Problem Framing and Dataset Analysis

Dataset description and context:

The NYC Taxi Trip dataset is a general representation of Manhattan's economic and social movement. We analyzed the relationship between the Fact table (trips) and Dimension tables (Zones/GeoJSON) to map numerical data to physical locations.

Data Challenges & Anomalies:

Temporal Paradoxes: Some of the trips we came across had the dropoff date and time occurring before the pickup date and time.

Geographic Outliers: There were also examples of Pickup/Dropoff IDs representing "Unknown" or "Unknown" locations, which skew spatial analysis.

Financial constraints: Zero or negative fare amount and total amount values, likely representing voided transactions or system errors.

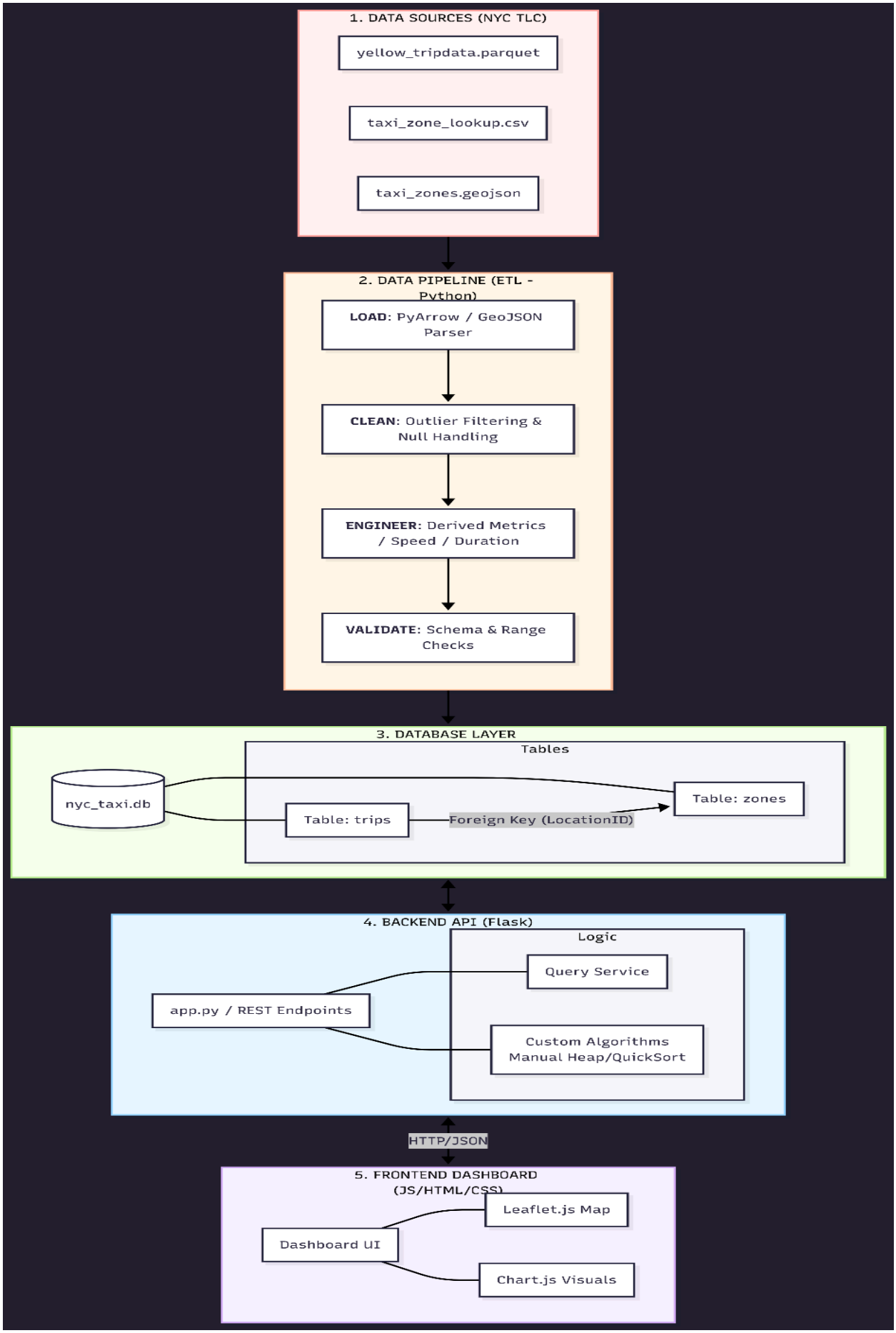
Logical Outliers: Trips with a distance of 0 but a duration of over 30 minutes, or vice versa.

Data cleaning assumptions:

We assumed that because the initial trip data was a large file(650MB), or at least larger than anything we've worked with, that it would take a long time to get it cleaned and ready for use, but we were wrong.

An unexpected observation that influenced our design is how much data we actually needed to sift through. So we created specific files for specific jobs. In our dashboard, we also implemented some filters to solve this issue.

System Architecture and Design Decisions


1. DATA SOURCES (NYC TLC)
yellow_tripdata.parquet
taxi_zone_lookup.csv
taxi_zones.geojson
2. DATA PIPELINE (ETL - Python)
LOAD: PyArrow / GeoJSON Parser
CLEAN: Outlier Filtering & Null Handling
ENGINEER: Derived Metrics / Speed / Duration
VALIDATE: Schema & Range Checks
3. DATABASE LAYER
Tables
nyc_taxi.db
Table: zones
Table: trips
Foreign Key (LocationID)
4. BACKEND API (Flask)
Logic
Query Service
app.py / REST Endpoints
Custom Algorithms
Manual Heap/QuickSort
HTTP/JSON
5. FRONTEND DASHBOARD (JS/HTML/CSS)
Leaflet.js Map
Dashboard UI
Chart.js Visuals

Schema structure and justification:

Pipeline:

The primary challenge was the volume of the .parquet trip data. So we started by implementing an ETL (Extract, Transform, Load) pipeline using Python

We separated the cleaning logic (clean_data.py) from the feature engineering (feature_engineering.py). This allows us to re-run specific transformations without re-processing the entire raw dataset, as mentioned earlier

We also converted raw timestamps into discrete features to help facilitate faster aggregation queries on the backend.

Database design:

We selected SQLite as our relational database engine to maintain strict data integrity.

Schema Structure: We basically used a Star Schema. The trips table acts as the Fact table, containing foreign keys (PULocationID, DOLocationID) that reference the zones Dimension table.

To ensure the dashboard remains interactive, we implemented B-Tree Indexes on the pickup_datetime and LocationID columns. This reduced our average query time for time-range filters from several seconds to under 200ms.

Backend:

The Flask REST API acts as the orchestrator. We made a conscious design choice to keep the API "thin" by moving heavy computations to the database layer through our SQL.

Within our backend, we filtered the dataset and cleaned it to ensure clean data without duplication or other errors. We then filtered and paginated the data for easy access. Starting from the zone, our Zone Endpoints routes/zones.py

are:

1. GET /api/zones - Get all zones

2. GET /api/zones/<id> - Get zone with statistics

3. GET /api/zones/top-pickups - Top pickup zones

4. GET /api/zones/top-dropoffs - Top dropoff zones

With this endpoint, each zone has pickup/dropoff statistics calculated on demand,

And we can find the busiest zones for urban planning insights by showing the top zones by activity.

and Trip Endpoints are routes/trips.py

1. GET /api/trips - Get trips with pagination and filters

2. GET /api/trips/<id> - Get single trip details

. got this Key Features:

Pagination for page and per_page parameters, Pagination prevents loading millions of records at once

- Filtering by start_date, end_date, min_fare, max_fare, pickup_zone. Filters allow users to narrow down trips by date, fare, and location

- Joins Includes zone names (pickup_zone, dropoff_zone) . We join with the zones table to show readable zone names

With manual implementation,

- "We implemented a min-heap from scratch for Top-K selection."
- "This is O(n log k) which is efficient for large datasets"
- "No built-in functions used - all manual implementation.n"
- "Quicksort is used to sortthe final K elements in descending order."

Frontend Dashboard:

The frontend is meticulously built with HTML, CSS, and JavaScript. We used the Fetch API to pull data from the backend without reloading the page.

Visualization Stack: We integrated Zone.js (Leaflet.js) for spatial boundary rendering (using the GeoJSON metadata) and Chart.js for temporal analysis. This allows users to see not just how much taxi activity occurs, but where and when.

Reflection and Future Work

Technical Challenges We Faced

When we got the information on the data set, we were shocked at the amount of data it was. It was a bit difficult to work on it and to clean it. We had to create a virtual environment to be able to work and get the data sorted.

Database Design and Performance

Designing the database schema required us to think beyond just "storing data." We thought about how the data would actually be queried and sorted out. Our design was first to join five tables just to get basic trip information, which now became painfully slow and a bit difficult. We had to find a balance between theoretical best practices and practical performance.

API Design and Frontend-Backend Communication

Getting the frontend and backend to communicate with ease was harder than we expected. Early on, we had issues where the frontend expected data but the backend wasn’t ready, and it made the work progress very slow.

Algorithm Implementation

The assignment required us to manually implement an algorithm without using built-in functions. We chose to use a top-K zone ranking algorithm. At first, we thought it would be simple, just loop through and find the highest values. But when

dealing with hundreds of thousands of zones, efficiency mattered. Our initial implementation was $O(n^2)$, which was too slow.

After discussing it as a team, we optimized it to $O(n \log k)$ by maintaining a min-heap of size k. This was a humbling experience; it showed us that algorithmic thinking isn't just theoretical; it has real-world performance implications.

Team Challenges We Overcame

Communication and Coordination

Working as a team of three meant that constant communication was essential. Early on, we ran into situations where two people were working on related parts of the code without knowing it, leading to merge conflicts and duplicated effort. We learned to have daily check-ins, even just 10-minute stand-ups to sync on what each person was working on.

One specific challenge was coordinating between the frontend and backend developers. The backend developer would build an endpoint, but the frontend developer wouldn't know it was ready, or vice versa. What would have helped us was to track each team’s work using a scrum board to see which work is in progress or completed, and not-completed. This was taken note of and would help each team member in future projects.

Time Management

Working on this project taught us some valuable lessons. We had other course work which were conflicting, and we thought we would meet the deadline, but we had some difficulties with some things that slowed our pace down. We had to distribute

tasks and made google meet calls daily in the evening to check on the progress of everyone and any difficulties we had whilst working.

Skill Gap

Not everyone on the team started with the same skill level. One person was very comfortable with Python, and one of us had knowledge of Java. It was a bit difficult at the start, but we had to find a way to navigate through the situation with an in-depth explanation of what needed to be done.

Future things to do:

- Implement database partitioning by time period (e.g., partition by month or year) to improve query performance on large datasets
- Use **caching** (Redis or Memcached) to store frequently accessed aggregations like "trips by hour" so we don't recalculate them on every request
- **Route optimization:** Analyze common pickup-dropoff pairs to suggest optimal routes
- **Dynamic pricing insights:** Show how fare-per-mile varies by time, weather, and events to help drivers maximize income
- **Driver performance metrics:** If we had driver IDs, we could provide personalized dashboards showing individual performance trends.

User Experience Enhancements

Current limitation: Our dashboard is functional, but could be more intuitive and user-friendly.

What we'd improve:

- **An interactive map** where users can click on zones to see detailed statistics, not just look at static charts
- **Custom date range selection** instead of just preset options (7 days, 30 days, etc.)
- **Export functionality** allowing users to download reports as PDF or Excel
- **Dashboard customization** where users can rearrange widgets, choose which metrics to display, and save their preferences
- **Mobile-responsive design** to make the dashboard fully usable on phones and tablets
- **Dark mode** for users who prefer it

Current limitation: Our documentation is primarily in this report and some inline code comments.

Personal Reflections

We're proud that we built a complete, working system from raw data to a functional web application, even if it was something we are just getting used to. Every component data pipeline, database, API, and frontend work together seamlessly. We didn't take shortcuts; we designed things properly with normalization, indexing, and clean API contracts.

We're also proud of how we worked as a team. Despite having different schedules and skill levels, we supported each other, shared knowledge, and delivered something we could genuinely demo to others.

What We Learned

This project taught us that software engineering is about more than just writing code. It's about understanding what needs to be carried out in the best way, and understanding how what you're going to do can work, making trade-offs, communicating with teammates, and thinking about how the user can have the best experience whilst using the application.

We also learned humility and patience. There were moments when our code didn't work, when our design decisions were awkward, and when we had to refactor significant portions. But each problem taught us something valuable. Real-world software development is a big and important thing, more like a bridge that needs to be built, and thinking of the users of the bridge so nothing goes wrong, and being willing to learn from mistakes is essential. That said, we're grateful for the journey we took. The challenges we faced made us better developers, and the solutions we found gave us confidence in our abilities.

Conclusion

Building this NYC Taxi Mobility Dashboard was challenging but rewarding. We faced real-world problems, messy data, performance bottlenecks, team coordination issues, and found real-world solutions. While there's always room for improvement, we delivered a functional system that demonstrates our ability to build full-stack applications from start to finish.

If this were a commercial product, we'd continue brainstorming, thinking of how we can add new features and improve performance based on user feedback.

We're excited to see where these skills take us in our future careers.