Ace the System Design Interview — Uber/Lyft

1. Introduction

When it comes to getting about in cities, many people choose ride-sharing apps like Uber or Lyft. Indeed, such apps make short-range commute very easy by offering competitive pricing, reasonably short wait time, and high availability. From a technical point of view, systems like these are very interesting because nearest-neighbor searching is hard. In this article, I want to share my design of large-scale ride-sharing apps like Uber/Lyft.

Requirements

Modern ride-sharing apps offer complex features such as carpooling, trip sharing, real-time chat in addition to ride-matching. In this article, I want to keep the discussion brief and focus only on the core functionality — booking a ride. Below are the **functional requirements** of our system:

- Users can request a ride and should be matched to a driver in close proximity
- Users can see all nearby drivers (although not choosing which one)
- Drivers can answer/decline requests from nearby riders.
- When a trip is created, both parties see each other's real-time location.

Needless to say, the system should be scalable and highly available.

Traffic Estimation

Before talking about traffics, let's examine a unique feature of Uber — its data usage is local. Unlike apps such as Slack or Instagram, inter-region data communication is rarely needed. If you are physically in New York, you can't book a ride in London. Hence, locational data and trips are not replicated across different regions (less-frequently accessed data such as profiles are replicated globally, of course). In this sense, discussing global traffic is not as meaningful as regional traffic.

The amount of ride requests hitting our system varies depending on the app's popularity. Here we assume significant regional traffic for a generalized solution:

- Group cities by proximity into regions. We need maybe a dozen regions to cover the whole of U.S
- We expect ~100K active drivers in one region.
- We expect ~1M active users in one region.
- The total amount of users globally is 10M.

Assume both drivers and riders emit their location ~5s with each message consisting of longitude, latitude, and additional metadata. In addition, riders regularly check nearby drivers (~5s) and sometimes make a ride request. The amount of QPS and bandwidth we can expect are:

- We expect ~200K location updates/writes per second
- We expect ~200K queries per second, double it for peak traffic handling.
- Each location message consists of ID (8 bytes), longitude and latitude (16 bytes), and other info (64 bytes). The total upload bandwidth is just shy of 88MB/s

2. High-level Design

Database Design

The access pattern of our application determines what schema is used. Let's investigate the requests that hit the database:

Read Operations

- Given a user ID, retrieve its profile
- Given a user ID, retrieve all trips completed by the user
- Given longitude and latitude, query all drivers nearby

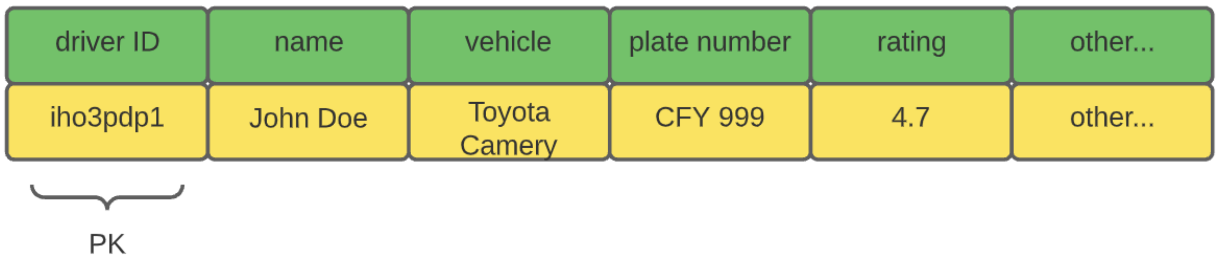
Write Operations

- Given a user ID, update its location
- Given a trip ID, the driver can either take/decline the request

Database Schema

Given the access patterns, sharded SQL is a good choice since no complex relationship queries are made. Note that NoSQL databases like Cassandra could also be used here if strong consistency is not a hard requirement.

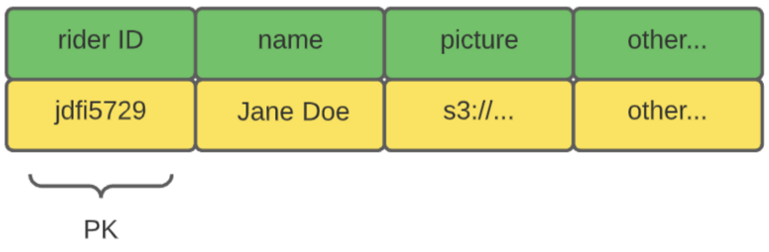
Driver profile table



driver ID	name	vehicle	plate number	rating	other...
iho3pdp1	John Doe	Toyota Camery	CFY 999	4.7	other...

figure 1. driver profile table, figure by author

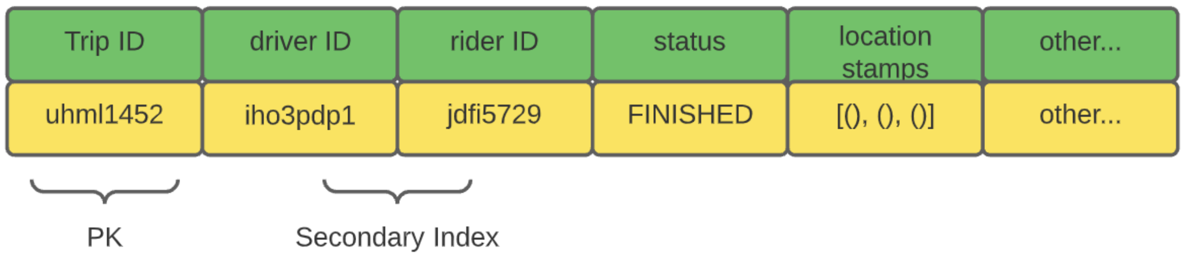
Rider profile table



rider ID	name	picture	other...
jdfi5729	Jane Doe	s3://...	other...

figure 2. rider profile table, figure by author

Trip details table



Trip ID	driver ID	rider ID	status	location stamps	other...
uhml1452	iho3pdp1	jdfi5729	FINISHED	[(), (), ()]	other...

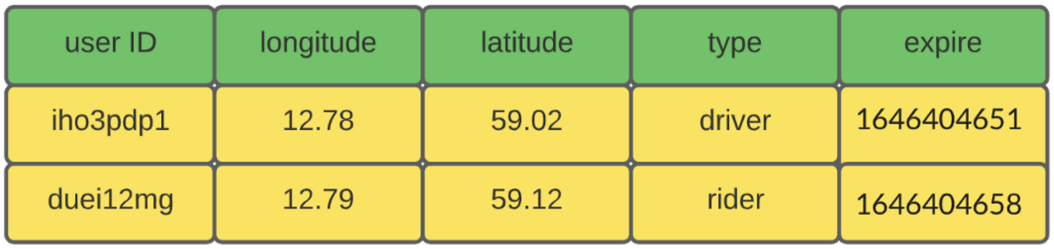
PK
Secondary Index

figure 3. trip details table, figure by author

In addition to these database tables, we need another high-performance storage to hold frequently-updated location data. Since live location data is inherently ephemeral, persisting them onto disks does not make sense. A good alternative would be using an in-memory cache such as Redis or Memcache.

Cache Schema

Location truth cache

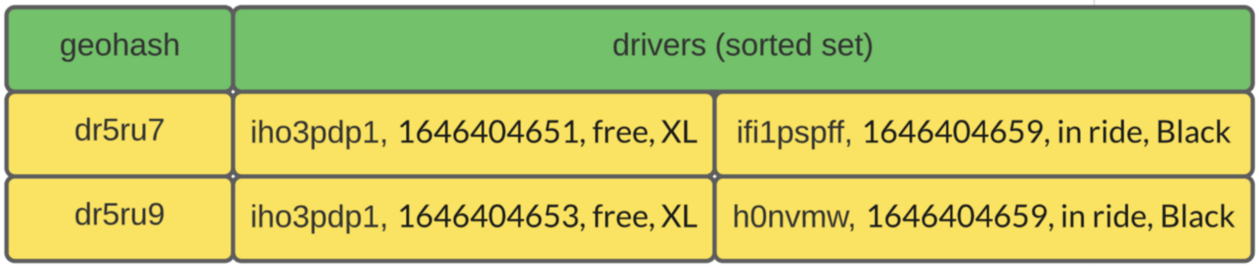


user ID	longitude	latitude	type	expire
iho3pdp1	12.78	59.02	driver	1646404651
duei12mg	12.79	59.12	rider	1646404658

figure 4. location truth cache, figure by author

This cache is the source of truth when it comes to user location. Phone apps send regular updates to maintain accuracy. If a user gets disconnected, its record expires in 30 seconds.

Driver proximity cache



geohash	drivers (sorted set)	
dr5ru7	iho3pdp1, 1646404651, free, XL	ifi1pspff, 1646404659, in ride, Black
dr5ru9	iho3pdp1, 1646404653, free, XL	h0nvmw, 1646404659, in ride, Black

figure 5. driver proximity cache, figure by author

The proximity cache is crucial for nearby driver search. Given a location, we can use GeoHash to compute its location key and retrieve all drivers in the grid. I'll talk more about this in the Details section.

Architecture

With a clearer understanding of what data to store, now it's time for service-oriented designs!

- **Notification Service**: whenever the backend needs to send information to the clients, the notification service is used to deliver the messages.
- **Trip Management Service**: When a trip is initiated, this service is needed to monitor the locations of all parties as well as plan routes
- **Ride Matching Service**: This service handles ride requests. It finds nearby drivers and matchmakes based on driver responses (either accept or decline)
- **Location Service**: All users must regularly update their locations via this service.

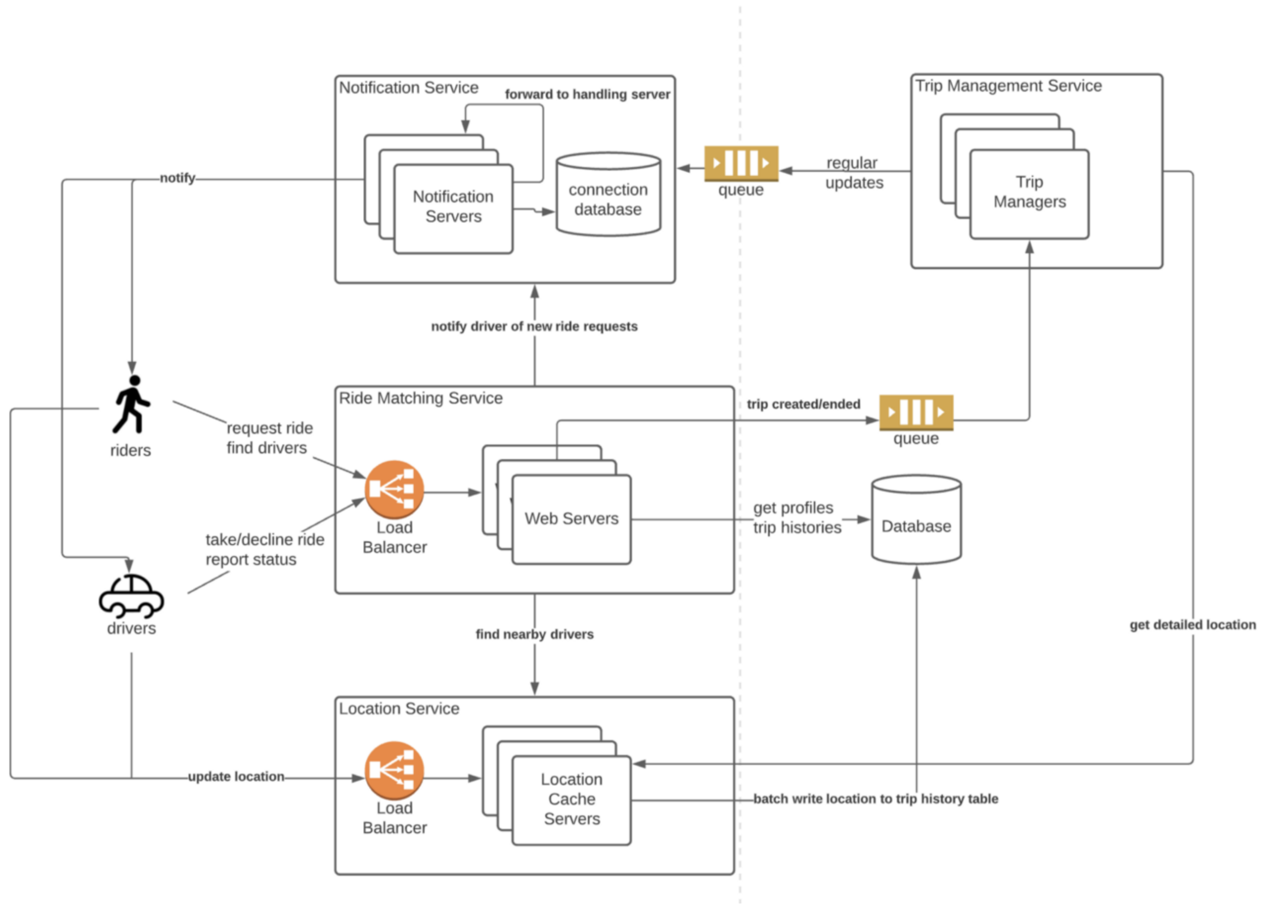


figure 6. high-level architecture, figure by author

Workflow

1. Bob sends a ride request to the Ride Matching service and hopes to pair with a driver.

2. The Rider Matching service contacts the Location service and finds all available drivers in the same region.
3. The Rider Matching service then conducts ranking/filtering and pushes the ride request to selected drivers via the Notification service.
4. Drivers can accept/decline the request. When received, the Ride Matching Service will send the trip details to the Trip Management service.
5. The Trip Management service monitors the trip progress and broadcasts driver&rider locations to all parties involved in the trip.

3. Details

The high-level architecture design is relatively straightforward. However, if you examine carefully, there are still a few wrinkles. For instance, I've ignored features such as trip pricing, carpooling, etc. In this article, I want to focus on location-related problems as they are the foundation of Uber/Lyft.

3.1 How to efficiently look up nearby drivers?

Nearest neighbor search is a hard problem and the scale of our system makes efficient lookup even more difficult. Instead of computing the distance between the rider and every driver in the database, we can use a technique called GeoHash to convert the user's location to a unique key that corresponds to one cell in figure 7 and confine the search to a few adjacent cells only.

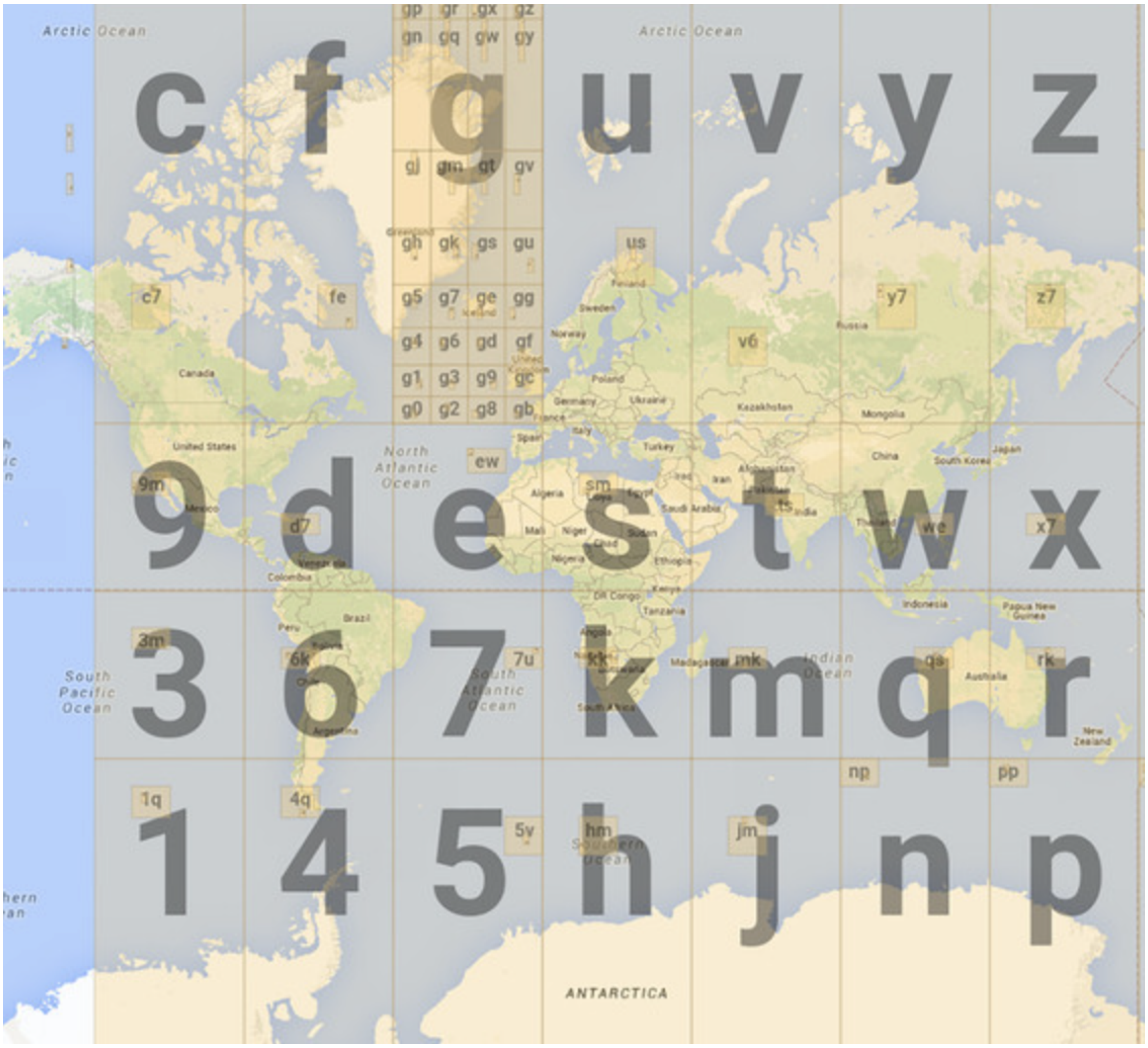
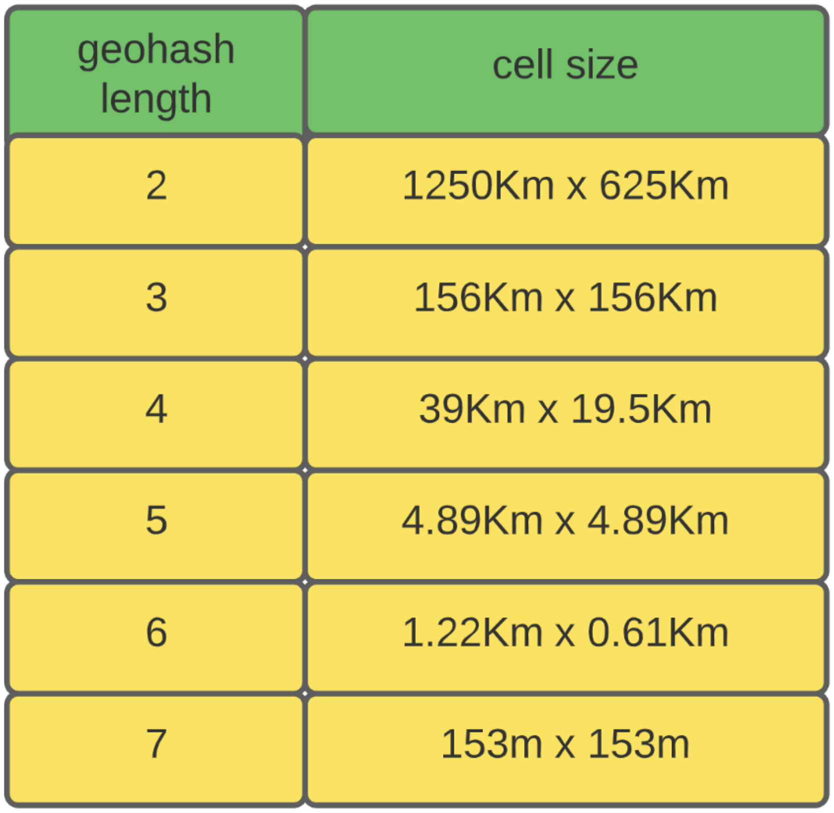


figure 7. location to GeoHash, PC credit: Movable Type Scripts

Figure 7 demonstrates the key property of GeoHash — a one-to-one mapping between keys and location cells. Therefore, we can use the following heuristic to quickly lookup drivers:

1. Given a user location, compute its GeoHash from longitude and latitude.
2. With the GeoHash key available, it's easy to compute the keys for the 8 nearby cells. (see this post)
3. Query the Location service with all 9 keys; retrieve drivers in those cells.

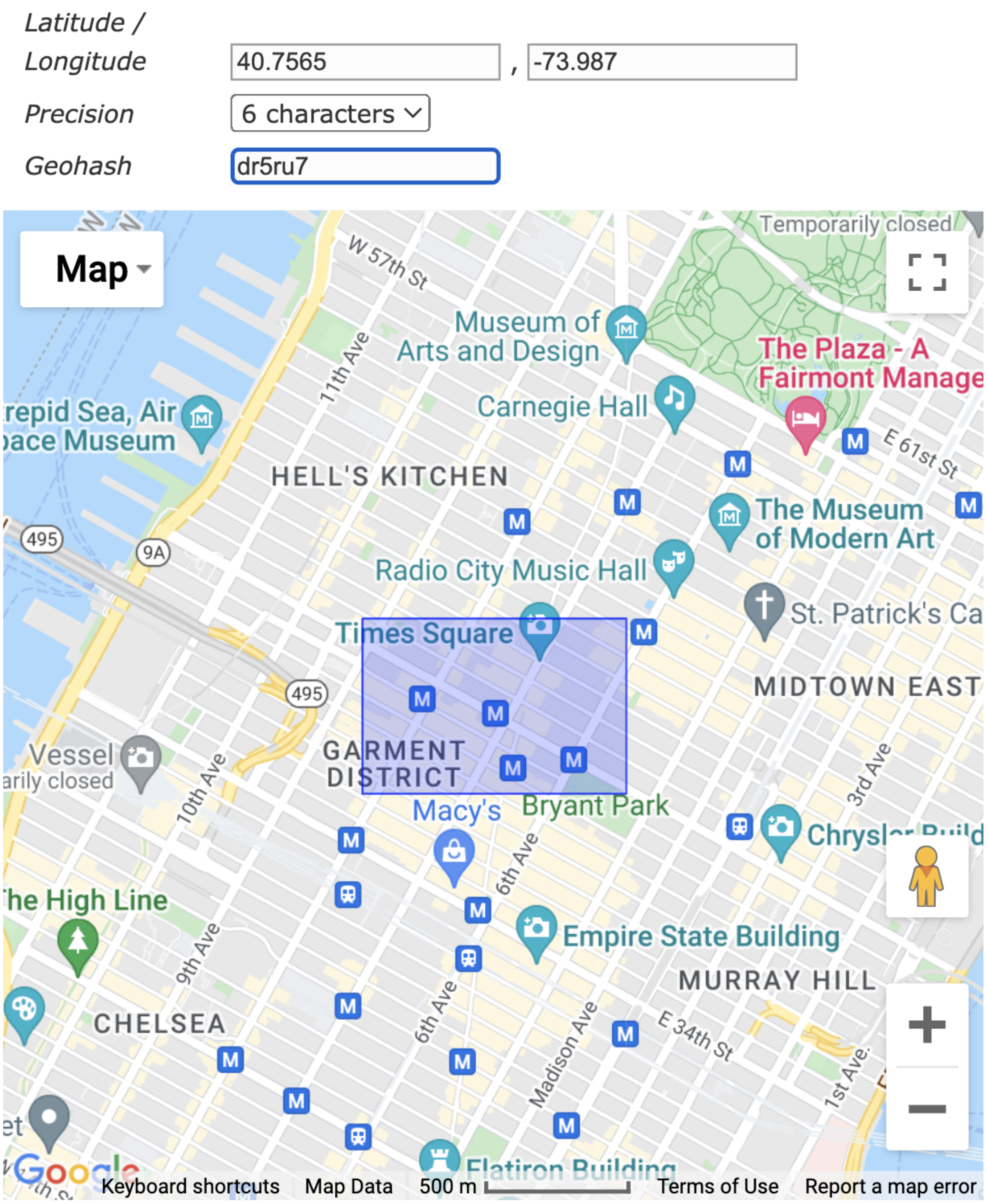
The accuracy of GeoHash is determined by the key length. The longer the key is, the smaller each cell would be.



geohash length	cell size
2	1250Km x 625Km
3	156Km x 156Km
4	39Km x 19.5Km
5	4.89Km x 4.89Km
6	1.22Km x 0.61Km
7	153m x 153m

figure 8. geohash key length and cell size, figure by author

What would be a good key size to use? In practice, the key size is determined iteratively based on the number of drivers and riders. In my opinion, a good starting point would be size 6, as it covers a few city blocks.

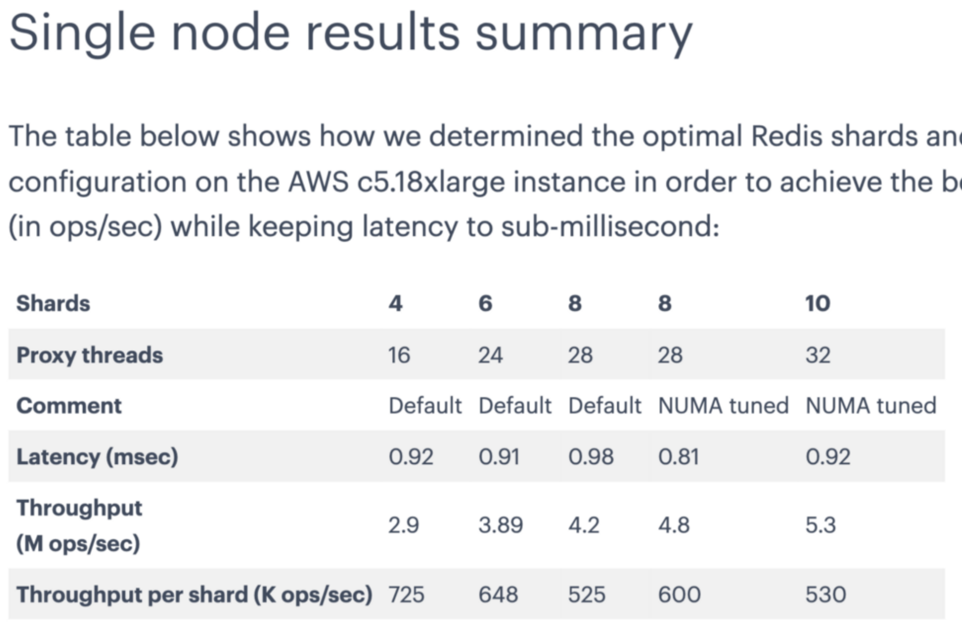


Latitude / Longitude	40.7565 , -73.987
Precision	6 characters
Geohash	dr5ru7

figure 9. size-6 GeoHash offers a good amount of coverage, figure by author

There are physical limitations for the maximum number of drivers and riders that can fit in one of these cells. Going back to figure 5, each cell will be an entry in the Driver Proximity cache in Redis.

You might be wondering if Redis can handle this level of traffic. If there are 1M active riders in the region, the number of requests hitting the cluster is around 200K writes/s (each user updates its location ~5s) and 2M reads/s (each user reads 9 Redis keys ~5s).



Single node results summary

The table below shows how we determined the optimal Redis shards and proxy threads configuration on the AWS c5.18xlarge instance in order to achieve the best possible throughput (in ops/sec) while keeping latency to sub-millisecond:

Shards	**4**	**6**	**8**	**8**	**10**
Proxy threads	16	24	28	28	32
Comment	Default	Default	Default	NUMA tuned	NUMA tuned
Latency (msec)	0.92	0.91	0.98	0.81	0.92
Throughput (M ops/sec)	2.9	3.89	4.2	4.8	5.3
Throughput per shard (K ops/sec)	725	648	525	600	530

Figure 10. Single Redis node capacity, PC by Redis.com

With even one single AWS c5.18xlarge node with 32 threads, the system can handle the traffic. In practice, we can distribute the workload to dozens of computers and achieve ~100M level RPS capacity.

What about memory limitations? If we use size 6 geohash, there are potentially 32^6 ~=10 billion keys in the cache, which is crazy. However, we will never reach this amount of keys if **empty keys are removed** (figure 5). In practice, the number of cache entries is constrained by the number of cars because each car can only be in one cell! Hence, memory is not the bottleneck.

3.2 Location update

Hopefully I've convinced you of the viability of hosting the Location service on Redis. Now let's examine how users update their locations.

There are two tables in the cache — the location truth table and the driver proximity table. The usage of the location truth table is simple. The user's mobile app sends out its location as well as the user ID to the Location service every 5 seconds. Whenever a user's accurate location is needed, the system can query this table by user ID.

The proximity table is more interesting in nature. Drivers move around in their cars, which means they often cross one cell to the other. When the backend receives updates from a driver, it doesn't know what cells they have been in the past. Hence, it is hard to remove drivers from their old cells. The implication is obvious; the same driver can appear in multiple cells because of the stale data.

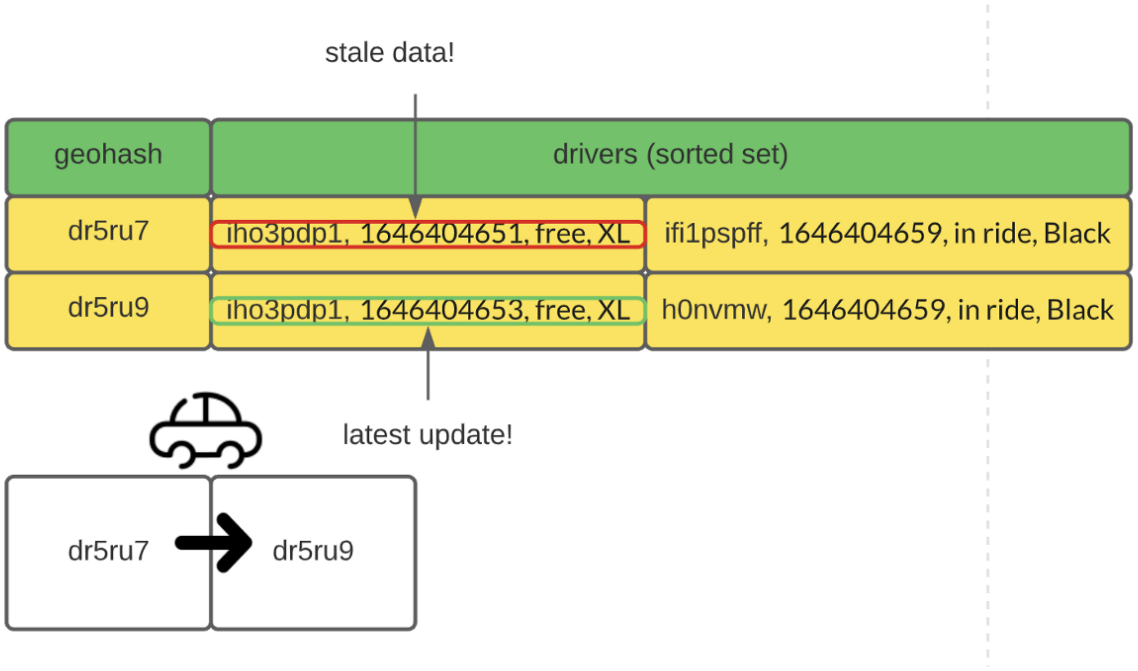

stale data!
geohash
drivers (sorted set)
dr5ru7
iho3pdp1, 1646404651, free, XL
ifi1pspff, 1646404659, in ride, Black
dr5ru9
iho3pdp1, 1646404653, free, XL
h0nvmw, 1646404659, in ride, Black
latest update!
dr5ru7
dr5ru9

Figure 11. stale records in Redis, figure by author

To deal with this problem, we could introduce a timestamp to each record. With timestamps, it is easy to filter out stale location data. In practice, the sorted set data structure in Redis is an efficient way to achieve such a feature.

In addition to the driver's location, we can keep information such as vehicle type, trip status (free, in ride) in the cache. With the additional information, ride-matching can be made quickly since the round trip to the database (query vehicle information and trip status) is skipped.

3.3 Trip recoding

Another important feature of our system is trip recording. For a completed trip, we want to store the route that was taken by the driver and make it available to the client for review.

There are many ways to achieve this feature; the simplest way would be relying on the drivers to report their location as well as their trip status. The Location service will batch write all location updates with *in-trip* status to the database for persistent storage.

Relying on the client app is risky. What if the client app loses all local data while on a trip? It will not know the driver's status before the reboot. To recover from failure, the client app should check if there's any unfinished trip in the database and confirm their status with the driver.

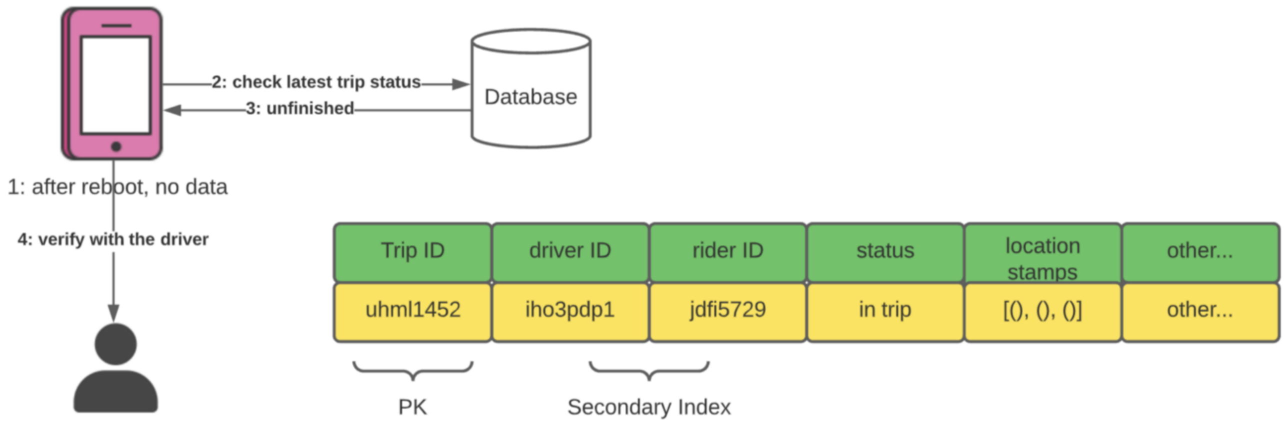


Figure 11. failure recovery, figure by author

4. Summary

In this article, we designed the foundations of ride-sharing apps such as Uber/Lyft. In particular, we've explored in detail how proximity search can be implemented. Keep in mind that there's no such thing as a perfect answer in an interview. All solutions have flaws; our job is to come up with a design that works well enough under the given assumptions and constraints.