Quaxi

CIS 550 Final Project Writeup

Sneha Advani, Arjun Lal, Romit Nagda, Rahul Shekhar

Introduction

Our objective throughout this project was to provide a service to both riders and drivers that would enhance their experience using taxi rides in NYC. Taking advantage of a taxi service in NYC is quite common, so we thought that providing tangible and easily understandable results to potential users would be useful. Furthermore, with the vast amount of taxi data currently available, it is certainly interesting to see what practical insights we can gain from querying the available data.

Specifically, we looked at data from NYC taxi rides spanning 2009 to 2019. Discussed in more detail below, the data contains attributes such as fare rates, location (zone and borough), and trip distance. We decided to make use of the data to provide dynamic insights for both drivers and riders, depending on various user-input parameters, such as target cost or target location. Some feature that we've implemented for riders include viewing taxi zones that are reachable from a rider's current location given a target cost, viewing payment statistics regarding both green and yellow taxis from a specified starting and ending location, and viewing breakdowns of what riders can expect their payment to be composed of (in terms of fare amount, toll amount, tip amount, etc.). The intention here is to help riders explore new parts of the city and provide them with knowledge on how they can get from any starting location in New York City to a particular ending location as cost-efficiently as possible. Similarly, a driver can use Quaxi to find zones that have higher average fare rates than that of their current zone, to help them optimize their trip's profit. They can also view a list of the most popular destinations given a driver's current trip trajectory, in order to plan the next ride, and they can view the average distance of a trip starting from any particular zone.

Our goal is to help both taxi drivers and riders gain as much insight as possible into traffic patterns, cost information, and more, so that they can optimize their taxi experience.

Data

Our data source was a compilation of data regarding taxi rides from 2009 to 2019 (broken down across months), across both green and yellow taxis. All 20 datasets can be found here: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page. We decided to focus on a subset of data from January 2019, discussed further in *Technical Challenges.* The full yellow taxi dataset from January 2019, as an example, is 687.1 MB, 7667792 rows, and contains 18 columns.

The taxi data was split over several data tables: the Ride table, Payment table, and Location table. The Ride table contains all information pertinent to each ride taken in a yellow or green taxi, such as the pickup and dropoff location, passenger count, and trip distance. The Payment table contains the total amount paid for each ride, broken down into the fare amount, tip amount, and more. The Location table contains more information about each Location ID, like the borough and zone. Below are several key summary statistics for some columns in this dataset:

1. Trip distance: mean = 2.80 miles, std = 3.74 miles
2. Fare amount: mean = $12.41, std = $262.07
3. Tip amount: mean = $1.83, std = $2.50
4. Total amount paid: mean = $15.68, std = $262.29

Discussed below in *Database*, we uploaded the data into several tables detailing ride information, payment information, and location information (keeping only the most salient attributes as we did so). By identifying the keys of each table, RideID, PaymentID, LocationID, we were then able to join the tables together to compute queries relevant to riders and drivers. For example, we could join the Location and Ride tables together to compute frequent pickup and dropoff locations.

Database

After obtaining the datasets as csv files from the New York City Taxi Database, we created a Python script using Pandas to add an additional column to the files indicating the type of taxi ride (yellow or green). This allowed us to store all yellow and green taxi rides in the same table (Ride). Then, since the size of the datasets was so large, we split each modified csv file into smaller files of around 100,000 rows so we could feed them into SQL Developer more easily. In SQL Developer, we selected only the columns necessary for our tables and inserted all rows into the Amazon AWS RDS instance. We also used SQL Developer to add additional columns for unique RideIDs, PaymentIDs, and LocationIDs, as well as a foreign key column in the Payment table for RideID. There was no other entity resolution necessary, since the ID columns we added each uniquely identified the Payments, Rides, and Locations.

Below are the number of rows in each of our tables.

Table	**NumRows**
Ride	4807697
Location	265
Payment	4807697

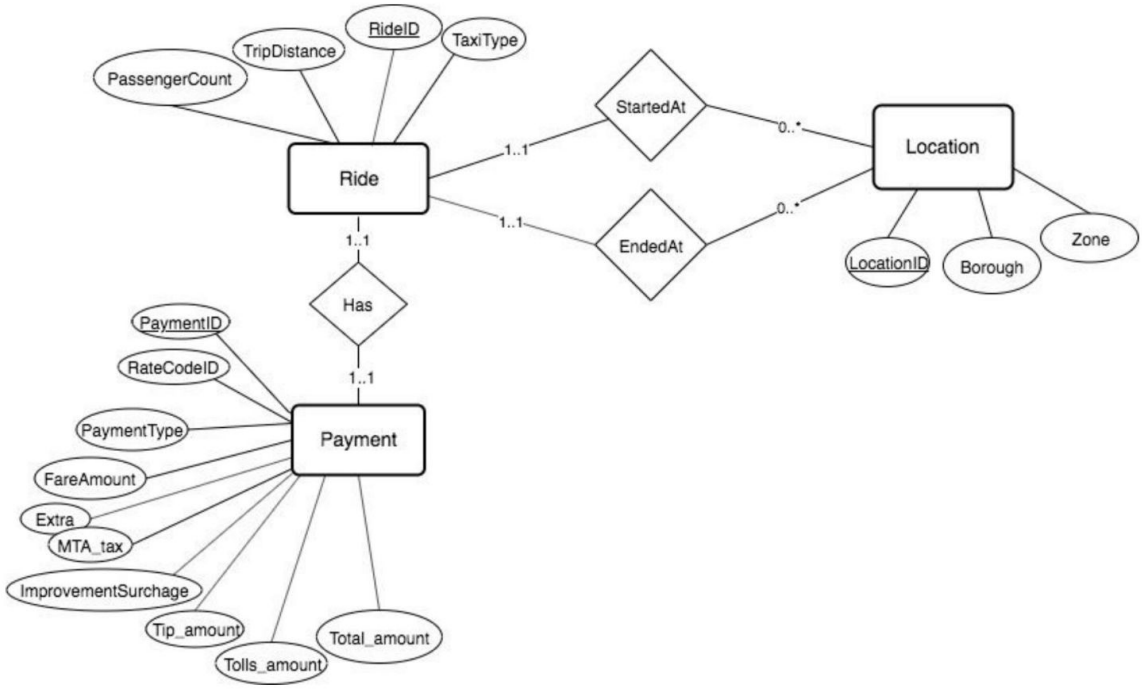
The schemas of our resulting tables are as follows:

Location (LocationID, Borough, Zone, PRIMARY KEY(LocationID))

Payment (PaymentID, RateCodeID, PaymentType, FareAmount, Extra, MTA_tax, ImprovementSurcharge, Tip_amount, Tolls_amount, Total_amount, RideID, PRIMARY KEY(PaymentId), FOREIGN KEY(RideID) REFERENCES Ride(RideID))

Ride (RideID, StartLocation, EndLocation, TripDistance, PassengerCount, TaxiType, PRIMARY KEY (RideID), FOREIGN KEY (StartLocation) REFERENCES Location(LocationID), FOREIGN KEY (EndLocation) REFERENCES Location(LocationID))

An ER diagram displaying the above dependencies is below:



As seen by the schemas above, our relations are BCNF. We can most easily see this with the **Location** table. The key here is LocationID which uniquely determines all of the other dependencies in that relation (including the fact that LocationID → Borough and LocationID → Zone). This same logic applies to the **Ride** and **Payment** tables, where the primary keys are RideID and PaymentID. Thus, the data is ensured to not have any redundancies in the relations.

Description of System Architecture

Below are the various web pages with their respective roles that they accomplish.

Login/Signup Page
If a user is not already logged in, the app will redirect users to the login/signup page, where they can either make a new account or log in with an existing account.

Home Page
The Home Page, which can be reached after logging in or signing up, has two buttons allowing the user to access either driver-side or rider-side tools.

Driver Tools

At the top of the web page, the driver can see three buttons representing links to the three driver side tool pages: (1) Zones with Higher Fares, which gets zones with higher average fares given a particular zone, (2) Popular Destinations, which gets the most popular destinations given a particular zone, and (3) Trip Distances by Zones, which shows the average trip distance for all trips starting from each zone. Each of these links takes the user to a page that allows the user to input their current zone and view their results in a table.

Rider Tools

At the top of the web page, the rider can see two buttons representing links to the two rider side functionalities: (1) Trip Cost Estimator, which shows the expected cost for a trip between two chosen zones, along with a breakdown of the fare if the rider clicks "See Breakdown". The breakdown of the cost is shown in an easy to understand format of a colorful pie chart, and the rider can click on the pie chart to view the various components of the total payment amount. (2) Destination Finder, which shows the zones a rider can travel to given a source zone and a budget. The zones are displayed as geo-markers on an interactive map, and the user can click into any of these destinations to view a Street View that gives them an immersive experience of what the location would look like.

Technologies Used

- Python and Pandas were used to clean the initial data and upload the data.
- We used AWS Oracle RDS for hosting the database.
- Our website was built using AngularJS, ExpressJS, HTML and CSS.
- We used MongoDB for user credentials when logging in.
- We used Crypto and SHA for encryption.
- We used the Google Maps/Street View API and CanvasJS API to dynamically display our data in an interactive way.

Queries

Below, we discuss the functionality of our five most complex queries.

```
"WITH pickup_locations AS (SELECT RideID, DoLocationID, LocationID\
                           FROM (SELECT LocationID FROM Location WHERE Zone = '" + req.params.zone.replace('-', '/') + "') L2\
                           JOIN Ride R ON R.PULocationID=L2.LocationID), \
  payment_amounts AS (SELECT DoLocationID, AVG(Total_Amount) AS avg_total\
                      FROM Payment P JOIN pickup_locations PL ON PL.RideID = P.RideID \
                      WHERE P.Total_Amount <= " + req.params.budget + " AND P.Total_Amount >= " + (req.params.budget - 0.1) +
                      "GROUP BY PL.DoLocationID), \
  all_dests AS (SELECT DISTINCT L.Zone AS potential_zone, P.avg_total\
                FROM payment_amounts P\
                JOIN Location L ON P.DoLocationID = L.LocationID\
                ORDER BY avg_total DESC)\
  SELECT * FROM all_dests\
  WHERE rownum <= 10";
```

The query above depends on a rider's input on their current zone and budget to spend on a taxi ride is. The query outputs possible zones he or she can visit with money within a 10 cent radius of the rider's target budget, to display potential destinations.

```
"WITH average_zone_costs AS \
(SELECT Zone, AVG(P.Fare_Amount) as average_fare_amount \
FROM Location L JOIN Ride R ON L.LocationID = R.PuLocationID JOIN Payment P ON R.RideID = P.RideID\
GROUP BY L.Zone), \
current_zone AS \
(SELECT average_fare_amount as curr_zone_fare FROM average_zone_costs A \
 WHERE A.Zone = '" + zone + "'), \
 other_zone as ( \
  select * from average_zone_costs A where A.Zone <> '" + zone + "' \
 ) \
SELECT Zone as zone FROM other_zone A \
WHERE average_fare_amount >=  \
(SELECT curr_zone_fare FROM current_zone) AND rownum <= 10";
```

The query above depends on a driver's input on their current zone. It then returns 10 neighboring zones that have average fare amounts greater than the average fare amount of the current zone, so that he or she can optimize their route for the near future.

```
"WITH LocationCounts AS \
(SELECT R1.DoLocationID, COUNT(R1.DoLocationID) AS Count \
FROM Ride R1 JOIN Location L1 ON R1.PuLocationID = L1.LocationID\
WHERE L1.Zone = '" + zone + "' \
GROUP BY R1.DoLocationID), \
BusyZones AS (SELECT L.Zone AS zone \
FROM LocationCounts LC JOIN Location L ON LC.DoLocationID = L.LocationID \
ORDER BY LC.Count DESC)\
SELECT * FROM BusyZones WHERE rownum <= 15";
```

The query above depends on a driver's input as to what zone he or she is currently driving in. This is a driver-side query that allows the driver to see where he or she may want to direct future rides upon completion of the current one. The driver inputs his or her current location and the query outputs the top 15 locations the driver should accept rides to such that the destination of the ride has a high chance of getting the driver a new passenger.

```
"SELECT taxi_type, AVG(P.Total_amount) AS average_total_cost \
 FROM Location L1 JOIN Ride R ON L1.LocationID = R.PuLocationID \
 JOIN Location L2 ON L2.LocationID = R.DoLocationID JOIN Payment P ON R.RideID = P.RideID \
 WHERE L1.Zone = '" + fromZone + "' AND L2.Zone = '" + toZone + "' \
 GROUP BY taxi_type \
 ORDER BY taxi_type DESC";
```

This is a driver side query that allows the driver to enter a source and destination zone. The query then returns the average fare amount the driver would earn in one trip between the source and destination zone based on the taxi type.

```
"WITH avg_trip_distances AS ( \
SELECT L.Zone, AVG(R.trip_distance) \
FROM Location L JOIN Ride R ON L.LocationID = R.PuLocationID \
GROUP BY L.Zone\
ORDER BY 2\
) \
SELECT * FROM avg_trip_distances";
```

This is a static query that accepts no input from a user. It displays the average trip distances per zone, ordered by average trip distance ascending. It serves as a simple knowledge-point for drivers, cleanly showing them these key metrics so that they may effectively begin their route.

We also have 3 additional queries used in our application. First, we have a query that finds all zones that are reachable with our data from a particular starting zone. This is used in the Trip Cost Estimator, so that when the user selects a particular starting zone, the destination zones list updates to only include zones that we are able to provide data for.

Second, we have a query that gets all zones from our Location table in alphabetical order. This query is used several times throughout the website; for example, this is how we populate the options that the user may select from in the Driver tools pages.

Lastly, we have a query that finds the estimated breakdown for a particular total payment amount into its constituent parts, including the fare amount, tip amount, tolls amount, improvement surcharge, MTA tax, and extra. This query looks at all payments in the Payment table within a specified range and outputs the average values for the breakdown. We use this in our web application to produce the pie chart after the user clicks "Show Breakdown" in the Trip Cost Estimator.

Performance Evaluation

We used several methods for optimizing our queries, including the creation of temporary tables to store intermediate data, indexing on commonly accessed columns, and pushing projections and filters deeper (before joins). Below is a table showing the times for running our most complicated queries before and after optimization.

Query	Before Optimization (ms)	After Optimization (ms)
Trip Cost Estimator	8417.929	1934.525
Destination Finder	3769.567	817.368
Zones with High Fares	16917.683	7507.466

For the first query (Trip Cost Estimator), we originally wrote a separate query for estimating the cost of the yellow and green taxi types. Instead, we combined these queries into one, where we grouped on the type of taxi in order to more efficiently compute the average cost for both of these types together. This was the most prominent improvement in our runtime, reducing the running time to 3819.851 ms. Then, we added an index on Payment.RideID, which again improved our runtime to 1934.525 ms, since one of the largest JOINs in this query was joining **Ride**.RideID onto **Payment**.RideID.

For the Destination Finder, our original query computed a lot of joins on large tables, like Ride and Payment, and it then filtered on the Total_Amount column in Payment to find all amounts less than the specified budget, returning the top 10 zones that are reachable within the budget. To optimize this query, we pushed many of the projections before the joins, and we added the filtering on the budget amount before we joined the Payment table with the Ride table. Since these tables are so large, this decreased the number of tuples from the Payment table that needed to be joined, which would decrease the runtime of the join significantly. We kept these intermediate results in temporary tables, and then we did the sorting and removing of duplicates at the end, in the last intermediate query. Additionally, we narrowed the range of the Total_Amount that was filtered on. Instead of selecting all tuples with Total_Amount less than the target cost amount, we selected a smaller range of amounts to use. This again decreased the size of the table that needed to be joined with Ride, improving the runtime of the query. Finally, we added an index on **Payment**.RideID, which is used to join with **Ride**.RideID, and this improved our runtime even more. The final runtime of this query was less than 1 second after using all these optimization techniques.

Our last query that needed to be optimized greatly was the query that finds other taxi zones that have a higher average fare amount than the current zone. The initial query took almost 17 seconds to run, which gave us a very slow user experience when the driver input a starting zone. Initially, the query had several joins that were executed multiple times, so we condensed the query using temporary tables so that the extraneous joins were removed. Then, we used the same techniques used to optimize the previous two queries (pushing selects and filters and adding the index on **Payment**.RideID) to get our final runtime down to around 7 seconds.

Technical Challenges

We were fortunate in that the compiled data was already quite well formatted and did not have any outliers or egregious values. Thus, we did not face too many difficulties in terms of pure data manipulation and wrangling, discussed in the *Database* section above. What did become immediately apparent to us upon inspection of the data, though, was that keeping all of it was not an option, as there was simply too much of it. To mitigate this issue, we decided to focus on data just from January 2019. There are a couple key points to address regarding why we felt this was justified. Firstly, even though one month is not entirely representative of data from the entire year (given that one might expect some seasonal variance in the frequency of taxi rides), the relative recency of January 2019 and the large number of rides during that month made its data generalizable. Thus, we felt that we would still be able to provide users with accurate information via our queries. Secondly, we determined that the user experience was one of our

highest priorities in terms of our finished product, so we did not want our queries to run too slowly just because we were working with too much data.

The majority of the technical challenges we faced stemmed from the sheer amount of time our queries initially took. Whether it was because of the relatively large size of our datasets or the initially un-optimized structure of our queries, some of them took almost *~20 seconds* to run initially (of course, there was some variance here depending on user input). The first step to reduce these runtimes was to push selects and filters from our queries as far towards their base relations as possible. Then, we created an index on **Payment**.RideID as this attribute plays a central role in a lot of our queries. Finally, we made our joins as modular as possible by splitting them up. We initially made the mistake of joining all of our tables together and then just selecting from these resulting amassed relations. These steps allowed us to cut our runtimes down by five-fold and provide a quick and easy-to-use user experience.

Extra Credit

1. We made use of Google's Maps API, Street View API, and Geocoder API to create a window for riders to to view potential pickup/dropoff points on an interactive map. Users can also select a particular location on the map that is returned by our database and view it through the Street View to get a better understanding of their query results.
2. We used the CanvasJS API to make pie charts to display fare information in a more user-friendly way. The charts are interactive and allow users to click different portions to see more detail about the query.
3. We added a moving circular animation that displays while queries are loading.
4. We created a login page (and signup page) to ensure only registered users are using the tool -- includes view-based access control so that any user that is not currently logged in is redirected to the home page.
5. User Crypto and SHA encryption to hide password in database. This is a form of double encryption. Using just sha512 results in any password like "abc1234" always being mapped to the same result. Combining sha512 with crypto using a user-specific salt, even if two people have "abc1234" as their password, they are hashed to different mappings, hence making it harder for a malicious user to hack multiple accounts.
6. Used MongoDB (NoSQL database) to store user information, password and the user-specific salt.