System Design - Case Study 5 (Design Uber)

Prerequisite: *System Design - S3 + Quad trees (nearest neighbors)*

We discussed nearest neighbors and quadtrees in the last class and left our discussion with a problem statements:

1. How to design for Uber? (earlier nearest-neighbor problem worked well with static neighbors, here taxis/cabs can move)

Uber: Case Study

We are building uber for intracity, not intercity.

Uber has users and cabs. Users can book a cab, and a cab has two states, either available for hire or unavailable. Only the available cabs will be considered when a user is trying to book.

Use case: I am a user at location X(latitude, longitude); match me with the nearest available cabs. (The notification goes to all nearby cabs for a rider in a round-robin fashion, and a driver can accept or reject). How will you design this?

Let's start our discussion with a question: Suppose there are 10 million cabs worldwide, and I am standing in Mumbai then do I need to care about all the cabs? What is the best sharding key for uber?

You might have guessed that the best sharding key is the city of the cab it belongs to. Cities can be a wider boundary (including more than one region). Every city is a hub and independent from the others. By sharding cabs by city, the scope of the problem will be reduced to a few hundred thousand cabs. Even the busiest cities in India will have around 50 thousand cabs.

Now suppose I am in Mumbai and requesting a cab. If the whole Mumbai region is already broken into grids of equal size, then that would not work well for finding the nearest cabs.
In the Mumbai region, only some areas can be dense (high traffic) while others have less traffic. So breaking Mumbai into equal size grids is not optimal.

- If I am in a heavy traffic area, then I want to avoid matching with cabs that are far(5km) away, but it's fine for sparse traffic areas because cabs will be available very soon.
- So we can say a uniform grid approach of quadtree will not work because different areas have different traffic. It's a function of time and location.

So what can be other approaches?

We can use the fact that cabs can ping their current location after every fixed time period (60 seconds), cab_id → (x,y), and we can maintain this mapping.

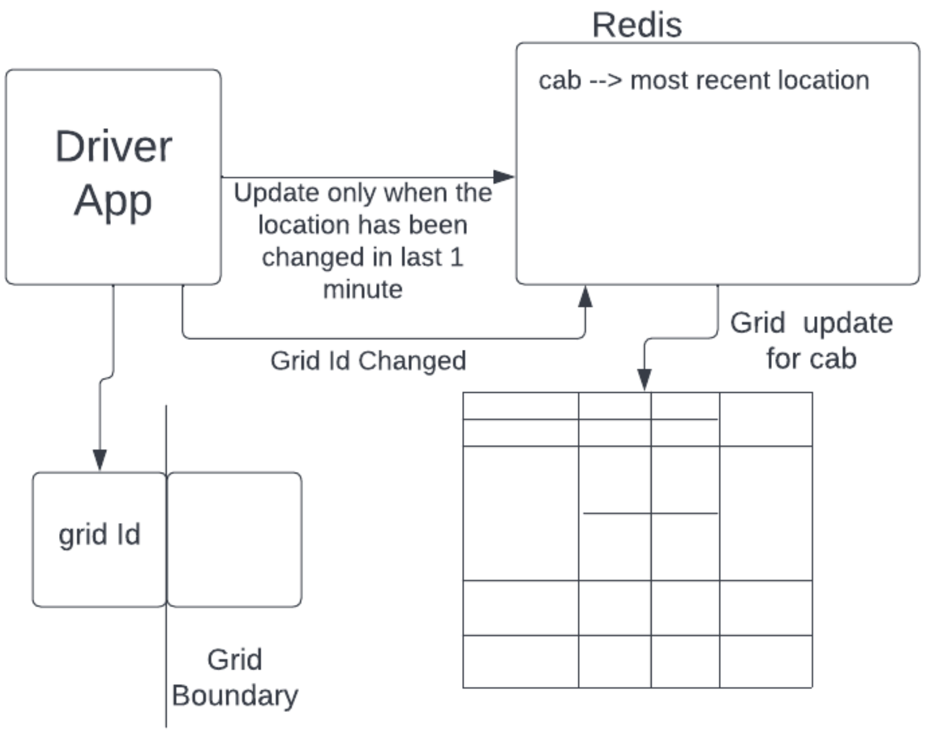
Bruteforce: We go through all the drivers/cabs in the city (since there will only be a few thousand) and then match with the nearest ones.

For an optimized approach, consider this: *If cabs are moving all the time, do we need to update the quadtree every time?*

Initially, we created a quadtree on the traffic pattern with some large and some small grids. When cabs move and change their grids (we get notified by the current location), then we have to update the quadtree (the previous grid and the new grid). There will be a lot of write queries for the update. *How can we optimize it?*

Note: Cabs will only send updates when their location has been changed at the last 1 minute. We can track this on the client side only.

Optimizations:
The driver app tells when the location changes and is aware of grid boundaries.
The optimization can be: cabs already knew the grid they are part of and the boundary. If the cab is moving within the boundary, then there is no need to update the quadtree. When the cab goes into a new grid, the cab_id and corresponding grid can be updated at the backend. We know we can maintain a memory map of cabs and the most recent know locations inside Redis.


Redis
cab --> most recent location
Driver App
Update only when the location has been changed in last 1 minute
Grid Id Changed
Grid update for cab
grid Id
Grid Boundary

We can modify the quadtree by using the above knowledge. However, one problem can occur if we start creating the grids instantaneously: If the number of cabs in a grid is fixed (say 50) and we create four children or merge four children if cabs move in/out of the grid. This will become an expensive operation and will cause more writes. We can also optimize the creation and merging of grids by not changing the grid dimensions instantly.

For example, the threshold for a cab is 50, and the moment we have more the 50 cabs in the grid, we split it into four parts (this was the usual rule).

- What we can do is, instead of fixing the threshold, we can have a range (like 50 to 75). If cabs exceed the range, then we can change the dimensions.
- Another approach could be running a job after every fixed time (3 hours or more) and checking for the number of cabs in leaf nodes. And we can consolidate or split the nodes accordingly. This would work because the traffic pattern doesn't change quickly.

To conclude the design of uber, we can say we need to follow the following steps:

Step 1: For a city, create a quadtree based on current cab locations. (At t = 0, dynamic grids will be created based on the density of cabs).
Step 2: Maintain **cab → (last known location)** and **quadtree** backend for nearest cabs.
Step 3: Don’t bombard quadtree and cab location updates. You can use optimizations:

- Driver app only sends location if cab location is changed in last 1 minute, handling this at client side only
- While sending a new location driver app also checks whether the grid has changed by checking the boundary. For a grid with boundaries (a,b) and (c,d), we can check location (x,y) is inside or not simply by a<=x<=c && d<=y<=b.
- If the grid Id changes, then delete it from the old grid and update it in the new grid.
- Another optimization is, in a quadtree, don't update grids' dimensions immediately. Instead, do it periodically.