1. Problem Facing and Dataset Analysis

This dataset(train.csv) contains detailed records of taxi trips from the New York City Trip dataset. It includes over 1.45 million rows and 11 columns, each describing attributes of a single trip. Trip attributes include timestamps, distances, durations, pickup/dropoff locations, and other metadata.

Description

- ID: Unique identifier for each trip.
- vendor_id: A code indicating the provider associated with the trip record.
- pickup_datetime: Date and time when the meter was engaged.
- dropoff_datetime: date and time when the meter was disengaged.
- passenger_count: the number of passengers in the vehicle (driver-entered value).
- pickup_longitude: the longitude where the meter was engaged.
- pickup_latitude: the latitude where the meter was engaged.
- dropoff_longitude: the longitude where the meter was disengaged.
- dropoff_latitude: the latitude where the meter was disengaged.
- store_and_fwd_flag: Whether the trip data was stored and forwarded later (Y for yes, N for no), often due to temporary connection loss.
- trip_duration: duration of the trip in seconds.

It aims to predict the total trip duration based on temporal, spatial, and vendor-related features.

It can also be analyzed to:

- Evaluate traffic congestion patterns.
- Estimate average trip speeds and distances.
- Analyze urban mobility and demand trends.
- Support data-driven transportation planning or ride optimization.

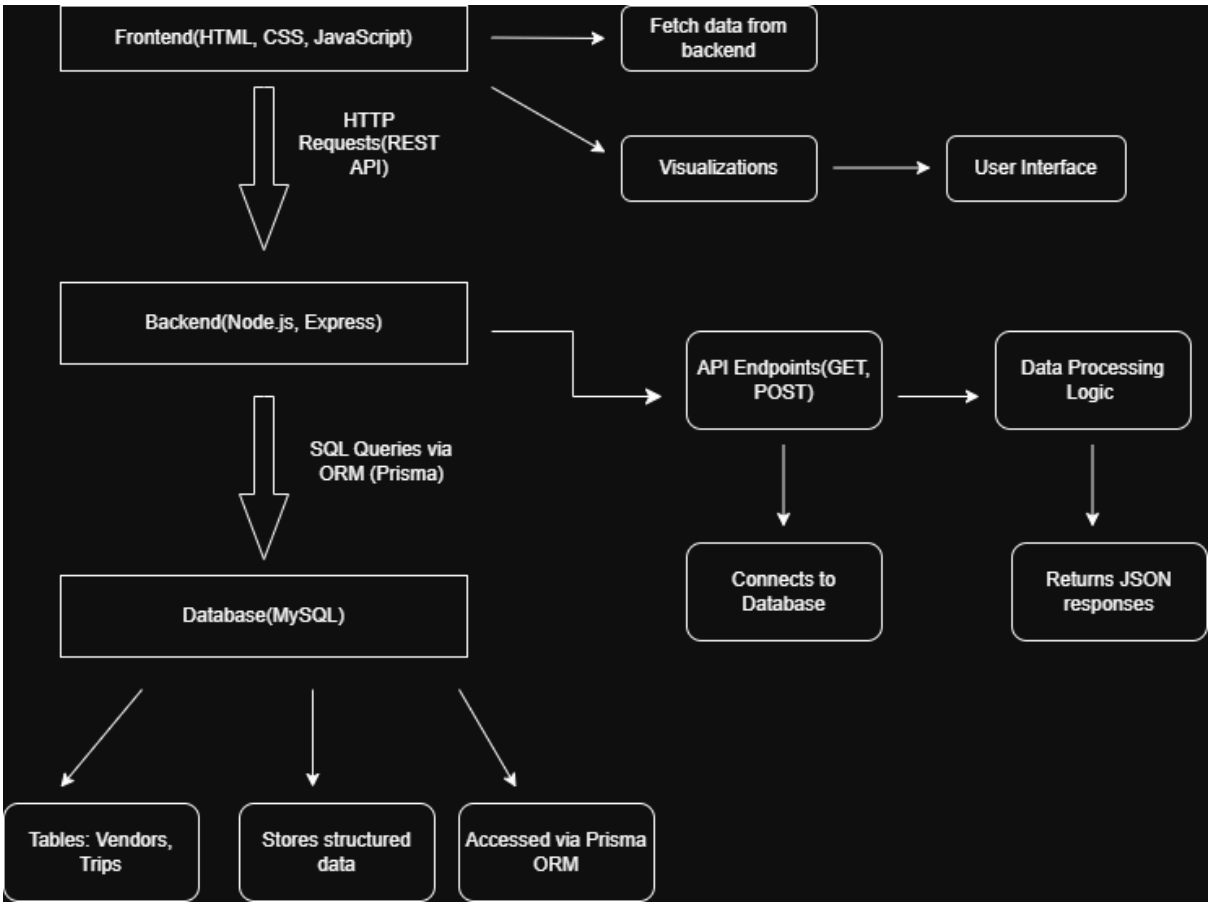
Data Challenges

- Invalid pickup and dropoff coordinates: These points go beyond the normal New York City geographic bounds.
- Passenger count anomaly: Some records have 0 passengers, which is not realistic for a taxi trip.

Assumptions made during data cleaning

- Trip duration was assumed to fall within a reasonable range for urban taxi rides. Specifically, any trip duration under one minute (60 seconds) was deemed unrealistic because it is unlikely that a trip would start and end in such a short amount of time.
- It was assumed that at least one passenger and no more than six passengers were riding per trip. A trip cannot happen without a passenger, and the highest number of passengers in most New York City taxis is six.

2. System Architecture and Design Decisions



Stack and Schema Structure Justification

- **Frontend**: The frontend of the system is developed using HTML, CSS, and JavaScript, which form the foundation of modern web development . The stack was chosen because it is lightweight, runs in the browser, and does not require any additional tools or build steps. Additionally, it enables the effortless integration of data visualization libraries like Chart.js.

- **Backend**: We built our backend with Node.js and Express.js to support a functional and efficient environment for handling API requests and server-side operations. We chose Node.js because it's designed to handle multiple requests simultaneously in large data applications like mobility tracking systems. Express.js was used to define RESTful API routes and manage requests.

- **Database(MySQL)**: MySQL was chosen as a primary database tool because it is robust, efficient, and scalable enough to handle structured and relational data. In the Mobility Data Explorer tool, MySQL is suitable because trip and vendor data are relational (each trip has a respective vendor).

- **ORM(Prisma)**: Prisma was chosen to simplify the data management and database maintenance. Prisma automatically generates functions to access this data, eliminating the need for raw SQL queries (which are prone to error and can expose our app to security threats like SQL injection).

Custom Algorithm: QuickSort for Trip Ranking

Real-world problem: Need to efficiently rank and sort trips by multiple criteria (distance, speed, duration) to identify:

- Longest/shortest trips
- Fastest/slowest trips
- Performance outliers

Why Custom Implementation?

- JavaScript's built-in `.sort()` is a black box
- Need to understand and control sorting behavior
- Demonstrate algorithmic thinking
- Track performance metrics (comparisons, swaps)

Algorithm: QuickSort

Overview

QuickSort is a divide-and-conquer sorting algorithm that:

1. Selects a 'pivot' element
2. Partitions the array into elements less than and greater than the pivot
3. Recursively sorts the sub-arrays

Why QuickSort?

- Efficient: O(n log n) average case
- In-place: O(log n) space complexity
- Practical: Widely used in production systems
- Demonstrable: Clear algorithmic logic

Pseudo-code

```
FUNCTION quickSort(trips, sortBy, order):
    IF length(trips) <= 1:
        RETURN trips

    RETURN quickSortRecursive(trips, 0, length-1, sortBy, order)
END FUNCTION

FUNCTION quickSortRecursive(trips, low, high, sortBy, order):
    IF low < high:
        pivotIndex = partition(trips, low, high, sortBy, order)
```

```
        quickSortRecursive(trips, low, pivotIndex-1, sortBy, order)
        quickSortRecursive(trips, pivotIndex+1, high, sortBy, order)

    RETURN trips
END FUNCTION

FUNCTION partition(trips, low, high, sortBy, order):
    pivot = getValue(trips[high], sortBy)
    i = low - 1

    FOR j = low TO high-1:
        currentValue = getValue(trips[j], sortBy)

        IF (order == 'desc' AND currentValue > pivot) OR
           (order == 'asc' AND currentValue < pivot):
            i = i + 1
            swap(trips[i], trips[j])

    swap(trips[i+1], trips[high])
    RETURN i + 1
END FUNCTION

FUNCTION getValue(trip, sortBy):
    SWITCH sortBy:
        CASE 'distance': RETURN trip.analytics.tripDistanceKm
        CASE 'speed': RETURN trip.analytics.tripSpeedKmh
        CASE 'duration': RETURN trip.tripDuration
        CASE 'fare': RETURN trip.fareAmount
    END SWITCH
END FUNCTION
```

Implementation Details

Key Features

1. **Generic Sorting:** Works with multiple criteria
    - Distance (km)
    - Speed (km/h)
    - Duration (seconds)
    - Fare (if available)

2. **Bi-directional**: Supports ascending and descending order.
3. **Performance Tracking**: Counts comparisons and swaps.
4. **Safe Extraction:** Uses optional chaining for nested properties.

Insights and Interpretation

1. **Peak Hours**

Using the trip data, we grouped the number of trips by the **hour of the day** based on the `pickup_datetime` field.

SQL Query:

```
SELECT
  HOUR(pickup_datetime) AS hour_of_day,
  COUNT(*) AS total_trips
FROM trips
GROUP BY hour_of_day
ORDER BY hour_of_day;
```

Interpretation

This is important to urban planners and operators of the transport systems because it helps them know the exact times when roads are likely to experience increased pressure and congestion.

2. **Relationship Between Trip Distance and Speed**

We examined the correlation between **trip distance** and **average speed** to understand mobility efficiency and possible congestion patterns.

SQL Query:

```
SELECT
  distance,
  speed
FROM trips
WHERE speed IS NOT NULL AND distance IS NOT NULL;
```

Interpretation

This correlation is useful for identifying locations with low traffic speed.
It also supports evidence-based decisions regarding traffic management and road design

3. **Average Trip Distance per Vendor**

We calculated the **average distance** covered by trips for each vendor to assess performance differences.

SQL Query:

```
SELECT
  v.name AS vendor_name,
  AVG(t.distance) AS avg_distance_km
FROM trips t
```

```
JOIN vendors v ON t.vendor_id = v.vendor_id
GROUP BY v.name;
```

Interpretation

In the context of urban mobility, it can help passengers decide which vendor is more suitable for short-term or long-term transport operations.

Reflection and Future Work

Technical Challenges

1. **Backend to Frontend Connection:**
   - Connecting the frontend JavaScript to backend API endpoints.
   - Fixed by ensuring the correct API base URL and restructuring the JS code.
2. **Server Connection Issues:**
   - The server initially failed to connect to my MySQL due to an incorrect DATABASE_URL.
   - Fixed by properly configuring the .env file and making sure the database credentials are correct and accessible.
3. **Prisma ORM Issues:**
   - Schema synchronization and migration errors.
   - Fixed by running the schema.prisma correctly and regenerating the Prisma client.

Team Challenges

1. **Coordination**
   - Due to the lack of integration between the front end and back end, there are overlaps in team efforts.
   - Solved by better use of GitHub for version control and improved communication.

2. **Debugging Collaboration**
   - Multiple members had to collaborate to troubleshoot the database and API errors.
   - Solved through teamwork and cooperation improved as we shared the configuration fixes.

Suggested Technical Improvements

1. Implement secure login (e.g., JWT-based) to allow different access levels for users, vendors, or administrators.
2. Use predictive models to forecast peak hours, demand patterns, or traffic congestion.
3. Deploy to AWS cloud and use a managed MySQL service for reliability and scalability.

4. Add security and encryption to protect sensitive mobility data.