Delivery Fare Estimation - Design Document

Project Overview

This project aims to build a high-performance, space- and time-efficient Golang program to estimate delivery fares based on GPS data logs. The program filters out invalid GPS points, calculates distances using the Haversine formula, and estimates the fare for each delivery. It produces a CSV file with the delivery ID and the corresponding fare.

Key requirements include:

- Filtering invalid GPS data points (based on speed)
- Fare calculation based on time of day, speed, distance, and idle time
- Handling large datasets (several gigabytes)
- Implementing concurrency for high performance
- Ensuring thread-safe file writing

Design Approach

1. Data Ingestion

The program reads data from a CSV file that contains GPS coordinates of delivery points in the format `(id_delivery, lat, lng, timestamp)`. Given the large potential size of the input data, the program uses **chunk-based processing**, where each chunk contains consecutive GPS points for one delivery. This avoids loading the entire dataset into memory, allowing the program to efficiently handle gigabyte-sized datasets.

2. Concurrency for Efficiency

To ensure the program runs efficiently, we make use of **Go's concurrency model** with **goroutines**. Each chunk of delivery data is processed in parallel by a separate goroutine, making the solution scalable for large datasets.

The program uses a **sync.WaitGroup** to ensure all goroutines complete before exiting. This guarantees that the main thread waits for all concurrent processes to finish their tasks, such as filtering data, calculating fares, and writing to output files.

3. Filtering Invalid Points

For each delivery, we calculate the speed between consecutive points using the **Haversine formula** to compute the distance between two latitude/longitude pairs.

A point is considered invalid if the speed between it and the previous point exceeds **100 km/h**. Such points are removed to ensure accurate fare estimation. The function `filterInvalidPoints()` handles this filtering logic.

4. Fare Calculation

Once the valid points are determined, the program calculates the fare based on the following rules:

- A base flag amount of **1.30** is charged for each delivery.
- Distance-based fare is calculated using the **Haversine formula**.
- Fares are adjusted based on the time of day: a **daytime rate** (5:00 AM to Midnight) and a **nighttime rate** (Midnight to 5:00 AM).
- Idle time is charged based on how long the delivery vehicle is stationary (speed <= 10 km/h).
- The minimum fare for any delivery is **3.47**.

5. Output and File Writing

The program produces two output files:

1. **Filtered Data**: A CSV file (`filtered_data.csv`) containing only the valid GPS points for each delivery.
2. **Fare Estimates**: A CSV file (`fares.csv`) with the format `(id_delivery, fare_estimate)`.

To ensure thread-safe writing of data (as multiple goroutines may attempt to write to the file at the same time), a **mutex (`sync.Mutex`)** is used to lock the file during write operations. This ensures that only one goroutine writes to the file at any given time, preventing data corruption.

6. Testing

The project includes a comprehensive suite of **unit tests** and **end-to-end tests**:

- **Unit tests**: Test each individual function, including the `haversine()` function for distance calculation, `filterInvalidPoints()` for filtering, and `calculateFare()` for fare computation.

- **End-to-end tests**: Simulate the entire flow from reading data, filtering invalid points, calculating fares, and writing results to the CSV files. Edge cases such as very short deliveries, idle time, and invalid GPS points are tested.

7. Handling Large Datasets

Given that the input dataset can be several gigabytes, the program is designed to work efficiently with **chunk-based processing** and **concurrency**.

By processing one delivery at a time (instead of loading the entire dataset into memory), the program scales effectively for large datasets, minimizing memory usage while maximizing processing speed. The chunked data is processed concurrently in multiple goroutines, making it highly performant even with large data sizes.

8. Best Practices & Documentation

The code is modular, with clearly defined functions and responsibilities, ensuring maintainability and extensibility. Each function has been documented with comments explaining its purpose, input, and output.

Concurrency is handled carefully with synchronization mechanisms (like `sync.WaitGroup` and `sync.Mutex`) to ensure correct execution without race conditions. The test suite provides comprehensive coverage of the code, ensuring that the project is robust and reliable.

Conclusion

This project implements a highly performant delivery fare estimation system using Go. The solution uses concurrency for efficiency, handles large datasets effectively, and includes thorough testing to ensure correctness and reliability.