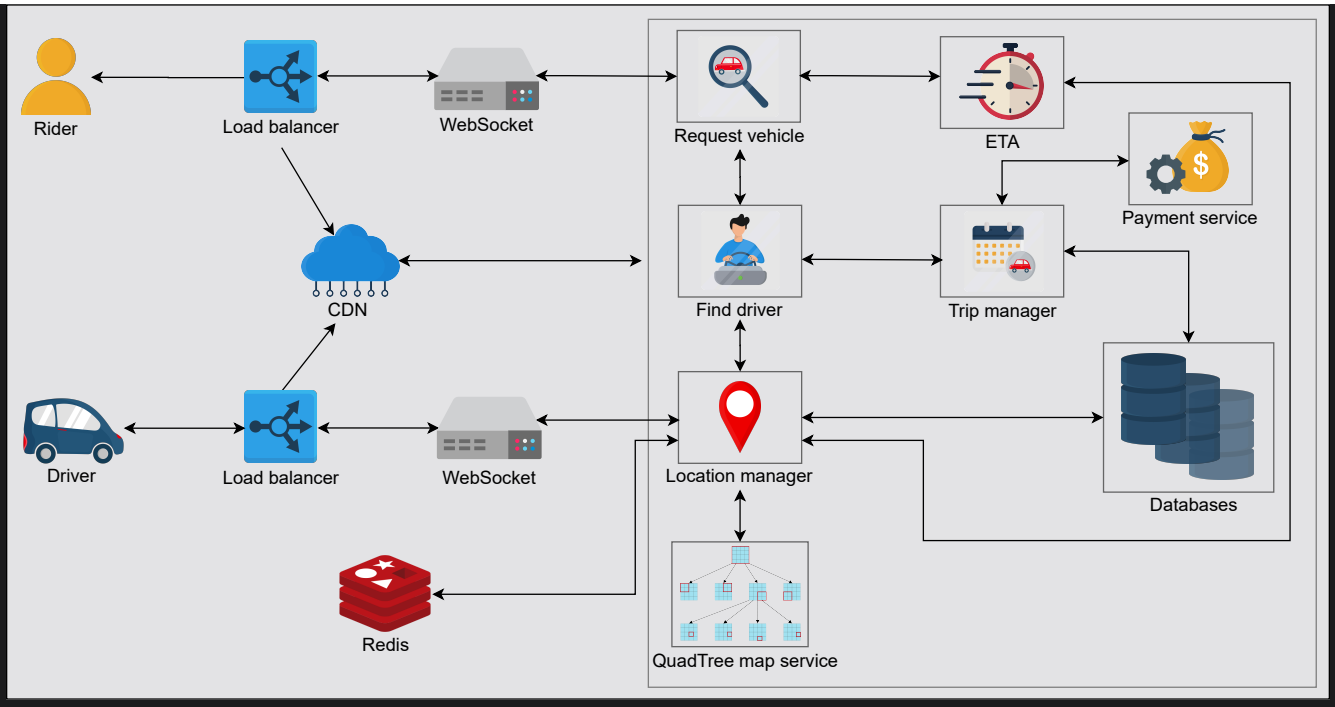
Detailed Design of Uber

Learn about the detailed design of the Uber system.

We'll cover the following

Let’s look at the detailed design of our Uber system and learn how the various components work together to offer a functioning service:



The detailed design of Uber

Components

Let’s discuss the components of our Uber system design in detail.

Location manager

The riders and drivers are connected to the **location manager** service. This service shows the nearby drivers to the riders when they open the application. This service also receives location updates from the drivers every four seconds. The location of drivers is then communicated to the QuadTree map service to determine which segment the driver belongs to on a map. The location manager saves the last location of all drivers in a database and saves the route followed by the drivers on a trip.

QuadTree map service

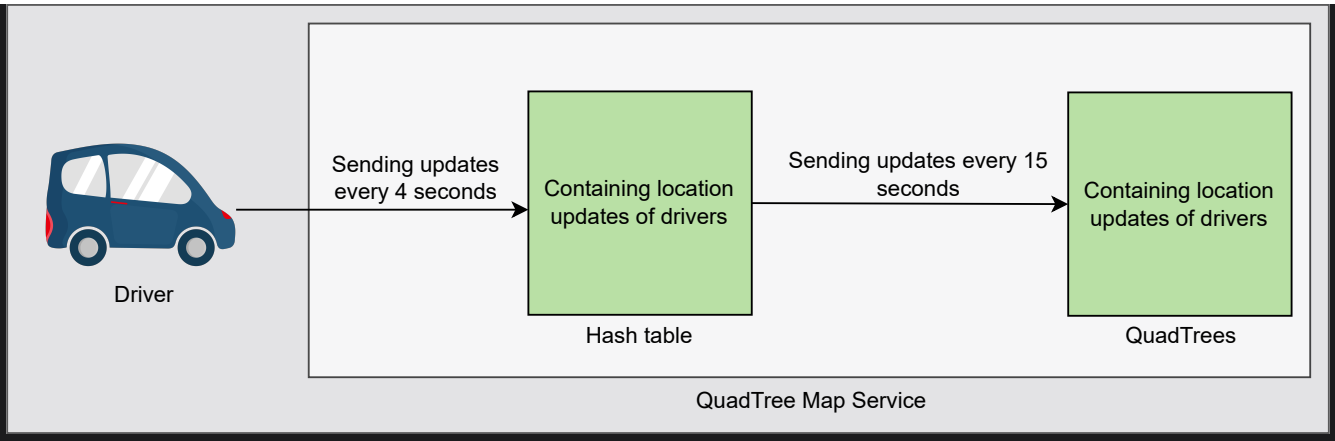
The **QuadTree map service** updates the location of the drivers. The main proble is how we deal with finding nearby drivers efficiently.

We'll modify the solution discussed in the Yelp chapter according to our requirements. We used QuadTrees on Yelp to find the location. QuadTrees help to divide the map into segments. If the number of drivers exceeds a certain limit, for example, 500, then we split that segment into four more child nodes and divide the drivers into them.

Each leaf node in QuadTrees contains segments that can't be divided further. We can use the same QuadTrees for finding the drivers. The most significant difference we have now is that our QuadTree wasn't designed with regular upgrades in consideration. So, we have the following issues with our dynamic segment solution.

We must update our data structures to point out that all active drivers update their location every four seconds. It takes a longer amount of time to modify the QuadTree whenever a driver's position changes. To identify the driver's new location, we must first find a proper grid depending on the driver's previous position. If the new location doesn't match the current grid, we should remove the driver from the current grid and shift it to the correct grid. We have to repartition the new grid if it exceeds the driver limit, which is the number of drivers for each region that we set initially. Furthermore, our platform must tell both the driver and the rider, of the car's current location while the ride is in progress.

To overcome the above problem, we can use a hash table to store the latest position of the drivers and update our QuadTree occasionally, say after 10–15 seconds. We can update the driver's location in the QuadTree around every 15 seconds instead of four seconds, and we use a hash table that updates every four seconds and reflects the drivers' latest location. By doing this, we use fewer resources and time.



Updating driver's location

Request vehicle

The rider contacts the **request vehicle** service to request a ride. The rider adds the drop-off location here. The request vehicle service then communicates with the find driver service to book a vehicle and get the details of the vehicle using the location manager service.

Find driver

The **find driver** service finds the driver who can complete the trip. It sends the information of the selected driver and the trip information back to the request vehicle service to communicate the details to the rider. The find driver service also contacts the trip manager to manage the trip information.

Trip manager

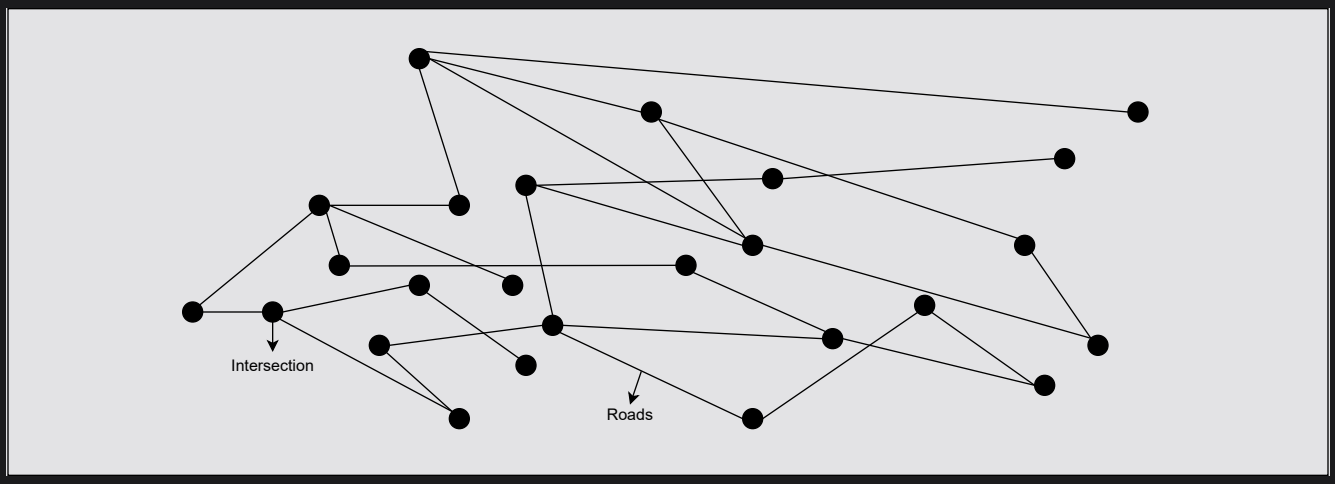
The **trip manager** service manages all the trip-related tasks. It creates a trip in the database and stores all the information of the trip in the database.

ETA service

The **ETA service** deals with the estimated time of arrival. It shows riders the pick ETA when their trip is scheduled. This service considers factors such as route and traffic. The two basic components of predicting an ETA given an origin and destination on a road network are the following:

- Calculate the shortest route from origin to destination.
- Compute the time required to travel the route.

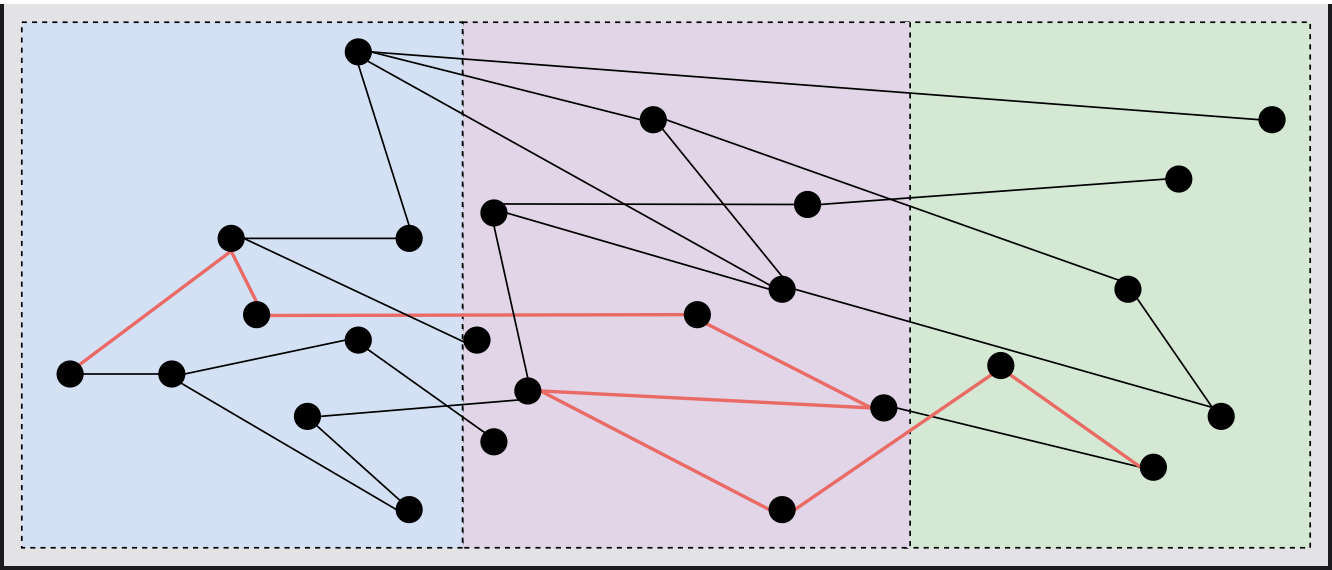
The whole road network is represented as a graph. Intersections are represented by nodes, while edges represent road segments. The graph also depicts one-way streets, turn limitations, and speed limits.



Graph representation

To identify the shortest path between source and destination, we can utilize routing algorithms such as Dijkstra's algorithm. However, Dijkstra, or any other algorithm that operates on top of an unprocessed graph, is quite slow for such a system. Therefore, this method is impractical at the scale at which these ride-hailing platforms operate.

To resolve these issues, we can split the whole graph into partitions. We preprocess the optimum path inside partitions using **contraction hierarchies** and deal with just the partition boundaries. This strategy can considerably reduce the time complexity since it partitions the graph into layers of tiny cells that are largely independent of one another. The preprocessing stage is executed in parallel in th partitions when necessary to increase speed. In the illustration below, all the partitions process the best route in parallel. For example, if each partition takes one second to find the path, we can have the complete path in one second since all partitions work in parallel.



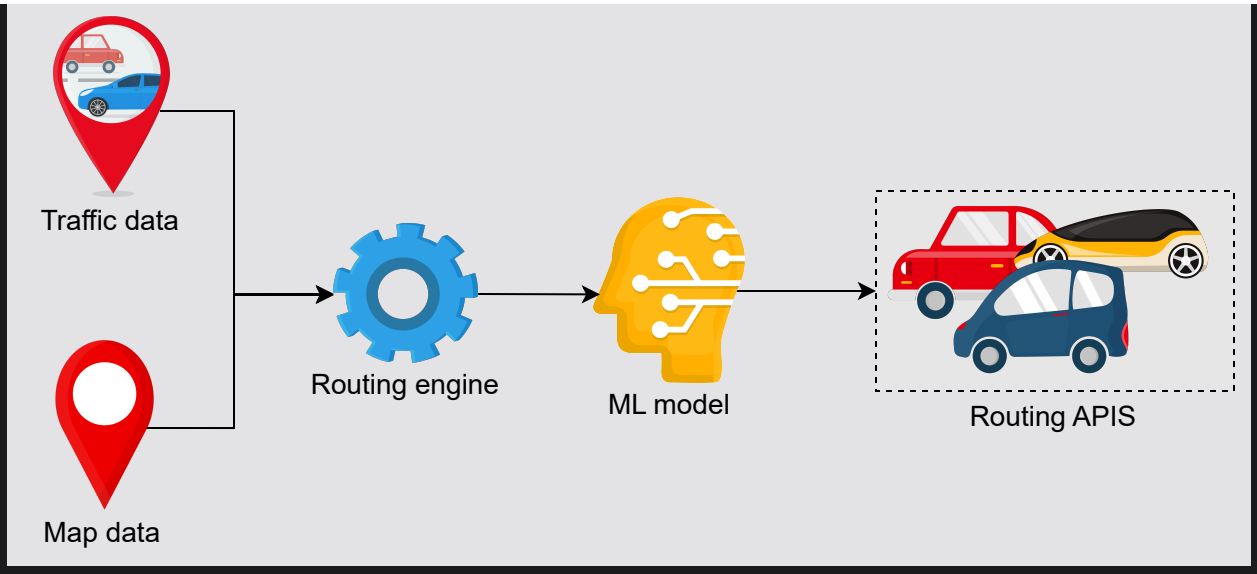
Partitioning the graph

Once we determine the best route, we calculate the expected time to travel the road segment while accounting for traffic. The traffic data will be the edge weights between the nodes.

DeepETA

We use a machine learning component named **DeepETA** to deliver an immediate improvement to metrics in production. It establishes a model foundation that can be reused for multiple consumer use cases.

We also use a routing engine that uses real-time traffic information and map data to predict an ETA to traverse the best path between the source and the destination. We use a post-processing ML model that considers spatial and temporal parameters, such as the source, destination, time of the request, and knowledge about real-time traffic, to forecast the ETA residual.



DeepETA

Database

Let's select the database according to our requirements:

- The database must be able to scale horizontally. There are new customers and drivers on a regular basis, so we should be able to add more storage without any problems.
- The database should handle a large number of reads and writes because the location of drivers is updated every four seconds.
- Our system should never be down.

We've already discussed the various database types and their specifications in the Database chapter. So, according to our understanding and requirements (high availability, high scalability, and fault tolerance), we can use Cassandra to store the driver's last known location and the trip information after the trip has been completed, and there will be no updates to it. We use Cassandra because the data we store is enormous and it increases continuously.

We can use a MySQL database to store trip information while it's in progress. W use MySQL for in-progress trips for frequent updates since the trip information is relational, and it needs to be consistent across tables.

> **Note:** Recently, Uber migrated their data storage to Google Cloud Spanner. It provides global transactions, strongly consistent reads, and automatic multisite replication and failover features.

We're using both MySQL and Cassandra databases in Uber's design. Why can't we use **only** MySQL or **only** a Cassandra database? In other words, what is the need for using both databases?

Want to know the correct answer?

Why can't we use only MySQL or only a Cassandra database?

H1 H2 H3 | B I | | | |

Provide your answer here!

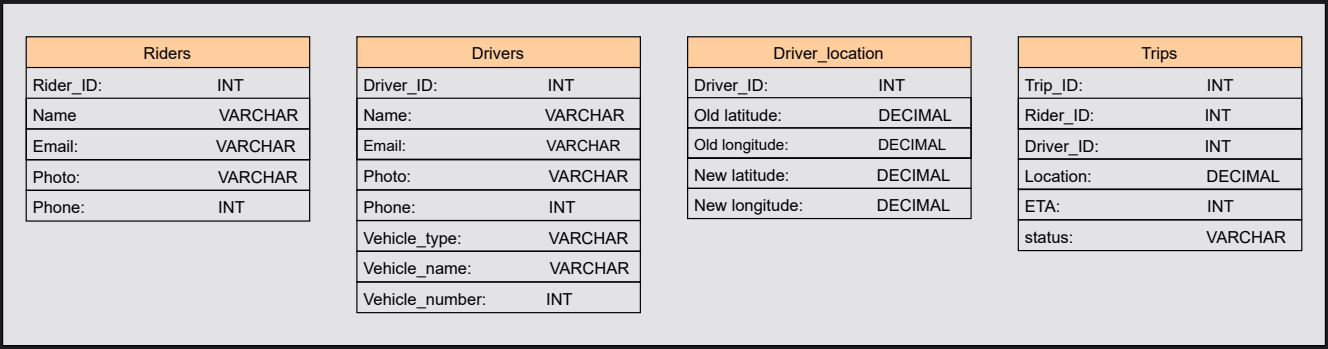
educative

Storage schema

On a basic level in the Uber application, we need the following tables:

- **Riders:** We store the rider's related information, such as ID, name, email, photo, phone number, and so on.
- **Drivers:** We store the driver's related information, such as ID, name, email, photo, phone number, vehicle name, vehicle type, and so on.
- **Driver_location:** We store the driver's last known location.
- **Trips:** We store the trip's related information, such as trip ID, rider ID, driver ID, status, ETA, location of the vehicle, and so on.

The following illustration visualizes the data model:



Riders	
Rider_ID:	INT
Name	VARCHAR
Email:	VARCHAR
Photo:	VARCHAR
Phone:	INT

Drivers	
Driver_ID:	INT
Name:	VARCHAR
Email:	VARCHAR
Photo:	VARCHAR
Phone:	INT
Vehicle_type:	VARCHAR
Vehicle_name:	VARCHAR
Vehicle_number:	INT

Driver_location	
Driver_ID:	INT
Old latitude:	DECIMAL
Old longitude:	DECIMAL
New latitude:	DECIMAL
New longitude:	DECIMAL

Trips	
Trip_ID:	INT
Rider_ID:	INT
Driver_ID:	INT
Location:	DECIMAL
ETA:	INT
status:	VARCHAR

The storage schema

Fault tolerance

For availability, we need to have replicas of our database. We use the primary-secondary replication model. We have one primary database and a few secondary databases. We synchronously replicate data from primary to secondary databases. Whenever our primary database is down, we can use a secondary database as a primary one.

Point to Ponder

Question

How will we handle a driver’s slow and disconnecting network?

Show Answer

Load balancers

We use load balancers between clients (drivers and riders) and the application servers to uniformly distribute the load among the servers. The requests are routed to the specified server that provides the requested service.