9. Design a Location System

Proximity System

Real-life examples

- Yelp

Requirements clarification

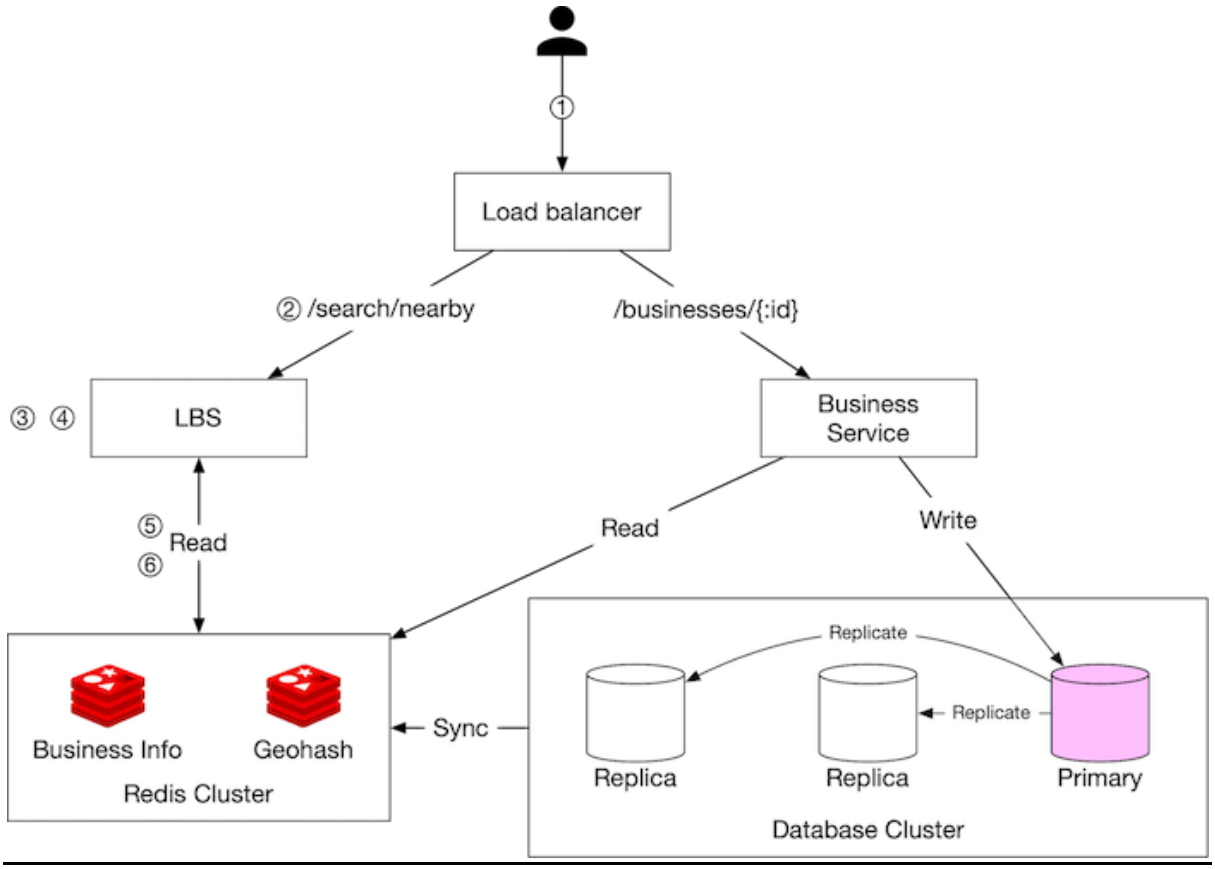
- **Functional requirements**
  - ✓ A user can search businesses by the search radius.
  - ✓ Business owners can add, delete or update a business.
  - ✓ A user can view detailed information about a business.
- **Non-functional requirements**
  - ✓ *Low latency*
  - ✓ Users should be able to see nearby businesses quickly.
  - ✓ *High availability and scalability*
  - ✓ Our system can handle the spike in traffic during peak hours in densely populated areas.

Estimation

- **Traffic estimation**
  - ✓ Read-heavy

Data model definition

High-level design



- **Load balancer**
  - ✓ Distributes incoming traffic across multiple services.
- **Location-based service (LBS)**
  - ✓ Finds nearby businesses for a given radius and location.
  - ✓ Characteristics:
  - ✓ It is a read-heavy service with no write requests.
  - ✓ QPS is high, especially during peak hours in dense areas.
  - ✓ This service is stateless so it's easy to scale horizontally.
- **Business service**
  - ✓ Allows business owners to create, update, or delete businesses.
  - ✓ Allows users to view detailed information about a business.

- **Database cluster**
  - ✓ Uses the primary-secondary setup.
  - ✓ The primary database handles all the write operations
  - ✓ The multiple replicas are used for read operations.

Detailed Design

Algorithms to fetch nearby businesses

Two-dimensional search

Concept

- ✓ Draw a circle with the predefined radius and find all the businesses within the circle.
- ✓ Use the similar query:

```
SELECT business_id, latitude, longitude,FROM businessWHERE (latitude BETWEEN {:my_lat} - radius AND {:my_lat} + radius) AND  (longitude BETWEEN {:my_long} - radius AND {:my_long} + radius)
```

Cons

- ✓ The query is not efficient because we need to scan the whole table.

Evenly divided grid

- ***Concepts***
  - Evenly divide the world into small grids
- ***Cons***
  - The distribution of businesses is not even (New York city vs. deserts).

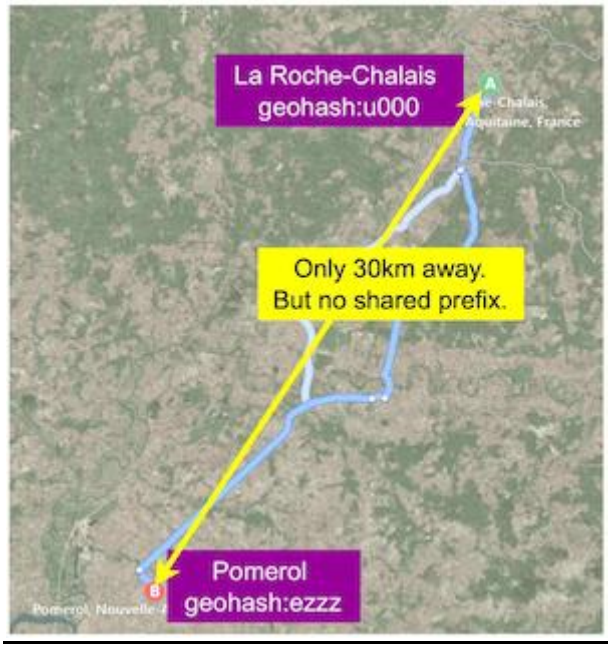
Geohash

ConceptS

- Reduce the two-dimensional longitude and latitude data into a one-dimensional string of letters and digits.
- Recursively Divide the world into smaller and smaller grids with each additional bit.
- The longer a shared prefix is between two geohashes, the closer they are.

Cons

- Have boundary issues:
- Two locations can be very close but have no shared prefix at all.

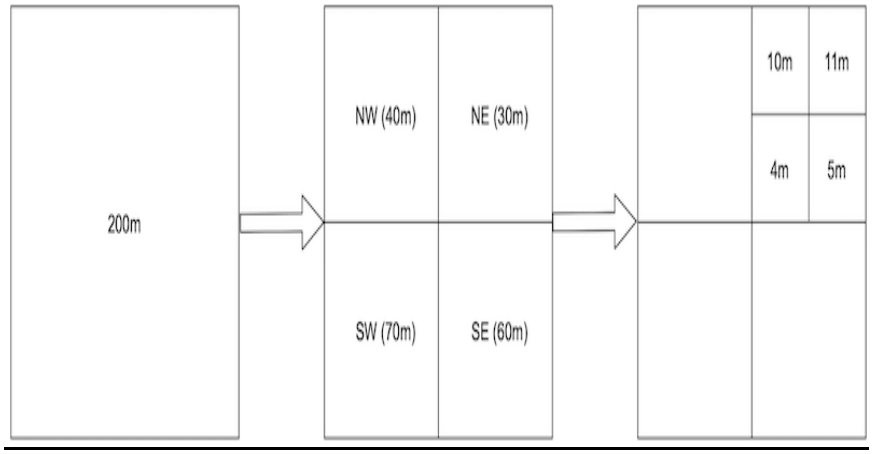


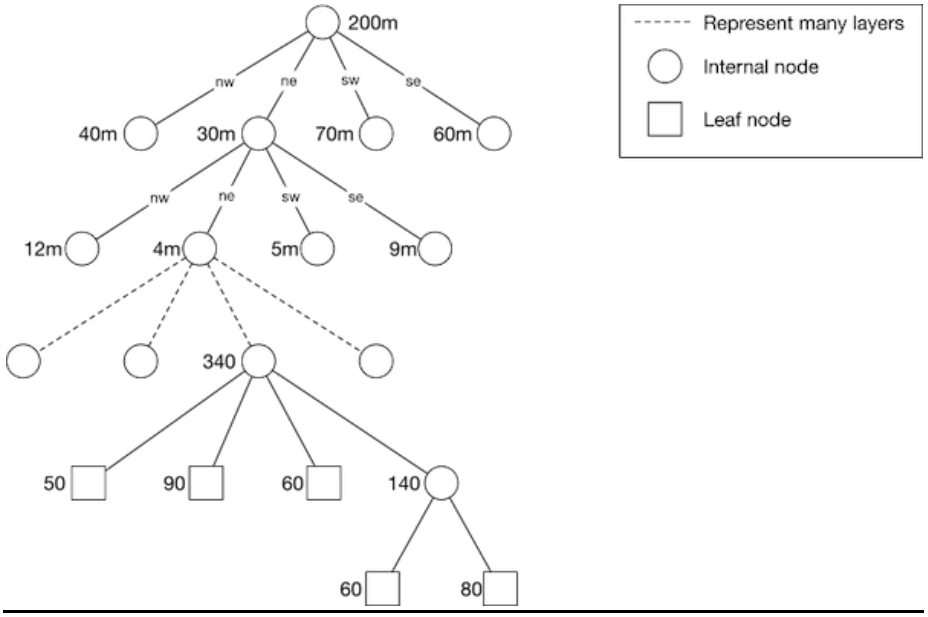
Two locations can have a long shared prefix, but they belong to different geohashes.

Quadtree

Concepts

- Partition a two-dimensional space by recursively subdividing it into four quadrants (grids) until the contents of the grids meet certain criteria.
- Quadtree is an in-memory data structure and it is not a database solution.


200m
NW (40m)
NE (30m)
SW (70m)
SE (60m)
10m
11m
4m
5m


200m
nw
ne
sw
se
40m
30m
70m
60m
12m
4m
5m
9m
340
50
90
60
140
60
80
Represent many layers
Internal node
Leaf node

Google S2

Concepts

- Map a sphere to a 1D index based on the Hilbert curve (a space-filling curve)
- Two points that are close to each other on the Hilbert curve are close in 1D.
- Search on 1D space is much more efficient than on 2D.
- Google S2 is an in-memory solution.

Pros

- S2 is great for geofencing because it can cover arbitrary areas with varying levels (A geofence is a virtual perimeter for a real-world geographic area).
- Region Cover algorithm
- Instead of having a fixed level (precision) as in geohash, we can specify min level, max level, and max cells in S2.
- The result returned by S2 is more granular because the cell sizes are flexible.

Nearby Friends System

Real-life examples

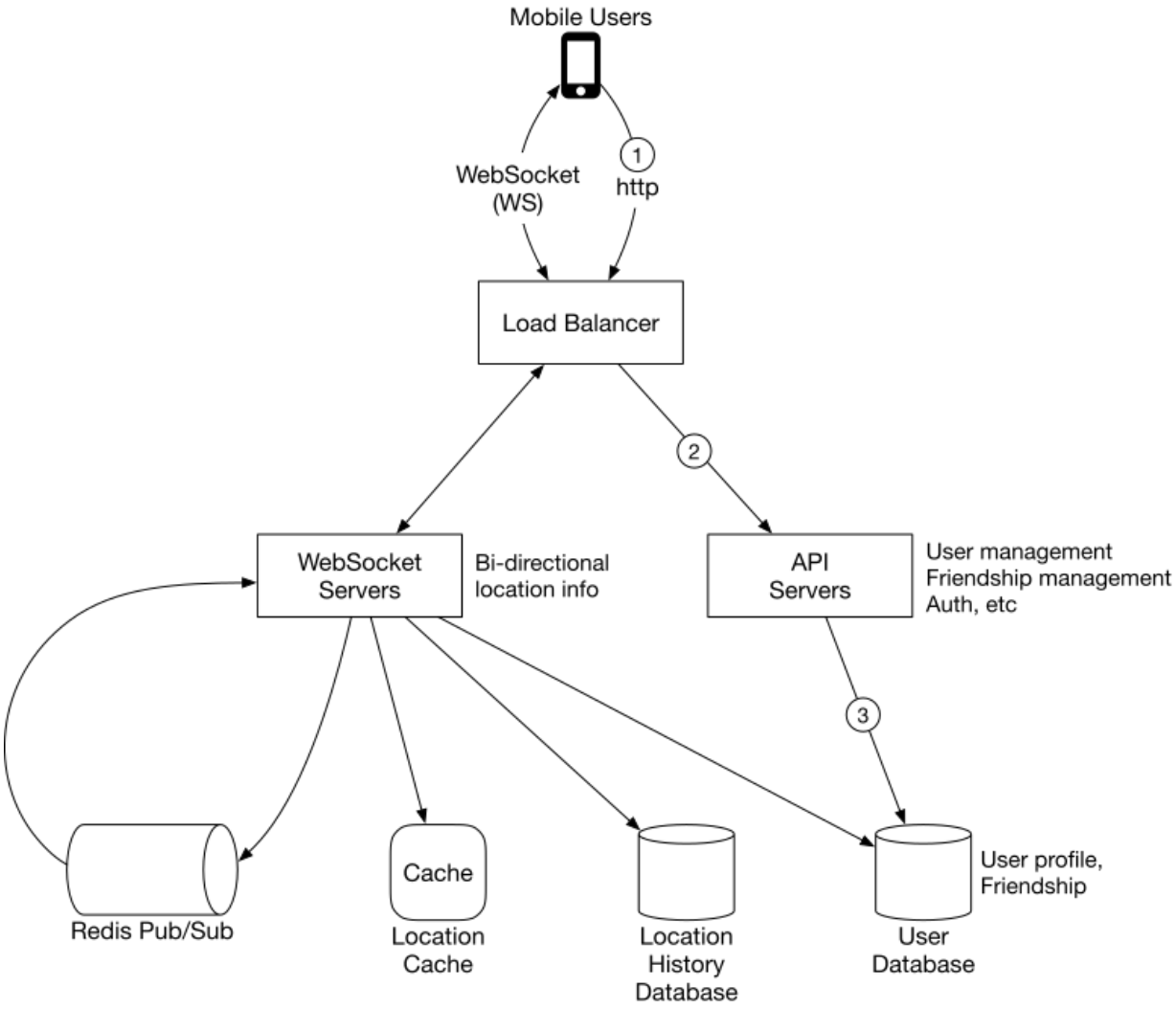
Requirements clarification

- **Functional requirements**
    - Users should be able to see nearby friends on their mobile apps.
    - Nearby friend lists should be updated every few seconds.
- **Non-functional requirements**
    - *Low latency*
    - Receive location updates from friends without too much delay.
    - *Moderate reliability*
    - Occasional data point loss is acceptable.

Eventual consistency

- A few seconds delay in receiving location data in different replicas is acceptable.

High-level design



Load Balancer

- Distributes traffic across those servers to spread out load evenly.

API Servers

- Handles auxiliary requests like adding/removing friends, updating user profiles.

WebSocket Servers

- Handles the near real-time update of friends' locations.
- Handles client initialization for the "nearby friends" feature.

Redis Pub/Sub

- A very lightweight message bus.
- Location updates received via the WebSocket server are published to the user's own channel in the Redis pub/sub server.

Location Cache

- Stores the most recent location data for each active user.
- Sets a Time to Live (TTL) on each entry in the cache.

Location History Database

- Stores users' historical location data (not directly related to the "nearby friends" feature).

User Database

- Stores user data and user friendship data.