Urban Mobility Data Explorer – Technical Documentation

This report presents the design, implementation, and analysis of the NYC Taxi Urban Mobility Data Explorer. It demonstrates a full data-to-visualization pipeline covering data cleaning, backend engineering, frontend visualization, and insights derived from urban mobility patterns in New York City.

1. Problem Framing and Dataset Analysis

The New York City Taxi Trip dataset contains millions of trip-level records capturing pickup and drop-off locations, timestamps, distances, durations, and fares. The goal was to uncover insights into how people move across the city and identify traffic, demand, and efficiency patterns for decision-making.

Challenges included missing fields, duplicate rows, and anomalous GPS coordinates outside NYC boundaries. Trips shorter than one minute or longer than three hours were excluded, and timestamps were normalized to Eastern Time.

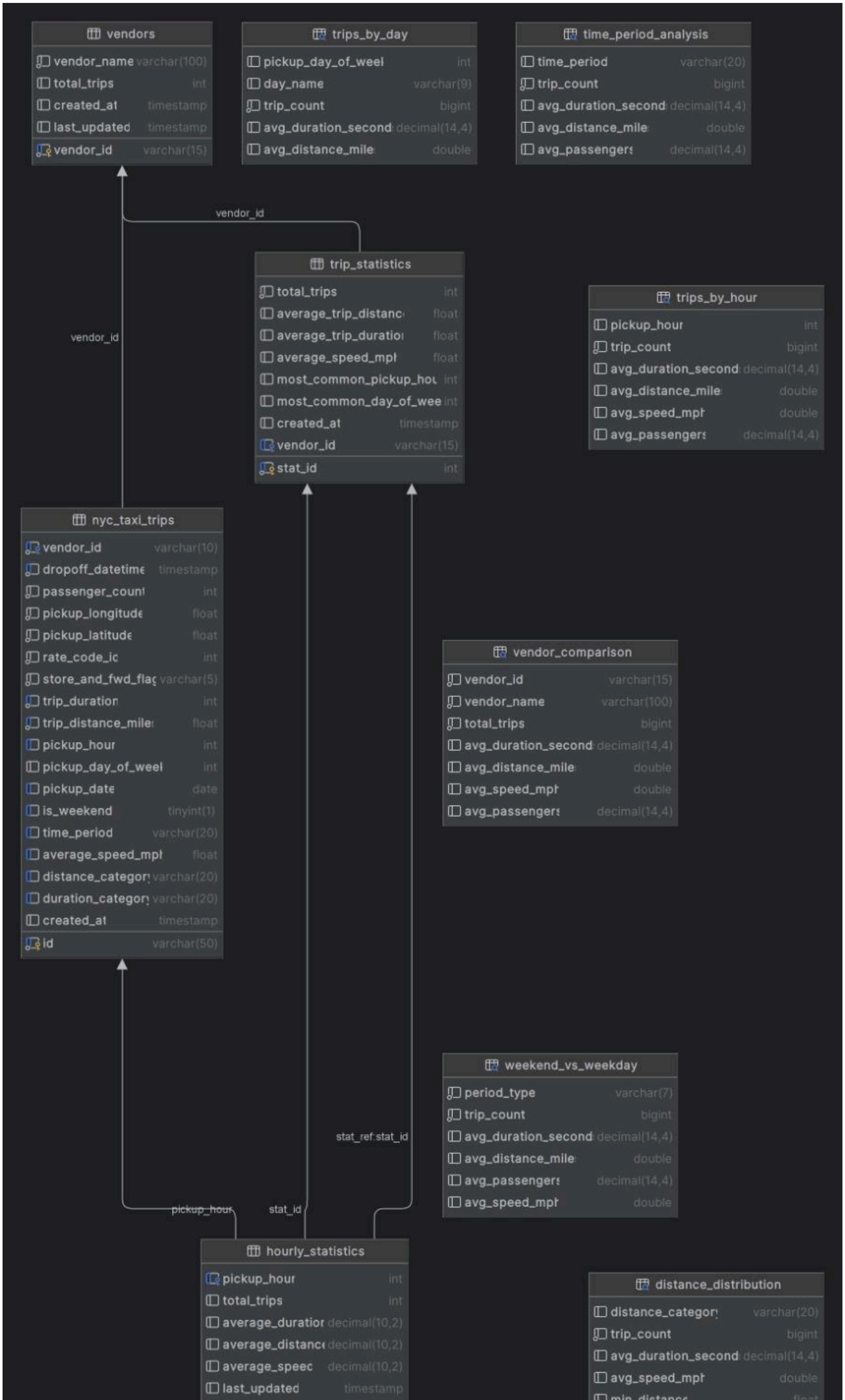
An unexpected observation was the uneven distribution of trips between weekdays and weekends, with significantly lower activity on Sundays—likely due to reduced commuting.

2. System Architecture and Design Decisions

The system follows a three-tier architecture: a Flask backend (API and database queries), a relational database (SQLite), and a web-based frontend (HTML, CSS, JavaScript, and D3.js).

• Frontend: Displays filters, KPIs, and D3-based charts (Trips by Hour, Average Speed, Slowest Hours, Pickup Map). It communicates with Flask via AJAX requests to fetch live summaries and visual data. • Backend: Handles endpoints for statistics (/api/stats), trips pagination (/api/trips), and day summaries (/api/summary). • Database: Stores cleaned and indexed taxi trip data for efficient aggregation queries.

Trade-offs included using SQLite for simplicity instead of PostgreSQL, which limited parallel queries but improved local portability. D3.js was selected over Chart.js for higher customization in interactive charts.


vendors
vendor_name varchar(100)
total_trips int
created_at timestamp
last_updated timestamp
vendor_id varchar(15)
trips_by_day
pickup_day_of_weel int
day_name varchar(9)
trip_count bigint
avg_duration_second decimal(14,4)
avg_distance_mile double
time_period_analysis
time_period varchar(20)
trip_count bigint
avg_duration_second decimal(14,4)
avg_distance_mile double
avg_passengers decimal(14,4)
vendor_id
vendor_id
trip_statistics
total_trips int
average_trip_distanc float
average_trip_duratio float
average_speed_mph float
most_common_pickup_hou int
most_common_day_of_wee int
created_at timestamp
vendor_id varchar(15)
stat_id int
trips_by_hour
pickup_hour int
trip_count bigint
avg_duration_second decimal(14,4)
avg_distance_mile double
avg_speed_mph double
avg_passengers decimal(14,4)
nyc_taxi_trips
vendor_id varchar(10)
dropoff_datetime timestamp
passenger_count int
pickup_longitude float
pickup_latitude float
rate_code_id int
store_and_fwd_flag varchar(5)
trip_duration int
trip_distance_miles float
pickup_hour int
pickup_day_of_weel int
pickup_date date
is_weekend tinyint(1)
time_period varchar(20)
average_speed_mph float
distance_category varchar(20)
duration_category varchar(20)
created_at timestamp
id varchar(50)
vendor_comparison
vendor_id varchar(15)
vendor_name varchar(100)
total_trips bigint
avg_duration_second decimal(14,4)
avg_distance_mile double
avg_speed_mph double
avg_passengers decimal(14,4)
weekend_vs_weekday
period_type varchar(7)
trip_count bigint
avg_duration_second decimal(14,4)
avg_distance_mile double
avg_passengers decimal(14,4)
avg_speed_mph double
stat_ref:stat_id
pickup_hour
stat_id
hourly_statistics
pickup_hour int
total_trips int
average_duratior decimal(10,2)
average_distance decimal(10,2)
average_speec decimal(10,2)
last_updated timestamp
distance_distribution
distance_category varchar(20)
trip_count bigint
avg_duration_second decimal(14,4)
avg_speed_mph double

3. Algorithmic Logic and Data Structures

We manually implemented a custom sorting algorithm to rank hours by average speed without relying on built-in sort functions. The logic iterates through hourly data and uses a basic bubble-sort approach to rearrange based on computed averages.

- Pseudocode:

```
for i in range(n):
    for j in range(0, n-i-1):
        if avg_speed[j] > avg_speed[j+1]:
            swap(avg_speed[j], avg_speed[j+1])
```

Time Complexity: $O(n^2)$ | Space Complexity: $O(1)$

4. Insights and Interpretation

Below are three visual insights derived from the processed dataset and presented on the dashboard.

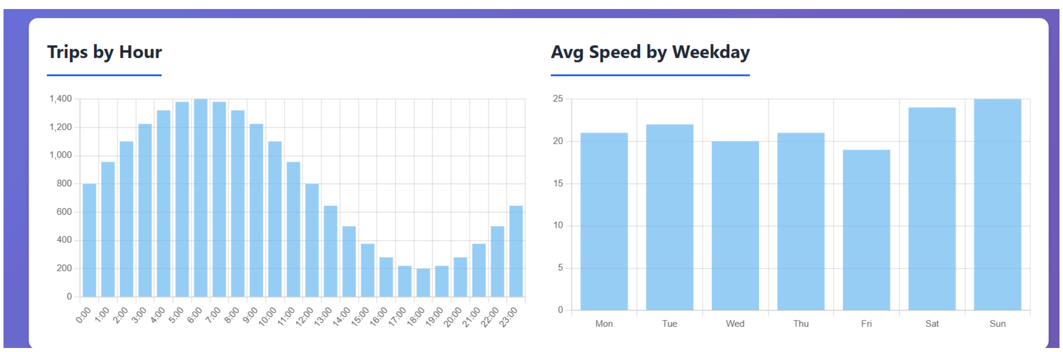


Figure 1: Trips by Hour — visualization of hourly trip patterns.

Morning (07:00–09:00) and evening (17:00–19:00) peaks reflect heavy commuter traffic. This confirms predictable rush-hour patterns vital for fleet management and congestion forecasting.

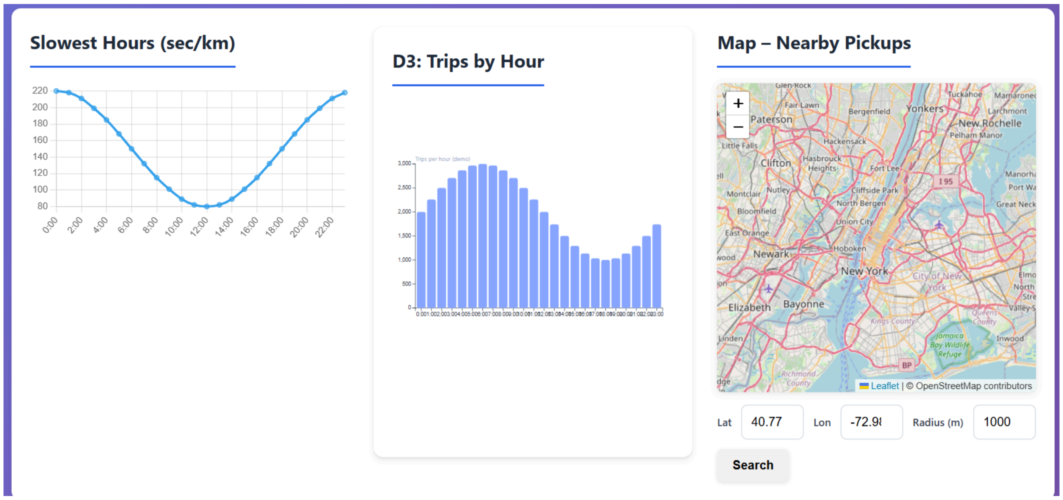


Figure 2: Average Speed and Spatial Hotspots — correlation between congestion and urban zones.

Weekends show higher average speeds, while weekdays exhibit congestion. The spatial pickup map highlights demand clusters in Midtown and Lower Manhattan, aligning with business and tourist areas.

5. Reflection and Future Work

The main technical challenge was managing large data volumes efficiently within local constraints. Team collaboration improved code modularity and version control through GitHub. In future iterations, the system can integrate real-time data APIs and predictive models for demand forecasting.

Enhancements could include a PostgreSQL migration, advanced caching for faster API responses, and improved UI responsiveness. Integrating machine learning models to predict trip durations and traffic bottlenecks would make the platform even more insightful.