VOLUME 2

SYSTEM DESIGN INTERVIEW

AN INSIDER'S GUIDE

ByteByteGo

Alex Xu & Sahn Lam

System Design Interview

An Insider's Guide

Volume 2

Alex Xu | Sahn Lam

ByteByteGo

SYSTEM DESIGN INTERVIEW - AN INSIDER'S GUIDE (VOLUME 2)

Join the community

We created a members-only Discord group. It is designed for community discussions on the following topics:

- System design fundamentals.
- Showcasing design diagrams and getting feedback.
- Finding mock interview buddies.
- General chat with community members.

Come join us and introduce yourself to the community, today! Use the link below or scan the barcode.

Invite link: http://bit.ly/systemdiscord

Contents

Foreword

We are delighted you are joining us to become better equipped for system design interviews. System design interviews are the most difficult to tackle of all technical interview questions. The questions test the interviewees' ability to design a scalable software system. This could be a news feed, Google search, chat application, or any other system. These questions are intimidating and there is no fixed pattern to follow when tackling them. The questions are usually very broad and vague. They are open-ended, with several plausible angles of attack, and often no perfect answer.

Many companies ask system design interview questions because the communication and problem-solving skills they test for are similar to the skills that software engineers use in their daily work. A candidate is evaluated on how they analyze a vague problem and how they solve it, step by step.

System design questions are open-ended. As in the real world, a design can have numerous variations. The desired outcome is an architecture that satisfies the agreed design goals. The discussions may go in different directions. Some interviewers may choose high-level architecture to cover all aspects of the challenge, whereas others might focus on one or more specific areas. Typically, system requirements, constraints, and bottlenecks should be well understood by the candidate, to shape the direction of the interview.

The objective of this book is to provide a reliable strategy and knowledge base for approaching a broad range of system design questions. The right strategy and knowledge are vital for the success of an interview.

This book also provides a step-by-step framework for how to tackle a system design question. It provides many examples to illustrate the systematic approach, with detailed steps that you can follow. With regular practice, you will be well-equipped to tackle system design interview questions.

This book can be seen as a sequel to the book: System Design Interview - An Insider's Guide (Volume 1: https://bit.ly/systemdesigning). Although reading Volume 1 is helpful, it is not a necessity to do so before you read this. This book should be accessible to readers who have a basic understanding of distributed systems. Let's get started!

Additional Resources

This book contains references at the end of each chapter. The following Github repository contains all the clickable links.

https://bit.ly/systemDesignLinks

You can connect with Alex on social media, where he shares system design interview tips every week.

twitter.com/alexxubyte

bit.ly/linkedinaxu

Acknowledgements

We wish we could say all the designs in this book are original. The truth is that most of the ideas discussed here can also be found elsewhere; in engineering blogs, research papers, code, tech talks, and other places. We have collected these elegant ideas and considered them, then added our personal experiences, to present them here in an easy-to-understand way. Additionally, this book has been written with the significant input and reviews of more than a dozen engineers and managers, some of whom made large writing contributions to the chapters. Thank you so much!

- Proximity Service, Meng Duan (Tencent)
- Nearby Friends, Yan Guo (Amazon)
- Google Maps, Ali Aminian (Adobe, Google)
- Distributed Message Queue, Lionel Liu (eBay)
- Distributed Message Queue, Tanmay Deshpande (Schlumberger)
- Ad Click Event Aggregation, Xinda Bian (Ant Group)
- Real-time Gaming Leaderboard, Jossie Haines (Tile)
- Distributed Email Servers, Kevin Henrikson (Instacart)
- Distributed Email Servers, JJ Zhuang (Instacart)
- S3-like Object Store, Zhiteng Huang (eBay)

We are particularly grateful to those who provided detailed feedback on an earlier draft of this book:

- Darshit Dave (Bloomberg)
- Dwaraknath Bakshi (Twitter)
- Fei Nan (Gusto, Airbnb)
- Richard Hsu (Amazon)
- Simon Gao (Google)
- Stanly Mathew Thomas (Microsoft)

- Wenhan Wang (Tiktok)
- Shiwakant Bharti (Amazon)

A huge thanks to our editors, Dominic Gover and Doug Warren. Your feedback was invaluable.

Last but not least, very special thanks to Elvis Ren and Hua Li for their invaluable contributions. This book wouldn't be what it is without them.

1 Proximity Service

In this chapter, we design a proximity service. A proximity service is used to discover nearby places such as restaurants, hotels, theaters, museums, etc., and is a core component that powers features like finding the best restaurants nearby on Yelp or finding k-nearest gas stations on Google Maps. Figure 1.1 shows the user interface via which you can search for nearby restaurants on Yelp [1]. Note the map tiles used in this book are from Stamen Design [2] and data are from OpenStreetMap [3].

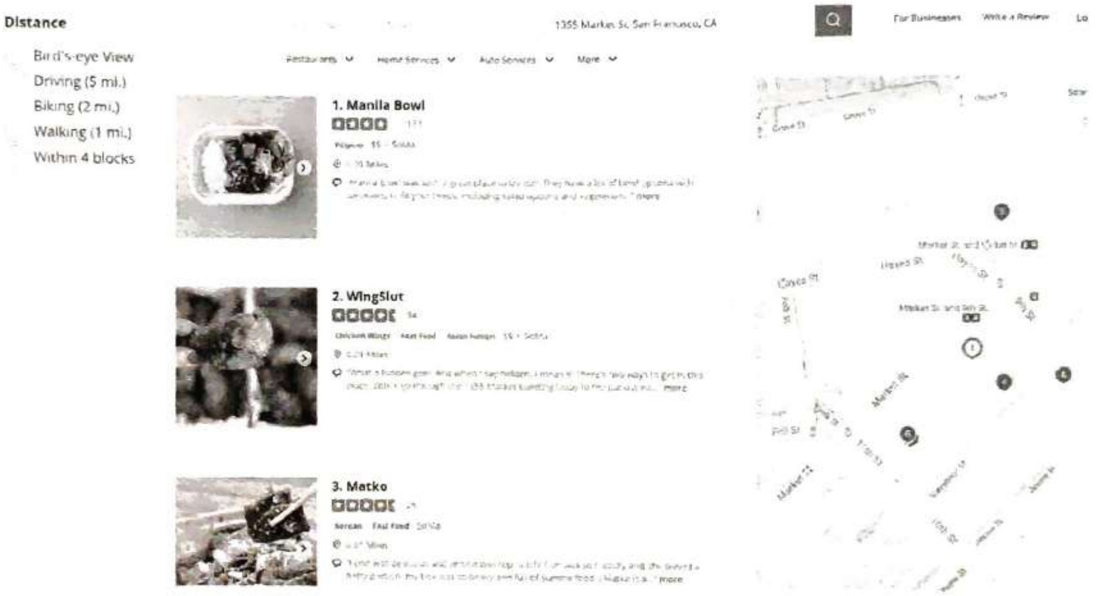


Figure 1.1: Nearby search on Yelp

Step 1 - Understand the Problem and Establish Design Scope

Yelp supports many features and it is not feasible to design all of them in an interview session, so it's important to narrow down the scope by asking questions. The interactions between the interviewer and the candidate could look like this:

Candidate: Can a user specify the search radius? If there are not enough businesses within the search radius, does the system expand the search?

Interviewer: That's a great question. Let's assume we only care about businesses within a specified radius. If time allows, we can then discuss how to expand the search if there are not enough businesses within the radius.

Candidate: What's the maximal radius allowed? Can I assume it's 20km (12.5 miles)?
Interviewer: That's a reasonable assumption.

Candidate: Can a user change the search radius on the UI?
Interviewer: Yes, we have the following options: 0.5km (0.31 mile), 1km (0.62 mile), 2km (1.24 mile), 5km (3.1 mile), and 20km (12.42 mile).

Candidate: How does business information get added, deleted, or updated? Do we need to reflect these operations in real-time?
Interviewer: Business owners can add, delete or update a business. Assume we have a business agreement upfront that newly added/updated businesses will be effective the next day.

Candidate: A user might be moving while using the app/website, so the search results could be slightly different after a while. Do we need to refresh the page to keep the results up to date?
Interviewer: Let's assume a user's moving speed is slow and we don't need to constantly refresh the page.

Functional requirements

Based on this conversation, we focus on 3 key features:

- Return all businesses based on a user's location (latitude and longitude pair) and radius.
- Business owners can add, delete or update a business, but this information doesn't need to be reflected in real-time.
- Customers can view detailed information about a business.

Non-functional requirements

From the business requirements, we can infer a list of non-functional requirements. You should also check these with the interviewer.

- Low latency. Users should be able to see nearby businesses quickly.
- Data privacy. Location info is sensitive data. When we design a location-based service (LBS), we should always take user privacy into consideration. We need to comply with data privacy laws like General Data Protection Regulation (GDPR) [4] and California Consumer Privacy Act (CCPA) [5], etc.
- High availability and scalability requirements. We should ensure our system can handle the spike in traffic during peak hours in densely populated areas.

Back-of-the-envelope estimation

Let's take a look at some back-of-the-envelope calculations to determine the potential scale and challenges our solution will need to address. Assume we have 100 million daily active users and 200 million businesses.

Calculate QPS

- Seconds in a day = $24 \times 60 \times 60 = 86{,}400$. We can round it up to 10^5 for easier calculation. **10^5 is used throughout this book** to represent seconds in a day.
- Assume a user makes 5 search queries per day.
- Search QPS = $\frac{100 \text{ million} \times 5}{10^5} = 5{,}000$

Step 2 - Propose High-level Design and Get Buy-in

In this section, we discuss the following:

- API design
- High-level design
- Algorithms to find nearby businesses
- Data model

API design

We use the RESTful API convention to design a simplified version of the APIs.

`GET /v1/search/nearby`

This endpoint returns businesses based on certain search criteria. In real-life applications, search results are usually paginated. Pagination [6] is not the focus of this chapter, but is worth mentioning during an interview.

Request Parameters:

Field	**Description**	**Type**
`latitude`	Latitude of a given location	decimal
`longitude`	Longitude of a given location	decimal
`radius`	Optional. Default is 5000 meters (about 3 miles)	int

Table 1.1: Request parameters

```
{
  "total": 10,
  "businesses":[{business object}]
}
```

The business object contains everything needed to render the search result page, but we may still need additional attributes such as pictures, reviews, star rating, etc., to render the business detail page. Therefore, when a user clicks on the business detail page, a new endpoint call to fetch the detailed information of a business is usually required.

APIs for a business

The APIs related to a business object are shown in the table below.

API	Detail
`GET /v1/businesses/:id`	Return detailed information about a business
`POST /v1/businesses`	Add a business
`PUT /v1/businesses/:id`	Update details of a business
`DELETE /v1/businesses/:id`	Delete a business

Table 1.2: APIs for a business

If you are interested in real-world APIs for place/business search, two examples are Google Places API [7] and Yelp business endpoints [8].

Data model

In this section, we discuss the read/write ratio and the schema design. The scalability of the database is covered in deep dive.

Read/write ratio

Read volume is high because the following two features are very commonly used:

- Search for nearby businesses.
- View the detailed information of a business.

On the other hand, the write volume is low because adding, removing, and editing business info are infrequent operations.

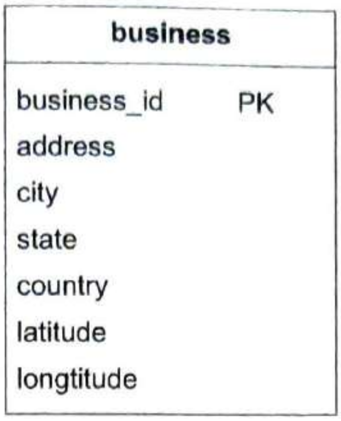
For a read-heavy system, a relational database such as MySQL can be a good fit. Let's take a closer look at the schema design.

Data schema

The key database tables are the business table and the geospatial (geo) index table.

Business table

The business table contains detailed information about a business. It is shown in Table 1.3 and the primary key is `business_id`.



business	
business_id	PK
address	
city	
state	
country	
latitude	
longtitude	

Table 1.3: Business table

Geo index table

A geo index table is used for the efficient processing of spatial operations. Since this table requires some knowledge about geohash, we will discuss it in the "Scale the database" section on page 24.

High-level design

The high-level design diagram is shown in Figure 1.2. The system comprises two parts: location-based service (LBS) and business-related service. Let's take a look at each component of the system.

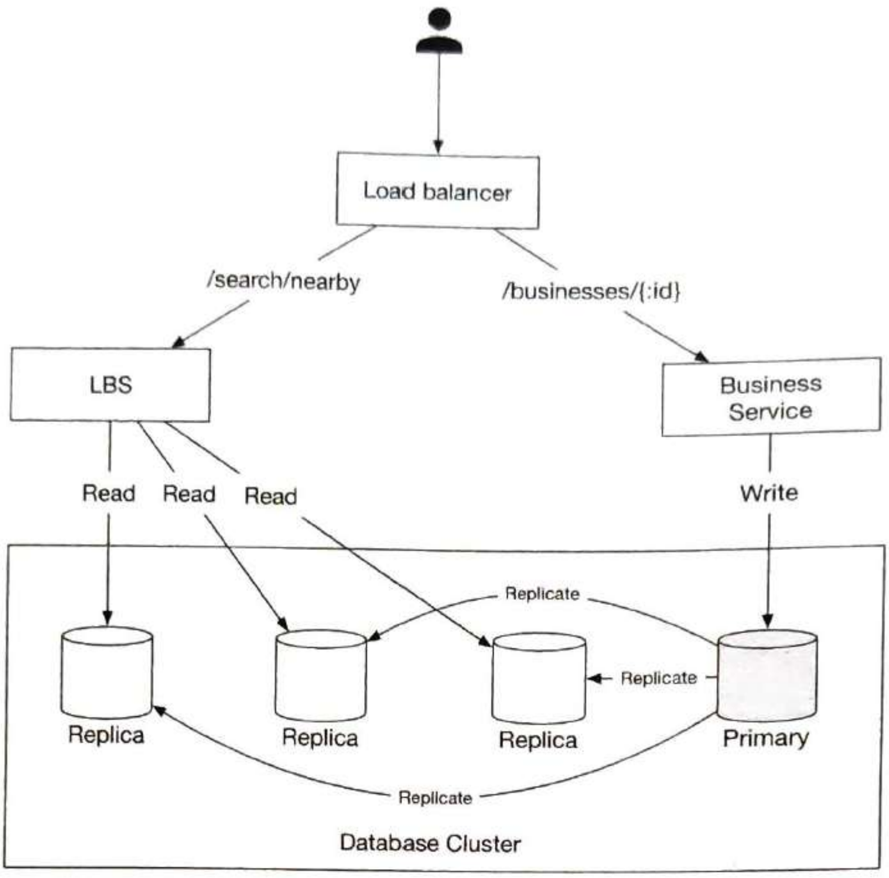


Figure 1.2: High-level design

Load balancer

The load balancer automatically distributes incoming traffic across multiple services. Normally, a company provides a single DNS entry point and internally routes the API calls to the appropriate services based on the URL paths.

Location-based service (LBS)

The LBS service is the core part of the system which finds nearby businesses for a given radius and location. The LBS has the following 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

Business service mainly deals with two types of requests:

- Business owners create, update, or delete businesses. Those requests are mainly write operations, and the QPS is not high.
- Customers view detailed information about a business. QPS is high during peak hours.

Database cluster

The database cluster can use the primary-secondary setup. In this setup, the primary database handles all the write operations, and multiple replicas are used for read operations. Data is saved to the primary database first and then replicated to replicas. Due to the replication delay, there might be some discrepancy between data read by the LBS and the data written to the primary database. This inconsistency is usually not an issue because business information doesn't need to be updated in real-time.

Scalability of business service and LBS

Both the business service and LBS are stateless services, so it's easy to automatically add more servers to accommodate peak traffic (e.g. mealtime) and remove servers during off-peak hours (e.g. sleep time). If the system operates on the cloud, we can set up different regions and availability zones to further improve availability [9]. We discuss this more in the deep dive.

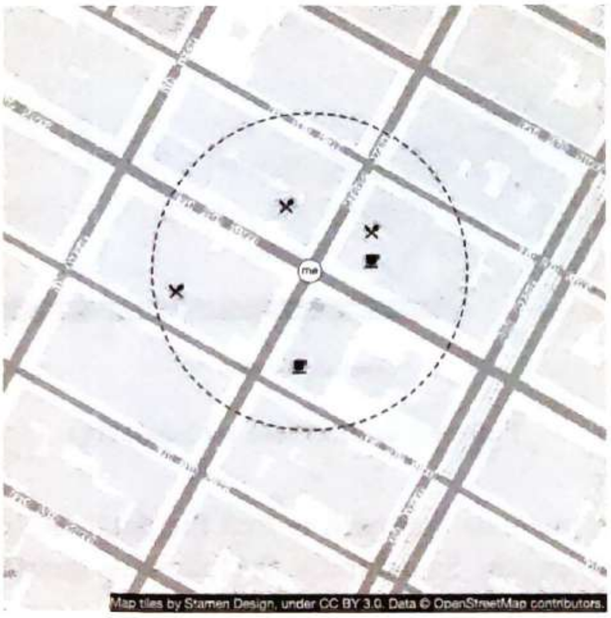
Algorithms to fetch nearby businesses

In real life, companies might use existing geospatial databases such as Geohash in Redis [10] or Postgres with PostGIS extension [11]. You are not expected to know the internals of those geospatial databases during an interview. It's better to demonstrate your problem-solving skills and technical knowledge by explaining how the geospatial index works, rather than to simply throw out database names.

The next step is to explore different options for fetching nearby businesses. We will list a few options, go over the thought process, and discuss trade-offs.

Option 1: Two-dimensional search

The most intuitive but naive way to get nearby businesses is to draw a circle with the predefined radius and find all the businesses within the circle as shown in Figure 1.3.



Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors.

Figure 1.3: Two dimensional search

This process can be translated into the following pseudo SQL query:

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

This query is not efficient because we need to scan the whole table. What if we build indexes on longitude and latitude columns? Would this improve the efficiency? The answer is not by much. The problem is that we have two-dimensional data and the dataset returned from each dimension could still be huge. For example, as shown in Figure 1.4, we can quickly retrieve dataset 1 and dataset 2, thanks to indexes on longitude and latitude columns. But to fetch businesses within the radius, we need to perform an intersect operation on those two datasets. This is not efficient because each dataset contains lots of data.

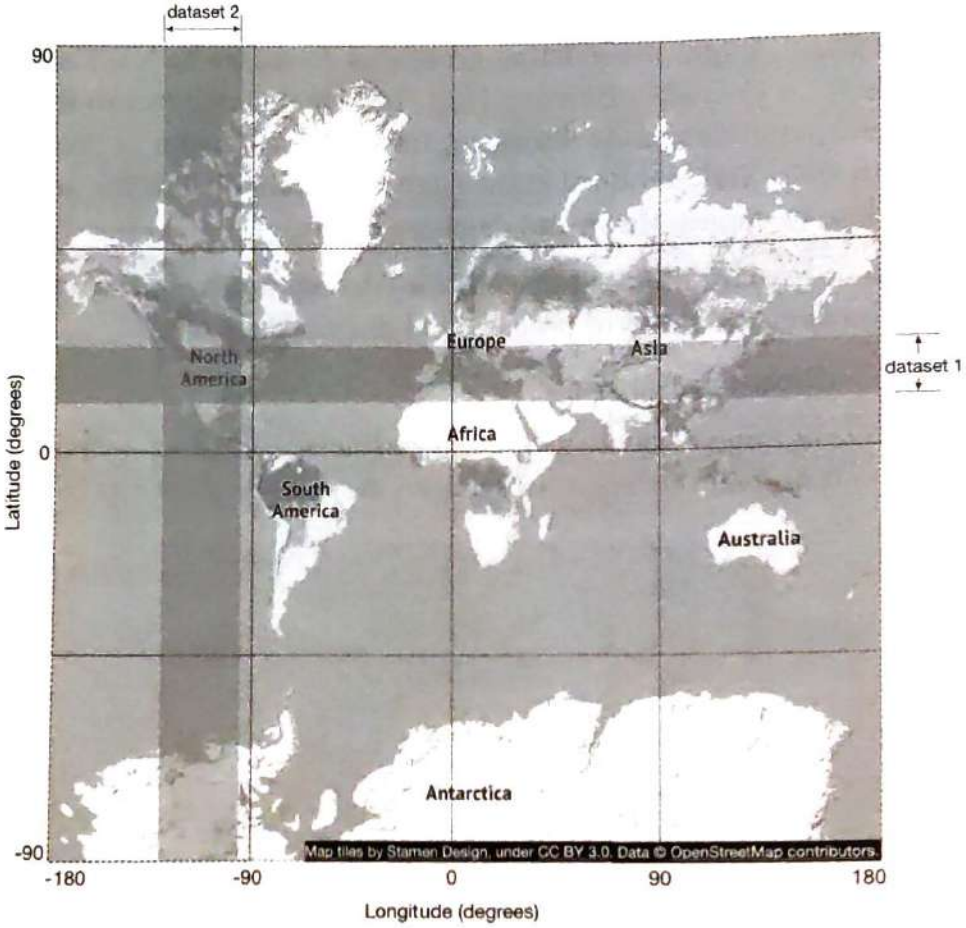


Figure 1.4: Intersect two datasets

The problem with the previous approach is that the database index can only improve search speed in one dimension. So naturally, the follow-up question is, can we map two-dimensional data to one dimension? The answer is yes.

Before we dive into the answers, let's take a look at different types of indexing methods.

In a broad sense, there are two types of geospatial indexing approaches, as shown in Figure 1.5. The highlighted ones are the algorithms we discuss in detail because they are commonly used in the industry.

- Hash: even grid, geohash, cartesian tiers [12], etc.
- Tree: quadtree, Google S2, RTree [13], etc.

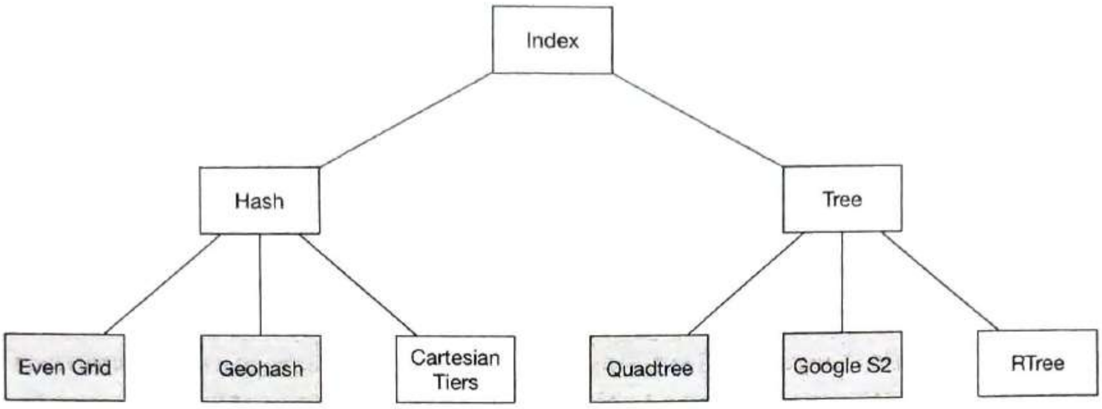


Figure 1.5: Different types of geospatial indexes

Even though the underlying implementations of those approaches are different, the high-level idea is the same, that is, **to divide the map into smaller areas and build indexes for fast search**. Among those, geohash, quadtree, and Google S2 are most widely used in real-world applications. Let's take a look at them one by one.

Reminder
In a real interview, you usually don't need to explain the implementation details of indexing options. However, it is important to have some basic understanding of the need for geospatial indexing, how it works at a high level, and also its limitations.

Option 2: Evenly divided grid

One simple approach is to evenly divide the world into small grids (Figure 1.6). This way, one grid could have multiple businesses, and each business on the map belongs to one grid.

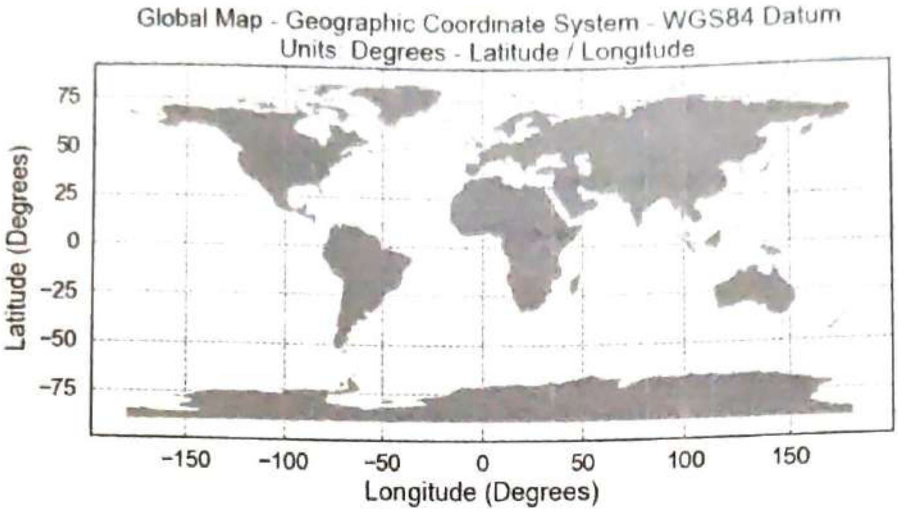

Global Map - Geographic Coordinate System - WGS84 Datum
Units Degrees - Latitude / Longitude
Latitude (Degrees)
75
50
25
0
-25
-50
-75
-150
-100
-50
0
50
100
150
Longitude (Degrees)

Figure 1.6: Global map (source: [14])

This approach works to some extent, but it has one major issue: the distribution of businesses is not even. There could be lots of businesses in downtown New York, while other grids in deserts or oceans have no business at all. By dividing the world into even grids, we produce a very uneven data distribution. Ideally, we want to use more granular grids for dense areas and large grids in sparse areas. Another potential challenge is to find neighboring grids of a fixed grid.

Option 3: Geohash

Geohash is better than the evenly divided grid option. It works by reducing the two-dimensional longitude and latitude data into a one-dimensional string of letters and digits. Geohash algorithms work by recursively dividing the world into smaller and smaller grids with each additional bit. Let's go over how geohash works at a high level.

First, divide the planet into four quadrants along with the prime meridian and equator.

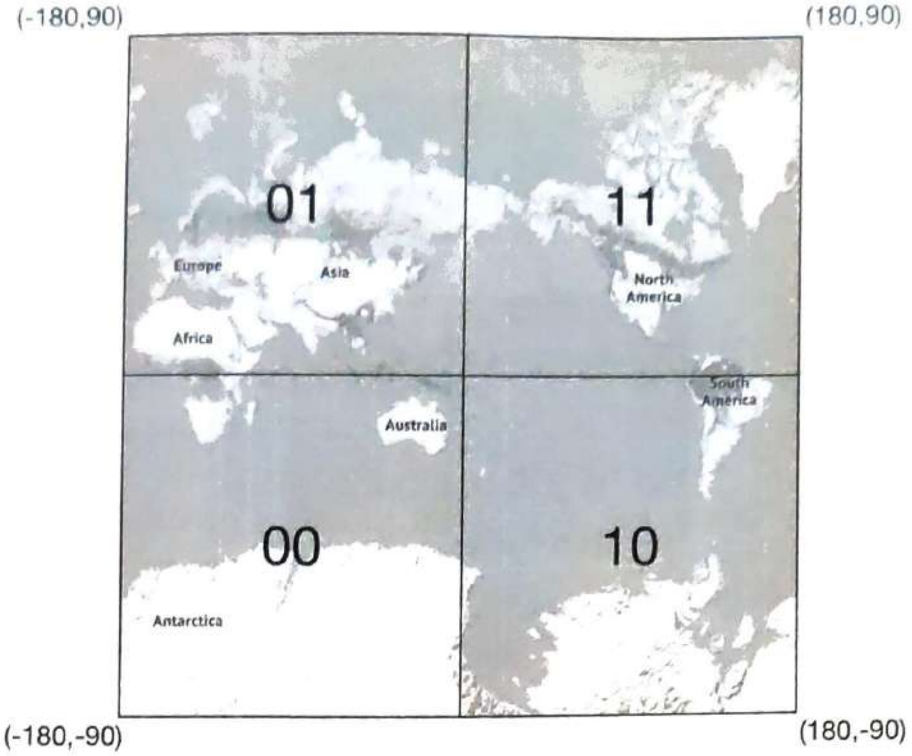


Figure 1.7: Geohash

- Latitude range $[-90, 0]$ is represented by 0
- Latitude range $[0, 90]$ is represented by 1
- Longitude range $[-180, 0]$ is represented by 0
- Longitude range $[0, 180]$ is represented by 1

Second, divide each grid into four smaller grids. Each grid can be represented by alternating between longitude bit and latitude bit.

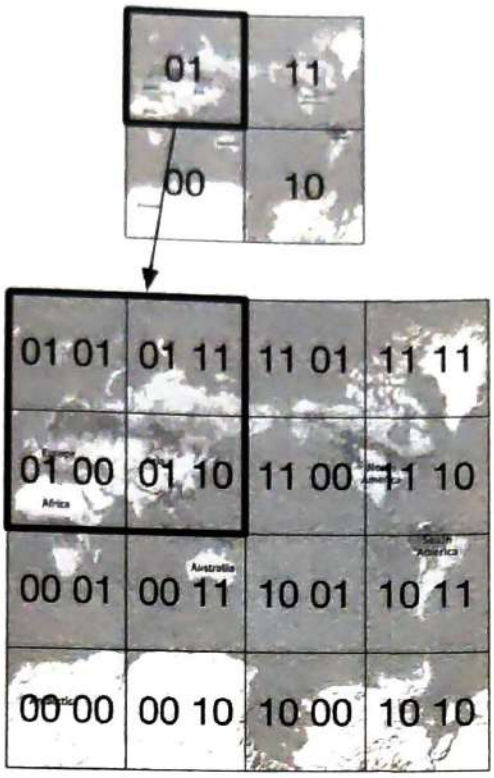


Figure 1.8: Divide grid

Repeat this subdivision until the grid size is within the precision desired. Geohash usually uses base32 representation [15]. Let's take a look at two examples.

- geohash of the Google headquarter (length = 6):
  `1001 10110 01001 10000 11011 11010` (base32 in binary) →
  9q9hvu (base32)
- geohash of the Facebook headquarter (length = 6):
  `1001 10110 01001 10001 10000 10111` (base32 in binary) →
  9q9jhr (base32)

Geohash has 12 precisions (also called levels) as shown in Table 1.4. The precision factor determines the size of the grid. We are only interested in geohashes with lengths between 4 and 6. This is because when it's longer than 6, the grid size is too small, while if it is smaller than 4, the grid size is too large (see Table 1.4).

geohash length	Grid width × height
1	5,009.4km × 4,992.6km (the size of the planet)
2	1,252.3km × 624.1km
3	156.5km × 156km
4	39.1km × 19.5km
5	4.9km × 4.9km
6	1.2km × 609.4m
7	152.9m × 152.4m
8	38.2m × 19m
9	4.8m × 4.8m
10	1.2m × 59.5cm
11	14.9cm × 14.9cm
12	3.7cm × 1.9cm

Table 1.4: Geohash length to grid size mapping (source: [16])

How do we choose the right precision? We want to find the minimal geohash length that covers the whole circle drawn by the user-defined radius. The corresponding relationship between the radius and the length of geohash is shown in the table below.

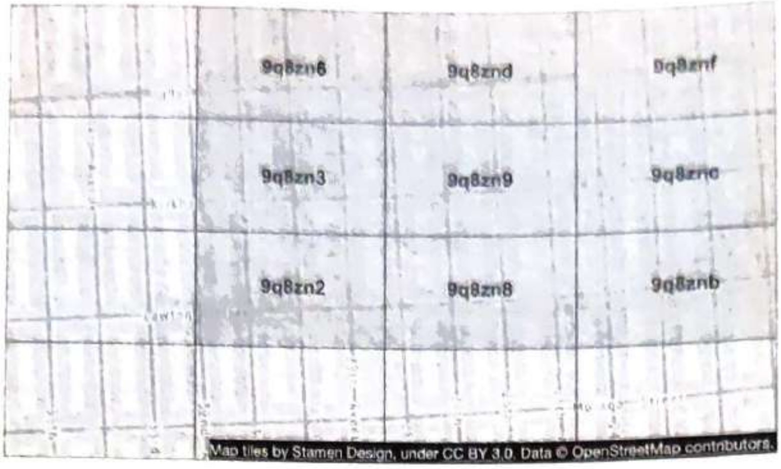
Radius (Kilometers)	Geohash length
0.5km (0.31 mile)	6
1km (0.62 mile)	5
2km (1.24 mile)	5
5km (3.1 mile)	4
20km (12.42 mile)	4

Table 1.5: Radius to geohash mapping

This approach works great most of the time, but there are some edge cases with how the geohash boundary is handled that we should discuss with the interviewer.

Boundary issues

Geohashing guarantees that the longer a shared prefix is between two geohashes, the closer they are. As shown in Figure 1.9, all the grids have a shared prefix: 9q8zn.



Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors.

Figure 1.9: Shared prefix

Boundary issue 1

However, the reverse is not true: two locations can be very close but have no shared prefix at all. This is because two close locations on either side of the equator or prime meridian belong to different "halves" of the world. For example, in France, La Roche-Chalais (geohash: `u000`) is just 30km from Pomerol (geohash: `ezzz`) but their geohashes have no shared prefix at all [17].

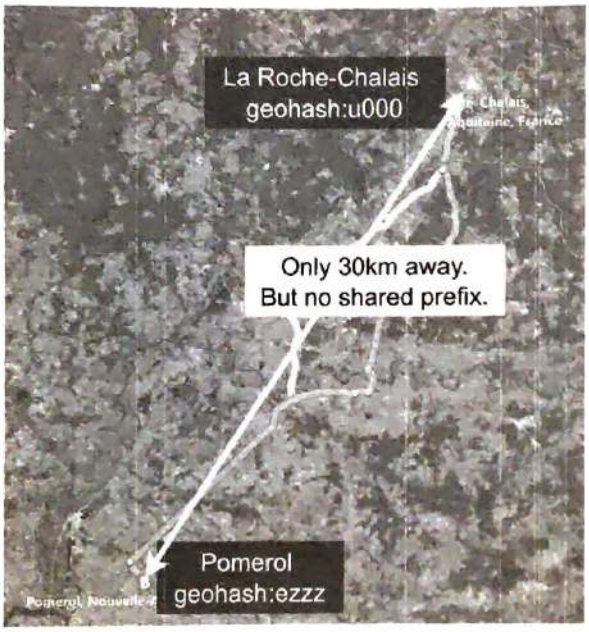


Figure 1.10: No shared prefix

Because of this boundary issue, a simple prefix SQL query below would fail to fetch all nearby businesses.

```
SELECT * FROM geohash_index WHERE geohash LIKE '9q8zn%'
```

Boundary issue 2

Another boundary issue is that two positions can have a long shared prefix, but they belong to different geohashes as shown in Figure 1.11.

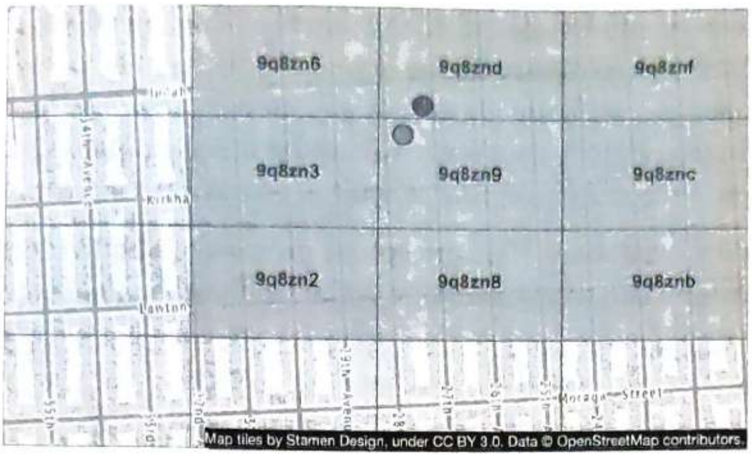

9q8zn6
9q8znd
9q8znf
9q8zn3
9q8zn9
9q8znc
9q8zn2
9q8zn8
9q8znb
Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors.

Figure 1.11: Boundary issue

A common solution is to fetch all businesses not only within the current grid but also from its neighbors. The geohashes of neighbors can be calculated in constant time and more details about this can be found here [17].

Not enough businesses

Now let's tackle the bonus question. What should we do if there are not enough businesses returned from the current grid and all the neighbors combined?

Option 1: only return businesses within the radius. This option is easy to implement, but the drawback is obvious. It doesn't return enough results to satisfy a user's needs.

Option 2: increase the search radius. We can remove the last digit of the geohash and use the new geohash to fetch nearby businesses. If there are not enough businesses, we continue to expand the scope by removing another digit. This way, the grid size is gradually expanded until the result is greater than the desired number of results. Figure 1.12 shows the conceptual diagram of the expanding search process.

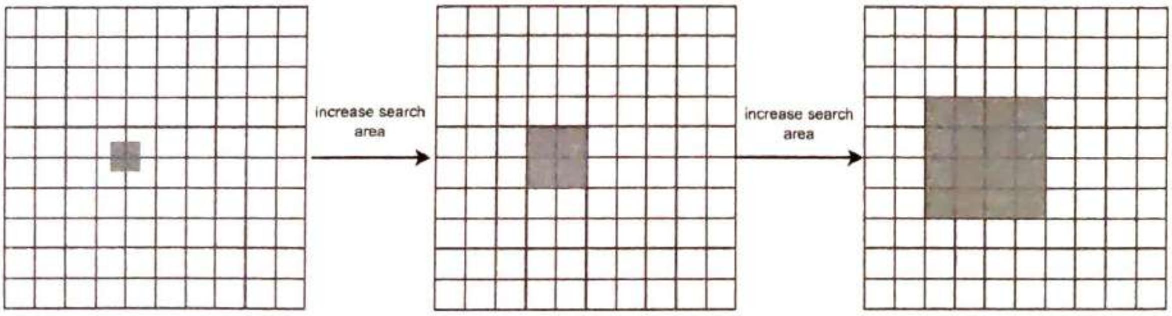


Figure 1.12: Expand the search process

Option 4: Quadtree

Another popular solution is quadtree. A quadtree [18] is a data structure that is commonly used to partition a two-dimensional space by recursively subdividing it into four quadrants (grids) until the contents of the grids meet certain criteria. For example, the criterion can be to keep subdividing until the number of businesses in the grid is not more than 100. This number is arbitrary as the actual number can be determined by business needs. With a quadtree, we build an in-memory tree structure to answer queries. Note that quadtree is an in-memory data structure and it is not a database solution. It runs on each LBS server, and the data structure is built at server start-up time.

The following figure visualizes the conceptual process of subdividing the world into a quadtree. Let's assume the world contains 200m (million) businesses.

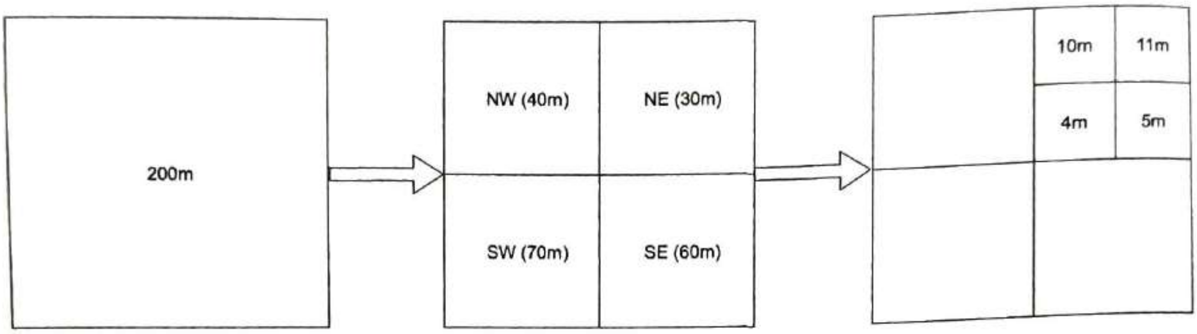


Figure 1.13: Quadtree

Figure 1.14 explains the quadtree building process in more detail. The root node represents the whole world map. The root node is recursively broken down into 4 quadrants until no nodes are left with more than 100 businesses.

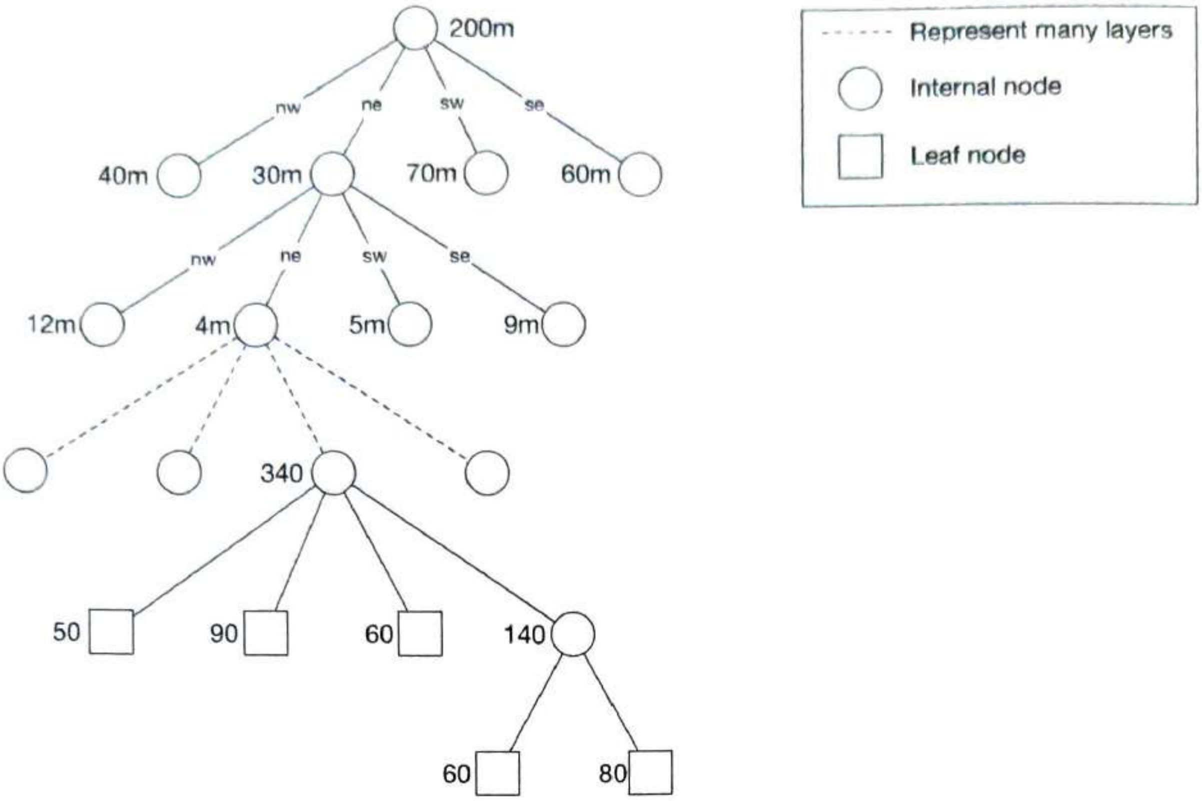


Figure 1.14: Build quadtree

The pseudocode for building quadtree is shown below:

```
public void buildQuadtree(TreeNode node) {
  if (countNumberOfBusinessesInCurrentGrid(node) > 100) {
    node.subdivide();
    for (TreeNode child : node.getChildren()) {
      buildQuadtree(child);
    }
  }
}
```

How much memory does it need to store the whole quadtree?

To answer this question, we need to know what kind of data is stored.

Data on a leaf node

Name	Size
Top left coordinates and bottom-right coordinates to identify the grid	32 bytes (8 bytes ×4)
List of business IDs in the grid	8 bytes per ID × 100 (maximal number of businesses allowed in one grid)
Total	832 bytes

Table 1.6: Leaf node

Data on internal node

Name	Size
Top left coordinates and bottom-right coordinates to identify the grid	32 bytes (8 bytes ×4)
Pointers to 4 children	32 bytes (8 bytes ×4)
Total	64 bytes

Table 1.7: Internal node

Even though the tree-building process depends on the number of businesses within a grid, this number does not need to be stored in the quadtree node because it can be inferred from records in the database.

Now that we know the data structure for each node, let's take a look at the memory usage.

- Each grid can store a maximal of 100 businesses
- Number of leaf nodes $=\sim \frac{200 \text{ million}}{100} =\sim 2$ million
- Number of internal nodes $= 2 \text{ million} \times \frac{1}{3} =\sim 0.67$ million. If you do not know why the number of internal nodes is one-third of the leaf nodes, please read the reference material [19].
- Total memory requirement $= 2 \text{ million} \times 832 \text{ bytes} + 0.67 \text{ million} \times 64 \text{ bytes} = \sim$ 1.71GB. Even if we add some overhead to build the tree, the memory requirement to build the tree is quite small.

In a real interview, we shouldn't need such detailed calculations. The key takeaway here is that the quadtree index doesn't take too much memory and can easily fit in one server. Does it mean we should use only one server to store the quadtree index? The answer is no. Depending on the read volume, a single quadtree server might not have enough CPU or network bandwidth to serve all read requests. If that is the case, it will be necessary to spread the read load among multiple quadtree servers.

How long does it take to build the whole quadtree?

Each leaf node contains approximately 100 business IDs. The time complexity to build the tree is $\frac{n}{100} \log \frac{n}{100}$, where n is the total number of businesses. It might take a few minutes to build the whole quadtree with 200 million businesses.

How to get nearby businesses with quadtree?

1. Build the quadtree in memory.
2. After the quadtree is built, start searching from the root and traverse the tree, until we find the leaf node where the search origin is. If that leaf node has 100 businesses, return the node. Otherwise, add businesses from its neighbors until enough businesses are returned.

Operational considerations for quadtree

As mentioned above, it may take a few minutes to build a quadtree with 200 million businesses at the server start-up time. It is important to consider the operational implications of such a long server start-up time. While the quadtree is being built, the server cannot serve traffic. Therefore, we should roll out a new release of the server incrementally to a small subset of servers at a time. This avoids taking a large swath of the server cluster offline and causes service brownout. Blue/green deployment [20] can also be used, but an entire cluster of new servers fetching 200 million businesses at the same time from the database service can put a lot of strain on the system. This can be done, but it may complicate the design and you should mention that in the interview.

Another operational consideration is how to update the quadtree as businesses are added and removed over time. The easiest approach would be to incrementally rebuild the quadtree, a small subset of servers at a time, across the entire cluster. But this would mean some servers would return stale data for a short period of time. However, this is generally an acceptable compromise based on the requirements. This can be further mitigated by setting up a business agreement that newly added/updated businesses will only be effective the next day. This means we can update the cache using a nightly job. One potential problem with this approach is that tons of keys will be invalidated at the same time, causing heavy load on cache servers.

It's also possible to update the quadtree on the fly as businesses are added and removed. This certainly complicates the design, especially if the quadtree data structure could be accessed by multiple threads. This will require some locking mechanism which could dramatically complicate the quadtree implementation.

Real-world quadtree example

Yext [21] provided an image (Figure 1.15) that shows a constructed quadtree near Denver [21]. We want smaller, more granular grids for dense areas and larger grids for sparse areas.

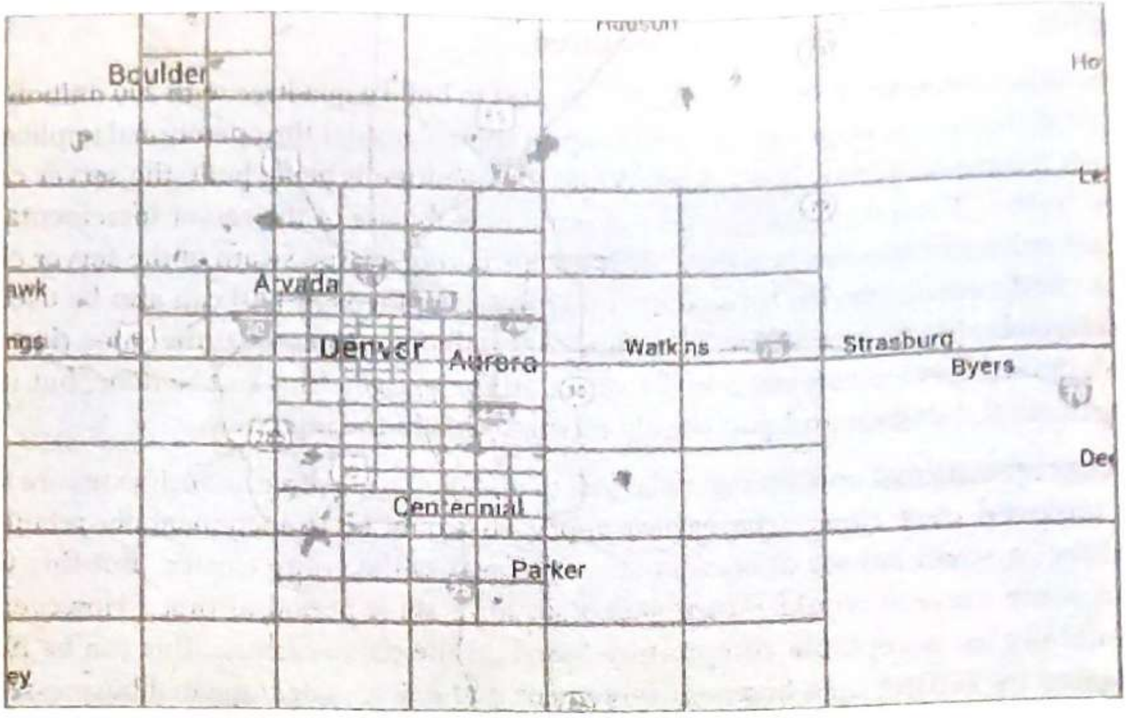


Figure 1.15: Real-world example of a quadtree

Option 5: Google S2

Google S2 geometry library [22] is another big player in this field. Similar to Quadtree, it is an in-memory solution. It maps a sphere to a 1D index based on the Hilbert curve (a space-filling curve) [23]. The Hilbert curve has a very important property: two points that are close to each other on the Hilbert curve are close in 1D space (Figure 1.16). Search on 1D space is much more efficient than on 2D. Interested readers can play with an online tool [24] for the Hilbert curve.

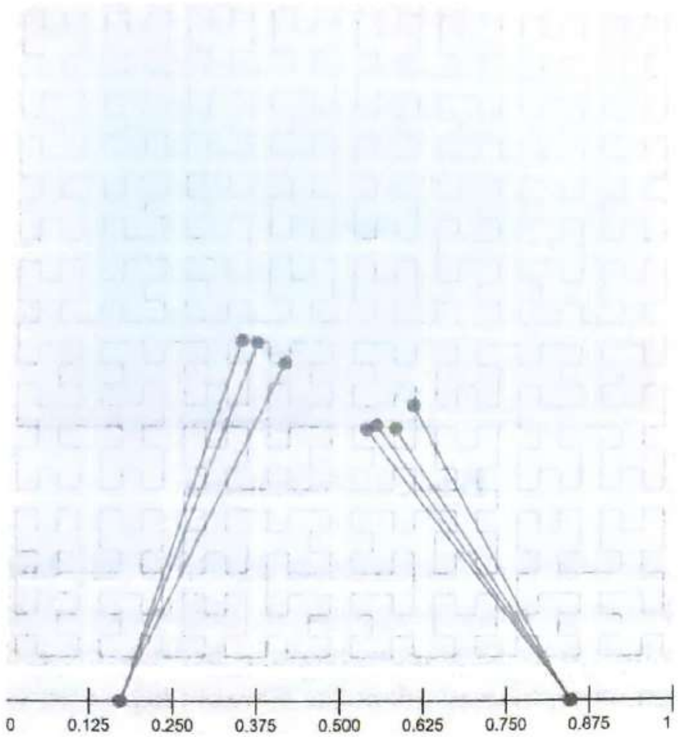


Figure 1.16: Hilbert curve (source: [24])

S2 is a complicated library and you are not expected to explain its internals during an interview. But because it's widely used in companies such as Google, Tinder, etc., we will briefly cover its advantages.

- S2 is great for geofencing because it can cover arbitrary areas with varying levels (Figure 1.17). According to Wikipedia, "A geofence is a virtual perimeter for a real-world geographic area. A geo-fence could be dynamically generated—as in a radius around a point location, or a geo-fence can be a predefined set of boundaries (such as school zones or neighborhood boundaries)" [25].

  Geofencing allows us to define perimeters that surround the areas of interest and to send notifications to users who are out of the areas. This can provide richer functionalities than just returning nearby businesses.

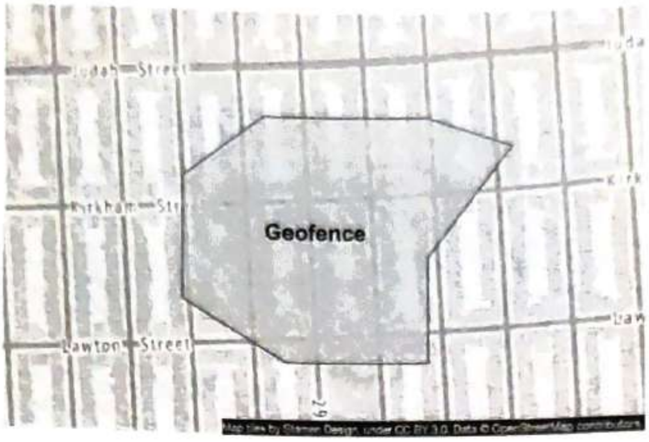

Geofence
Judah Street
Kirkham St
Lawton Street
Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors

Figure 1.17: Geofence

- Another advantage of S2 is its Region Cover algorithm [26]. 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. If you want to learn more, take a look at the S2 tool [26].

Recommendation

To find nearby businesses efficiently, we have discussed a few options: geohash, quadtree and S2. As you can see from Table 1.8, different companies or technologies adopt different options.

Geo Index	Companies
geohash	Bing map [27], Redis [10], MongoDB [28], Lyft [29]
quadtree	Yext [21]
Both geohash and quadtree	Elasticsearch [30]
S2	Google Maps, Tinder [31]

Table 1.8: Different types of geo indexes

During an interview, we suggest choosing **geohash or quadtree** because S2 is more complicated to explain clearly in an interview.

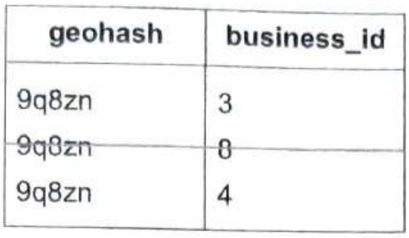
Geohash vs quadtree

Before we conclude this section, let's do a quick comparison between geohash and quadtree.

Geohash

- Easy to use and implement. No need to build a tree.
- Supports returning businesses within a specified radius.
- When the precision (level) of geohash is fixed, the size of the grid is fixed as well. It cannot dynamically adjust the grid size, based on population density. More complex logic is needed to support this.
- Updating the index is easy. For example, to remove a business from the index,

we just need to remove it from the corresponding row with the same geohash and business_id. See Figure 1.18 for a concrete example.



geohash	business_id
9q8zn	3
~~9q8zn~~	~~8~~
9q8zn	4

Figure 1.18: Remove a business

Quadtree

- Slightly harder to implement because it needs to build the tree.
- Supports fetching k-nearest businesses. Sometimes we just want to return k-nearest businesses and don't care if businesses are within a specified radius. For example, when you are traveling and your car is low on gas, you just want to find the nearest k gas stations. These gas stations may not be near you, but the app needs to return the nearest k results. For this type of query, a quadtree is a good fit because its subdividing process is based on the number k and it can automatically adjust the query range until it returns k results.
- It can dynamically adjust the grid size based on population density (see the Denver example in Figure 1.15).
- Updating the index is more complicated than geohash. A quadtree is a tree structure. If a business is removed, we need to traverse from the root to the leaf node, to remove the business. For example, if we want to remove the business with ID $= 2$, we have to travel from the root all the way down to the leaf node, as shown in Figure 1.19. Updating the index takes $O(\log n)$, but the implementation is complicated if the data structure is accessed by a multi-threaded program, as locking is required. Also, rebalancing the tree can be complicated. Rebalancing is necessary if, for example, a leaf node has no room for a new addition. A possible fix is to over-allocate the ranges.

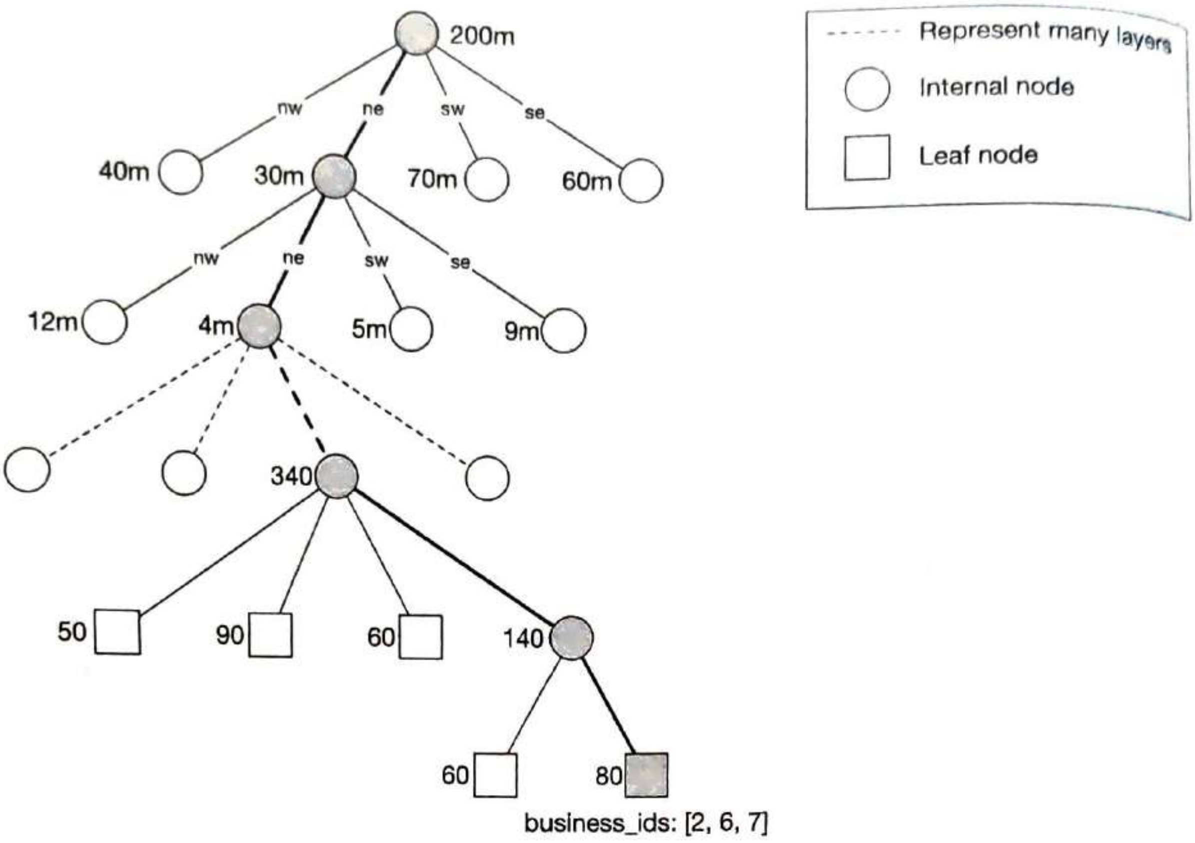


Figure 1.19: Update quadtree

Step 3 - Design Deep Dive

By now you should have a good picture of what the overall system looks like. Now let's dive deeper into a few areas.

- Scale the database
- Caching
- Region and availability zones
- Filter results by time or business type
- Final architecture diagram

Scale the database

We will discuss how to scale two of the most important tables: the business table and the geospatial index table.

Business table

The data for the business table may not all fit in one server, so it is a good candidate for sharding. The easiest approach is to shard everything by business ID. This sharding scheme ensures that load is evenly distributed among all the shards, and operationally it is easy to maintain.

Geospatial index table

Both geohash and quadtree are widely used. Due to geohash's simplicity, we use it as an example. There are two ways to structure the table.

Option 1: For each geohash key, there is a JSON array of business IDs in a single row. This means all business IDs within a geohash are stored in one row.

geospatial_index
geohash
list_of_business_ids

Table 1.9: `list_of_business_ids` is a JSON array

Option 2: If there are multiple businesses in the same geohash, there will be multiple rows, one for each business. This means different business IDs within a geohash are stored in different rows.

geospatial_index
geohash
business_id

Table 1.10: `business_id` is a single ID

Here are some sample rows for option 2.

geohash	business_id
32feac	343
32feac	347
f31cad	112
f31cad	113

Table 1.11: Sample rows of the geospatial index table

Recommendation: we recommend option 2 because of the following reasons:

For option 1, to update a business, we need to fetch the array of `business_ids` and scan the whole array to find the business to update. When inserting a new business, we have to scan the entire array to make sure there is no duplicate. We also need to lock the row to prevent concurrent updates. There are a lot of edge cases to handle.

For option 2, if we have two columns with a compound key of (`geohash, business_id`), the addition and removal of a business are very simple. There would be no need to lock anything.

Scale the geospatial index

One common mistake about scaling the geospatial index is to quickly jump to a sharding scheme without considering the actual data size of the table. In our case, the full dataset for the geospatial index table is not large (quadtree index only takes 1.71G memory and storage requirement for geohash index is similar). The whole geospatial index can easily fit in the working set of a modern database server. However, depending on the read volume, a single database server might not have enough CPU or network bandwidth to handle all read requests. If that is the case, it is necessary to spread the read load among multiple database servers.

There are two general approaches for spreading the load of a relational database server. We can add read replicas, or shard the database.

Many engineers like to talk about sharding during interviews. However, it might not be a good fit for the geohash table as sharding is complicated. For instance, the sharding logic has to be added to the application layer. Sometimes, sharding is the only option. In this case, though, everything can fit in the working set of a database server, so there is no strong technical reason to shard the data among multiple servers.

A better approach, in this case, is to have a series of read replicas to help with the read load. This method is much simpler to develop and maintain. For this reason, scaling the geospatial index table through replicas is recommended.

Caching

Before introducing a cache layer we have to ask ourselves, do we really need a cache layer?

It is not immediately obvious that caching is a solid win:

- The workload is read-heavy, and the dataset is relatively small. The data could fit in the working set of any modern database server. Therefore, the queries are not I/O bound and they should run almost as fast as an in-memory cache.
- If read performance is a bottleneck, we can add database read replicas to improve the read throughput.

Be mindful when discussing caching with the interviewer, as it will require careful benchmarking and cost analysis. If you find out that caching does fit the business requirements, then you can proceed with discussions about caching strategy.

Cache key

The most straightforward cache key choice is the location coordinates (latitude and longitude) of the user. However, this choice has a few issues:

- Location coordinates returned from mobile phones are not accurate as they are just the best estimation [32]. Even if you don't move, the results might be slightly different each time you fetch coordinates on your phone.
- A user can move from one location to another, causing location coordinates to change slightly. For most applications, this change is not meaningful.

Therefore, location coordinates are not a good cache key. Ideally, small changes in location should still map to the same cache key. The geohash/quadtree solution mentioned earlier handles this problem well because all businesses within a grid map to the same geohash.

Types of data to cache

As shown in Table 1.12, there are two types of data that can be cached to improve the overall performance of the system:

Key	Value
geohash	List of business IDs in the grid
business_id	Business object

Table 1.12: Key-value pairs in cache

List of business IDs in a grid

Since business data is relatively stable, we precompute the list of business IDs for a given geohash and store it in a key-value store such as Redis. Let's take a look at a concrete example of getting nearby businesses with caching enabled.

1. Get the list of business IDs for a given geohash.

```
SELECT business_id FROM geohash_index WHERE geohash LIKE `{:
geohash}%`
```

2. Store the result in the Redis cache if cache misses.

```
public List<String> getNearbyBusinessIds(String geohash) {
  String cacheKey = hash(geohash);
  List<string> listOfBusinessIds = Redis.get(cacheKey);
  if (listOfBusinessIDs == null) {
    listOfBusinessIds = Run the select SQL query above;
    Cache.set(cacheKey, listOfBusinessIds, "1d");
  }
  return listOfBusinessIds;
}
```

When a new business is added, edited, or deleted, the database is updated and the cache invalidated. Since the volume of those operations is relatively small and no locking mechanism is needed for the geohash approach, update operations are easy to deal with.

According to the requirements, a user can choose the following 4 radii on the client: 500m, 1km, 2km, and 5km. Those radii are mapped to geohash lengths of 4, 5, 5, and 6, respectively. To quickly fetch nearby businesses for different radii, we cache data in Redis on all three precisions (geohash_4, geohash_5, and geohash_6).

As mentioned earlier, we have 200 million businesses and each business belongs to 1 grid in a given precision. Therefore the total memory required is:

- Storage for Redis values: 8 bytes × 200 million × 3 precisions = $\sim$ 5GB
- Storage for Redis keys: negligible

- Total memory required: ~ 5GB

We can get away with one modern Redis server from the memory usage perspective, but to ensure high availability and reduce cross continent latency, we deploy the Redis cluster across the globe. Given the estimated data size, we can have the same copy of cache data deployed globally. We call this Redis cache "Geohash" in our final architecture diagram (Figure 1.21).

Business data needed to render pages on the client

This type of data is quite straightforward to cache. The key is the `business_id` and the value is the business object which contains the business name, address, image URLs, etc. We call this Redis cache "Business info" in our final architecture diagram (Figure 1.21).

Region and availability zones

We deploy a location-based service to multiple regions and availability zones as shown in Figure 1.20. This has a few advantages:

- Makes users physically "closer" to the system. Users from the US West are connected to the data centers in that region, and users from Europe are connected with data centers in Europe.
- Gives us the flexibility to spread the traffic evenly across the population. Some regions such as Japan and Korea have high population densities. It might be wise to put them in separate regions, or even deploy location-based services in multiple availability zones to spread the load.
- Privacy laws. Some countries may require user data to be used and stored locally. In this case, we could set up a region in that country and employ DNS routing to restrict all requests from the country to only that region.

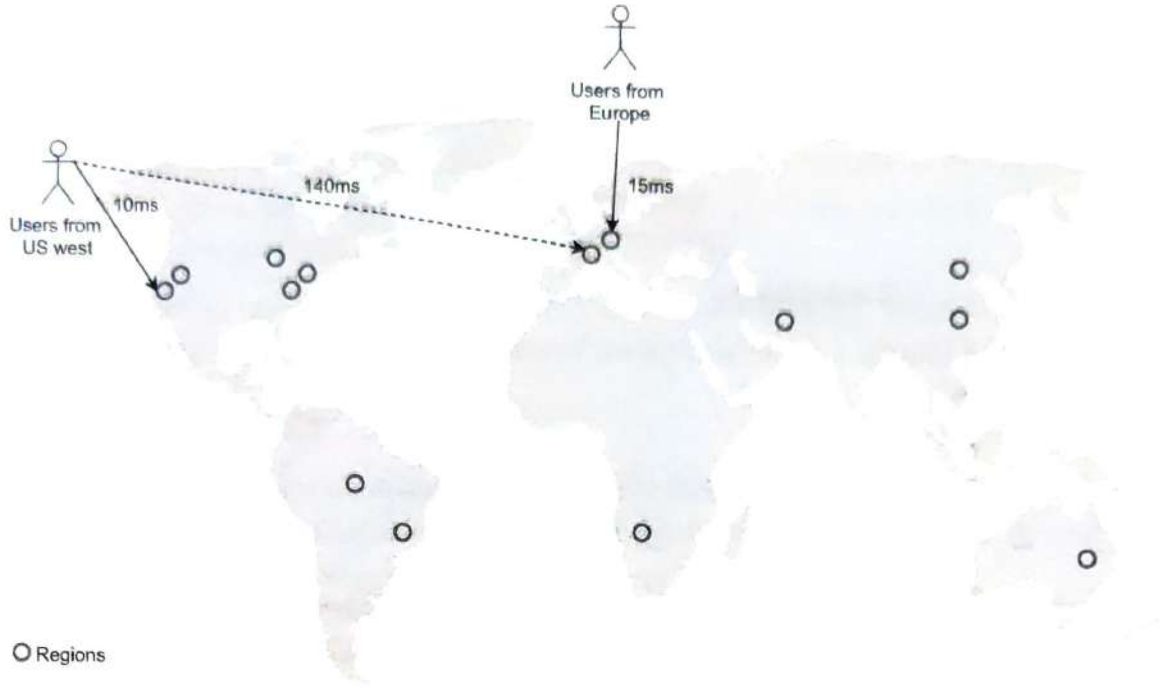


Figure 1.20: Deploy LBS "closer" to the user

Follow-up question: filter results by time or business type

The interviewer might ask a follow-up question: how to return businesses that are open now, or only return businesses that are restaurants?

Candidate: When the world is divided into small grids with geohash or quadtree, the number of businesses returned from the search result is relatively small. Therefore, it is acceptable to return business IDs first, hydrate business objects, and filter them based on opening time or business type. This solution assumes opening time and business type are stored in the business table.

Final design diagram

Putting everything together, we come up with the following design diagram.

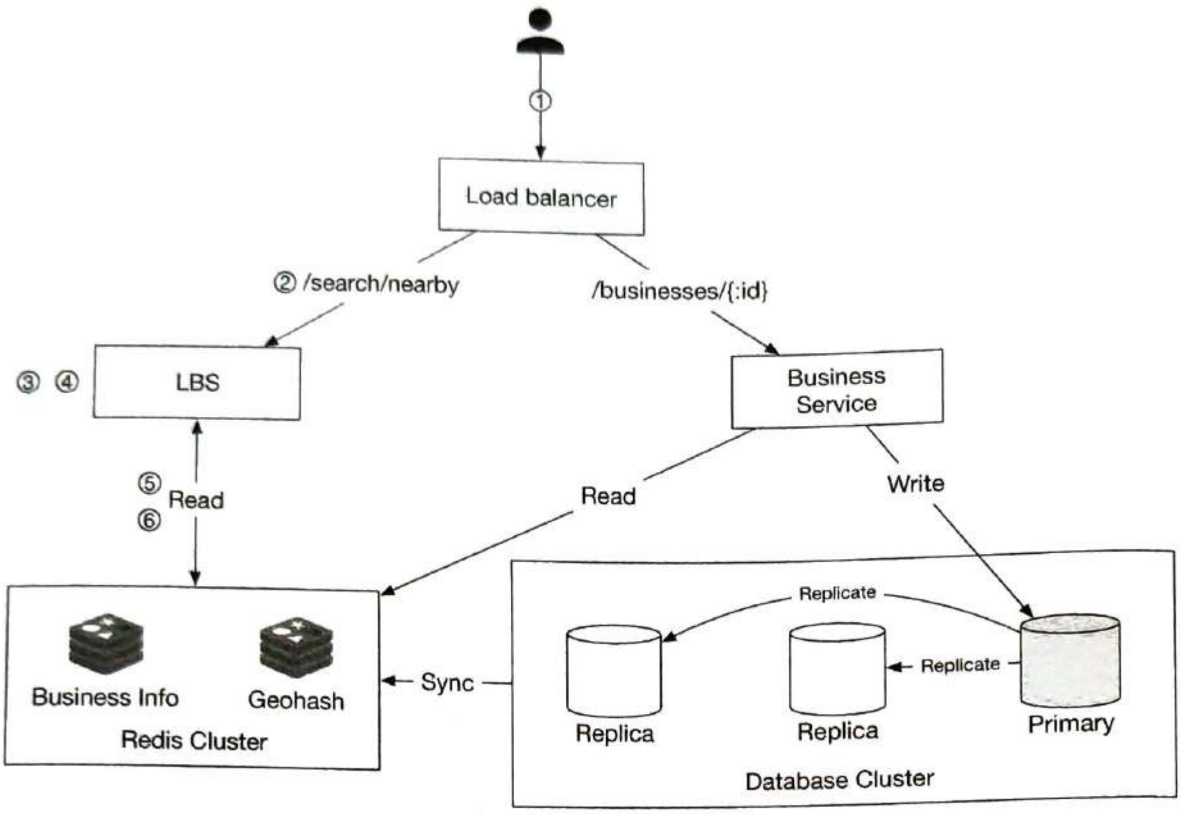


Figure 1.21: Design diagram

Get nearby businesses

1. You try to find restaurants within 500 meters on Yelp. The client sends the user location (latitude = 37.776720, longitude = −122.416730) and radius (500m) to the load balancer.
2. The load balancer forwards the request to the LBS.
3. Based on the user location and radius info, the LBS finds the geohash length that matches the search. By checking Table 1.5, 500m map to geohash length = 6.
4. LBS calculates neighboring geohashes and adds them to the list. The result looks like this:
   `list_of_geohashes = [my_geohash, neighbor1_geohash, neighbor2_geohash, ..., neighbor8_geohash]`.
5. For each geohash in `list_of_geohashes`, LBS calls the "Geohash" Redis server to fetch corresponding business IDs. Calls to fetch business IDs for each geohash can be made in parallel to reduce latency.
6. Based on the list of business IDs returned, LBS fetches fully hydrated business information from the "Business info" Redis server, then calculates distances between a user and businesses, ranks them, and returns the result to the client.

View, update, add or delete a business

All business-related APIs are separated from the LBS. To view the detailed information about a business, the business service first checks if the data is stored in the "Business info" Redis cache. If it is, cached data will be returned to the client. If not, data is fetched from the database cluster and then stored in the Redis cache, allowing subsequent requests to get results from the cache directly.

Since we have an upfront business agreement that newly added/updated businesses will be effective the next day, cached business data is updated by a nightly job.

Step 4 - Wrap Up

In this chapter, we have presented the design for proximity service. The system is a typical LBS that leverages geospatial indexing. We discussed several indexing options:

- Two-dimensional search
- Evenly divided grid
- Geohash
- Quadtree
- Google S2

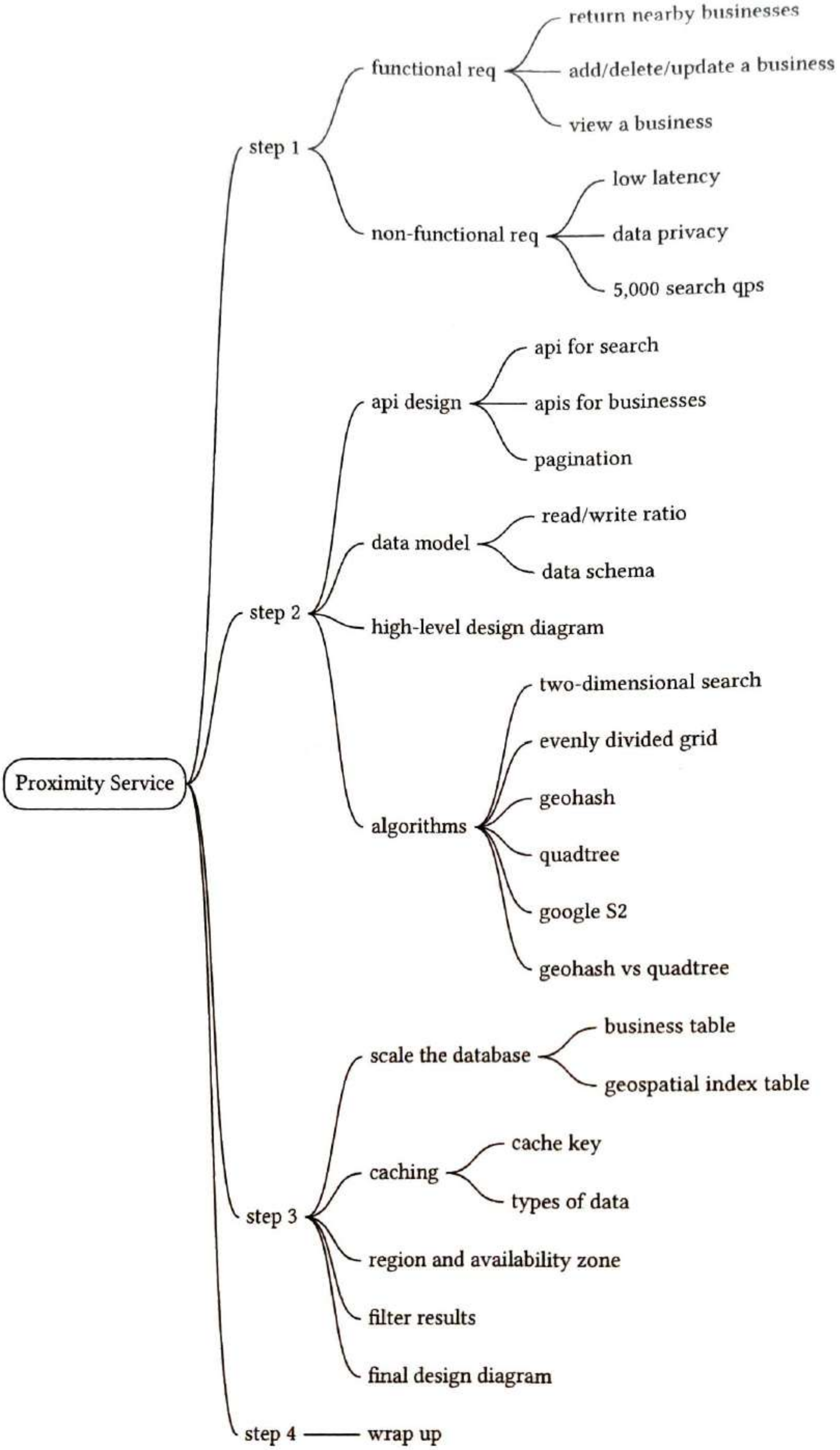
Geohash, quadtree, and S2 are widely used by different tech companies. We choose geohash as an example to show how a geospatial index works.

In the deep dive, we discussed why caching is effective in reducing the latency, what should be cached and how to use cache to retrieve nearby businesses fast. We also discussed how to scale the database with replication and sharding.

We then looked at deploying LBS in different regions and availability zones to improve availability, to make users physically closer to the servers, and to comply better with local privacy laws.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Yelp. https://www.yelp.com/.

[2] Map tiles by Stamen Design. http://maps.stamen.com/.

[3] OpenStreetMap. https://www.openstreetmap.org.

[4] GDPR. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation.

[5] CCPA. https://en.wikipedia.org/wiki/California_Consumer_Privacy_Act.

[6] Pagination in the REST API. https://developer.atlassian.com/server/confluence/pagination-in-the-rest-api/.

[7] Google places API. https://developers.google.com/maps/documentation/places/web-service/search.

[8] Yelp business endpoints. https://www.yelp.com/developers/documentation/v3/business_search.

[9] Regions and Zones. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

[10] Redis GEOHASH. https://redis.io/commands/GEOHASH.

[11] POSTGIS. https://postgis.net/.

[12] Cartesian tiers. http://www.nsshutdown.com/projects/lucene/whitepaper/locallucene_v2.html.

[13] R-tree. https://en.wikipedia.org/wiki/R-tree.

[14] Global map in a Geographic Coordinate Reference System. https://bit.ly/3DsjAwg.

[15] Base32. https://en.wikipedia.org/wiki/Base32.

[16] Geohash grid aggregation. https://bit.ly/3kKl4e6.

[17] Geohash. https://www.movable-type.co.uk/scripts/geohash.html.

[18] Quadtree. https://en.wikipedia.org/wiki/Quadtree.

[19] How many leaves has a quadtree. https://stackoverflow.com/questions/35976444/how-many-leaves-has-a-quadtree.

[20] Blue green deployment. https://martinfowler.com/bliki/BlueGreenDeployment.html.

[21] Improved Location Caching with Quadtrees. https://engblog.yext.com/post/geolocation-caching.

[22] S2. https://s2geometry.io/.

[23] Hilbert curve. https://en.wikipedia.org/wiki/Hilbert_curve.

[24] Hilbert mapping. http://bit-player.org/extras/hilbert/hilbert-mapping.html.

[25] Geo-fence. https://en.wikipedia.org/wiki/Geo-fence.

[26] Region cover. https://s2.sidewalklabs.com/regioncoverer/.

[27] Bing map. https://bit.ly/30ytSfG.

[28] MongoDB. https://docs.mongodb.com/manual/tutorial/build-a-2d-index/.

[29] Geospatial Indexing: The 10 Million QPS Redis Architecture Powering Lyft. https://www.youtube.com/watch?v=cSFWlF96Sds&t=2155s.

[30] Geo Shape Type. https://www.elastic.co/guide/en/elasticsearch/reference/1.6/mapping-geo-shape-type.html.

[31] Geosharded Recommendations Part 1: Sharding Approach. https://medium.com/tinder-engineering/geosharded-recommendations-part-1-sharding-approach-d5d54e0ec77a.

[32] Get the last known location. https://developer.android.com/training/location/retrieve-current#Challenges.

2 Nearby Friends

In this chapter, we design a scalable backend system for a new mobile app feature called "Nearby Friends". For an opt-in user who grants permission to access their location, the mobile client presents a list of friends who are geographically nearby. If you are looking for a real-world example, please refer to this article [1] about a similar feature in the Facebook app.

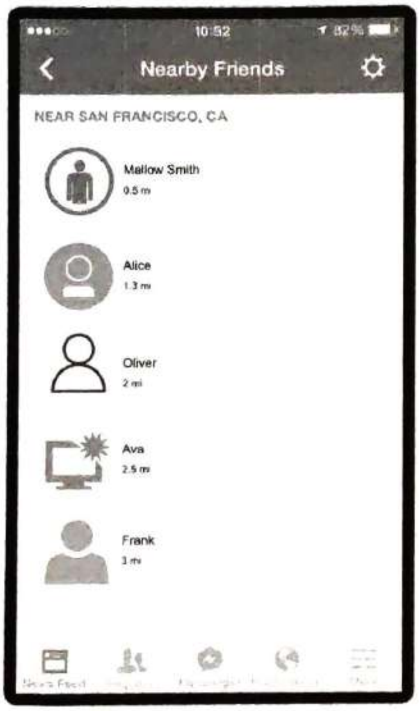


Figure 2.1: Facebook's nearby friends

If you read Chapter 1 Proximity Service, you may wonder why we need a separate chapter for designing "nearby friends" since it looks similar to proximity services. If you think carefully though, you will find major differences. In proximity services, the addresses for businesses are static as their locations do not change, while in "nearby friends", data is more dynamic because user locations change frequently.

Step 1 - Understand the Problem and Establish Design Scope

Any backend system at the Facebook scale is complicated. Before starting with the design, we need to ask clarification questions to narrow down the scope.

Candidate: How geographically close is considered to be "nearby"?
Interviewer: 5 miles. This number should be configurable.

Candidate: Can I assume the distance is calculated as the straight-line distance between two users? In real life, there could be, for example, a river in between the users, resulting in a longer travel distance.
Interviewer: Yes, that's a reasonable assumption.

Candidate: How many users does the app have? Can I assume 1 billion users and 10% of them use the nearby friends feature?
Interviewer: Yes, that's a reasonable assumption.

Candidate: Do we need to store location history?
Interviewer: Yes, location history can be valuable for different purposes such as machine learning.

Candidate: Could we assume if a friend is inactive for more than 10 minutes, that friend will disappear from the nearby friend list? Or should we display the last known location?
Interviewer: We can assume inactive friends will no longer be shown.

Candidate: Do we need to worry about privacy and data laws such as GDPR or CCPA?
Interviewer: Good question. For simplicity, don't worry about it for now.

Functional requirements

- Users should be able to see nearby friends on their mobile apps. Each entry in the nearby friend list has a distance and a timestamp indicating when the distance was last updated.
- Nearby friend lists should be updated every few seconds.

Non-functional requirements

- Low latency. It's important to receive location updates from friends without too much delay.
- Reliability. The system needs to be reliable overall, but occasional data point loss is acceptable.
- Eventual consistency. The location data store doesn't need strong consistency. A few seconds delay in receiving location data in different replicas is acceptable.

Back-of-the-envelope estimation

Let's do a back-of-the-envelope estimation to determine the potential scale and challenges our solution will need to address. Some constraints and assumptions are listed below:

- Nearby friends are defined as friends whose locations are within a 5-mile radius.
- The location refresh interval is 30 seconds. The reason for this is that human walking speed is slow (average $3 \sim 4$ miles per hour). The distance traveled in 30 seconds does not make a significant difference on the "nearby friends" feature.
- On average, 100 million users use the "nearby friends" feature every day.
- Assume the number of concurrent users is 10% of DAU (Daily Active Users), so the number of concurrent users is 10 million.
- On average, a user has 400 friends. Assume all of them use the "nearby friends" feature.
- The app displays 20 nearby friends per page and may load more nearby friends upon request.

Calculate QPS

- 100 million DAU
- Concurrent users: $10\% \times 100$ million $= 10$ million
- Users report their locations every 30 seconds.
- Location update QPS = $\frac{10 \text{ million}}{30} = \sim 334{,}000$

Step 2 - Propose High-level Design and Get Buy-in

In this section, we will discuss the following:

- High-level design
- API design
- Data model

In other chapters, we usually discuss API design and data model before the high-level design. However, for this problem, the communication protocol between client and server might not be a straightforward HTTP protocol, as we need to push location data to all friends. Without understanding the high-level design, it's difficult to know what the APIs look like. Therefore, we discuss the high-level design first.

High-level design

At a high level, this problem calls for a design with efficient message passing. Conceptually, a user would like to receive location updates from every active friend nearby. It could in theory be done purely peer-to-peer, that is, a user could maintain a persistent connection to every other active friend in the vicinity (Figure 2.2).

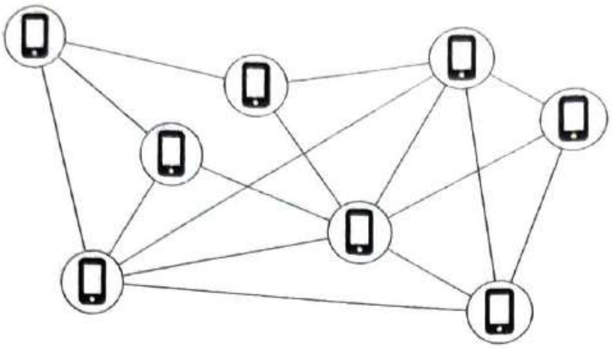


Figure 2.2: Peer-to-peer

This solution is not practical for a mobile device with sometimes flaky connections and a tight power consumption budget, but the idea sheds some light on the general design direction.

A more practical design would have a shared backend and look like this:

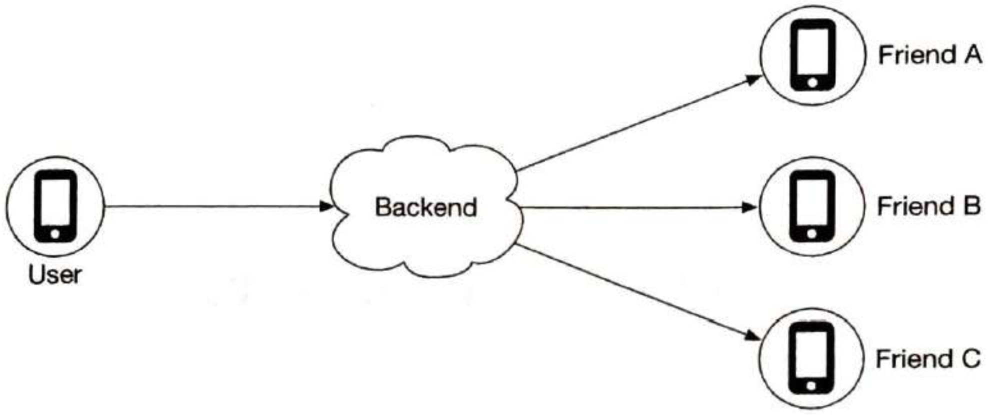


Figure 2.3: Shared backend

What are the responsibilities of the backend in Figure 2.3?

- Receive location updates from all active users.
- For each location update, find all the active friends who should receive it and forward it to those users' devices.
- If the distance between two users is over a certain threshold, do not forward it to the recipient's device.

This sounds pretty simple. What is the issue? Well, to do this at scale is not easy. We have 10 million active users. With each user updating the location information every 30 seconds, there are 334K updates per second. If on average each user has 400 friends, and we further assume that roughly 10% of those friends are online and nearby, every second the backend forwards 334K × 400 × 10% = 14 million location updates per second. That is a lot of updates to forward.

Proposed design

We will first come up with a high-level design for the backend at a lower scale. Later in the deep dive section, we will optimize the design for scale.

Figure 2.4 shows the basic design that should satisfy the functional requirements. Let's go over each component in the design.

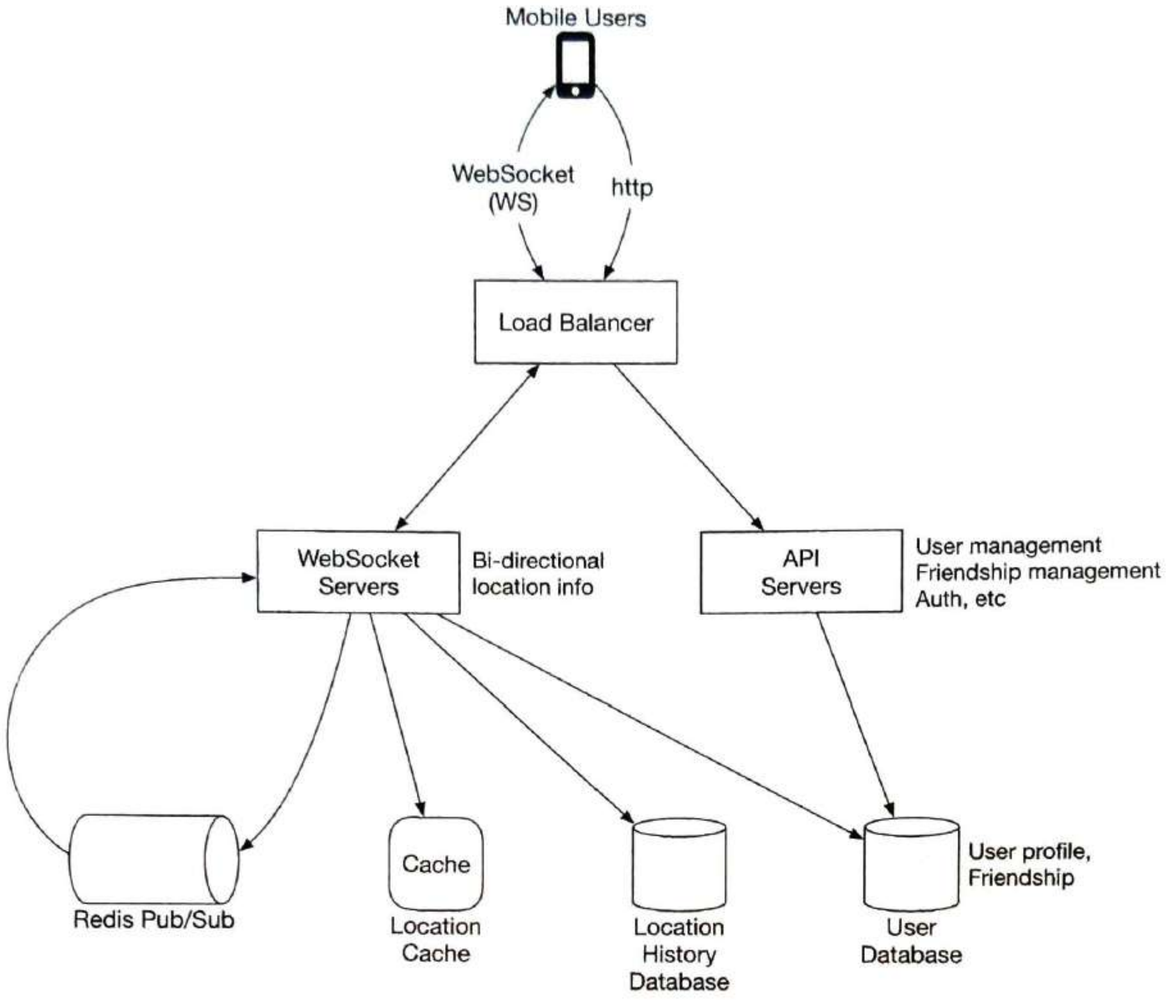


Figure 2.4: High-level design

Load balancer

The load balancer sits in front of the RESTful API servers and the stateful, bi-directional WebSocket servers. It distributes traffic across those servers to spread out load evenly.

RESTful API servers

This is a cluster of stateless HTTP servers that handles the typical request/response traffic. The API request flow is highlighted in Figure 2.5. This API layer handles auxiliary tasks like adding/removing friends, updating user profiles, etc. These are very common and we will not go into more detail.

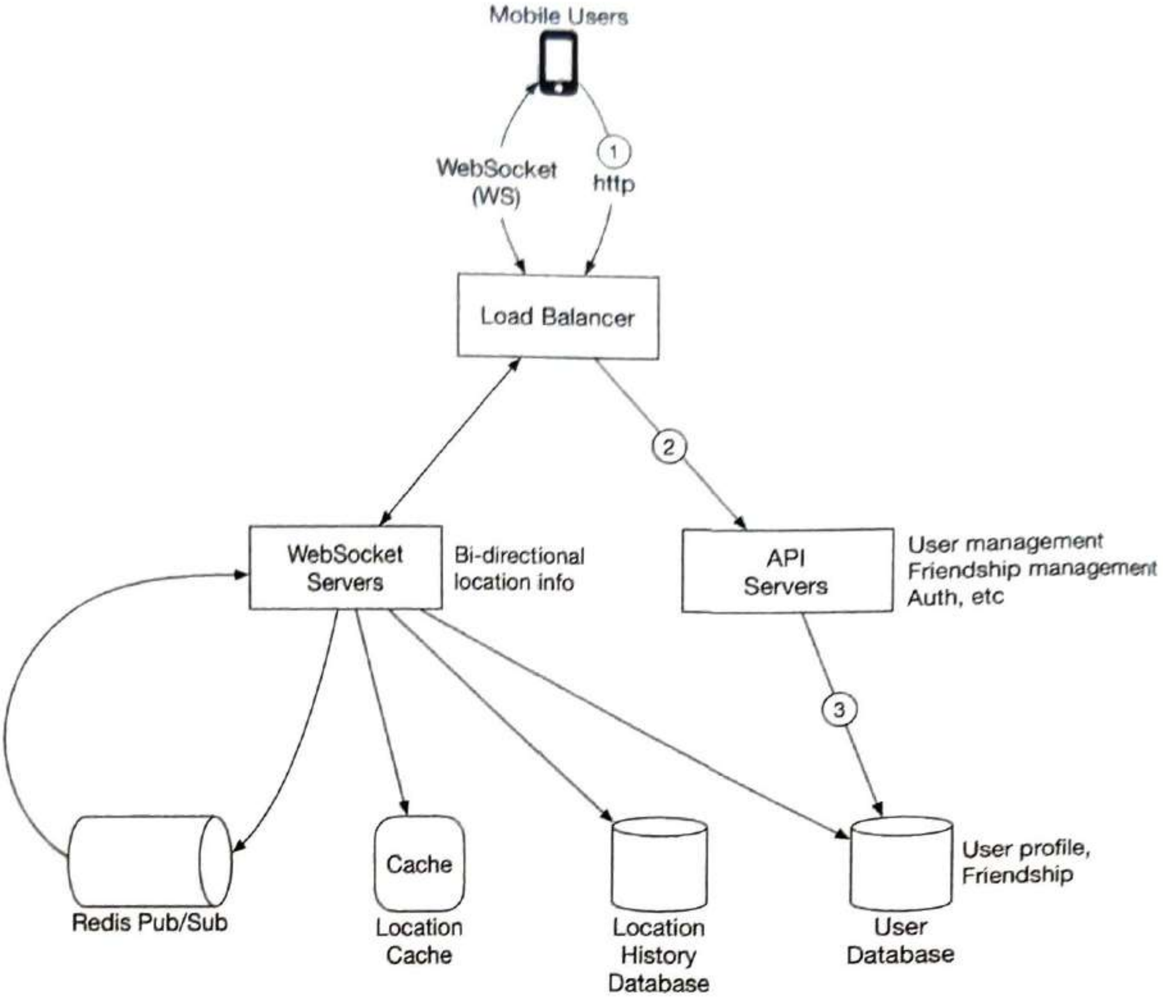


Figure 2.5: RESTful API request flow

WebSocket servers

This is a cluster of stateful servers that handles the near real-time update of friends' locations. Each client maintains one persistent WebSocket connection to one of these servers. When there is a location update from a friend who is within the search radius, the update is sent on this connection to the client.

Another major responsibility of the WebSocket servers is to handle client initialization for the "nearby friends" feature. It seeds the mobile client with the locations of all nearby online friends. We will discuss how this is done in more detail later.

Note "WebSocket connection" and "WebSocket connection handler" are interchangeable in this chapter.

Redis location cache

Redis is used to store the most recent location data for each active user. There is a Time to Live (TTL) set on each entry in the cache. When the TTL expires, the user is no longer active and the location data is expunged from the cache. Every update refreshes the TTL. Other KV stores that support TTL could also be used.

User database

The user database stores user data and user friendship data. Either a relational database or a NoSQL database can be used for this.

Location history database

This database stores users' historical location data. It is not directly related to the "nearby friends" feature.

Redis Pub/Sub server

Redis Pub/Sub [2] is a very lightweight message bus. Channels in Redis Pub/Sub are very cheap to create. A modern Redis server with GBs of memory could hold millions of channels (also called topics). Figure 2.6 shows how Redis Pub/Sub works.

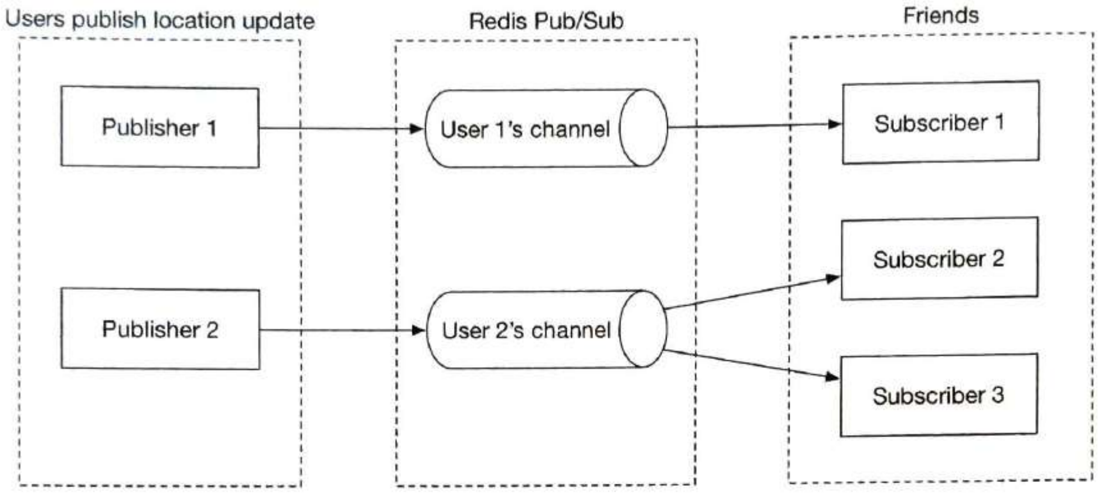


Figure 2.6: Redis Pub/Sub

In this design, location updates received via the WebSocket server are published to the user's own channel in the Redis Pub/Sub server. A dedicated WebSocket connection handler for each active friend subscribes to the channel. When there is a location update, the WebSocket handler function gets invoked, and for each active friend, the function recomputes the distance. If the new distance is within the search radius, the new location and timestamp are sent via the WebSocket connection to the friend's client. Other message buses with lightweight channels could also be used.

Now that we understand what each component does, let's examine what happens when a user's location changes from the system's perspective.

Periodic location update

The mobile client sends periodic location updates over the persistent WebSocket connection. The flow is shown in Figure 2.7.

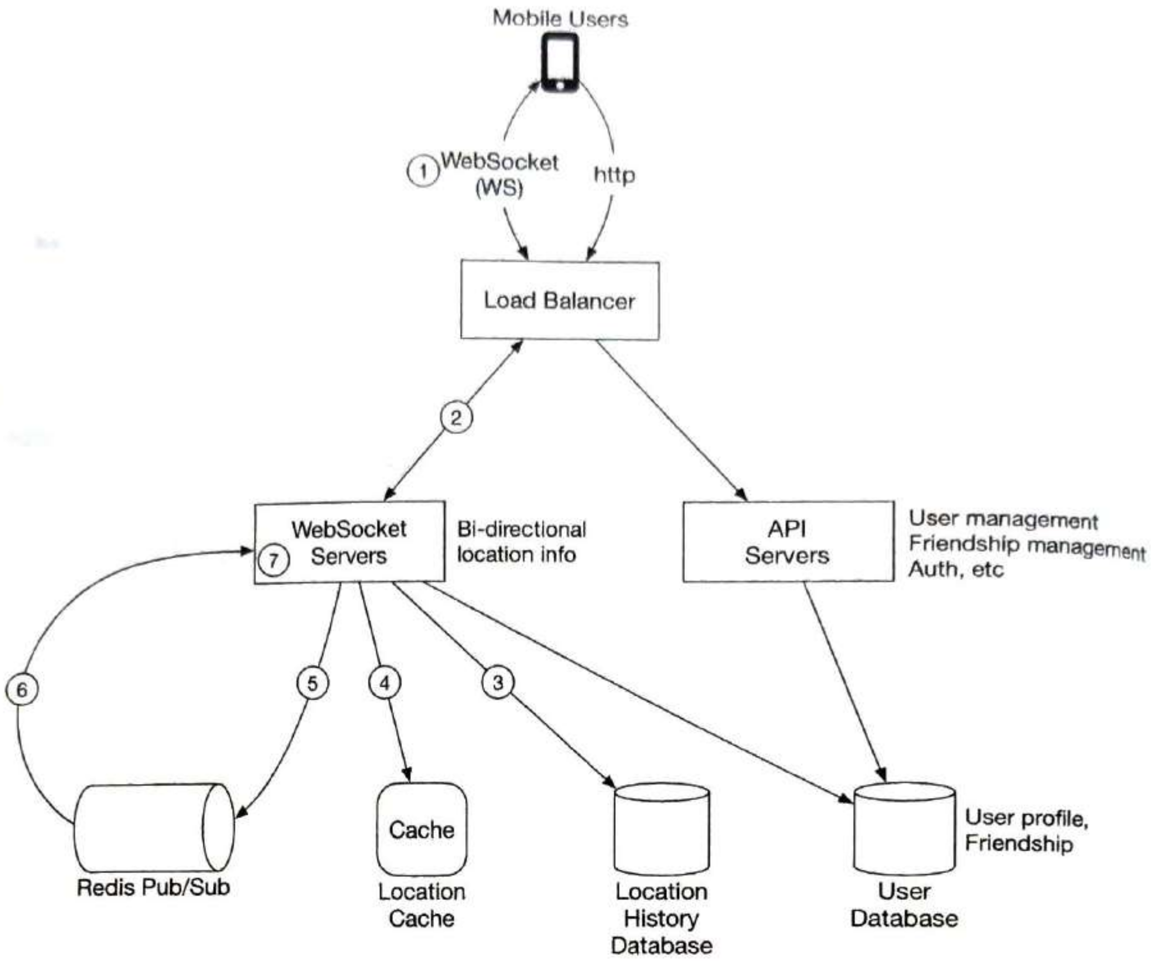


Figure 2.7: Periodic location update

1. The mobile client sends a location update to the load balancer.
2. The load balancer forwards the location update to the persistent connection on the WebSocket server for that client.
3. The WebSocket server saves the location data to the location history database.
4. The WebSocket server updates the new location in the location cache. The update refreshes the TTL. The WebSocket server also saves the new location in a variable in the user's WebSocket connection handler for subsequent distance calculations.
5. The WebSocket server publishes the new location to the user's channel in the Redis Pub/Sub server. Steps 3 to 5 can be executed in parallel.
6. When Redis Pub/Sub receives a location update on a channel, it broadcasts the update to all the subscribers (WebSocket connection handlers). In this case, the subscribers are all the online friends of the user sending the update. For each subscriber (i.e., for each of the user's friends), its WebSocket connection handler would receive the user location update.
7. On receiving the message, the WebSocket server, on which the connection handler lives, computes the distance between the user sending the new location (the location

data is in the message) and the subscriber (the location data is stored in a variable with the WebSocket connection handler for the subscriber).

8. This step is not drawn on the diagram. If the distance does not exceed the search radius, the new location and the last updated timestamp are sent to the subscriber's client. Otherwise, the update is dropped.

Since understanding this flow is extremely important, let's examine it again with a concrete example, as shown in Figure 2.8. Before we start, let's make a few assumptions.

- User 1's friends: User 2, User 3, and User 4.
- User 5's friends: User 4 and User 6.

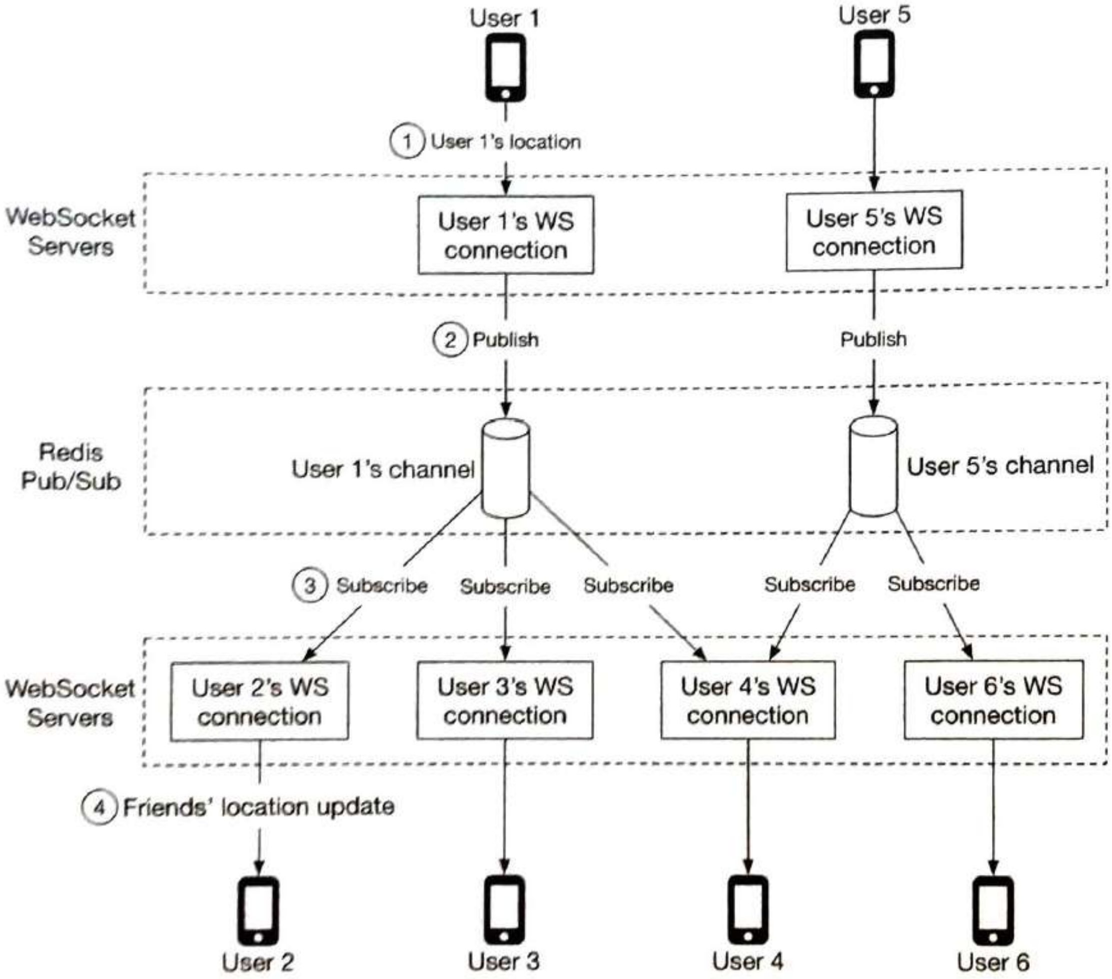


Figure 2.8: Send location update to friends

1. When User 1's location changes, their location update is sent to the WebSocket server which holds User 1's connection.
2. The location is published to User 1's channel in Redis Pub/Sub server.
3. Redis Pub/Sub server broadcasts the location update to all subscribers. In this case,

subscribers are WebSocket connection handlers (User 1's friends).

4. If the distance between the user sending the location (User 1) and the subscriber (User 2) doesn't exceed the search radius, the new location is sent to the client (User 2).

This computation is repeated for every subscriber to the channel. Since there are 400 friends on average, and we assume that 10% of those friends are online and nearby, there are about 40 location updates to forward for each user's location update.

API design

Now that we have created a high-level design, let's list APIs needed.

WebSocket: Users send and receive location updates through the WebSocket protocol. At the minimum, we need the following APIs.

1. Periodic location update
Request: Client sends latitude, longitude, and timestamp.
Response: Nothing.

2. Client receives location updates
Data sent: Friend location data and timestamp.

3. WebSocket initialization
Request: Client sends latitude, longitude, and timestamp.
Response: Client receives friends' location data.

4. Subscribe to a new friend
Request: WebSocket server sends friend ID.
Response: Friend's latest latitude, longitude, and timestamp.

5. Unsubscribe a friend
Request: WebSocket server sends friend ID.
Response. Nothing.

HTTP requests: the API servers handle tasks like adding/removing friends, updating user profiles, etc. These are very common and we will not go into detail here.

Data model

Another important element to discuss is the data model. We already talked about the User DB in the high-level design, so let's focus on the location cache and location history database.

Location cache

The location cache stores the latest locations of all active users who have had the nearby friends feature turned on. We use Redis for this cache. The key/value of the cache is shown in Table 2.1.

key	value
user_id	{latitude, longitude, timestamp}

Table 2.1: Location cache

Why don't we use a database to store location data?

The "nearby friends" feature only cares about the **current** location of a user. Therefore, we only need to store one location per user. Redis is an excellent choice because it provides super-fast read and write operations. It supports TTL, which we use to auto-purge users from the cache who are no longer active. The current locations do not need to be durably stored. If the Redis instance goes down, we could replace it with an empty new instance and let the cache be filled as new location updates stream in. The active users could miss location updates from friends for an update cycle or two while the new cache warms. It is an acceptable tradeoff. In the deep dive section, we will discuss ways to lessen the impact on users when the cache gets replaced.

Location history database

The location history database stores users' historical location data and the schema looks like this:

user_id	latitude	longitude	timestamp

We need a database that handles the heavy-write workload well and can be horizontally scaled. Cassandra is a good candidate. We could also use a relational database. However, with a relational database, the historical data would not fit in a single instance so we need to shard that data. The most basic approach is to shard by user ID. This sharding scheme ensures that load is evenly distributed among all the shards, and operationally, it is easy to maintain.

Step 3 - Design Deep Dive

The high-level design we created in the previous section works in most cases, but it will likely break at our scale. In this section, we work together to uncover the bottlenecks as we increase the scale, and along the way work on solutions to eliminate those bottlenecks.

How well does each component scale?

API servers

The methods to scale the RESTful API tiers are well understood. These are stateless servers, and there are many ways to auto-scale the clusters based on CPU usage, load, or I/O. We will not go into detail here.

WebSocket servers

For the WebSocket cluster, it is not difficult to auto-scale based on usage. However, the WebSocket servers are stateful, so care must be taken when removing existing nodes. Before a node can be removed, all existing connections should be allowed to drain. To achieve that, we can mark a node as "draining" at the load balancer so that no new WebSocket connections will be routed to the draining server. Once all the existing connections are closed (or after a reasonably long wait), the server is then removed.

Releasing a new version of the application software on a WebSocket server requires the same level of care.

It is worth noting that effective auto-scaling of stateful servers is the job of a good load balancer. Most cloud load balancers handle this job very well.

Client initialization

The mobile client on startup establishes a persistent WebSocket connection with one of the WebSocket server instances. Each connection is long-running. Most modern languages are capable of maintaining many long-running connections with a reasonably small memory footprint.

When a WebSocket connection is initialized, the client sends the initial location of the user, and the server performs the following tasks in the WebSocket connection handler.

1. It updates the user's location in the location cache.
2. It saves the location in a variable of the connection handler for subsequent calculations.
3. It loads all the user's friends from the user database.
4. It makes a batched request to the location cache to fetch the locations for all the friends. Note that because we set a TTL on each entry in the location cache to match our inactivity timeout period, if a friend is inactive then their location will not be in the location cache.
5. For each location returned by the cache, the server computes the distance between the user and the friend at that location. If the distance is within the search radius, the friend's profile, location, and last updated timestamp are returned over the WebSocket connection to the client.
6. For each friend, the server subscribes to the friend's channel in the Redis Pub/Sub server. We will explain our use of Redis Pub/Sub shortly. Since creating a new channel is cheap, the user subscribes to all active and inactive friends. The inactive friends will take up a small amount of memory on the Redis Pub/Sub server, but they will not consume any CPU or I/O (since they do not publish updates) until they come online.
7. It sends the user's current location to the user's channel in the Redis Pub/Sub server.

User database

The user database holds two distinct sets of data: user profiles (user ID, username, profile URL, etc.) and friendships. These datasets at our design scale will likely not fit in a single relational database instance. The good news is that the data is horizontally scalable by sharding based on user ID. Relational database sharding is a very common technique.

As a side note, at the scale we are designing for, the user and friendship datasets will likely be managed by a dedicated team and be available via an internal API. In this scenario, the WebSocket servers will use the internal API instead of querying the database directly to fetch user and friendship-related data. Whether accessing via API or direct database queries, it does not make much difference in terms of functionality or performance.

Location cache

We choose Redis to cache the most recent locations of all the active users. As mentioned earlier, we also set a TTL on each key. The TTL is renewed upon every location update. This puts a cap on the maximum amount of memory used. With 10 million active users at peak, and with each location taking no more than 100 bytes, a single modern Redis server with many GBs of memory should be able to easily hold the location information for all users.

However, with 10 million active users roughly updating every 30 seconds, the Redis server will have to handle 334K updates per second. That is likely a little too high, even for a modern high-end server. Luckily, this cache data is easy to shard. The location data for each user is independent, and we can evenly spread the load among several Redis servers by sharding the location data based on user ID.

To improve availability, we could replicate the location data on each shard to a standby node. If the primary node goes down, the standby could be quickly promoted to minimize downtime.

Redis Pub/Sub server

The Pub/Sub server is used as a routing layer to direct messages (location updates) from one user to all the online friends. As mentioned earlier, we choose Redis Pub/Sub because it is very lightweight to create new channels. A new channel is created when someone subscribes to it. If a message is published to a channel that has no subscribers, the message is dropped, placing very little load on the server. When a channel is created, Redis uses a small amount of memory to maintain a hash table and a linked list [3] to track the subscribers. If there is no update on a channel when a user is offline, no CPU cycles are used after a channel is created. We take advantage of this in our design in the following ways:

1. We assign a unique channel to every user who uses the "nearby friends" feature. A user would, upon app initialization, subscribe to each friend's channel, whether the friend is online or not. This simplifies the design since the backend does not need to handle subscribing to a friend's channel when the friend becomes active, or handling unsubscribing when the friend becomes inactive.
2. The tradeoff is that the design would use more memory. As we will see later, memory use is unlikely to be the bottleneck. Trading higher memory use for a simpler architecture is worth it in this case.

How many Redis Pub/Sub servers do we need?

Let's do some math on memory and CPU usage.

Memory usage

Assuming a channel is allocated for each user who uses the nearby friends feature, we need 100 million channels (1 billion × 10%). Assuming that on average a user has 100 active friends using this feature (this includes friends who are nearby, or not), and it takes about 20 bytes of pointers in the internal hash table and linked list to track each subscriber, it will need about 200GB (100 million × 20 bytes × 100 friends / 10^9 = 200GB) to hold all the channels. For a modern server with 100GB of memory, we will need about 2 Redis Pub/Sub servers to hold all the channels.

CPU usage

As previously calculated, the Pub/Sub server pushes about 14 million updates per second to subscribers. Even though it is not easy to estimate with any accuracy how many messages a modern Redis server could push a second without actual benchmarking, it is safe to assume that a single Redis server will not be able to handle that load. Let's pick a conservative number and assume that a modern server with a gigabit network could handle about 100,000 subscriber pushes per second. Given how small our location update messages are, this number is likely to be conservative. Using this conservative estimate, we will need to distribute the load among 14 million / 100,000 = 140 Redis servers. Again, this number is likely too conservative, and the actual number of servers could be much lower.

From the math, we conclude that:

- The bottleneck of Redis Pub/Sub server is the CPU usage, not the memory usage.
- To support our scale, we need a distributed Redis Pub/Sub cluster.

Distributed Redis Pub/Sub server cluster

How do we distribute the channels to hundreds of Redis servers? The good news is that the channels are independent of each other. This makes it relatively easy to spread the channels among multiple Pub/Sub servers by sharding, based on the publisher's user ID. Practically speaking though, with hundreds of Pub/Sub servers, we should go into a bit more detail on how this is done so that operationally it is somewhat manageable, as servers inevitably go down from time to time.

Here, we introduce a service discovery component to our design. There are many service discovery packages available, with etcd [4] and ZooKeeper [5] among the most popular ones. Our need for the service discovery component is very basic. We need these two features:

1. The ability to keep a list of servers in the service discovery component, and a simple UI or API to update it. Fundamentally, service discovery is a small key-value store for holding configuration data. Using Figure 2.9 as an example, the key and value for

the hash ring could look like this:

Key: `/config/pub_sub_ring`
Value: `["p_1", "p_2", "p_3", "p_4"]`

2. The ability for clients (in this case, the WebSocket servers) to subscribe to any updates to the "Value" (Redis Pub/Sub servers).

Under the "Key" mentioned in point 1, we store a hash ring of all the active Redis Pub/Sub servers in the service discovery component (See the consistent hashing chapter in Volume 1 of the System Design Interview book or [6] on details of a hash ring). The hash ring is used by the publishers and subscribers of the Redis Pub/Sub servers to determine the Pub/Sub server to talk to for each channel. For example, channel 2 lives in Redis Pub/Sub server 1 in Figure 2.9.

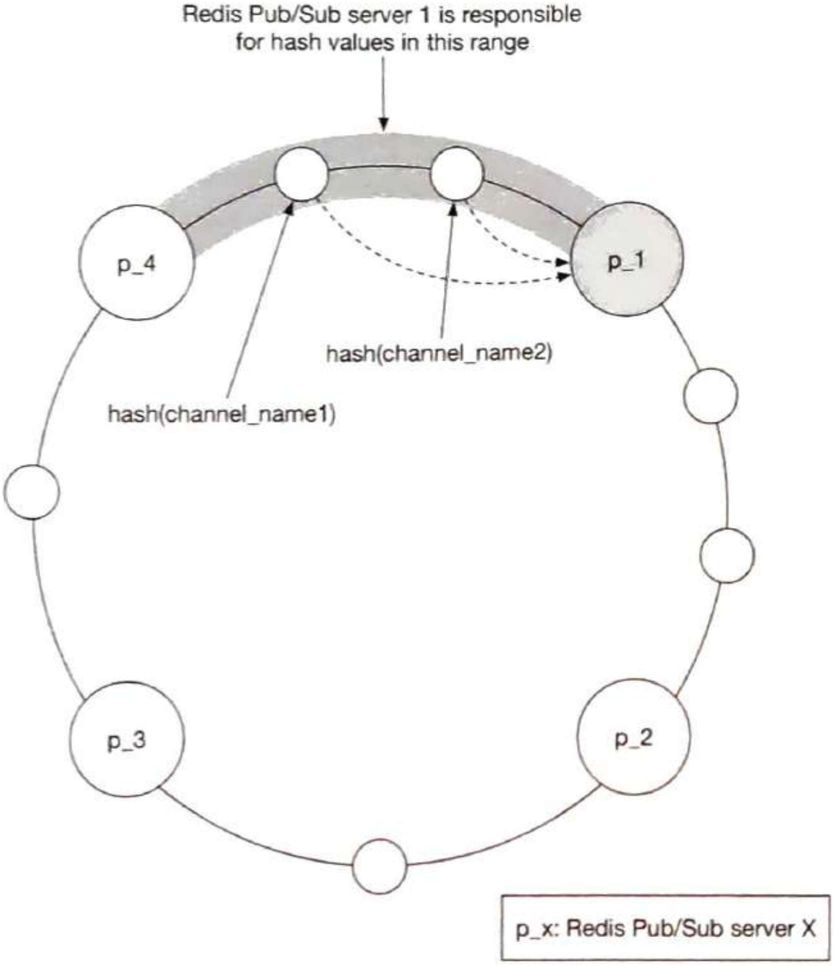

Redis Pub/Sub server 1 is responsible
for hash values in this range
p_4
p_1
hash(channel_name2)
hash(channel_name1)
p_3
p_2
p_x: Redis Pub/Sub server X

Figure 2.9: Consistent hashing

Figure 2.10 shows what happens when a WebSocket server publishes a location update to a user's channel.

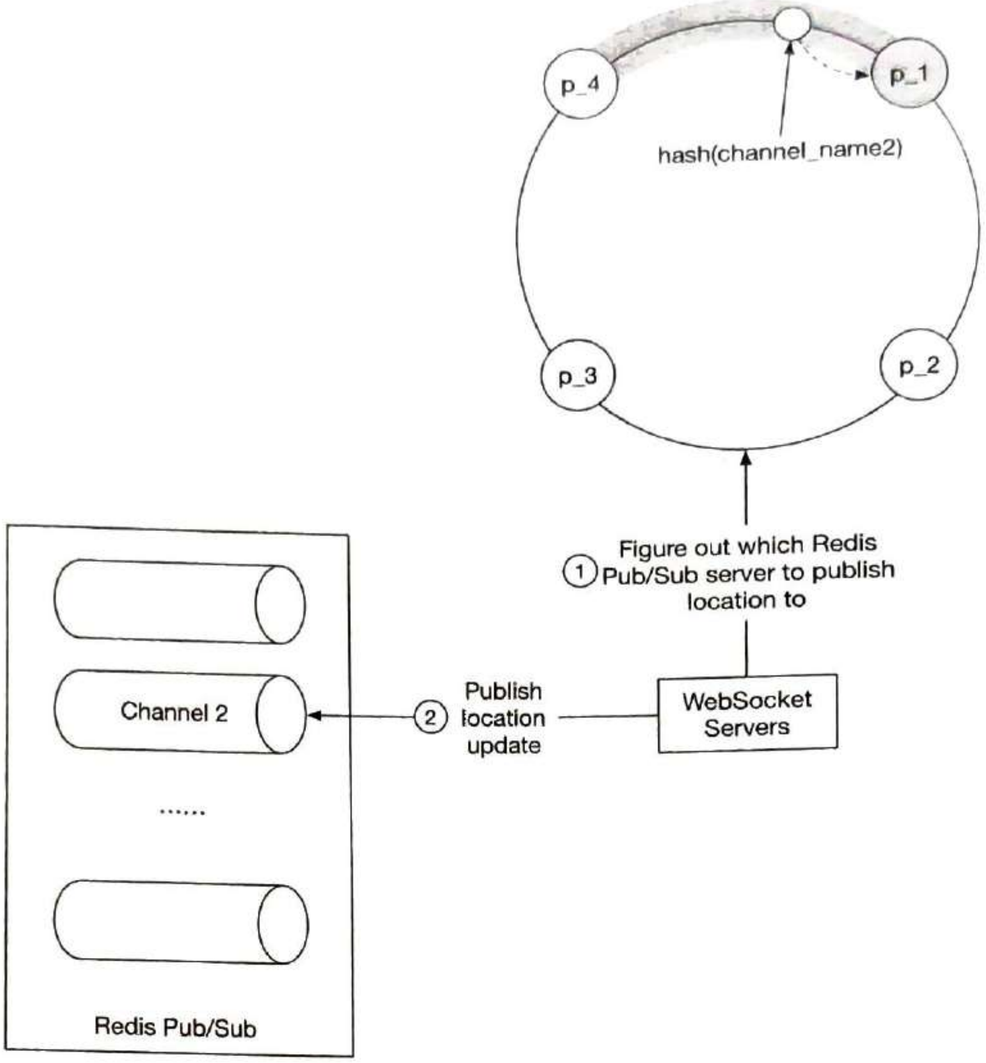

p_4
p_1
hash(channel_name2)
p_3
p_2
1 Figure out which Redis Pub/Sub server to publish location to
Channel 2
......
Redis Pub/Sub
2 Publish location update
WebSocket Servers

Figure 2.10: Figure out the correct Redis Pub/Sub server

1. The WebSocket server consults the hash ring to determine the Redis Pub/Sub server to write to. The source of truth is stored in service discovery, but for efficiency, a copy of the hash ring could be cached on each WebSocket server. The WebSocket server subscribes to any updates on the hash ring to keep its local in-memory copy up to date.
2. WebSocket server publishes the location update to the user's channel on that Redis Pub/Sub server.

Subscribing to a channel for location updates uses the same mechanism.

Scaling considerations for Redis Pub/Sub servers

How should we scale the Redis Pub/Sub server cluster? Should we scale it up and down daily, based on traffic patterns? This is a very common practice for stateless servers because it is low risk and saves costs. To answer these questions, let's examine some of the properties of the Redis Pub/Sub server cluster.

1. The messages sent on a Pub/Sub channel are not persisted in memory or on disk. They are sent to all subscribers of the channel and removed immediately after. If there are no subscribers, the messages are just dropped. In this sense, the data going through the Pub/Sub channel is stateless.
2. However, there are indeed states stored in the Pub/Sub servers for the channels. Specifically, the subscriber list for each channel is a key piece of the states tracked by the Pub/Sub servers. If a channel is moved, which could happen when the channel's Pub/Sub server is replaced, or if a new server is added or an old server removed on the hash ring, then every subscriber to the moved channel must know about it, so they could unsubscribe from the channel on the old server and resubscribe to the replacement channel on the new server. In this sense, a Pub/Sub server is stateful, and coordination with all subscribers to the server must be orchestrated to minimize service interruptions.

For these reasons, we should treat the Redis Pub/Sub cluster more like a stateful cluster, similar to how we would handle a storage cluster. With stateful clusters, scaling up or down has some operational overhead and risks, so it should be done with careful planning. The cluster is normally over-provisioned to make sure it can handle daily peak traffic with some comfortable headroom to avoid unnecessary resizing of the cluster.

When we inevitably have to scale, be mindful of these potential issues:

- When we resize a cluster, many channels will be moved to different servers on the hash ring. When the service discovery component notifies all the WebSocket servers of the hash ring update, there will be a ton of resubscription requests.
- During these mass resubscription events, some location updates might be missed by the clients. Although occasional misses are acceptable for our design, we should minimize the occurrences.
- Because of the potential interruptions, resizing should be done when usage is at its lowest in the day.

How is resizing actually done? It is quite simple. Follow these steps:

- Determine the new ring size, and if scaling up, provision enough new servers.
- Update the keys of the hash ring with the new content.
- Monitor your dashboard. There should be some spike in CPU usage in the WebSocket cluster.

Using the hash ring from Figure 2.9 above, if we were to add 2 new nodes, say, p_5, and p_6, the hash ring would be updated like this:

```
Old: ["p_1", "p_2", "p_3", "p_4"]
New: ["p_1", "p_2", "p_3", "p_4", "p_5", "p_6"]
```

Operational considerations for Redis Pub/Sub servers

The operational risk of replacing an existing Redis Pub/Sub server is much, much lower. It does not cause a large number of channels to be moved. Only the channels on the server being replaced will need to be handled. This is good because servers inevitably go down and need to be replaced regularly.

When a Pub/Sub server goes down, the monitoring software should alert the on-call operator. Precisely how the monitoring software monitors the health of a Pub/Sub server is beyond the scope of this chapter, so it is not covered. The on-call operator updates the hash ring key in service discovery to replace the dead node with a fresh standby node. The WebSocket servers are notified about the update and each one then notifies its connection handlers to re-subscribe to the channels on the new Pub/Sub server. Each WebSocket handler keeps a list of all channels it has subscribed to, and upon receiving the notification from the server, it checks each channel against the hash ring to determine if a channel needs to be re-subscribed on a new server.

Using the hash ring from Figure 2.9 above, if `p_1` went down, and we replace it with `p1_new`, the hash ring would be updated like so:

Old: `["p_1", "p_2", "p_3", "p_4"]`

New: `["p_1_new", "p_2", "p_3", "p_4"]`

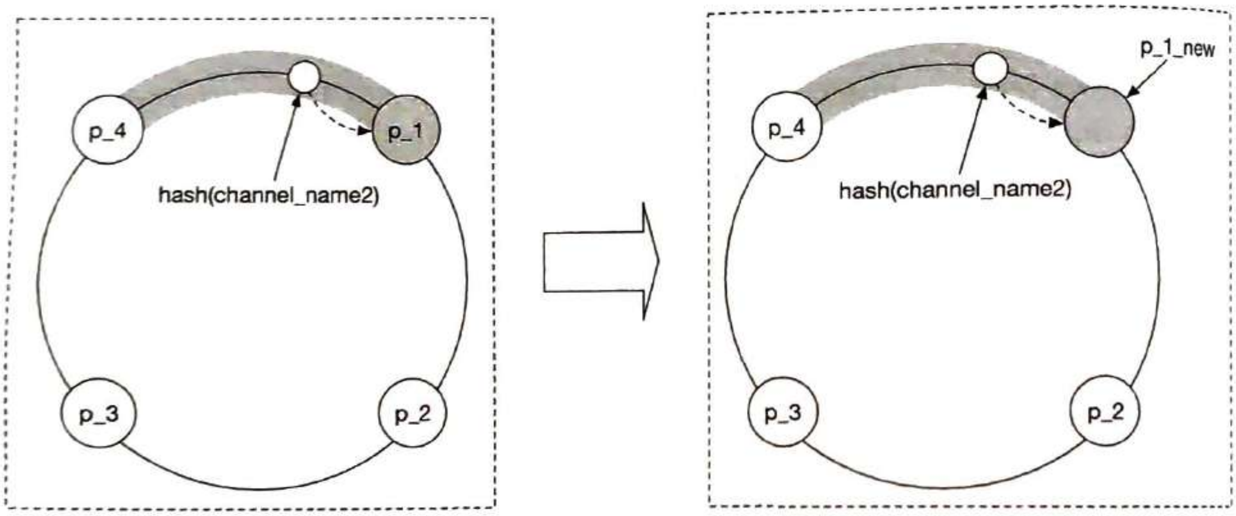


Figure 2.11: Replace Pub/Sub serverver

Adding/removing friends

What should the client do when the user adds or removes a friend? When a new friend is added, the client's WebSocket connection handler on the server needs to be notified, so it can subscribe to the new friend's Pub/Sub channel.

Since the "nearby friends" feature is within the ecosystem of a larger app, we can assume that the "nearby friends" feature could register a callback on the mobile client whenever a new friend is added. The callback, upon invocation, sends a message to the WebSocket server to subscribe to the new friend's Pub/Sub channel. The WebSocket server also returns a message containing the new friend's latest location and timestamp, if they are active.

Likewise, the client could register a callback in the application whenever a friend is removed. The callback would send a message to the WebSocket server to unsubscribe from the friend's Pub/Sub channel.

This subscribe/unsubscribe callback could also be used whenever a friend has opted in or out of the location update.

Users with many friends

It is worth discussing whether a user with many friends could cause performance hotspots in our design. We assume here that there is a hard cap on the number of friends. (Facebook has a cap of 5,000 friends, for example). Friendships are bi-directional. We are not talking about a follower model in which a celebrity could have millions of followers.

In a scenario with thousands of friends, the Pub/Sub subscribers will be scattered among the many WebSocket servers in the cluster. The update load would be spread among them and it's unlikely to cause any hotspots.

The user would place a bit more load on the Pub/Sub server where their channel lives. Since there are over 100 Pub/Sub servers, these "whale" users would be spread out among the Pub/Sub servers and the incremental load should not overwhelm any single one.

Nearby random person

You might call this section an extra credit, as it's not in the initial functional requirements. What if the interviewer wants to update the design to show random people who opted-in to location-sharing?

One way to do this while leveraging our design is to add a pool of Pub/Sub channels by geohash. (See Chapter 1 Proximity Service for details on geohash). As shown in Figure 2.12, an area is divided into four geohash grids and a channel is created for each grid.

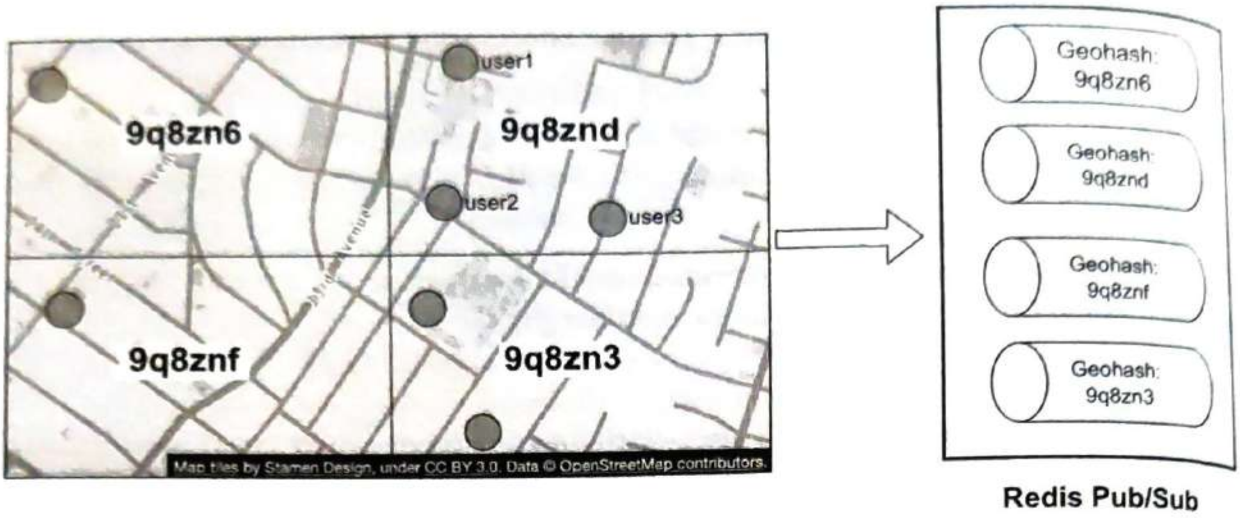


Figure 2.12: Redis Pub/Sub channels

Anyone within the grid subscribes to the same channel. Let's take grid 9q8znd for example as shown in Figure 2.13.

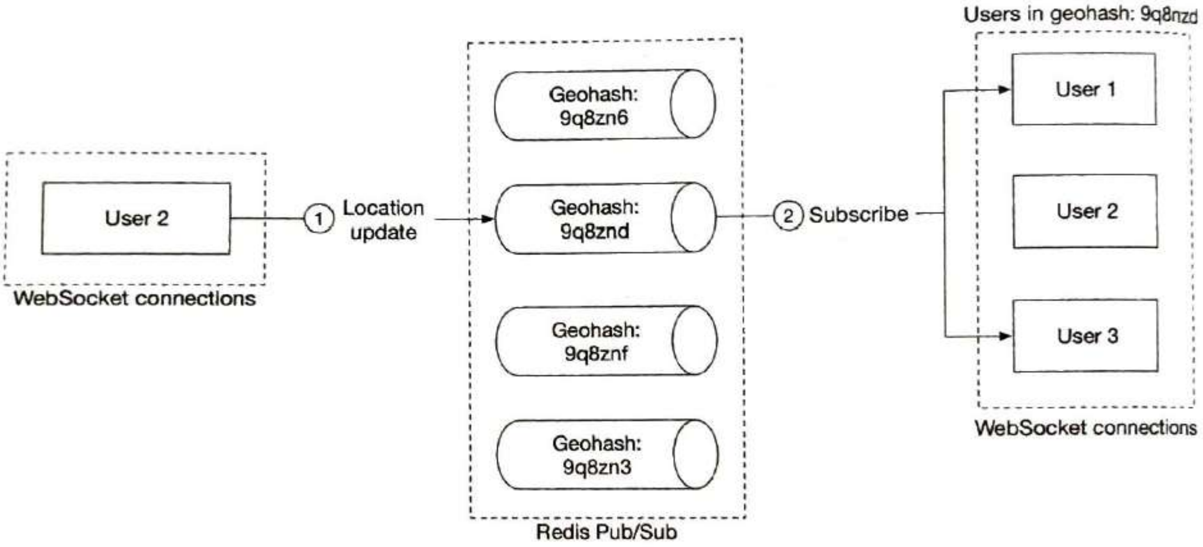
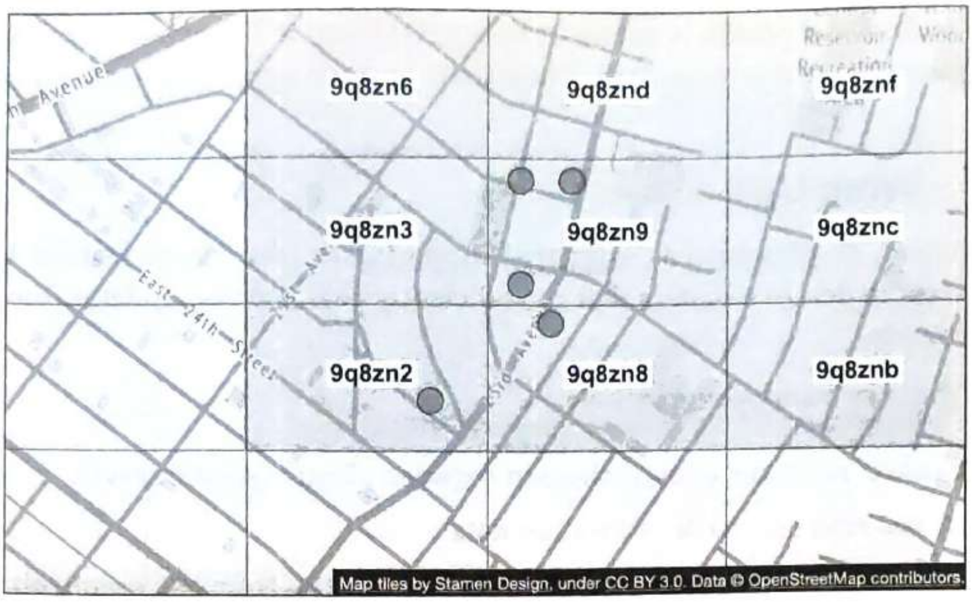


Figure 2.13: Publish location update to random nearby person

1. Here, when user 2 updates their location, the WebSocket connection handler computes the user's geohash ID and sends the location to the channel for that geohash.
2. Anyone nearby who subscribes to the channel (exclude the sender) will receive a location update message.

To handle people who are close to the border of a geohash grid, every client could subscribe to the geohash the user is in and the eight surrounding geohash grids. An example with all 9 geohash grids highlighted is shown in Figure 2.14.



Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors.

Figure 2.14: Nine geohash grids

Alternative to Redis Pub/Sub

Is there any good alternative to using Redis Pub/Sub as the routing layer? The answer is a resounding yes. Erlang [7] is a great solution for this particular problem. We would argue that Erlang is a better solution than the Redis Pub/Sub proposed above. However, Erlang is quite a niche, and hiring good Erlang programmers is hard. But if your team has Erlang expertise, this is a great option.

So, why Erlang? Erlang is a general programming language and runtime environment built for highly distributed and concurrent applications. When we say Erlang here, we specifically talk about the Erlang ecosystem itself. This includes the language component (Erlang or Elixir [8]) and the runtime environment and libraries (the Erlang virtual machine called BEAM [9] and the Erlang runtime libraries called OTP [10]).

The power of Erlang lies in its lightweight processes. An Erlang process is an entity running on the BEAM VM. It is several orders of magnitude cheaper to create than a Linux process. A minimal Erlang process takes about 300 bytes, and we can have millions of these processes on a single modern server. If there is no work to do in an Erlang process, it just sits there without using any CPU cycles at all. In other words, it is extremely cheap to model each of the 10 million active users in our design as an individual Erlang process.

Erlang is also very easy to distribute among many Erlang servers. The operational overhead is very low, and there are great tools to support debugging live production issues, safely. The deployment tools are also very strong.

How would we use Erlang in our design? We would implement the WebSocket service in Erlang, and also replace the entire cluster of Redis Pub/Sub with a distributed Erlang application. In this application, each user is modeled as an Erlang process. The user process would receive updates from the WebSocket server when a user's location is updated by the client. The user process also subscribes to updates from the Erlang processes of the user's friends. Subscription is native in Erlang/OTP and it's easy to build. This forms a mesh of connections that would efficiently route location updates from one user to many friends.

Step 4 - Wrap Up

In this chapter, we presented a design that supports a nearby friends feature. Conceptually, we want to design a system that can efficiently pass location updates from one user to their friends.

Some of the core components include:

- WebSocket: real-time communication between clients and the server.
- Redis: fast read and write of location data.
- Redis Pub/Sub: routing layer to direct location updates from one user to all the online friends.

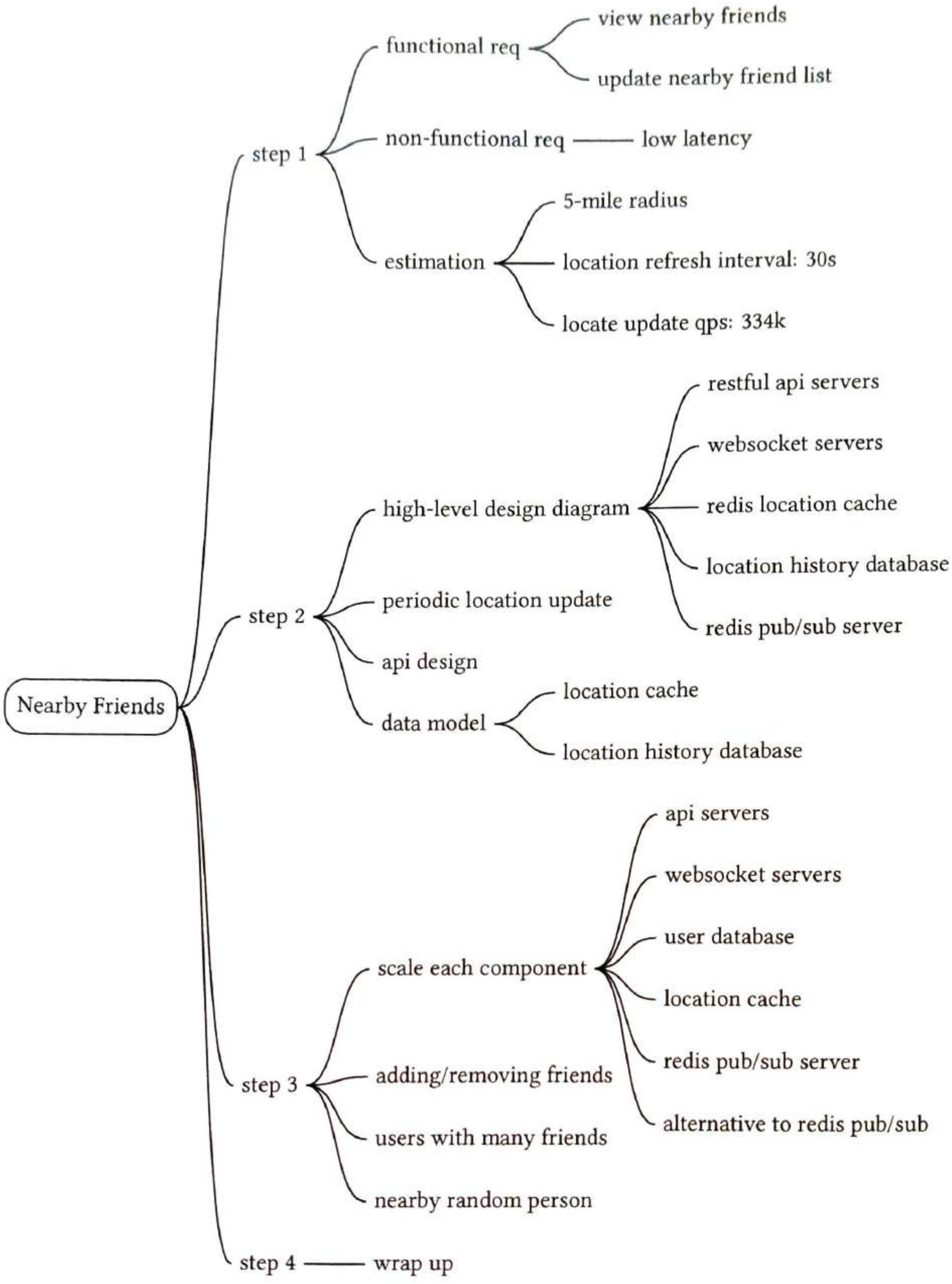
We first came up with a high-level design at a lower scale and then discussed challenges that arise as the scale increases. We explored how to scale the following:

- RESTful API servers
- WebSocket servers
- Data layer
- Redis Pub/Sub servers
- Alternative to Redis Pub/Sub

Finally, we discussed potential bottlenecks when a user has many friends and we proposed a design for the "nearby random person" feature.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Facebook Launches "Nearby Friends". https://techcrunch.com/2014/04/17/facebook-nearby-friends/.

[2] Redis Pub/Sub. https://redis.io/topics/pubsub.

[3] Redis Pub/Sub under the hood. https://making.pusher.com/redis-pubsub-under-the-hood/.

[4] etcd. https://etcd.io/.

[5] ZooKeeper. https://zookeeper.apache.org/.

[6] Consistent hashingones. https://www.toptal.com/big-data/consistent-hashing.

[7] Erlang. https://www.erlang.org/.

[8] Elixir. https://elixir-lang.org/.

[9] A brief introduction to BEAM. https://www.erlang.org/blog/a-brief-beam-primer/.

[10] OTP. https://www.erlang.org/doc/design_principles/des_princ.html.

3 Google Maps

In this chapter, we design a simple version of Google Maps. Before we proceed to the system design, let's learn a bit about Google Maps. Google started Project Google Maps in 2005 and developed a web mapping service. It provides many services such as satellite imagery, street maps, real-time traffic conditions, and route planning [1].

Google Maps helps users find directions and navigate to their destination. As of March 2021, Google Maps had one billion daily active users, 99% coverage of the world, and 25 million updates daily of accurate and real-time location information [2]. Given the enormous complexity of Google Maps, it is important to nail down which features our version of it supports.

Step 1 - Understand the Problem and Establish Design Scope

The interaction between the interviewer and the candidate could look like this:

Candidate: How many daily active users are we expecting?
Interviewer: 1 billion DAU.

Candidate: Which features should we focus on? Direction, navigation, and estimated time of arrival (ETA)?
Interviewer: Let's focus on location update, navigation, ETA, and map rendering.

Candidate: How large is the road data? Can we assume we have access to it?
Interviewer: Great questions. Yes, let's assume we obtained the road data from different sources. It is terabytes (TBs) of raw data.

Candidate: Should our system take traffic conditions into consideration?
Interviewer: Yes, traffic conditions are very important for accurate time estimation.

Candidate: How about different travel modes such as driving, walking, bus, etc?
Interviewer: We should be able to support different travel modes.

Candidate: Should it support multi-stop directions?
Interviewer: It is good to allow a user to define multiple stops, but let's not focus on it.

Candidate: How about business places and photos? How many photos are we expecting?
Interviewer: I am happy you asked and considered these. We do not need to design those.

In the rest of the chapter, we focus on three key features. The main devices that we need to support are mobile phones.

- User location update.
- Navigation service, including ETA service.
- Map rendering.

Non-functional requirements and constraints

- Accuracy: Users should not be given the wrong directions.
- Smooth navigation: On the client-side, users should experience very smooth map rendering.
- Data and battery usage: The client should use as little data and battery as possible. This is very important for mobile devices.
- General availability and scalability requirements.

Before jumping into the design, we will briefly introduce some basic concepts and terminologies that are helpful in designing Google Maps.

Map 101

Positioning system

The world is a sphere that rotates on its axis. At the very top, there is the north pole, and the very bottom is the south pole.

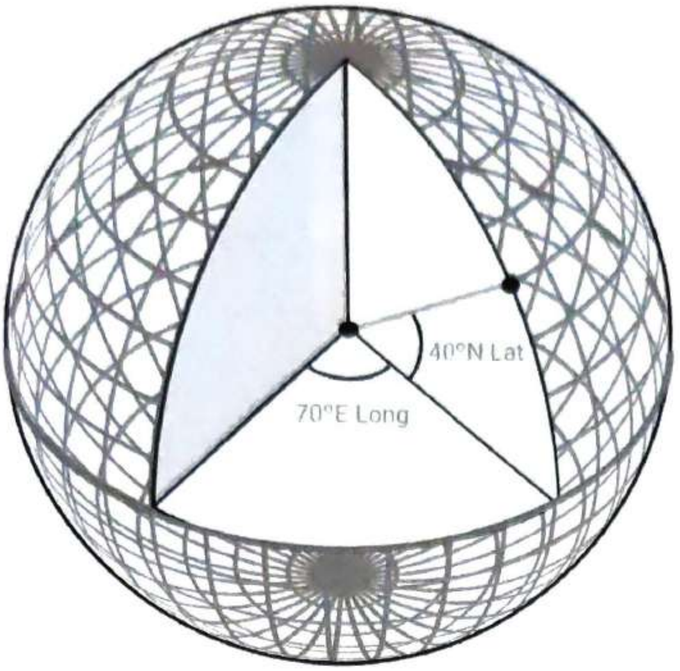


Figure 3.1: Latitude and longitude (source: [3])

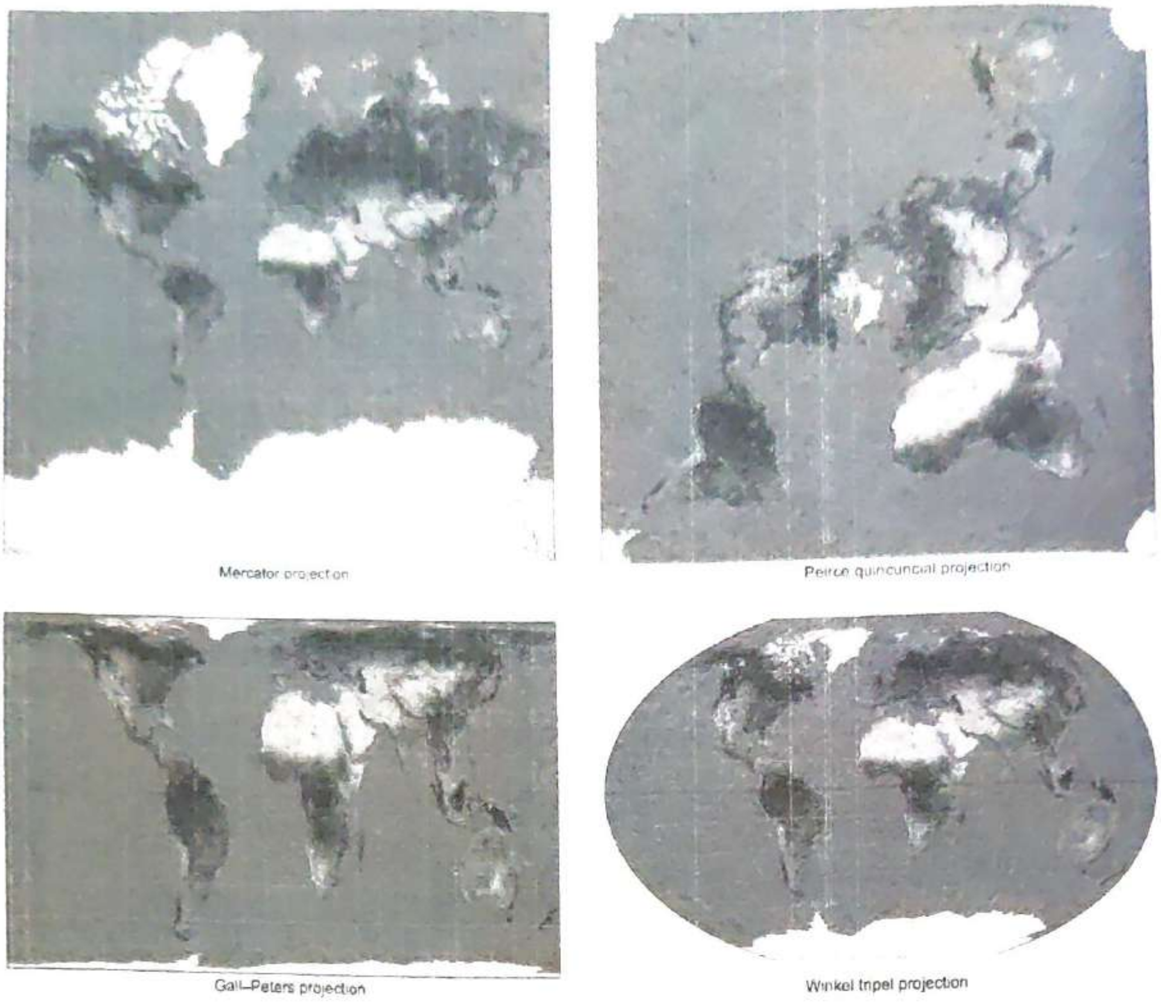
Lat (Latitude): denotes how far north or south we are

Long (Longitude): denotes how far east or west we are

Going from 3D to 2D

The process of translating the points from a 3D globe to a 2D plane is called "Map Projection".

There are different ways to do map projection, and each comes with its own strengths and limitations. Almost all of them distort the actual geometry. Below we can see some examples.



Mercator projection

Peirce quincuncial projection

Gall–Peters projection

Winkel tripel projection

Figure 3.2: Map projections (source: Wikipedia [4] [5] [6] [7])

Google Maps selected a modified version of Mercator projection called Web Mercator. For more details on positioning systems and projections, please refer to [3].

Geocoding

Geocoding is the process of converting addresses to geographic coordinates. For instance, "1600 Amphitheatre Parkway, Mountain View, CA" is geocoded to a latitude/longitude pair of (latitude 37.423021, longitude −122.083739).

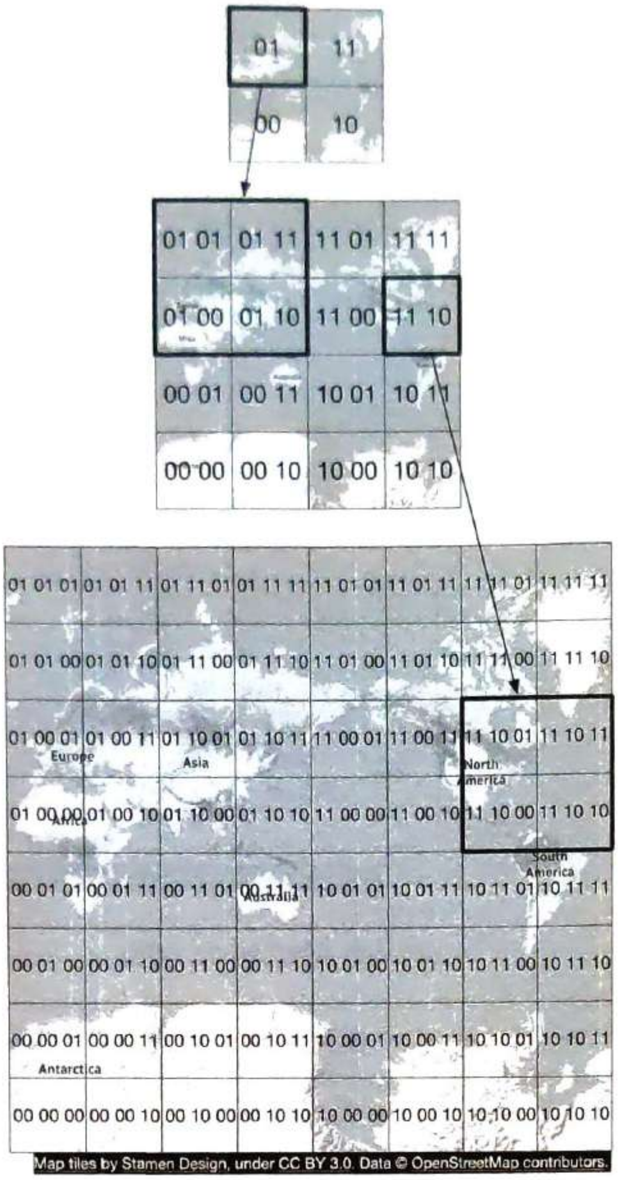
In the other direction, the conversion from the latitude/longitude pair to the actual human-readable address is called reverse geocoding.

One way to geocode is interpolation [8]. This method leverages the data from different sources such as geographic information systems (GIS) where the street network is mapped to the geographic coordinate space.

Geohashing

Geohashing is an encoding system that encodes a geographic area into a short string of letters and digits. At its core, it depicts the earth as a flattened surface and recursively divides the grids into sub-grids, which can be square or rectangular. We represent each grid with a string of numbers between 0 to 3 that are created recursively.

Let's assume the initial flattened surface is of size 20,000km × 10,000km. After the first division, we would have 4 grids of size 10,000km × 5,000km. We represent them as 00, 01, 10, and 11 as shown in Figure 3.3. We further divide each grid into 4 grids and use the same naming strategy. Each sub-grid is now of size 5,000km × 2,500km. We recursively divide the grids until each grid reaches a certain size threshold.



Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors.

Figure 3.3: Geohashing

Geohashing has many uses. In our design, we use geohashing for map tiling. For more details on geohashing and its benefits, please refer to [9].

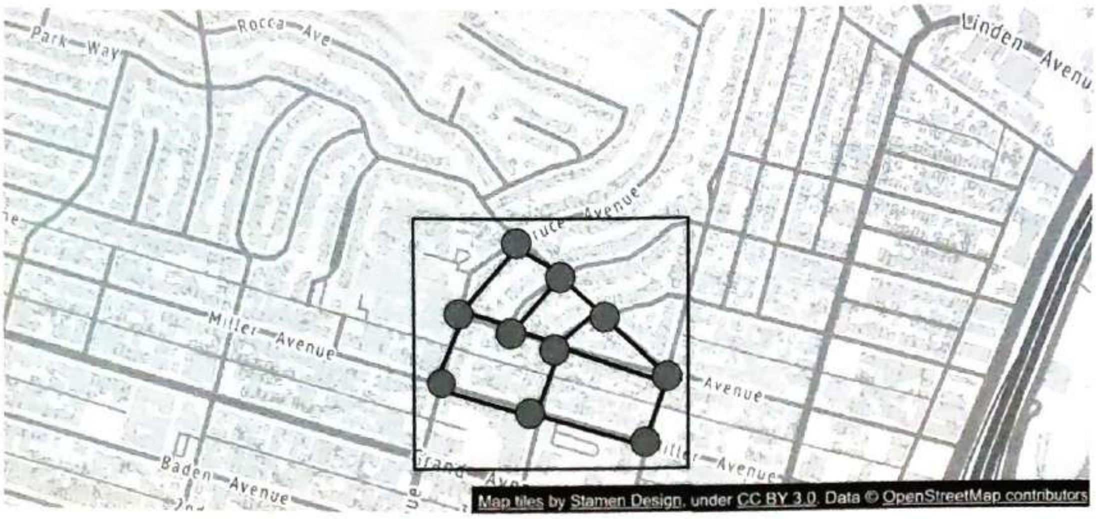
Map rendering

We won't go into a lot of detail about map rendering here, but it is worth mentioning the basics. One foundational concept in map rendering is tiling. Instead of rendering the entire map as one large custom image, the world is broken up into smaller tiles. The client only downloads the relevant tiles for the area the user is in and stitches them together like a mosaic for display.

There are distinct sets of tiles at different zoom levels. The client chooses the set of tiles appropriate for the zoom level of the map viewport on the client. This provides the right level of map details without consuming excess bandwidth. To illustrate with an extreme example, when the client is zoomed all the way out to show the entire world, we don't want to have to download hundreds of thousands of tiles for a very high zoom level. All the details would go to waste. Instead, the client would download one tile at the lowest zoom level, which represents the entire world with a single 256 × 256 pixel image.

Road data processing for navigation algorithms

Most routing algorithms are variations of Dijkstra's or A* pathfinding algorithms. The exact algorithm choice is a complex topic and we won't go into much detail in this chapter. What is important to note is that all these algorithms operate on a graph data structure, where intersections are nodes and roads are edges of the graph. See Figure 3.4 for an example:



Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors

Figure 3.4: Map as a graph

The pathfinding performance for most of these algorithms is extremely sensitive to the size of the graph. Representing the entire world of road networks as a single graph would consume too much memory and is likely too large for any of these algorithms to run efficiently. The graph needs to be broken up into manageable units for these algorithms to work at our design scale.

One way to break up road networks around the world is very similar to the tiling concept we discussed for map rendering. By employing a similar subdivision technique as geohashing, we divide the world into small grids. For each grid, we convert the roads within the grid into a small graph data structure that consists of the nodes (intersections) and edges (roads) inside the geographical area covered by the grid. We call these grids routing tiles. Each routing tile holds references to all the other tiles it connects to. This is how the routing algorithms can stitch together a bigger road graph as it traverses these interconnected routing tiles.

By breaking up road networks into routing tiles that can be loaded on demand, the routing algorithms can significantly reduce memory consumption and improve pathfinding performance by only consuming a small subset of the routing tiles at a time, and only loading additional tiles as needed.

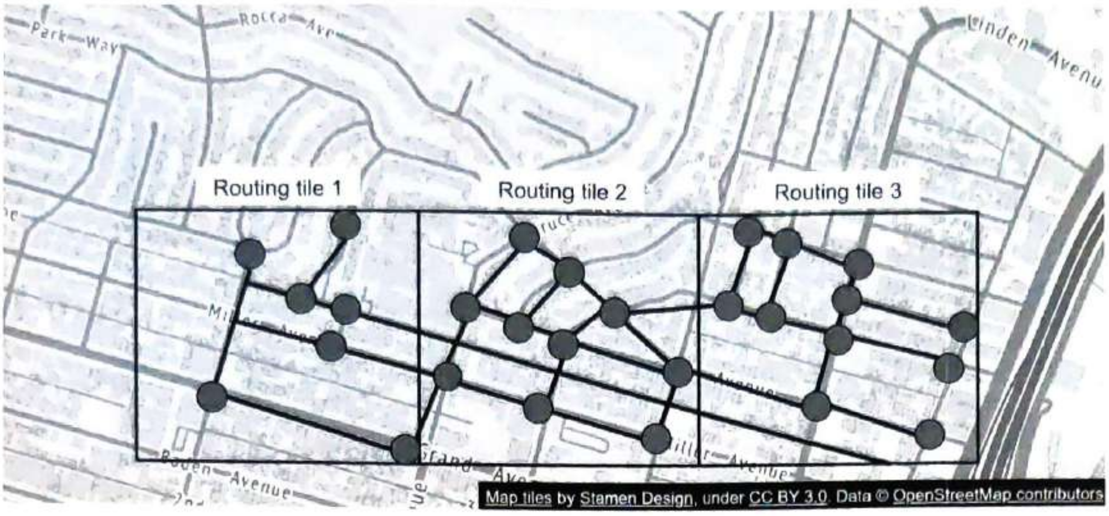

Routing tile 1
Routing tile 2
Routing tile 3
Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors

Figure 3.5: Routing tiles

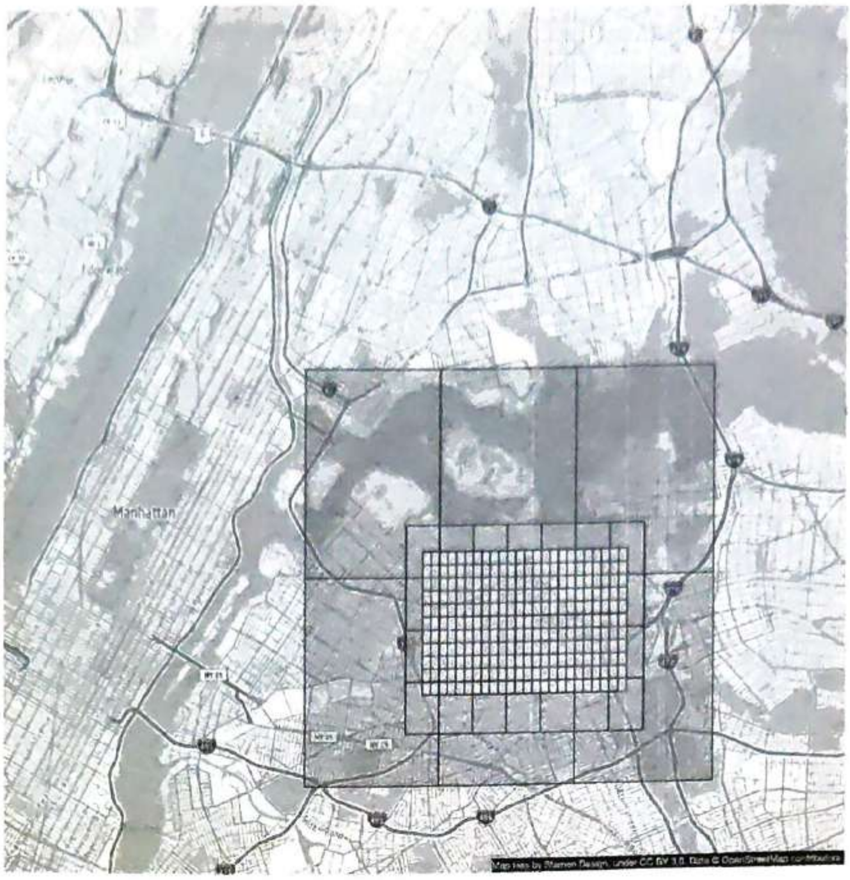
Reminder

In Figure 3.5, we call these grids routing tiles. Routing tiles are similar to map tiles in that both are grids covering certain geographical areas. Map tiles are PNG images, while routing tiles are binary files of road data for the area covered by the tiles.

Hierarchical routing tiles

Efficient navigation routing also requires having road data at the right level of detail. For example, for cross country routing, it would be slow to run the routing algorithm against a highly detailed set of street-level routing tiles. The graph stitched together from these detailed routing tiles would likely be too large and consume too much memory.

There are typically three sets of routing tiles with different levels of detail. At the most detailed level, the routing tiles are small and contain only local roads. At the next level, the tiles are bigger and contain only arterial roads connecting districts together. At the lowest level of detail, the tiles cover large areas and contain only major highways connecting cities and states together. At each level, there could be edges connecting to tiles at a different zoom level. For example, for a freeway entrance from local street A to freeway F, there would be a reference from the node (street A) in the small tile to the node (freeway F) in the big tile. See Figure 3.6 for an example of routing tiles of varying sizes.



Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors

Figure 3.6: Routing tiles of varying sizes

Back-of-the-envelope estimation

Now that we understand the basics, let's do a back-of-the-envelope estimation. Since the focus of the design is mobile, data usage and battery consumption are two important factors to consider.

Before we dive into the estimation, here are some imperial/metric conversions for reference.

- 1 foot = 0.3048 meters
- 1 kilometer (km) = 0.6214 miles
- 1 km = 1,000 meters

Storage usage

We need to store three types of data.

- Map of the world: A detailed calculation is shown below.
- Metadata: Given that the metadata for each map tile could be negligible in size, we can skip the metadata in our computation.
- Road info: The interviewer told us there are TBs of road data from external sources. We transform this dataset into routing tiles, which are also likely to be terabytes in size.

Map of the world

We discussed the concept of map tiling in the "Map 101" section on page 60. There are many sets of map tiles, with one at each zoom level. To get an idea of the storage requirement for the entire collection of map tile images, it would be informative to estimate the size of the largest tile set at the highest zoom level first. At zoom level 21, there are about 4.3 trillion tiles (Table 3.1). Let's assume that each tile is a 256 × 256 pixel compressed PNG image, with the image size of about 100KB. The entire set at the highest zoom level would need about 4.4 trillion × 100KB = 440PB.

In Table 3.1, we show the progression of tile counts at every zoom level.

Zoom	Number of Tiles
0	1
1	4
2	16
3	64
4	256
5	1 024
6	4 096
7	16 384
8	65 536
9	262 144
10	1 048 576
11	4 194 304
12	16 777 216
13	67 108 864
14	268 435 456
15	1 073 741 824
16	4 294 967 296
17	17 179 869 184
18	68 719 476 736
19	274 877 906 944
20	1 099 511 627 776
21	4 398 046 511 104

Table 3.1: Zoom levels

However, keep in mind that about 90% of the world's surface is natural and mostly uninhabited areas like oceans, deserts, lakes, and mountains. Since these areas are highly compressible as images, we could conservatively reduce the storage estimate by $80 \sim 90\%$. That would reduce the storage size to a range of 44 to 88PB. Let's pick a simple round number of 50PB.

Next, let's estimate how much storage each subsequent lower zoom level would take. At each lower zoom level, the number of tiles for both north-south and east-west directions drops by half. This results in a total reduction of the number of tiles by 4x, which drops the storage size for the zoom level also by 4x. With the storage size reduced by 4x at each lower zoom level, the math for the total size is a series: $50 + \frac{50}{4} + \frac{50}{16} + \frac{50}{64} + \cdots =\sim 67$PB. This is just a rough estimate. It is good enough to know that we need roughly about 100PB to store all the map tiles at varying levels of detail.

Server throughput

To estimate the server throughput, let's review the types of requests we need to support. There are two main types of requests. The first is navigation requests. These are sent by the clients to initiate a navigation session. The second is location update requests. These are sent by the client as the user moves around during a navigation session. The location data is used by downstream services in many different ways. For example, location data is one of the inputs for live traffic data. We will cover the use cases of location data in the design deep dive section.

Now we can analyze the server throughput for navigation requests. Let's assume we have 1 billion DAU, and each user on average uses navigation for a total of 35 minutes per week. This translates to 35 billion minutes per week or 5 billion minutes per day.

One simple approach would be to send GPS coordinates every second, which results in 300 billion (5 billion minutes × 60) requests per day, or 3 million QPS $\left(\frac{\text{300 billion requests}}{10^5} = 3 \text{ million}\right)$. However, the client may not need to send a GPS update every second. We can batch these on the client and send them at a much lower frequency (for example, every 15 seconds or 30 seconds) to reduce the write QPS. The actual frequency could depend on factors such as how fast the user moves. If they are stuck in traffic, a client can slow down the GPS updates. In our design, we assume GPS updates are batched and then sent to the server every 15 seconds. With this batched approach, the QPS is reduced to 200,000 $\left(\frac{\text{3 million}}{15}\right)$.

Assume peak QPS is five times the average. Peak QPS for location updates $= 200{,}000 \times 5 = 1$ million.

Step 2 - Propose High-level Design and Get Buy-in

Now that we have more knowledge about Google Maps, we are ready to propose a high-level design (Figure 3.7).

High-level design

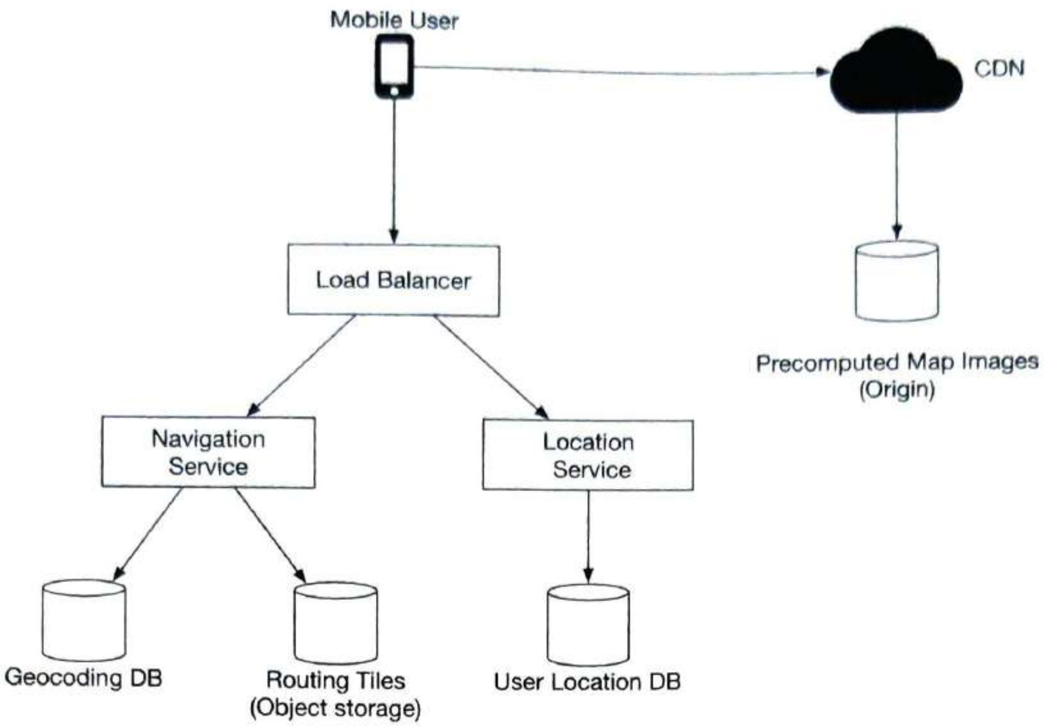


Figure 3.7: High-level design

The high-level design supports three features. Let's take a look at them one by one.

1. Location service
2. Navigation service
3. Map rendering

Location service

The location service is responsible for recording a user's location update. The architecture is shown in Figure 3.8.

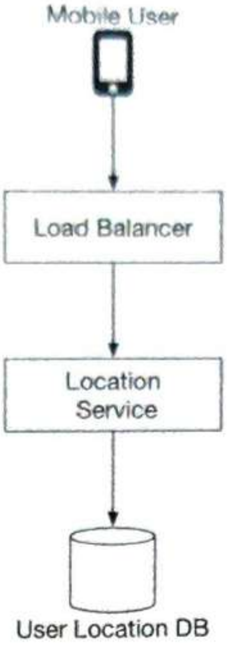


Figure 3.8: Location service

The basic design calls for the clients to send location updates every t seconds, where t is a configurable interval. The periodic updates have several benefits. First, we can leverage the streams of location data to improve our system over time. We can use the data to monitor live traffic, detect new or closed roads, and analyze user behavior to enable personalization, for example. Second, we can leverage the location data in near real-time to provide more accurate ETA estimates to the users and to reroute around traffic, if necessary.

But do we really need to send every location update to the server immediately? The answer is probably no. Location history can be buffered on the client and sent in batch to the server at a much lower frequency. For example, as shown in Figure 3.9, the location updates are recorded every second, but are only sent to the server as part of a batch every 15 seconds. This significantly reduces the total update traffic sent by all the clients.

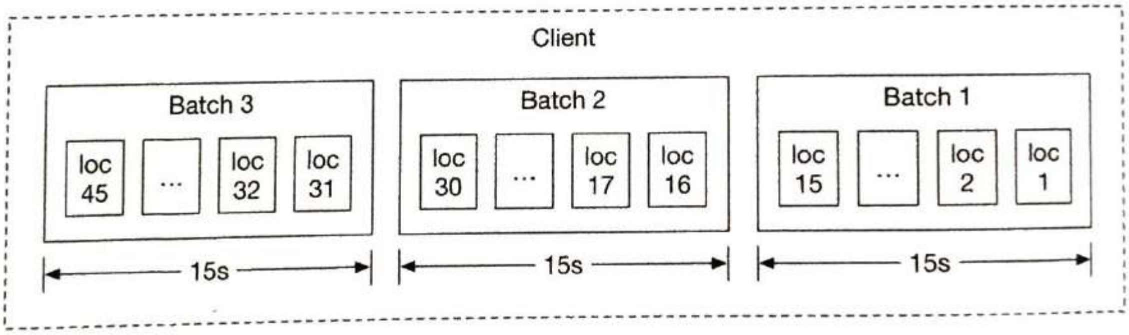


Figure 3.9: Batch requests

For a system like Google Maps, even when location updates are batched, the write volume is still very high. We need a database that is optimized for high write volume and is highly scalable, such as Cassandra. We may also need to log location data using a stream processing engine such as Kafka for further processing. We discuss this in detail in the deep dive section.

What communication protocol might be a good fit here? HTTP with the keep-alive option [10] is a good choice because it is very efficient. The HTTP request might look like this:

```
POST /v1/locations
Parameters
locs: JSON encoded array of (latitude, longitude, timestamp)
  tuples.
```

Navigation service

This component is responsible for finding a reasonably fast route from point A to point B. We can tolerate a little bit of latency. The calculated route does not have to be the fastest, but accuracy is critical.

As shown in Figure 3.8, the user sends an HTTP request to the navigation service through a load balancer. The request includes origin and destination as the parameters. The API might look like this:

```
GET /v1/nav?origin=1355+market+street,SF&destination=
Disneyland
```

Here is an example of what the navigation result could look like:

```
{
  'distance': {'text':'0.2 mi', 'value': 259},
  'duration': {'text': '1 min', 'value': 83},
  'end_location': {'lat': 37.4038943, 'Ing': -121.9410454},
  'html_instructions': 'Head <b>northeast</b> on <b>Brandon St
      </b> toward <b>Lumin Way</b><div style="font-size:0.9em">
      Restricted usage road</div>',
  'polyline': {'points': '_fhcFjbhgVuAwDsCal'},
  'start_location': {'lat': 37.4027165, 'lng': -121.9435809},
  'geocoded_waypoints': [
  {
    "geocoder_status" : "OK",
    "partial_match" : true,
    "place_id" : "ChIJwZNMti1fawwRO2aVVVX2yKg",
    "types" : [ "locality", "political" ]
  },
  {
    "geocoder_status" : "OK",
    "partial_match" : true,
    "place_id" : "ChIJ3aPgQGtXawwRLYeiBMUi7bM",
    "types" : [ "locality", "political" ]
  }
  ],
  'travel_mode': 'DRIVING'
}
```

Please refer to [11] for more details on Google Maps' official APIs.

So far we have not taken reroute and traffic changes into consideration. Those problems are tackled by the Adaptive ETA service in the deep dive section.

Map rendering

As we discussed in the back-of-the-envelope estimation, the entire collection of map tiles at various zoom levels is about a hundred petabytes in size. It is not practical to hold the entire dataset on the client. The map tiles must be fetched on-demand from the server based on the client's location and the zoom level of the client viewport.

When should the client fetch new map tiles from the server? Here are some scenarios:

- The user is zooming and panning the map viewpoint on the client to explore their surroundings.
- During navigation, the user moves out of the current map tile into a nearby tile.

We are dealing with a lot of data. Let's see how we could serve these map tiles from the server efficiently.

Option 1

The server builds the map tiles on the fly, based on the client location and zoom level of the client viewport. Considering that there is an infinite number of location and zoom level combinations, generating map tiles dynamically has a few severe disadvantages:

- It puts a huge load on the server cluster to generate every map tile dynamically.
- Since the map tiles are dynamically generated, it is hard to take advantage of caching.

Option 2

Another option is to serve a pre-generated set of map tiles at each zoom level. The map tiles are static, with each tile covering a fixed rectangular grid using a subdivision scheme like geohashing. Each tile is therefore represented by its geohash. In other words, there is a unique geohash associated with each grid. When a client needs a map tile, it first determines the map tile collection to use based on its zoom level. It then computes the map tile URL by converting its location to the geohash at the appropriate zoom level.

These static, pre-generated images are served by a CDN as shown in Figure 3.10.

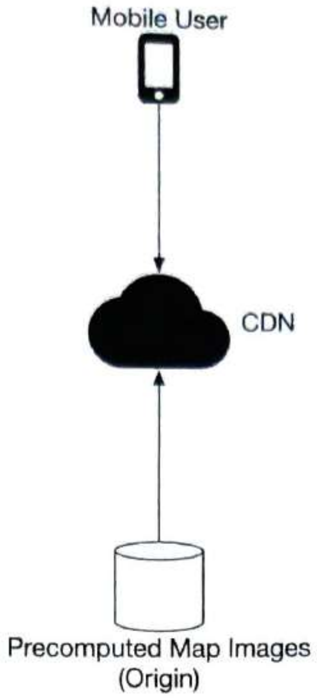


Figure 3.10: Pre-generated images are served by a CDN

In the diagram above, the mobile user makes an HTTP request to fetch a tile from the CDN. If the CDN has not yet served that specific tile before, it fetches a copy from the origin server, caches it locally, and returns it to the user. On subsequent requests, even if those requests are from different users, the CDN returns a cached copy without contacting the origin server.

This approach is more scalable and performant because the map tiles are served from the nearest point of presence (POP) to the client, as shown in Figure 3.11. The static nature of the map tiles makes them highly cacheable.

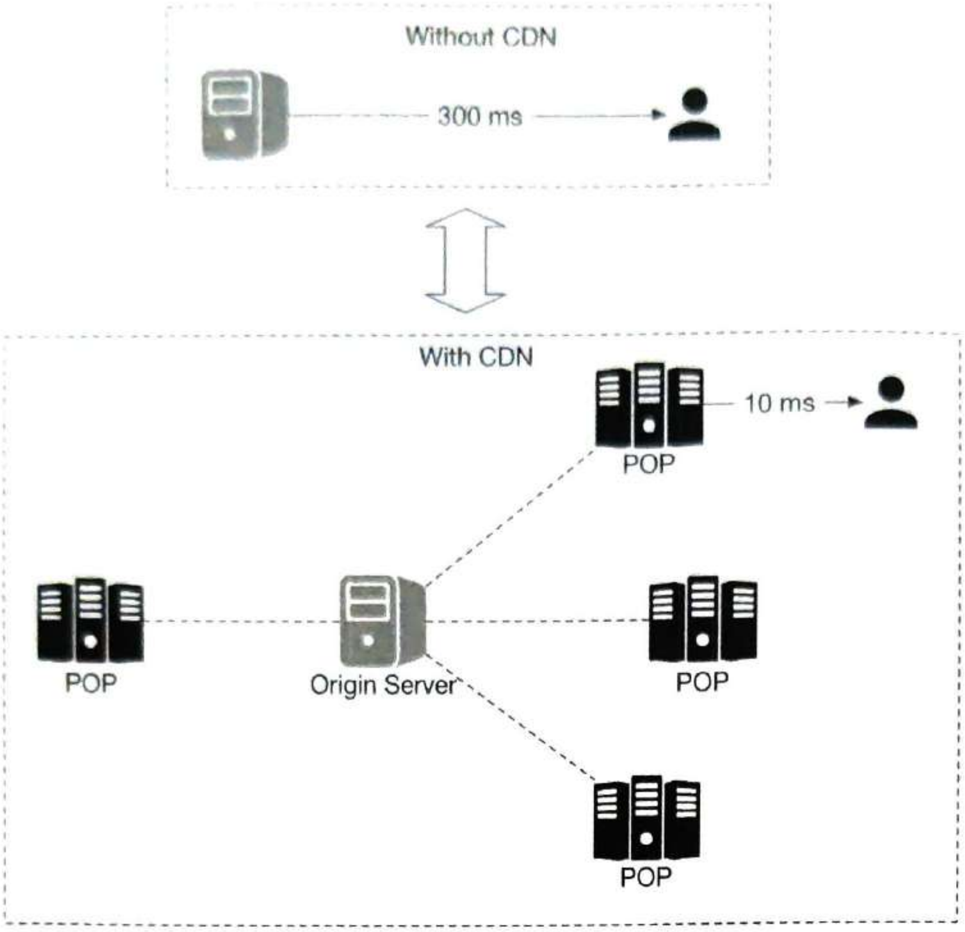


Figure 3.11: Without CDN vs with CDN

It is important to keep mobile data usage low. Let's calculate the amount of data the client needs to load during a typical navigation session. Note the following calculations don't take client-side caching into consideration. Since the routes a user takes could be similar each day, the data usage is likely to be a lot lower with client-side caching.

Data usage
Let's assume a user moves at 30km/h, and at a zoom level where each image covers a block of 200m × 200m (a block can be represented by a 256-pixel by 256-pixel image and the average image size is 100KB). For an area of 1km × 1km, we need 25 images or 2.5MB (25 × 100KB) of data. Therefore, if the speed is 30km/h, we need 75MB (30 × 2.5MB) of data per hour or 1.25MB of data per minute.

Next, we estimate the CDN data usage. At our scale, the cost is an important factor to consider.

Traffic through CDN

As mentioned earlier, we serve 5 billion minutes of navigation per day. This translates to 5 billion × 1.25MB = 6.25 billion MB per day. Hence, we serve 62,500MB $\left(\frac{6.25\text{ billion}}{10^5\text{ seconds in a day}}\right)$ of map data per second. With a CDN, these map images are going to be served from the POPs all over the world. Let's assume there are 200 POPs. Each POP would only need to serve a few hundred MBs $\left(\frac{62,500}{200}\right)$ per second.

There is one final detail in the map rendering design we have only briefly touched on. How does the client know which URLs to use to fetch the map tiles from the CDN? Keep in mind that we are using option 2 as discussed above. With that option, the map tiles are static and pre-generated based on fixed sets of grids, with each set representing a discrete zoom level.

Since we encode the grids in geohash, and there is one unique geohash per grid, computationally it is very efficient to go from the client's location (in latitude and longitude) and zoom level to the geohash, for the map tile. This calculation can be done on the client and we can fetch any static image tile from the CDN. For example, the URL for the image tile of Google headquarter could look like this: `https://cdn.map-provider.com/tiles/9q9hvu.png`

Refer to Chapter 1 Proximity Service on page 10 for a more detailed discussion of geohash encoding.

Calculating geohash on the client should work well. However, keep in mind that this algorithm is hardcoded in all the clients on all different platforms. Shipping changes to mobile apps is a time-consuming and somewhat risky process. We have to be sure that geohashing is the method we plan to use long-term to encode the collection of map tiles and that it is unlikely to change. If we need to switch to another encoding method for some reason, it will take a lot of effort and the risk is not low.

Here is another option worth considering. Instead of using a hardcoded client-side algorithm to convert a latitude/longitude (lat/lng) pair and zoom level to a tile URL, we could introduce a service as an intermediary whose job is to construct the tile URLs based on the same inputs mentioned above. This is a very simple service. The added operational flexibility might be worth it. This is a very interesting tradeoff discussion we could have with the interviewer. The alternative map rendering flow is shown in Figure 3.12.

When a user moves to a new location or to a new zoom level, the map tile service determines which tiles are needed and translates that information into a set of tile URLs to retrieve.

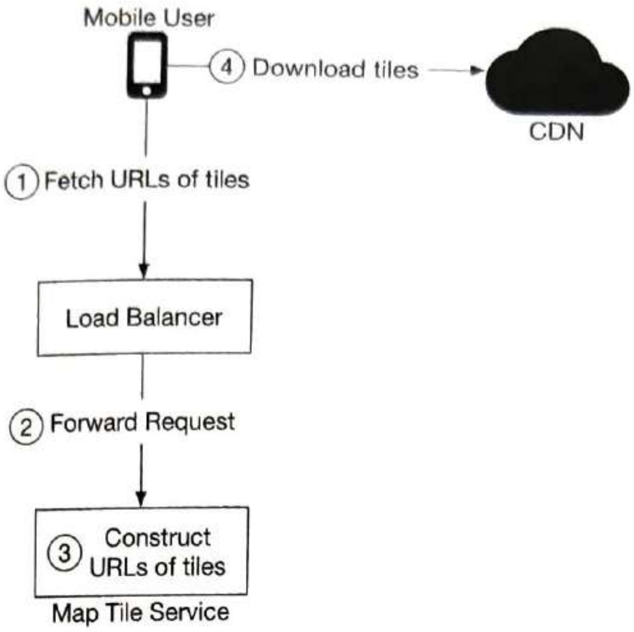


Figure 3.12: Map rendering

1. A mobile user calls the map tile service to fetch the tile URLs. The request is sent to the load balancer.
2. The load balancer forwards the request to the map tile service.
3. The map tile service takes the client's location and zoom level as inputs and returns 9 URLs of the tiles to the client. These tiles include the tile to render and the eight surrounding tiles.
4. The mobile client downloads the tiles from the CDN.

We will go into more detail on the precomputed map tiles in the design deep dive section.

Step 3 - Design Deep Dive

In this section, we first have a discussion about the data model. Then we talk about location service, navigation service, and map rendering in more detail.

Data model

We are dealing with four types of data: routing tiles, user location data, geocoding data, and precomputed map tiles of the world.

Routing tiles

As mentioned previously, the initial road dataset is obtained from different sources and authorities. It contains terabytes of data. The dataset is improved over time by the location data the application continuously collects from the users as they use the application.

This dataset contains a large number of roads and associated metadata such as names, county, longitude, and latitude. This data is not organized as graph data structures and is not usable by most routing algorithms. We run a periodic offline processing pipeline, called routing tile processing service, to transform this dataset into the routing tiles we introduced. The service runs periodically to capture new changes to the road data.

The output of the routing tile processing service is routing tiles. There are three sets of these tiles at different resolutions, as described in the "Map 101" section on page 60. Each tile contains a list of graph nodes and edges representing the intersections and roads within the area covered by the tile. It also contains references to other tiles its roads connect to. These tiles together form an interconnected network of roads that the routing algorithms can consume incrementally.

Where should the routing tile processing service store these tiles? Most graph data is represented as adjacency lists [12] in memory. There are too many tiles to keep the entire set of adjacency lists in memory. We could store the nodes and edges as rows in a database, but we would only be using the database as storage, and it seems an expensive way to store bits of data. We also don't need any database features for routing tiles.

The more efficient way to store these tiles is in object storage like S3 and cache it aggressively on the routing service that uses those tiles. There are many high-performance software packages we could use to serialize the adjacency lists to a binary file. We could organize these tiles by their geohashes in object storage. This provides a fast lookup mechanism to locate a tile by lat/lng pair.

We discuss how the shortest path service uses these routing tiles shortly.

User location data

User location data is valuable. We use it to update our road data and routing tiles. We also use it to build a database of live and historical traffic data. This location data is also consumed by multiple data stream processing services to update the map data.

For user location data, we need a database that can handle the write-heavy workload well and can be horizontally scaled. Cassandra could be a good candidate.

Here is what a single row could look like:

user_id	timestamp	user_mode	driving_mode	location
101	1635740977	active	driving	(20.0, 30.5)

Table 3.2: Location table

Geocoding database

This database stores places and their corresponding lat/lng pair. We can use a key-value database such as Redis for fast reads, since we have frequent reads and infrequent writes. We use it to convert an origin or destination to a lat/lng pair before passing it to the route planner service.

Precomputed images of the world map

When a device asks for a map of a particular area, we need to get nearby roads and compute an image that represents that area with all the roads and related details. These computations would be heavy and redundant, so it could be helpful to compute them once and then cache the images. We precompute images at different zoom levels and store them on a CDN, which is backed by cloud storage such as Amazon S3. Here is an example of such an image:

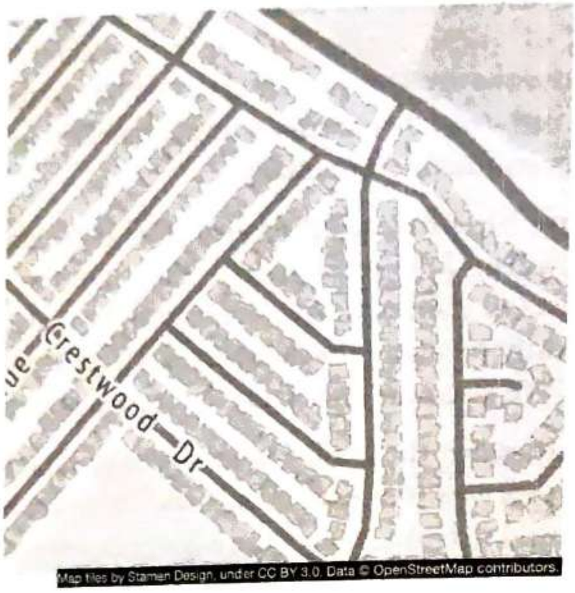

ue
Crestwood Dr
Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors.

Figure 3.13: Precomputed tiles

Services

Now that we have discussed the data model, let's take a close look at some of the most important services: location service, map rendering service, and navigation service.

Location service

In the high-level design, we discussed how location service works. In this section, we focus on the database design for this service and also how user location is used in detail.

In Figure 3.14, the key-value store is used to store user location data. Let's take a close look.

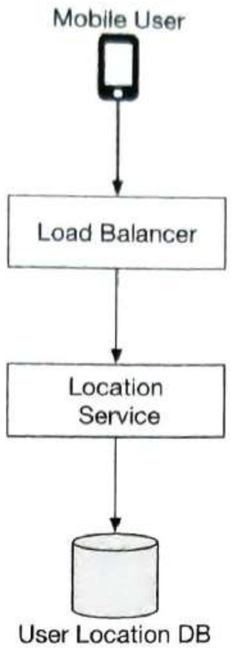


Figure 3.14: User location database

Given the fact we have 1 million location updates every second, we need to have a database that supports fast writes. A NoSQL key-value database or column-oriented database would be a good choice here. In addition, a user's location is continuously changing and becomes stale as soon as a new update arrives. Therefore, we can prioritize availability over consistency. The CAP theorem [13] states that we could choose two attributes among consistency, availability, and partition tolerance. Given our constraints, we would go with availability and partition tolerance. One database that is a good fit is Cassandra. It can handle our scale with a strong availability guarantee.

The key is the combination of `(user_id, timestamp)` and the value is a lat/lng pair. In this setup, `user_id` is the primary key and `timestamp` is the clustering key. The advantage of using `user_id` as the partition key is that we can quickly read the latest position of a specific user. All the data with the same partition key are stored together, sorted by `timestamp`. With this arrangement, the retrieval of the location data for a specific user within a time range is very efficient.

Below is an example of what the table may look like.

key (user_id)	timestamp	lat	long	user_mode	navigation_mode
51	132053000	21.9	89.8	active	driving

Table 3.3: Location data

How do we use the user location data?

User location data is essential. It supports many use cases. We use the data to detect new and recently closed roads. We use it as one of the inputs to improve the accuracy of our map over time. It is also an input for live traffic data.

To support these use cases, in addition to writing current user locations in our database, we log this information into a message queue, such as Kafka. Kafka is a unified low-latency, high-throughput data streaming platform designed for real-time data feeds. Figure 3.15 shows how Kafka is used in the improved design.

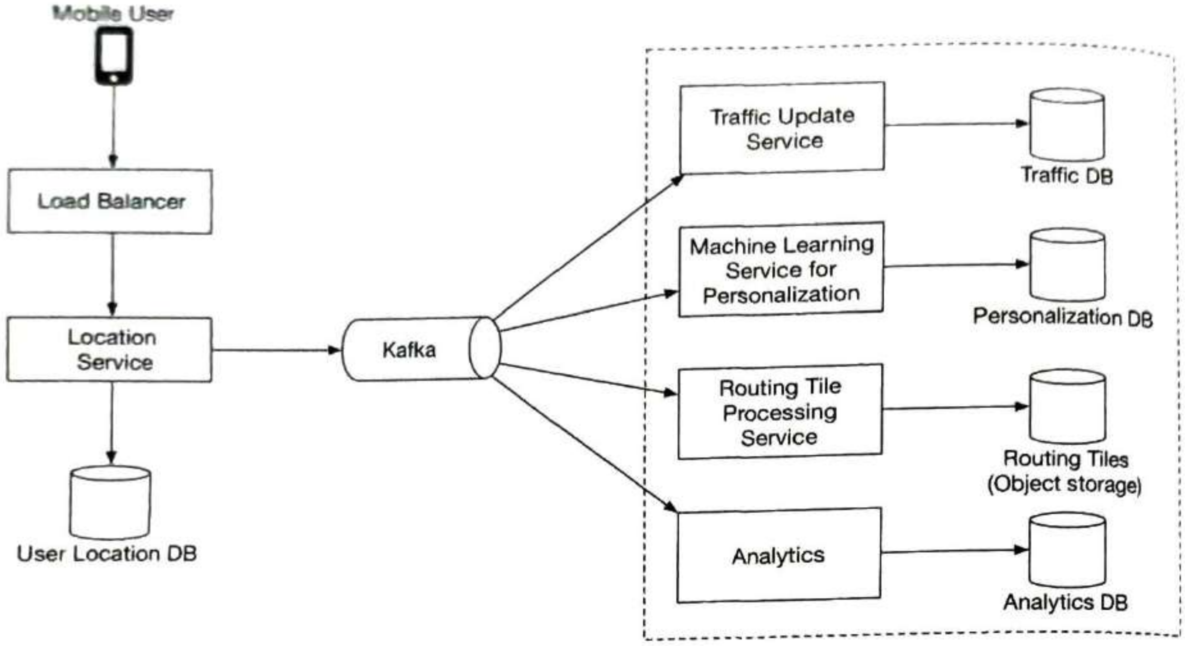


Figure 3.15: Location data is used by other services

Other services consume the location data stream from Kafka for various use cases. For instance, the live traffic service digests the output stream and updates the live traffic database. The routing tile processing service improves the map of the world by detecting new or closed roads and updating the affected routing tiles in object storage. Other services can also tap into the stream for different purposes.

Rendering map

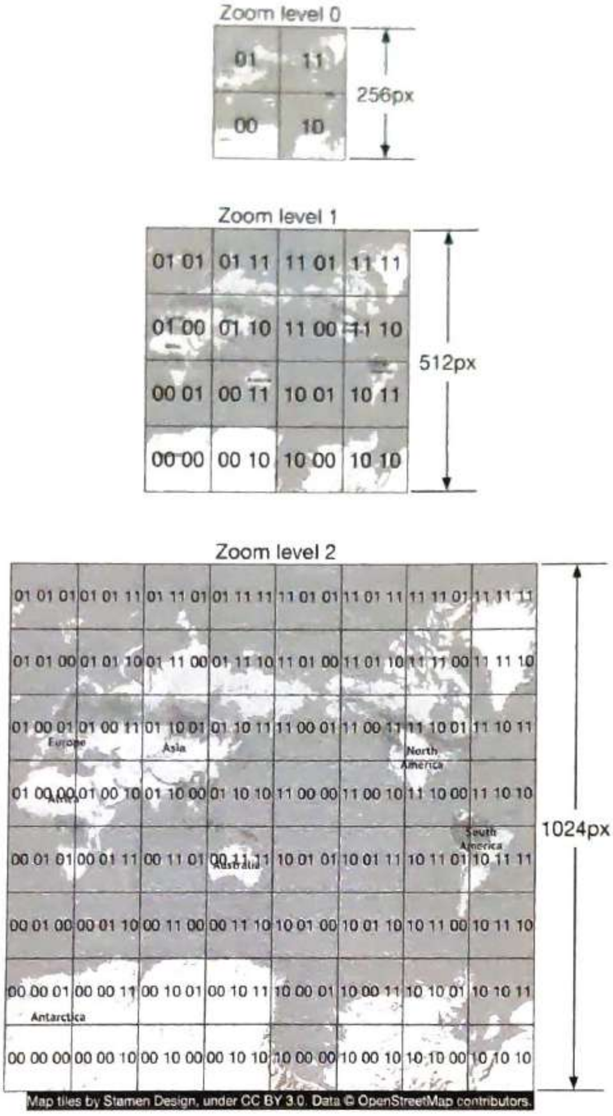
In this section, we dive deep into precomputed map tiles and map rendering optimization. They are primarily inspired by the work of Google Design [3].

Precomputed tiles

As mentioned previously, there are different sets of precomputed map tiles at various distinct zoom levels to provide the appropriate level of map detail to the user, based on the client's viewport size and zoom level. Google Maps uses 21 zoom levels (Table 3.1). This is what we use, as well.

Level 0 is the most zoomed-out level. The entire map is represented by a single tile of size 256 × 256 pixels.

With each increment of the zoom level, the number of map tiles doubles in both north-south and east-west directions, while each tile stays at 256 × 256 pixels. As shown in Figure 3.16, at zoom level 1, there are 2 × 2 tiles, with a total combined resolution of 512 × 512 pixels. At zoom level 2, there are 4 × 4 tiles, with a total combined resolution of 1024 × 1024 pixels. With each increment, the entire set of tiles has 4x as many pixels as the previous level. The increased pixel count provides an increasing level of detail to the user. This allows the client to render the map at the best granularities depending on the client's zoom level, without consuming excessive bandwidth to download tiles in excessive detail.



Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors.

Figure 3.16: Zoom levels

Optimization: use vectors

With the development and implementation of WebGL, one potential improvement is to change the design from sending the images over the network, to sending the vector information (paths and polygons) instead. The client draws the paths and polygons from the vector information.

One obvious advantage of vector tiles is that vector data compresses much better than images do. The bandwidth saving is substantial.

A less obvious benefit is that vector tiles provide a much better zooming experience. With rasterized images, as the client zooms in from one level to another, everything gets

Shortest-path service

The shortest-path service receives the origin and the destination in lat/lng pairs and returns the top-k shortest paths without considering traffic or current conditions. This computation only depends on the structure of the roads. Here, caching the routes could be beneficial because the graph rarely changes.

The shortest-path service runs a variation of A* pathfinding algorithms against the routing tiles in object storage. Here is an overview:

- The algorithm receives the origin and destination in lat/lng pairs. The lat/lng pairs are converted to geohashes which are then used to load the start and end-points of routing tiles.
- The algorithm starts from the origin routing tile, traverses the graph data structure, and hydrates additional neighboring tiles from object storage (or its local cache if it has loaded it before) as it expands the search area. It's worth noting that there are connections from one level of tile to another covering the same area. This is how the algorithm could "enter" the bigger tiles containing only highways, for example. The algorithm continues to expand its search by hydrating more neighboring tiles (or tiles at different resolutions) as needed until a set of best routes is found.

Figure 3.18 (based on [14]) gives a conceptual overview of the tiles used in the graph traversal.

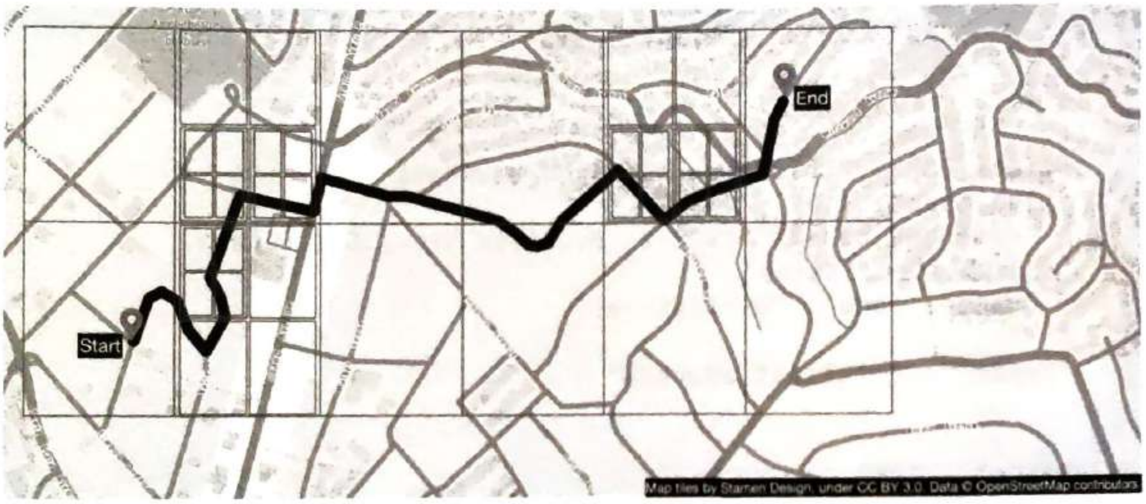

End
Start
Map tiles by Stamen Design, under CC BY 3.0. Data © OpenStreetMap contributors

Figure 3.18: Graph traversal

ETA service

Once the route planner receives a list of possible shortest paths, it calls the ETA service for each possible route and gets a time estimate. For this, the ETA service uses machine learning to predict the ETAs based on the current traffic and historical data.

One of the challenges here is that we not only need to have real-time traffic data but also to predict how the traffic will look like in 10 or 20 minutes. These kinds of challenges need to be addressed at an algorithmic level and will not be discussed in this section. If you are interested, refer to [15] and [16].

Ranker service

Finally, after the route planner obtains the ETA predictions, it passes this info to the ranker to apply possible filters as defined by the user. Some example filters include options to avoid toll roads or to avoid freeways. The ranker service then ranks the possible routes from fastest to slowest and returns top-k results to the navigation service.

Updater services

These services tap into the Kafka location update stream and asynchronously update some of the important databases to keep them up-to-date. The traffic database and the routing tiles are some examples.

The routing tile processing service is responsible for transforming the road dataset with newly found roads and road closures into a continuously updated set of routing tiles. This helps the shortest path service to be more accurate.

The traffic update service extracts traffic conditions from the streams of location updates sent by the active users. This insight is fed into the live traffic database. This enables the ETA service to provide more accurate estimates.

Improvement: adaptive ETA and rerouting

The current design does not support adaptive ETA and rerouting. To address this, the server needs to keep track of all the active navigating users and update them on ETA continuously, whenever traffic conditions change. Here we need to answer a few important questions:

- How do we track actively navigating users?
- How do we store the data, so that we can efficiently locate the users affected by traffic changes among millions of navigation routes?

Let's start with a naive solution. In Figure 3.19, user_1's navigation route is represented by routing tiles r_1, r_2, r_3, ..., r_7.

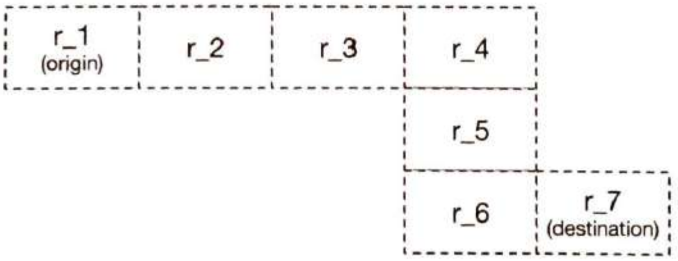


Figure 3.19: Navigation route

The database stores actively navigating users and routes information which might look like this:

```
user_1: r_1, r_2, r_3, ..., r_k
```

```
user_2: r_4, r_6, r_9, ..., r_n
user_3: r_2, r_8, r_9, ..., r_m
...
user_n: r_2, r_10, r_21, ..., r_l
```

Let's say there is a traffic incident in routing tile 2 (r_2). To figure out which users are affected, we scan through each row and check if routing tile 2 is in our list of routing tiles (see example below).

```
user_1: r_1, r_2, r_3, ..., r_k
user_2: r_4, r_6, r_9, ..., r_n
user_3: r_2, r_8, r_9, ..., r_m
...
user_n: r_2, r_10, r_21, ..., r_l
```

Assume the number of rows in the table is n and the average length of the navigation route is m. The time complexity to find all users affected by the traffic change is $O(n \times m)$.

Can we make this process faster? Let's explore a different approach. For each actively navigating user, we keep the current routing tile, the routing tile at the next resolution level that contains it, and recursively find the routing tile at the next resolution level until we find the user's destination in the tile as well (Figure 3.20). By doing this, we can get a row of the database table like this.

```
user_1, r_1, super(r_1), super(super(r_1)), ...
```

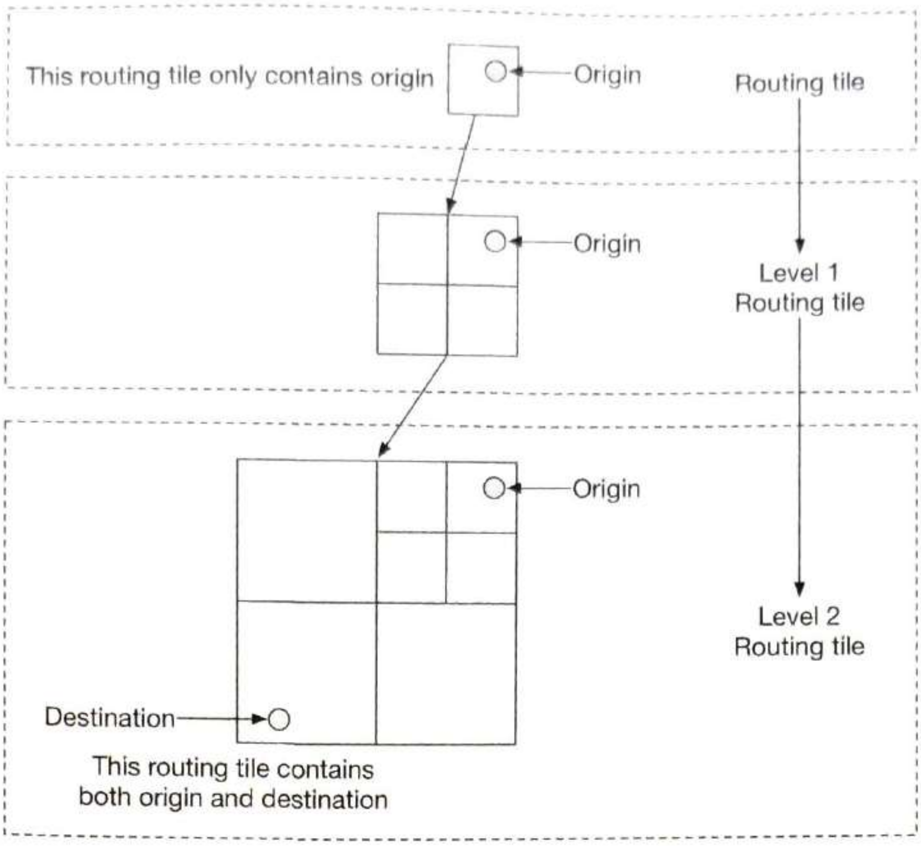

This routing tile only contains origin
Origin
Routing tile
Origin
Level 1
Routing tile
Origin
Level 2
Routing tile
Destination
This routing tile contains
both origin and destination

Figure 3.20: Build routing tiles

To find out if a user is affected by the traffic change, we need only check if a routing tile is inside the last routing tile of a row in the database. If not, the user is not impacted. If it is, the user is affected. By doing this, we can quickly filter out many users.

This approach doesn't specify what happens when traffic clears. For example, if routing tile 2 clears and users can go back to the old route, how do users know rerouting is available? One idea is to keep track of all possible routes for a navigating user, recalculate the ETAs regularly and notify the user if a new route with a shorter ETA is found.

Delivery protocols

It is a reality that during navigation, route conditions can change and the server needs a reliable way to push data to mobile clients. For delivery protocol from the server to the client, our options include mobile push notification, long polling, WebSocket, and Server-Sent Events (SSE).

- Mobile push notification is not a great option because the payload size is very limited (4,096 bytes for iOS) and it doesn't support web applications.
- WebSocket is generally considered to be a better option than long polling because it has a very light footprint on servers.
- Since we have ruled out the mobile push notification and long polling, the choice is mainly between WebSocket and SSE. Even though both can work, we lean towards WebSocket because it supports bi-directional communication and features such as

last-mile delivery might require bi-directional real-time communication.

For more details about ETA and rerouting, please refer to [15].

Now we have every piece of the design together. Please see the updated design in Figure 3.21.

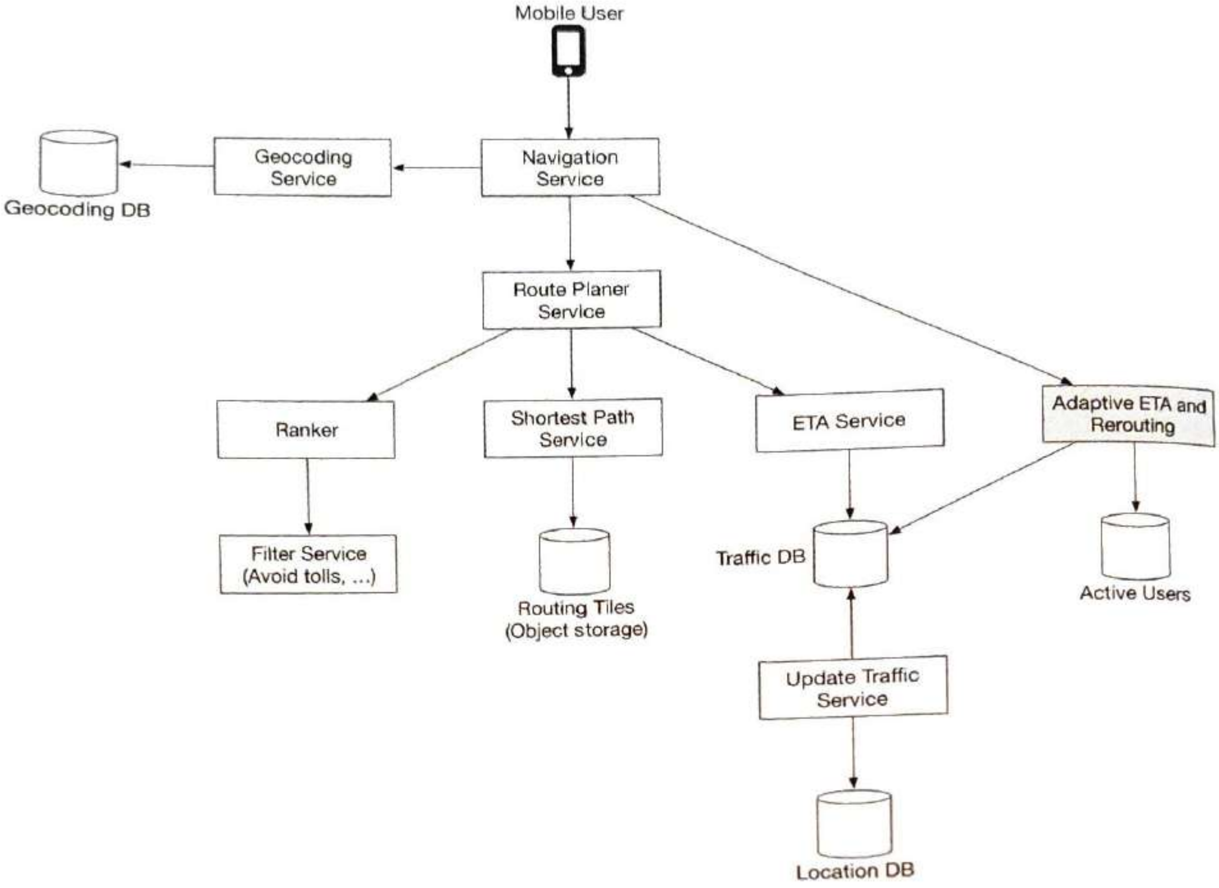


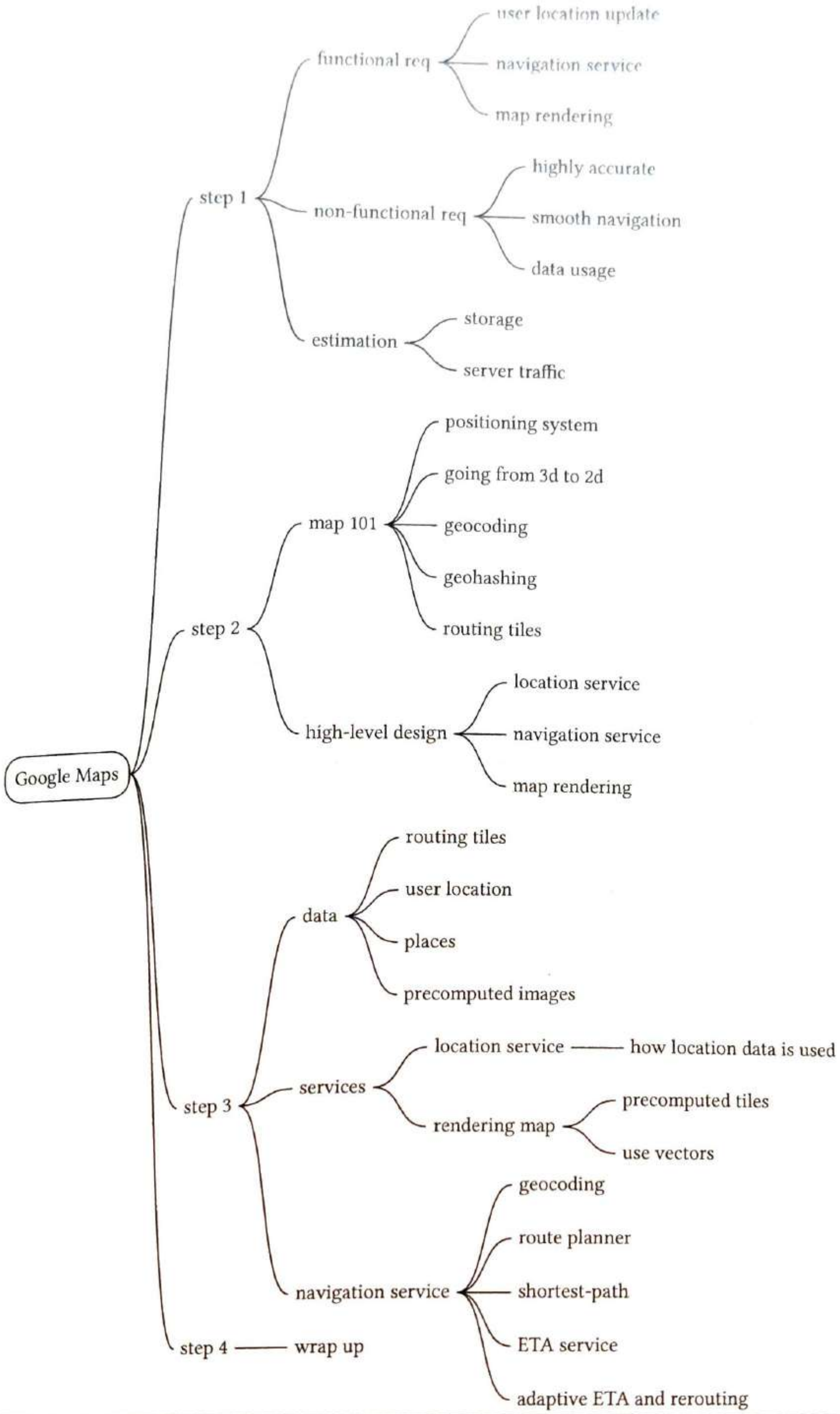
Figure 3.21: Final design

Step 4 - Wrap Up

In this chapter, we designed a simplified Google Maps application with key features such as location update, ETAs, route planning, and map rendering. If you are interested in expanding the system, one potential improvement would be to provide multi-stop navigation capability for enterprise customers. For example, for a given set of destinations, we have to find the optimal order in which to visit them all and provide proper navigation, based on live traffic conditions. This could be helpful for delivery services such as DoorDash, Uber, Lyft, etc.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Google Maps. https://developers.google.com/maps?hl=en_US.

[2] Google Maps Platform. https://cloud.google.com/maps-platform/.

[3] Prototyping a Smoother Map. https://medium.com/google-design/google-maps-cb0326d165f5.

[4] Mercator projection. https://en.wikipedia.org/wiki/Mercator_projection.

[5] Peirce quincuncial projection. https://en.wikipedia.org/wiki/Peirce_quincuncial_projection.

[6] Gall–Peters projection. https://en.wikipedia.org/wiki/Gall–Peters_projection.

[7] Winkel tripel projection. https://en.wikipedia.org/wiki/Winkel_tripel_projection.

[8] Address geocoding. https://en.wikipedia.org/wiki/Address_geocoding.

[9] Geohashing. https://kousiknath.medium.com/system-design-design-a-geo-spatial-index-for-real-time-location-search-10968fe62b9c.

[10] HTTP keep-alive. https://en.wikipedia.org/wiki/HTTP_persistent_connection.

[11] Directions API. https://developers.google.com/maps/documentation/directions/start?hl=en_US.

[12] Adjacency list. https://en.wikipedia.org/wiki/Adjacency_list.

[13] CAP theorem. https://en.wikipedia.org/wiki/CAP_theorem.

[14] Routing Tiles. https://valhalla.readthedocs.io/en/latest/mjolnir/why_tiles/.

[15] ETAs with GNNs. https://deepmind.com/blog/article/traffic-prediction-with-advanced-graph-neural-networks.

[16] Google Maps 101: How AI helps predict traffic and determine routes. https://blog.google/products/maps/google-maps-101-how-ai-helps-predict-traffic-and-determine-routes/.

4 Distributed Message Queue

In this chapter, we explore a popular question in system design interviews: design a distributed message queue. In modern architecture, systems are broken up into small and independent building blocks with well-defined interfaces between them. Message queues provide communication and coordination for those building blocks. What benefits do message queues bring?

- Decoupling. Message queues eliminate the tight coupling between components so they can be updated independently.
- Improved scalability. We can scale producers and consumers independently based on traffic load. For example, during peak hours, more consumers can be added to handle the increased traffic.
- Increased availability. If one part of the system goes offline, the other components can continue to interact with the queue.
- Better performance. Message queues make asynchronous communication easy. Producers can add messages to a queue without waiting for the response and consumers consume messages whenever they are available. They don't need to wait for each other.

Figure 4.1 shows some of the most popular distributed message queues on the market.

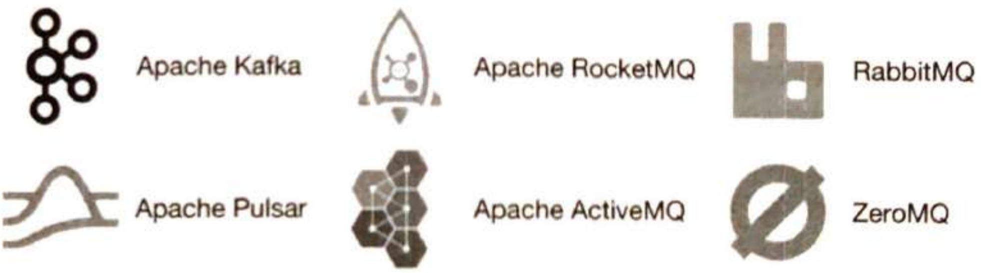


Figure 4.1: Popular distributed message queues

Message queues vs event streaming platforms

Strictly speaking, Apache Kafka and Pulsar are not message queues as they are event streaming platforms. However, there is a convergence of features that starts to blur the distinction between message queues (RocketMQ, ActiveMQ, RabbitMQ, ZeroMQ, etc.) and event streaming platforms (Kafka, Pulsar). For example, RabbitMQ, which is a typical message queue, added an optional streams feature to allow repeated message consumption and long message retention, and its implementation uses an append-only log, much like an event streaming platform would. Apache Pulsar is primarily a Kafka competitor, but it is also flexible and performant enough to be used as a typical distributed message queue.

In this chapter, we will design a distributed message queue with **additional features, such as long data retention, repeated consumption of messages, etc.**, that are typically only available on event streaming platforms. These additional features make the design more complicated. Throughout the chapter, we will highlight places where the design could be simplified if the focus of your interview centers around the more traditional distributed message queues.

Step 1 - Understand the Problem and Establish Design Scope

In a nutshell, the basic functionality of a message queue is straightforward: producers send messages to a queue, and consumers consume messages from it. Beyond this basic functionality, there are other considerations including performance, message delivery semantics, data detention, etc. The following set of questions will help clarify requirements and narrow down the scope.

Candidate: What's the format and average size of messages? Is it text only? Is multimedia allowed?
Interviewer: Text messages only. Messages are generally measured in the range of kilobytes (KBs).

Candidate: Can messages be repeatedly consumed?
Interviewer: Yes, messages can be repeatedly consumed by different consumers. Note that this is an added feature. A traditional distributed message queue does not retain a message once it has been successfully delivered to a consumer. Therefore, a message cannot be repeatedly consumed in a traditional message queue.

Candidate: Are messages consumed in the same order they were produced?
Interviewer: Yes, messages should be consumed in the same order they were produced. Note that this is an added feature. A traditional distributed message queue does not usually guarantee delivery orders.

Candidate: Does data need to be persisted and what is the data retention?
Interviewer: Yes, let's assume data retention is two weeks. This is an added feature. A traditional distributed message queue does not retain messages.

Candidate: How many producers and consumers are we going to support?

Interviewer: The more the better.

Candidate: What's the data delivery semantic we need to support? For example, at-most-once, at-least-once, and exactly once.
Interviewer: We definitely want to support at-least-once. Ideally, we should support all of them and make them configurable.

Candidate: What's the target throughput and end-to-end latency?
Interviewer: It should support high throughput for use cases like log aggregation. It should also support low latency delivery for more traditional message queue use cases.

With the above conversation, let's assume we have the following functional requirements:

- Producers send messages to a message queue.
- Consumers consume messages from a message queue.
- Messages can be consumed repeatedly or only once.
- Historical data can be truncated.
- Message size is in the kilobyte range.
- Ability to deliver messages to consumers in the order they were added to the queue.
- Data delivery semantics (at-least once, at-most once, or exactly once) can be configured by users.

Non-functional requirements

- High throughput or low latency, configurable based on use cases.
- Scalable. The system should be distributed in nature. It should be able to support a sudden surge in message volume.
- Persistent and durable. Data should be persisted on disk and replicated across multiple nodes.

Adjustments for traditional message queues

Traditional message queues like RabbitMQ do not have as strong a retention requirement as event streaming platforms. Traditional queues retain messages in memory just long enough for them to be consumed. They provide on-disk overflow capacity [1] which is several orders of magnitude smaller than the capacity required for event streaming platforms. Traditional message queues do not typically maintain message ordering. The messages can be consumed in a different order than they were produced. These differences greatly simplify the design which we will discuss where appropriate.

Step 2 - Propose High-level Design and Get Buy-in

First, let's discuss the basic functionalities of a message queue.

Figure 4.2 shows the key components of a message queue and the simplified interactions between these components.

tab
caps lock
shift
control