065145467432615	3.1983713340497317	42.05911098316022	42.05911098316022
42.673347090802146	5.834995291635298	40.718978078141355	34.88990008193729	3.8202284535504427	36.842996830921436	36.842996830921436
9.365431044379138	8.65482369674569	29.655876304054345	23.687789902390254	22.76495210199404	11.117683042472306	11.117683042472306
12.57713638778915	4.470951344243721	34.23857397693118	32.40822403149594	21.878840337121517	11.888051020734302	11.888051020734302

Time_o2_to_d2	Time_d1_to_d2	Time_d2_to_d1	Sequence1Dist	Sequence2Dist	Sequence3Dist	Sequence4Dist	totalDistanceP2P
2.83959976139986	41.562986294619705	41.562986294619705	18.389556443298545	21.846537852973505	16.93102144491034	26.076632285447694	9.79271334597429
8.398647238108165	45.2334585767611	45.2334585767611	17.977184794190904	20.83050632937089	16.035639858925393	22.906039994825086	9.672063408356623
8.6068185662896	28.395392479320517	28.395392479320517	19.557795502063065	22.945435883272253	18.17459823214896	27.46770746856476	9.64803458903442
5.31577956592627	15.488159795217248	15.488159795217248	13.372204522139347	17.139317413864795	9.509335646106914	12.76272638009147	9.631902397737086
1.564449580749834	22.63952172527555	22.63952172527555	13.29103749057715	17.85748036543957	13.11715096599567	15.788988834214262	9.617526337355027
4.964745609995596	37.4328647949271	37.4328647949271	17.50074219781291	22.156893609921987	14.74948863189487	23.269694656986793	9.409325608006322
4.018477365247584	10.480982888279259	10.480982888279259	17.814448080419123	18.76947732491313	11.381896769862957	12.576165164951071	9.389501185029381
4.918166127068616	6.982467944637355	6.982467944637355	17.628952408549655	18.30296113574038	10.924018157580992	12.385475513472421	9.380863608138595
7.15207547502813	23.195028289179987	23.195028289179987	13.816999479580566	18.17666361123505	10.83905155843129	15.381403601392785	9.339532682782966
1.901428910319765	1.5761424948512706	1.5761424948512706	18.074302015809376	17.778700909717564	10.501744932533866	10.6344653997193	9.278876607162905
0.83274474708589	32.41291547459925	32.41291547459925	19.924533135675993	22.61033431411152	18.224168657076913	26.332967894642536	9.26336259494573
2.02073925079799	6.071792209226833	6.071792209226833	14.811111247975392	16.318469558362178	11.93730672162601	10.414576886295041	9.258359774647293
9.346189991103707	22.949540050985377	22.949540050985377	17.547048399212606	20.15814391771071	15.807682459638086	23.76101104612617	9.227658395983747
4.84268838913021	2.544936027680855	2.544936027680855	11.503493794935686	11.507816076195022	7.264779541050214	7.188345863998934	9.120233473055933
3.79712336165777	23.421225317711006	23.421225317711006	23.2784062747085	24.131298151573887	17.079004408316273	24.18067043473325	9.054214373236817

As we can see here, these results make it possible now to start analyzing if one sequence total distance is going to be greater than or less than the total distance from A to B. By looking at the column name “totalDistanceP2P” and “Sequence3Dist, we find two trips, that are satisfying the condition that the total sequence distance being less than the distance of trip A and trip B makes those records candidates for a mergeable ride.

These are just candidate trips since we haven’t applied the condition yet where we check if one trip can get to the other’s trip pickup location 5 min before that trip starts.

The next last query window used in mySQL for this project was centered around the idea of splitting up the database into two parts:

1) o1 is less than o2 :

   This means that for a taxi C, with the average speed of Taxi A and Taxi B, to travel to the pick up location of Taxi B from the location of Taxi A, it had to have a pick up datetime less than the one from Taxi B. This way the condition of arriving at that location 5 min before Taxi B’s trip started could be achieved.
2) o2 is less than o1 :

   This shares the same logic as the first one. For a Taxi C, with the average speed calculated, to go from Taxi B pick up location to Taxi A pick up location it had to have a pick up datetime less than the one from taxi A. This way the condition of arriving at that location 5 min before Taxi A’s trip started could be achieved.

As shown here:

```
SELECT *,((Time_o1_to_o2 + Time_o2_to_d1 + Time_d1_to_d2) - Time_o2_to_d2) AS checkSequence1Timing
FROM greenandyellowimportantview
WHERE tpep_pickup_datetime < Second_pickup
AND (totalDistanceP2P > Sequence1Dist OR totalDistanceP2P > Sequence2Dist OR totalDistanceP2P > Sequence3Dist
OR totalDistanceP2P > Sequence4Dist)
AND ((Time_o1_to_o2 + Time_o2_to_d1 + Time_d1_to_d2) - Time_o2_to_d2) <= 5
```

We also have three other different queries in this query window, for the other 3 sequences.

Tpep_pickup_datetime is the pickup time for the yellow taxi and Second_pickup is the pickup time for green taxi. By splitting up the data, we achieved a better logic for using the sequences. This allowed the algorithm results in python to output the correct results either when

yellow had to go to green's pick up location or green had to go to yellow's pick up location.

This query is also computing if Taxi C could get from the pickup time to the drop off time of the original trip. As shown there, we start at o1 pickup time and as we go dropping off the customers we end up at d2 which means that we have to compute the difference of the original drop off time with the time it takes Taxi C to get to the original drop off time. If Taxi C gets to d2 within 5 min of the original drop off time then it will be considered a shareable ride.

The Average Query Run Time

The very first strategy tried was taking a time period of 16 minutes per 50000 rows of data. Each sequence is 50000 roads of data and each month had four sequences analyzed. Having each of the queries requiring 16 minutes on average meant that each month took 1 hour and 4 minutes. And since the time period analyzed for green taxis was 6 months, running these 4 sequences for 6 months took about 6.4 hours. Running these additionally for the yellow taxis took even longer and took about 12.8 hours. The 16 minutes that were calculated with our very first strategy only included the running time of the MySQL queries. With our later developed Python script, the time may have been even longer.

The strategy that was used during our presentation had altered the queries slightly which made our time much faster. By making small changes, the query running time was improved by 40%. During the presentation, one of the queries would run for an average of 10 minutes per 50000. Again, each month had 4 sequences, and therefore, the running of 200000 rows of data for each month took 40 minutes on average. That meant running 6 months of data, nearly 1.2 million rows of data, and required 4 hours. Running both the colors of taxis meant, the queries were analyzing 2.4 million rows of data and took on average 8 hours.

However, after merging the two databases, these runtimes changed for both mySQL and Python.We still run a total of 200,000 records per month, by using the 50000 limit per sequence. The average runtime for 50,000 records was 220min in mySQL. This average is the result of combining the query times for the other 3 sequences and the times it took to run the same queries in other computers with different hardware and software specifications. Moreover, it is important to mention again, that this runtime average is for mySQL only, not python.The average runtime in python for the same 50,000 records was about 30 min by considering again the different hardware and software constraints of different computers.

The Effectiveness in Terms of Miles Saved

To calculate the miles that were going to be saved we first made sure that the algorithm that was in charge of determining if a sequence or more were valid, was working.

```
#Loops through to check whoch sequence gives the best saving
i = 0
for row in jan1.itertuples():
    if(jan1.at[i, 'TotalDistanceP2P'] > jan1.at[i, 'Sequence1Dist']):
        myList11.append(row)
    elif(jan1.at[i, 'TotalDistanceP2P'] > jan1.at[i, 'Sequence2Dist']):
        myList12.append(row)
    elif (jan1.at[i, 'TotalDistanceP2P'] > jan1.at[i, 'Sequence3Dist']):
        myList13.append(row)
    elif (jan1.at[i, 'TotalDistanceP2P'] > jan1.at[i, 'Sequence4Dist']):
        myList14.append(row)

    i = i + 1


#Convert the lists into DataFrame
jan1DF1 = pd.DataFrame(myList11)
jan1DF2 = pd.DataFrame(myList12)
jan1DF3 = pd.DataFrame(myList13)
jan1DF4 = pd.DataFrame(myList14)

#Calculate the savings in a new column
for row in jan1DF1.itertuples():
    jan1DF1["Savings1"] = jan1DF1["TotalDistanceP2P"] - jan1DF1["Sequenc
e1Dist"]


#Display the dataframes
jan1DF1
```

Here, we use python to iterate over the 50000 records of the combined data of the green and yellow taxi database that we retrieved from mySQL. Every single record that satisfies the sequence conditions is stored in a list which will be converted to a data frame. The records in this data frame will be subtracted from the totalDistanceP2P that was retrieved from mySQL.

Out[20]:

	Index	Sequence1Dist	Sequence2Dist	Sequence3Dist	Sequence4Dist	TotalDistanceP2P
0	0	15.047159	25.101836	14.966485	24.364701	15.301878
1	1	6.481197	10.362479	8.452358	12.266569	7.076648
2	2	10.169923	17.749392	12.404225	19.958742	10.511694
3	3	6.016234	9.269882	6.662602	9.854305	7.773807
4	4	3.833584	4.624430	4.526146	5.122151	5.264811
...	...	...	...	...	...	...
41843	49996	14.381329	24.916217	15.839432	26.148268	16.360446
41844	49997	8.383139	14.246413	9.688732	15.462511	9.121680
41845	49998	3.483071	3.875122	3.686487	3.997730	4.752634
41846	49999	3.281154	3.279675	3.323967	3.355555	4.519707
41847	50000	7.340245	11.161622	7.543860	10.676651	8.181760

41848 rows × 8 columns

In [22]:
```
#Total amount in savings
jan1DF1['Savings1'].sum()
```

Out[22]: 88417.3064613629

In [24]:
```
#The percentage
index = jan1DF1.index
perc = len(index)/50000
print(perc * 100)
```

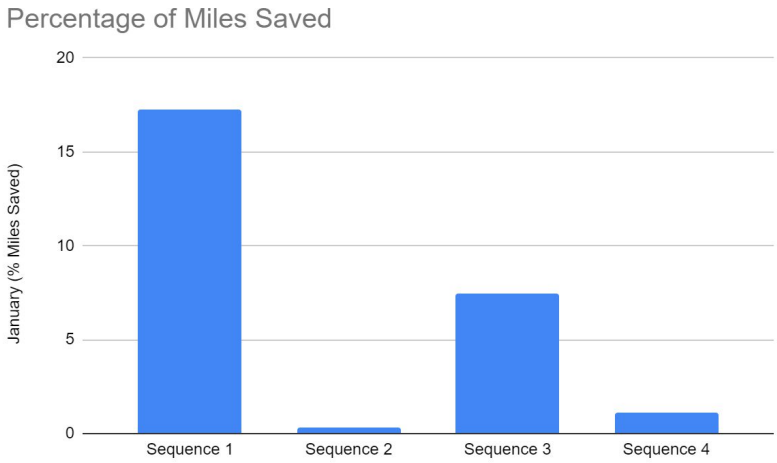
83.696

In this picture we first calculate the percent of trips that could be shared for sequence 1. We get the total amount, 41848, and divide that by the total amount of trips analyzed in the sequence, 50000. The result for this sequence is 83.696%. We then retrieve the next 50000 records of data for the remaining 3 sequences for January so we can have a consolidated average for shareable trips of 63.22%.

Every single record that is retrieved from mySQL follows the same logic. At the end the results will be added up so we can have a consolidated amount of miles saved.

There was a pdf generated from the values that were calculated and it contained an aggregation of all the miles saved sums from January to June, in pools of 4. Since the first strategy was to separate the green and yellow taxi data, the result was two groups of 6 months data on how many miles were saved.

Percentage of Miles Saved from each sequence compared with total distance for mergeable trips from the green and yellow data::



The Period of Time for Which Ride-Sharing was Analyzed

The first set of data analyzed was only for the month of January 2015 for green taxis. The first analysis was divided into two parts, one analyzing the green taxi and the second analyzing the yellow taxi data. The first queries were all written in January 2015 of the green taxi. For this there were pools of 4 sequences, with 50000 rows of data each with a total of 20000 rows of data for each month, meaning that only partial data of each month was used for analysis. After the first month of analysis on the green taxi, the months February, March, April, May, and June, were analyzed for the green taxi, and finally, the same was repeated for months part of the yellow taxi data. The sequences were run the same way, for a partial amount of data of the months of January, February, March, April, May, and June of 2015 for the yellow taxis trips.

The total amount of records analyzed for the green data and yellow data was 2.4M. We achieved this amount by running 200,000 records per month for each database for 6 months each. 1.2M for green and 1.2M for yellow.

All these same amounts were retrieved by Python to continue running them through the algorithms where the savings were calculated.

However, it is important to mention again that we thought we had run every database separately. After the video presentation, we merged the databases, green and yellow, so we could try to achieve new results based on this.

We were able to run all the same queries in mySQL, 200,000 records of the new combined database was analyzed for 6 months, giving a total of 1.2M of records analyzed. However, due to logistic and computer efficiency issues, we could only run the month of January in Python.

Chart showing the percent of shareable rides for green taxi:

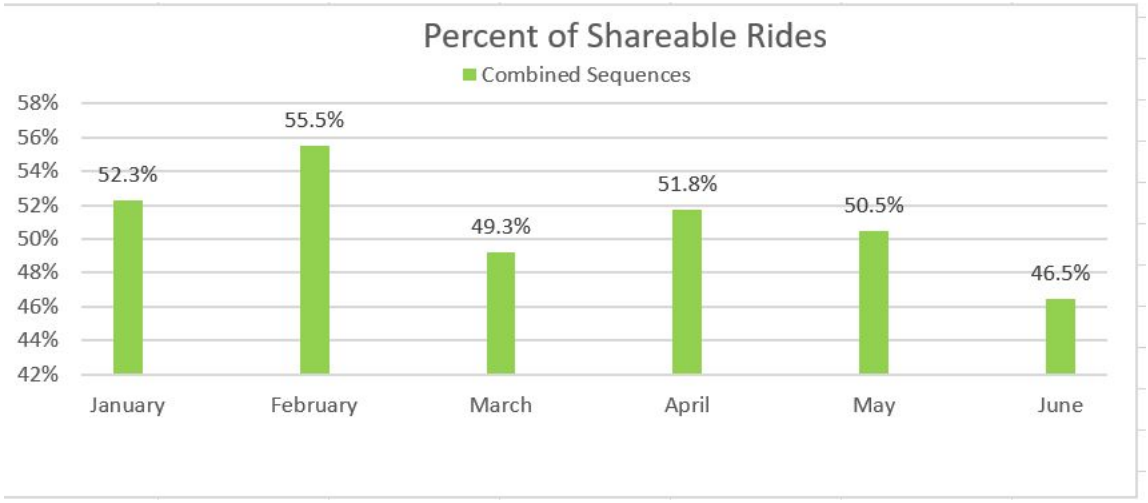


Chart showing the percent of shareable rides for yellow taxi:

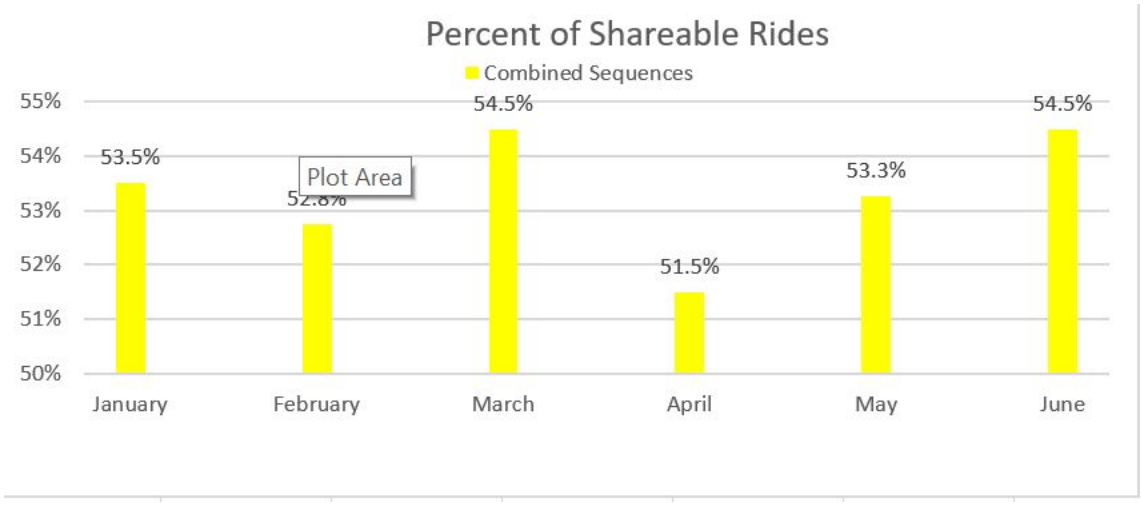
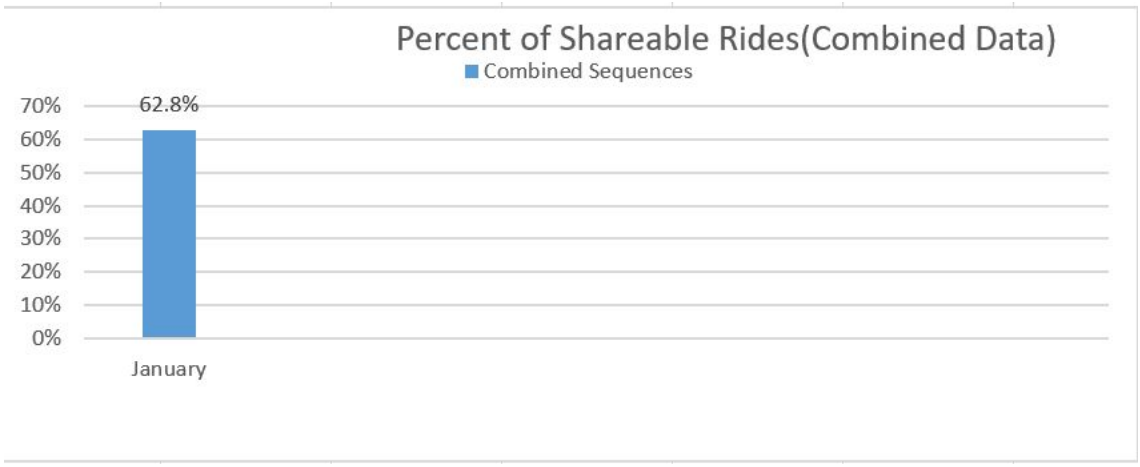


Chart showing the percent of shareable rides for the combined databases:



The Values of Delay Considered

The first delay that was considered due to time limitations was a 5 minutes delay. Since such a large pool of data was analyzed, 200000 results of data per month, only a 5-minute delay was considered at first. From the feedback that we received, we considered more delays including 10 minutes and 15 minutes. This was done by editing the query below. For changing the query, we changed the 5 in the query underneath to 10 and 15.

```
SELECT *,((Time_o1_to_o2 + Time_o2_to_d1 + Time_d1_to_d2) - Time_o2_to_d2) AS checkSequence1Timing
FROM greenandyellowimportantview
WHERE tpep_pickup_datetime < Second_pickup
AND (totalDistanceP2P > Sequence1Dist OR totalDistanceP2P > Sequence2Dist OR totalDistanceP2P > Sequence3Dist
OR totalDistanceP2P > Sequence4Dist)
AND ((Time_o1_to_o2 + Time_o2_to_d1 + Time_d1_to_d2) - Time_o2_to_d2) <= 5
```

Limitations and Issues Raised in Presentations

During the video presentation, only two issues were brought up by the professor about our analysis so far. One of the main issues raised during the presentation was how we choose to combine trips. The question asked was if a trip has two potential matches that can be combined, which trip did the query combine the trip with. The answer to that question was based on our sequences. We combined the trip with the trip that resulted in the most amount of miles saved according to our 4 sequences.

Another important point brought up by the professor was the percentage of savings. Instead of the actual savings amount, he wanted the percentage of savings calculated, which is something that we said would be considered in the report and it was considered in the effectiveness of the miles saved.

Teamwork Retrospective Analysis

The project was first started as soon as a detailed description was explained. During the first few weeks, we focused on loading all the data into MySQL. Since the data files are large we loaded in two years' worth of data, which included loading in twelve separate CSV format files.

All members of the team took about a week to load the data into MySQL and then had weekly team meetings to work on queries that analyzed the mergeable trips in the data. The team used many different forms of communication to hold these meetings.

The main modes of communication that the team used were WhatsApp, Discord, and Google Meet. Team meetings were held weekly starting in the middle of the semester with most of the team members present. During these meetings, the team worked on writing queries by sharing screens and working through any issues. All files and scripts shared between team members were done through discord. And most of the communication about meeting times was done through WhatsApp. As the semester progressed, the meetings were more frequent due to the upcoming deadlines.

The roles and responsibilities in the group were divided amongst all members of the group. Although all members worked on all aspects of the project, different roles emerged. Filmon was responsible for beginning querying and distance calculation queries. Edison was mainly responsible for creating the MySQL queries and came up with the queries for each of the sequences and distances. Danny also worked on the queries, running each of the queries on all the data for Green and Yellow Taxis. Having some background knowledge in Python, Danny used Jupyter Notebooks to create a MySQL connection and create python scripts that ran and performed calculations on our MySQL data. Mallika worked mainly on compiling all the information in a readable format for the presentation and recording the video explaining the data collected during the semester. Mallika was also responsible for compiling all the information in a final project report.

Appendix

First, we use the data downloaded to create the first view:

```
USE yellowtripdatajan2015;

CREATE VIEW YDataSpeedTimeJan AS
SELECT  RIGHT(sec_to_time((TIME_TO_SEC(y1.tpep_dropoff_datetime) -
TIME_TO_SEC(y1.tpep_pickup_datetime))),5)  AS Trip_time_in_minutes,
                    (y1.trip_distance / ((TIME_TO_SEC(y1.tpep_dropoff_datetime) -
TIME_TO_SEC(y1.tpep_pickup_datetime)) / 3600) ) AS Trip_Speed,
      y1.passenger_count, y1.trip_distance, ST_Distance_Sphere(
 point(pickup_longitude, pickup_latitude),
 point(dropoff_longitude, dropoff_latitude)
 ) * .000621371192 AS P2PDistance,
      y1.fare_amount, y1.total_amount, y1.tpep_pickup_datetime, y1.tpep_dropoff_datetime,
      y1.pickup_longitude, y1.pickup_latitude, y1.dropoff_Longitude, y1.dropoff_Latitude
FROM yellowtripdatajan y1
WHERE y1.passenger_count < 3
```

****

Second, we use the first view to create a second view with all the sequences:

```
CREATE VIEW YellowTaxiViewJan AS
```

```
SELECT A.Trip_time_in_minutes ,B.Trip_time_in_minutes AS Second_trip_time,
A.Trip_distance, B.Trip_distance AS Second_trip_distance, A.Trip_speed, B.Trip_speed AS
Second_speed ,
            A.passenger_count, B.passenger_count AS Second_trip_passenger,
A.tpep_pickup_datetime, B.tpep_pickup_datetime AS Second_pickup, A.tpep_dropoff_datetime,
B.tpep_dropoff_datetime AS Second_dropoff,
      (A.Trip_speed + B.Trip_speed)/2 AS Average_speed,
(ST_Distance_Sphere(
 point(A.pickup_longitude, A.pickup_latitude),
 point(B.pickup_longitude, B.pickup_latitude)
 ) * .000621371192) / ((A.Trip_speed + B.Trip_speed)/2) * 60 AS Time_o1_to_o2,
 ST_Distance_Sphere(
 point(A.pickup_longitude, A.pickup_latitude),
 point(B.dropoff_longitude, B.dropoff_latitude)
 ) * .000621371192 AS o1_to_d2Dist,
 ST_Distance_Sphere(
 point(A.pickup_longitude, A.pickup_latitude),
 point(A.dropoff_longitude, A.dropoff_latitude)
 ) * .000621371192 AS o1_to_d1Dist,
 ST_Distance_Sphere(
 point(A.pickup_longitude, A.pickup_latitude),
 point(B.pickup_longitude, B.pickup_latitude)
 ) * .000621371192 AS o1_to_o2Dist,
 ST_Distance_Sphere(
 point(B.pickup_longitude, B.pickup_latitude),
 point(A.pickup_longitude, A.pickup_latitude)
 ) * .000621371192 AS o2_to_o1Dist,
 ST_Distance_Sphere(
 point(B.pickup_longitude, B.pickup_latitude),
 point(A.dropoff_longitude, A.dropoff_latitude)
 ) * .000621371192 AS o2_to_d1Dist,
 ST_Distance_Sphere(
 point(B.pickup_longitude, B.pickup_latitude),
 point(B.dropoff_longitude, B.dropoff_latitude)
 ) * .000621371192 AS o2_to_d2Dist,
 ST_Distance_Sphere(
 point(A.dropoff_longitude, A.dropoff_latitude),
 point(B.dropoff_longitude, B.dropoff_latitude)
 ) * .000621371192 AS d1_to_d2Dist,
 ST_Distance_Sphere(
 point(B.dropoff_longitude, B.dropoff_latitude),
 point(A.dropoff_longitude, A.dropoff_latitude)
 ) * .000621371192 AS d2_to_d1Dist
FROM ydataspeedtimejan AS B
         CROSS JOIN
   ydataspeedtimejan AS A
```

```
WHERE B.Passenger_count + A.Passenger_count < 4
```

****

Third, we calculate mergeable trips from the final view:

```
CREATE VIEW YELLOWallImportantData AS
SELECT *, (o1_to_d2Dist / Average_speed) * 60 AS Time_o1_to_d2,
        (o1_to_d1Dist / Average_speed) * 60 AS Time_o1_to_d1,
    (o2_to_o1Dist / Average_speed) * 60 AS Time_o2_to_o1,
    (o2_to_d1Dist / Average_speed) * 60 AS Time_o2_to_d1,
    (o2_to_d2Dist / Average_speed) * 60 AS Time_o2_to_d2,
    (d1_to_d2Dist / Average_speed) * 60 AS Time_d1_to_d2,
    (d2_to_d1Dist / Average_speed) * 60 AS Time_d2_to_d1,
    (o1_to_o2Dist+o2_to_d1Dist+d1_to_d2Dist) AS Sequence1Dist,
    (o1_to_o2Dist+ o2_to_d2Dist + d2_to_d1Dist) AS Sequence2Dist,
    (o2_to_o1Dist + o1_to_d1Dist + d1_to_d2Dist) AS Sequence3Dist,
    (o2_to_o1Dist + o1_to_d2Dist + d2_to_d1Dist) AS Sequence4Dist,
    (o1_to_d1Dist + o2_to_d2Dist) AS totalDistanceP2P
FROM yellowtaxiviewjan;
```

****

Fourth step we took was to go to the Python code and we ran the Python code for each sequence, which resulted in the miles saved total. The Python file is on our Github.