Urban Mobility Data Explorer — Technical Report

SECTION 1. Problem Framing and Dataset Analysis

Dataset and context

There are three NYC TLC datasets used in the project. The main is data of yellow trips: each row constitutes one trip and contains pickup time, dropoff time, and distance, fare structure (fare-amount, tip-amount, total-amount, extras, tolls), rate code, payment type, pickup/dropoff location IDs (PULocationID, DO LocationID). A monthly (one in our case in January 2019) data in Parquet or CSV. The second one is the taxi zone lookup which matches each LocationID with a Borough, Zone name and Service Zone in order to give the trip IDs real place names. The third is optional - Geojson or shapefile zones boundaries, of each zone, in the event of a map; we store it in the DB when available.

The concept is to consider the movement of people within the city: at what locations and at what times people take trips, average distances and prices, and how it varies by location and time of day (e.g. rush hour vs off-peak).

Data challenges

- Missing values: we simply kept the rows, in which the following important fields were applied (pickup/dropoff datetime, trip distance, total amount, pu-location id, do-location id). In our month there were in fact no missing columns, the result being no drop there.
- Duplicates: we eliminated rows that had identical duplicate rows on pickup time, dropoff time and both location IDs and total amount. That was only 4 rows in our run.
- Invalid place ID: PULocationID is required to be in the official taxi zone look up, and so is DOLocationID. Anything else we dropped. In our sample, we had no invalid ID.
- Invalid bounds: we have discarded trips with trip-distance of zero or less, or total-amount or fare-amount of less than zero or of pickup of less than or equal to dropoff. That tableted away most of the evident bad--60 793 of our run.

- Outliers: in the case of trip and total amount we applied a 1.5IQR cutoff. Any value that is less than Q1 -1.5xIQR or more than Q3 +1.5xIQR was considered an outlier. We used the sort and quartiles by writing our own code (merge sort, there is no default sort) to complete the assignment. This reduced much of the extreme trips, e.g., 872k trips of trip distance, and 182k of total amount. Many of them are most likely legitimate long or expensive journeys (e.g. to airports), but they would distort the analytics so we excluded them and put all that in the cleaning log.

Inferences in the cleaning process

We assume a row is useful in case we have all of: pickup/dropoff datetime, trip distance, total amount and both locationIn case that all that is missing then we drop the row. The location IDs should be the official look up otherwise we drop. We also assume that trip distance must be non-negative and fares non-negative and dropoff must be post pick up. In the case of the outliers we apply the 1.5x IQR to trip distance and total amount individually, a row can get dropped by either. Derived items: duration is simply the difference between dropoff and pickup, speed is distance over duration (limited), fare per mile and tip per cent are taken out of the cleaned items and we indicate the peak hour as 7-9 and 16-19. All the exclusions are logged in the pipeline (steps and counts per cause) to have the idea of what was removed.

This is one of the surprising details that contributed to design.

The trip distance outlier step eliminated way too many rows than the total amount step (872k vs 182k). Thus a long-tail of really long trips, which are still realistic (airports, out of town), would dominate the numbers. The same 1.5x IQR condition was maintained to both to ensure the dashboard and API revolve around the common city visits. We documented it in this way because in case a person repeats a similar pipeline run they are able to adjust the threshold to higher or lower values or apply it only to a single variable (in case they wish to retain long-range journeys).

SECTION 2 :System Architecture and Design Decisions

The API Architecture of the backend.

We have introduced a Flask-based RESTful API that will be used as an interface between the database and the frontend. We chose flask because of its:
The simple python support of our SQLite database.
Inbuilt testing development server.
Lightweight architecture that is suitable in rapid development.

API DESIGN
We have seven REST endpoints that are grouped by som
e functionality on our backend:

Data Retrieval
'/api/trips' - This will give filtered trip records containing information on the zones.
/api/zones' - Gives all 265 taxi areas in NYC and the boroughs.

Statistics

/api/statistics' - The total metrics of 6.5M trips.
api/statistics/by-borough - Breakdown by Borough.
api/statistics/peak-vs-offpeak - Rush hour analysis.

Analytics
/api/top-routes' - Routes being most popular with custom merge sort.
/api/insights' - The key findings are pre-calculated.

DESIGN DECISIONS

Database Connection Strategy: The client aims to connect via the internet to a database server containing comprehensive data on the company's financial performance.<|human|>Database Connection Strategy: The client intends to access the internet to get to a database server, which contains detailed information on the financial performance of the company.

We created a simulated SQLite database (10k records) at first so that we could build a database in parallel with the database itself. This approach allowed us to :

Architecture the API interfaces on their own.
Smooth transfer to the production database(6.5M records).

Query Optimization
In order to process 6.5 millions records, we:
Used the JOIN operation to combine trips and zone look up in single queries.
Create leverage indexes on location-id and date fields created by Person 1.

RESPONSE FORMAT

All endpoints are returning a JSON of the same format.

'''json
{
"count":,
 "data":[...]
}
'''
This standardization made it easy to integrate the frontend and error handling.

TRADEOFFS WE DID

SQLite vs PostgreSQL :
We selected SQLite due to his ease and portability. PostgreSQL would be more appropriate to the deployment, however, SQLite removed the problem of setting up the server and allowed the entire database to be version managed which was a nice plus.

Real-time vs Pre-computed Statistics: /api/insights endpoint pre-computes the most common metrics. This replaces the memory with speed that the dashboard takes less than 2 seconds to load instead of around 30 seconds.

CORS Set up: We have had CORS turned on to all the origins in the development to allow frontend testing. This would be limited to certain areas of production in order to be secure.

SECTION 3: ALGORITHMIC LOGIC

Problem Statement

The top-routes endpoint API should prioritize pairs of pickup-dropoff routes by.
popularity (number of trips) in thousands of possible combinations. Using
The built-in sorted () or.sort by Python breaches the assignment.
manual implementation of algorithms.

Solution: Merge Sort

We have used merge sort in its purest format to sort routes in the descending order by.
trip count. Merge sort was selected since:
O(n log n) time assured when large dataset is involved.
Stable sort maintains order in equally-counted routes.
Divide-and-conquer algorithm is memory-efficient to our problem.

IMPLEMENTATION

Pseudo Code

```
fuction merge_sort(routes, key, reverse):
    if length(routes) ≤ 1:
        return routes

    mid ← length(routes) / 2
    left ← merge_sort(routes[0:mid], key, reverse)
    right ← merge_sort(routes[mid:end], key, reverse)
    return merge(left, right, key, reverse)

function merge(left, right, key, reverse):
    result ← empty array
    i ← 0, j ← 0

    while i < length(left) AND j < length(right):
        if reverse = true:
            if left[i][key] ≥ right[j][key]:
                append left[i] to result
                i ← i + 1
            else:
                append right[j] to result
                j ← j + 1
        else:
            if left[i][key] ≤ right[j][key]:
                append left[i] to result
                i ← i + 1
            else:
                append right[j] to result
                j ← j + 1

    append remaining elements from left and right to result
    return result
```

Actual Code (Python):

```python
def merge_sort(routes, key='trip_count', reverse=True):
    if len(routes) <= 1:
        return routes

    mid = len(routes) // 2
    left = merge_sort(routes[:mid], key, reverse)
    right = merge_sort(routes[mid:], key, reverse)

    return merge(left, right, key, reverse)

def merge(left, right, key, reverse):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if reverse:
            if left[i][key] >= right[j][key]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1
        else:
            if left[i][key] <= right[j][key]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```
‘‘‘

<u>Complexity Analysis</u>

Time Complexity: O(n log n)
Dividing: The array is divided into one half and the other half, recursively.

in the recursion tree
Merging: In every level, we merge the n elements in all the subarrays.
Total: O(n) operations -O(log n) levels = O(n log n)

Space Complexity: O(n)
During merge operation, creates temporary arrays: O(n).
Depth of recursive call stack: O(log n)
On average controlled by array storage: O(n).

Performance in Practice
In our case of 2,500 unique route pairs in our data:
Sorting time: <50ms
In comparison to possible O(n 2) bubble sort: approximately 3000 times faster.
Supports API real-time interactions.

All combinations of routes and returns are successfully processed with the help of this implementation.
the most popular N routes, running the Most Popular Routes.
representation at the front end dashboard.

SECTION 4: INSIGHTS

Learning Point 2: NYC Traffic Congestion Effect.

The /api/statistics/by-borough endpoint was used to query average speed by borough:

```
 SELECT
 Z.borough,
 AVG(t.speedmph) as avgspeed
 FROM trips t
 SELECT taxizones z.locationid = t.pulocationid
GROUP BY z.borough
```

Finding:

The average speed of Manhattan taxis stands at 10.16 mph when compared to Queens which has an average speed of 15.66 mph.
-- a 54 percent difference with the same distance of the trip.

Insert scatter diagram of results of avg speed by borough.

Interpretation:
The congestion of the transport in Manhattan is very extreme resulting to serious implications to the city mobility:
The greater the distance covered by the trip, the greater the operation cost of the taxi companies.
The time based fares are charged at a higher price.
Wasting time in traffic jam enhances carbon emission.
The information would help in policy making on the areas of congestion pricing or.
motivating travelling off peak time.

The fact that route 52 percent of the total number of trips take place through Manhattan (96 out of 100 trips) (6.2M out of 6.5M) is due to the fact that it is the kind of place where 96 percent of all trips are made.
the slowest borough means that the demand is made instead by the need of the destination.
than efficiency it must be improved to the public transit, or
other means of mobility made available in the borough.

SECTION 5: Reflection

Technical Issues - Back Office Development.
Problem 1: Time of Database Integration.
Day 1 at the evening, Person 1 had failed to get his database online in time, which implied a potential.
blocker. We minimized this by developing a faked SQLite database.
The development and representation information can be built parallel to each other. Only a transition was needed.
renewing table association strings and column titles.

Problem 2: Query Performance.
The first time queries on 6.5M records took 20-30 seconds. We optimized by:
Being on the right indexes on the join columns.
Being in possession of the already calculated statistics.
Combining sets of results with default LIMIT.

Challenge 3: Real-time Sorting
It was initially slow to rank thousands of route combinations with each API call.
Specialized writing of merge sort actually did better than generic Python.
classifying based on optimization of our case.

Future Improvements

Backend Enhancements:
Caching Layer: Introduction of Redis of the commonly accessed statistics.
Pagination: Add relevant pagination of large result sets.
Authentication: Sign production APIs using the JWT tokens.
Rate Limiting: Throttling of abuse.
Database Migration: Migrate to postgreSQL, to support concurrency in a better way.
Versioning of API: Add prefix /v1/ to ensure future changes of API can be performed without.
breaking existing clients

GRAPHIC: System Architecture Diagram (All 3 Design as a System)

This will be required to be created between you and your team members. Here's what it should show:

```
+---------------------------------------------------------+
                    FRONTEND
  (Person 3 - HTML/CSS/JavaScript)
   Dashboard UI
   Charts (Chart.js)
   Filters & Controls
```

```
+-------------------+-------------------------------------+
                  HTTP Requests
                  (fetch API)


+---------------------------------------------------------+
                 BACKEND API
  (Person 2 - Flask/Python)
   REST Endpoints (7 routes)
   Custom Merge Sort Algorithm
   Query Logic & Filtering
+-------------------+-------------------------------------+
                  SQL Queries
                  (SQLite)


+---------------------------------------------------------+
                  DATABASE
  (Person 1 - SQLite)
   trips table (6.5M records)
   taxizones table (265 zones)
   The features which are inferred (speed, farepermile, etc.).
   Locationid, datetime Primary key.
+---------------------------------------------------------+


                  Data Pipeline


+---------------------------------------------------------+
               RAW DATA SOURCES
   yellowtripdata2019-01.csv (7.6M records).
   taxizonelookup.csv
  * taxi_zones.geojson
+---------------------------------------------------------+
```