Smart Cab Allocation System for Efficient Trip Planning

Introduction

With the increase in traffic in urban areas and surge in demand of the transportation system we see traffic congestion, longer travel times and environmental issues. In this case study we explore smart and efficient implementation of the cab allocation system.

Objective

The primary objective of this study is to design and implement a smart cab allocation system that optimizes the trip planning, enhances user experience and contributes to reducing traffic congestion.

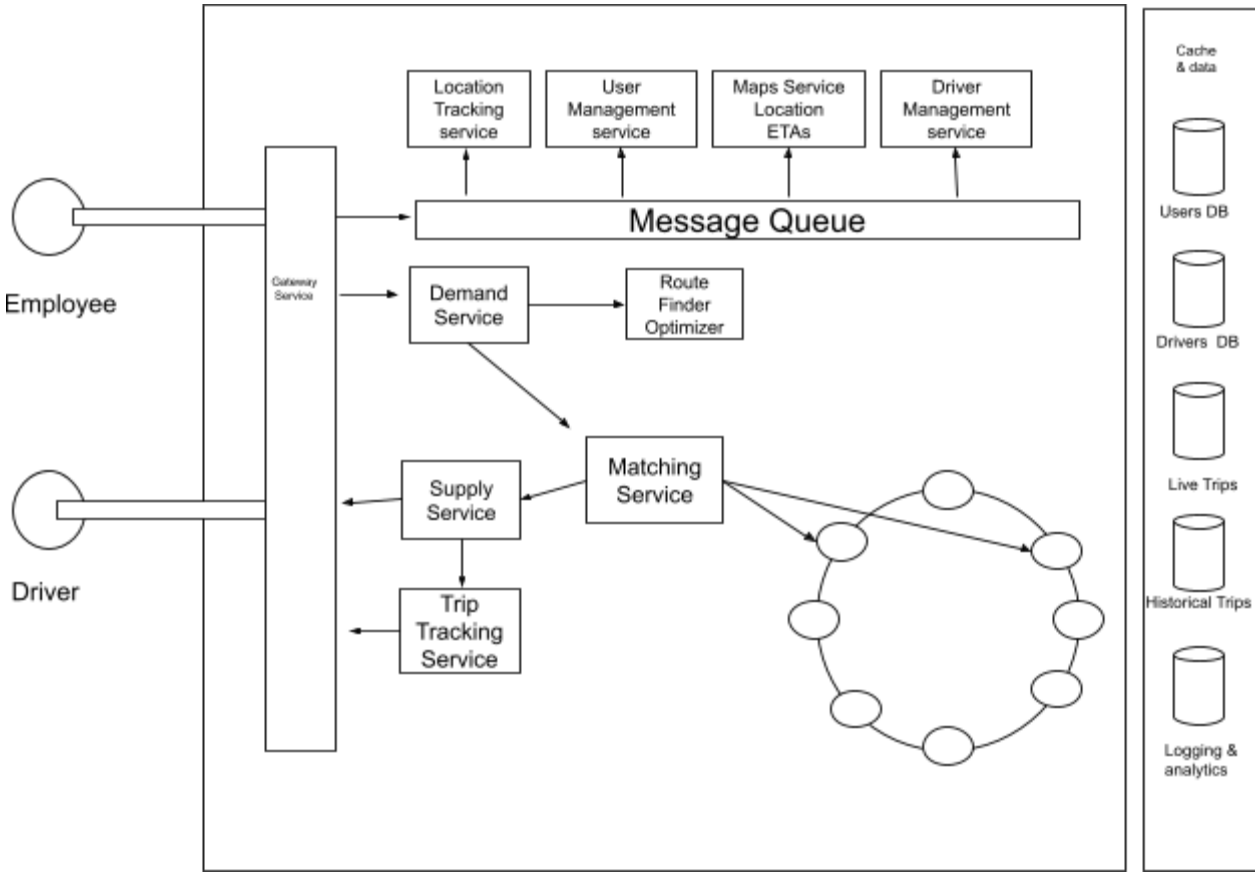
System Design

Functional Requirements

- **Customer**
  - Request Rides
  - Cancel Rides
  - View Nearby Vehicles
  - Destinations
  - Trip History
- **Driver**
  - Request for Rides
  - Accept/Deny Rides
  - Manage Trips once started
  - Trip History
  - Payment History
- **System**
  - Efficient match based on proximity, destination and other rides.
  - Minimize and predict ETAs and delays.
  - Handle surge in demand

Non Functional Requirements

- Scalable
- Handle both sparse and dense areas
- Reliable and durable



The figure shows the proposed flow of the trip management system. In addition to services shown in flow, this flow is convenient to implement consistent hashing, Gossip between two servers to share the state information, Scaling or descaling like adding or removing servers, adding or launching new regions.
Here we choose a micro service architecture, where we use different services to perform different tasks. Drivers and employees interact with the server through the gateway. It all starts when an employee demands service, the request goes through route finding optimizer and matches the employee with a driver, then we use a matching service to notify the driver using supply service. We store separate databases for users, drivers, live trips, historical trips and logging and analytics. As we have spatial proximity factors. It will be optimal to use spatial caching, more details about spatial caching are discussed in the below sections.

1. Admin's Cab Allocation Optimization

We can use a Quadtree to solve this problem. Using a Quadtree to find the nearest cab driver is a spatial indexing technique that can efficiently organize and query spatial data. Quadtree is a tree data structure in which each internal node has exactly four children corresponding to four quadrants of the space. Here's a general approach using a Quadtree for finding the nearest cab driver:

Construction of the Quadtree:

We divide the space into quadrants and build the Quadtree by recursively subdividing each quadrant until a certain condition is met (e.g., a minimum number of cars in a quadrant).We Store the cab driver locations at the leaf nodes of the tree.

Query for Nearest Cab Driver:

Start at the root of the Quadtree. Recursively traverse down the tree to the leaf node that contains the point of interest (the passenger's location).Perform this option until all the passengers are allocated with a taxi or all the taxis are booked.

Checking Neighbors:

Once we reach the leaf node containing the passenger's location, check the cab driver locations within that quadrant and potentially in neighboring quadrants.

Calculate Distance:

Calculate the distance between the passenger's location and the cab drivers found in the relevant quadrants.

Find Nearest Cab Driver:

Identify the cab driver with the minimum distance as the nearest one.

Optimizations:

We can Implement heuristics to prune branches of the Quadtree based on distance or other criteria to reduce unnecessary computations.
We can also use techniques like caching or storing precomputed distances to speed up the process. More details about caching

Time Complexity:

Construction of Quadtree:

The time complexity for constructing the Quadtree is O(n log n), where n is the number of cab driver locations. Each level of the Quadtree involves iterating through all the data points, and there are log n levels.

Querying for Nearest Cab Driver:
In the worst case, the time complexity for querying the Quadtree is O(log n) for each passenger.This assumes a balanced tree and involves traversing down from the root to the leaf node. So In worst case we need O(klogn) time for all the passengers combined.

Checking Neighbors and Calculating Distance:
The complexity of checking neighbors and calculating distances depends on the number of cab drivers in the relevant quadrants. In the worst case, it can be O(n), where n is the number of drivers.

Space Complexity:

Quadtree Storage:
The space complexity for storing the Quadtree structure is O(n).Each node in the Quadtree represents a subdivision of space and contains references to cab drivers.

Leaf Node Data:
The space complexity for storing cab driver locations at the leaf nodes is also O(n). Each cab driver location is stored in a leaf node.

2. Employee's Cab Search Optimization

To address this challenge, we propose the utilization of quadtrees, coupled with strategic caching mechanisms, as a robust solution for spatial data organization and retrieval.

Key Components:

Spatial Data Organization with Quadtrees:
Quadtrees provide a hierarchical spatial indexing structure that partitions the geographical area into quadrants.
Each node in the quadtree represents a spatial region, allowing for efficient organization and retrieval of cab locations based on their geographical coordinates.

Quadtree Construction and Maintenance:
During system initialization,we construct a quadtree using the geographical coordinates of cab locations. We regularly update the quadtree to reflect real-time changes in cab positions, ensuring accuracy and responsiveness.

Querying Nearby Cabs:
When a user requests nearby cabs at a specific location, traverse the quadtree to identify the leaf node corresponding to that location. Retrieve cab locations stored in the identified leaf node and neighboring nodes to efficiently find nearby cabs.

Caching Strategies:
Spatial Index Caching:
Cache the quadtree structure to avoid repeated reconstruction for each query, reducing computational overhead. Refresh the cached quadtree periodically to incorporate changes in cab locations.
Cab Location Caching:
We cache the cab locations associated with each leaf node of the quadtree.
Implement cache invalidation mechanisms to ensure that the cached information is up-to-date.
Query Result Caching:
We cache the results of frequent queries for specific locations to improve response time.
Implement expiration and eviction policies to manage the size of the query result cache.

Adaptive Caching:
We tailor caching strategies based on the popularity and demand of specific geographical areas.Allocate more detailed or frequently updated cached information to high-demand areas.

Geofencing and Efficient Area Management:
We use quadtrees to define geofenced areas, allowing for optimized caching and efficient area-specific data retrieval. Geofencing ensures that cached information is tailored to the unique characteristics of different regions.

Benefits:

Optimized Spatial Queries:
Quadtrees enable quick identification of nearby cabs, significantly reducing the search space and enhancing query performance.
Real-time Responsiveness: Caching ensures that frequently accessed information is readily available, minimizing computational delays and improving the real-time responsiveness of the system.

Adaptability and Scalability:
The adaptive caching strategies and hierarchical organization provided by quadtrees make the system adaptable to varying levels of demand and scalable to handle increased user activity.

Conclusion:
By integrating quadtrees and caching strategies, our proposed solution optimizes the retrieval of nearby cab locations, creating a robust foundation for efficient and scalable ride-sharing services. This approach not only enhances the user experience by providing timely and accurate information but also ensures that computational resources are utilized optimally.

3. Real-Time Location Data Integration:

Firebase, a mobile and web application development platform owned by Google, offers a real-time NoSQL database that can be a powerful solution for integrating and managing real-time location data. Below is a proposal on how Firebase can be utilized for this purpose:

1. Firebase Realtime Database:
We can use Firebase Realtime Database to store and synchronize real-time location data. The database is a JSON-like structure that allows instant updates to connected clients whenever data changes.

2. Authentication and Security:
We can leverage Firebase Authentication to secure access to our database. This ensures that only authorized users or devices can read or write location data.

3. Mobile SDKs:
We can integrate Firebase SDKs into mobile applications for seamless communication with the Realtime Database. Firebase SDKs are available for iOS, Android, and web platforms.

4. Real-Time Updates:
We can utilize Firebase's real-time capabilities to automatically push location updates to all connected clients. This ensures that all users receive the latest location information without the need for manual refreshing.

5. Geolocation Integration:
Combine Firebase with the device's geolocation APIs (such as HTML5 Geolocation for web or Core Location for iOS) to continuously update and push the device's location to the Firebase Realtime Database.

6. Firestore for Advanced Querying:
Consider using Firestore, Firebase's cloud-native NoSQL database, for more advanced querying capabilities if needed. Firestore allows for complex queries and is scalable for larger datasets.

7. Cloud Messaging:
We can Integrate Firebase Cloud Messaging (FCM) to send push notifications to users when relevant location-based events occur. For instance, notify users when a tracked vehicle reaches a specific destination.

8. Monitoring and Analytics:
We can leverage Firebase Analytics to gain insights into user behavior and app usage. Firebase Performance Monitoring can help identify and resolve performance issues in real-time.

By combining Firebase's real-time capabilities with geolocation services and other Firebase features, we can create a robust and scalable real-time location data integration solution.

Conclusion:

In conclusion, the proposed smart cab allocation system represents a robust solution aimed at enhancing efficiency in trip allocation. Through careful system design, we have addressed key objectives such as minimizing wait times, optimizing routes, and prioritizing user satisfaction. The utilization of spatial data structures, notably the Quadtree, showcases a strategic approach to handle location-based queries, ensuring swift and accurate identification of the nearest cab drivers.

The implementation considerations encompass scalability, real-time processing, data storage, and retrieval mechanisms, all geared towards creating a reliable and responsive system. Moreover, the incorporation of optimization techniques and heuristics adds a layer of intelligence to the allocation process, contributing to an overall enhanced user experience.