Uber is a service provider which allows user to book a cab and a driver to go to some other place similar to taxi. It is available on web and mobile platforms.

**REQUIREMENTS**

**FUNCTIONAL REQUIREMENTS**

**Customers :**

1. Customers should be able to see the cabs around its area or vicinity with estimated arrival time(ETA) and price info.
2. Customer should be able to book the cab.
3. Customer should be able to see location of the driver.

**Drivers :**

1. Driver should be able to accept or deny the customer request.
2. Once driver accept, he should be able to see the pickup location of the customer.
3. Driver should be able to mark the trip as complete.

**NON FUNCTIONAL REQUIREMENTS**

Low latency

High reliability

High Availability

System should be efficient and scalable.

**Extended Requirements**

1. Customers giving ratings to drivers.
2. Payment Integration.
3. Metrics and analytics.

**ESTIMATIONS**

Daily active users = 100 million

Daily Average rides = 10 million

Average actions perform by 1 user daily = 10

Daily Action performed = 100 \* 10 = 1000 million = 1 billion

Request Per Second = 1B/(24\*3600) = 12k request/second

Storage

Each action consumes = 400 bytes

Every day storage requirement = 400 \* 1B = 400GB/day

Bandwidth

Storage = 400GB/Day

= 400/(24\*3600) = 5MB/sec

**DATA DESIGN MODEL**

**Tables**

**Customers**(userId, Name, Email, Mobile Number, createdAt)

**Driver**(driverId, name, Email, VehicleNo, CreatedAt)

**Cabs**(cabId, Type, Name, DriverId)

**Payment**(paymentId, tripId, paymentType, amount, createdAt)

**Trip**(tripId, customerId, driverId, paymentId, status, source, destination, createdAt)

**Ratings**(ratingId, driverId, userId, tripId, rating, feedback)

**What kind of databases we should use**

We will split the database between different services each having ownership of a particular table. Then we can use a relational database such as PostgreSQL or SQL or a distributed NoSQL database such as Apache Cassandra for our use case.

**API DESIGN**

**RequestRide**(customerId, source, destination, cabType, paymentType);

**CancelRide**(rideId, customerId, reason)

**acceptRide**(driverId, tripId);

**cancelRide**(driverId, tripId);

**rateTrip**(tripId, customerId, driverId, rating)

**High Level Design**

**Architecture**

We will be using microservice architecture since it is easier to horizontally scale and decouple the services. Each service will have ownership of its own data model.Lets try to divide the system into more core services.

Customer service, Driver service, Ride service, payment serivce , trip service, Notification service, analytics service

Since out architecture is microservice architecture, services will be communicating with one another.

**Location Tracking**

Sending the live location from the driver and customer to the backend, we can use 2 options:

1. Pull method : The client can periodically send the HTTP request to report its current location and receive ETA and pricing information.
2. Push method : We can use Websockets if we want a long-lived continuous connection with the server and once a new data is available, it will be pushed to the client.

To minimize latency, using the push model withWebsockets is a better choice because then we can push data to the client once it's available without any delay given the connection is open with the client. Also, WebSockets provide full-duplex communication.

**Ride Matching**

We need a way to efficiently store and query nearby drivers, we use Geohashing.

[Geohashing](https://dev.to/courses/sytem-design/geohashing-and-quadtrees#geohashing) is a [geocoding](https://en.wikipedia.org/wiki/Address_geocoding) method used to encode geographic coordinates such as latitude and longitude into short alphanumeric strings. It was created by [Gustavo Niemeyer](https://twitter.com/gniemeyer) in 2008.

Geohash is a hierarchical spatial index that uses Base-32 alphabet encoding, the first character in a geohash identifies the initial location as one of the 32 cells. This cell will also contain 32 cells. This means that to represent a point, the world is recursively divided into smaller and smaller cells with each additional bit until the desired precision is attained. The precision factor also determines the size of the cell.

**QUADTREE**

A [Quadtree](https://github.com/karanpratapsingh/system-design/blob/main/courses/sytem-design/geohashing-and-quadtrees#quadtrees) is a tree data structure in which each internal node has exactly four children.

[Quadtree](https://github.com/karanpratapsingh/system-design/blob/main/courses/sytem-design/geohashing-and-quadtrees#quadtrees) seems perfect for our use case, we can update the Quadtree every time we receive a new location update from the driver. To reduce the load on the quadtree servers we can use an in-memory datastore such as [Redis](https://redis.io) to cache the latest updates. And with the application of mapping algorithms such as the [Hilbert curve](https://en.wikipedia.org/wiki/Hilbert_curve), we can perform efficient range queries to find nearby drivers for the customer.

Let us identify and resolve bottlenecks such as single points of failure in our design:

* "What if one of our services crashes?"
* "How will we distribute our traffic between our components?"
* "How can we reduce the load on our database?"
* "How to improve the availability of our cache?"
* "How can we make our notification system more robust?"

To make our system more resilient we can do the following:

* Running multiple instances of each of our services.
* Introducing [load balancers](https://karanpratapsingh.com/courses/system-design/load-balancing) between clients, servers, databases, and cache servers.
* Using multiple read replicas for our databases.
* Multiple instances and replicas for our distributed cache.
* Exactly once delivery and message ordering is challenging in a distributed system, we can use a dedicated [message broker](https://karanpratapsingh.com/courses/system-design/message-brokers) such as [Apache Kafka](https://kafka.apache.org) or [NATS](https://nats.io) to make our notification system more robust.

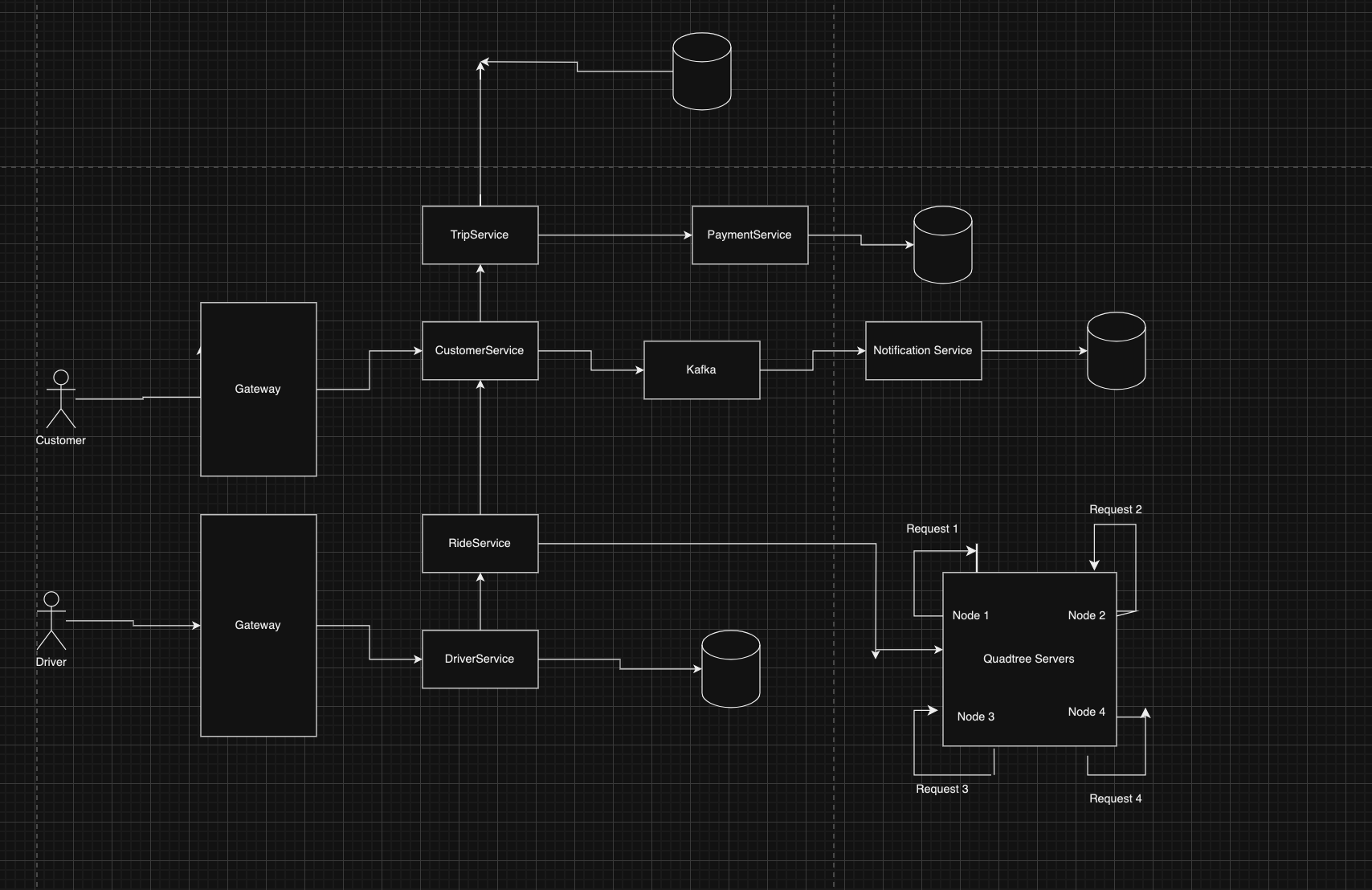
## **What Databases Uber used to use?**

Earlier they have used RDBMS to save profile-related data and GPS points and everything. But they identified they couldn’t scale when they got more and more users as well as cities. Then they moved to NoSQL databases that are built on top of MYSQL something called schemaless. They considered about

* Horizontally scalable — You can add multiple nodes in different regions and altogether acts as one database
* Write and the Read availability — every 4-sec cab will be sending the GPS location to the database. So that there are tons of reading and write happens to the system.
* No downtime — The system will be always available and what uber adds or remove from the system or while they are doing some maintenance for the system then the system should be up and there should be no downtime.
* Nearest datacenters — When they add new cities to the system they try to add new data centers or else they store data on the nearest datacenters to the newly added city to give the seamless service.

**REFERENCE**

**https://github.com/karanpratapsingh/system-design#uber**



In the Uber system design, the choice of database depends on various factors, including the scale of the system, the data requirements, the read and write patterns, and the overall design goals. Uber operates at a massive scale, handling millions of rides and transactions daily. Therefore, the database used should be able to handle high throughput, low latency, and provide high availability.

In a typical Uber system, a combination of databases and data storage solutions is used to serve different purposes:

1. **Relational Database (RDBMS):** RDBMSs like PostgreSQL or MySQL are used for managing critical data and ensuring data consistency. Relational databases are ideal for handling user data, ride history, payment information, and other structured data.
   * User Data: Information about drivers and riders, including user profiles, account details, and preferences, is well-suited for storage in a relational database.
   * Ride History: The history of completed rides, including trip details, fares, and driver ratings, can be stored in a relational database.
   * Payment Information: Payment-related data, such as transaction details and payment methods, can be stored securely in an RDBMS.
2. **NoSQL Databases:** NoSQL databases like Apache Cassandra or Amazon DynamoDB are used for managing highly scalable and distributed data. NoSQL databases are suitable for handling data that needs to be replicated across multiple data centers for high availability and partitioned for distributed storage.
   * Real-Time Analytics: To support real-time analytics and monitoring, NoSQL databases can be used to store event data, tracking user locations, ride requests, and driver availability.
   * Geospatial Data: To efficiently manage location-based queries for finding nearby drivers and estimating ride distances, a geospatial database like Redis with geospatial capabilities can be used.