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 loca-

should still map to the same cache key. The geohash/quadtree solution mentioned 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: 0.5km, 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 with given precision. Therefore the total memory required is:

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

- Total memory required: $\sim$ 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.

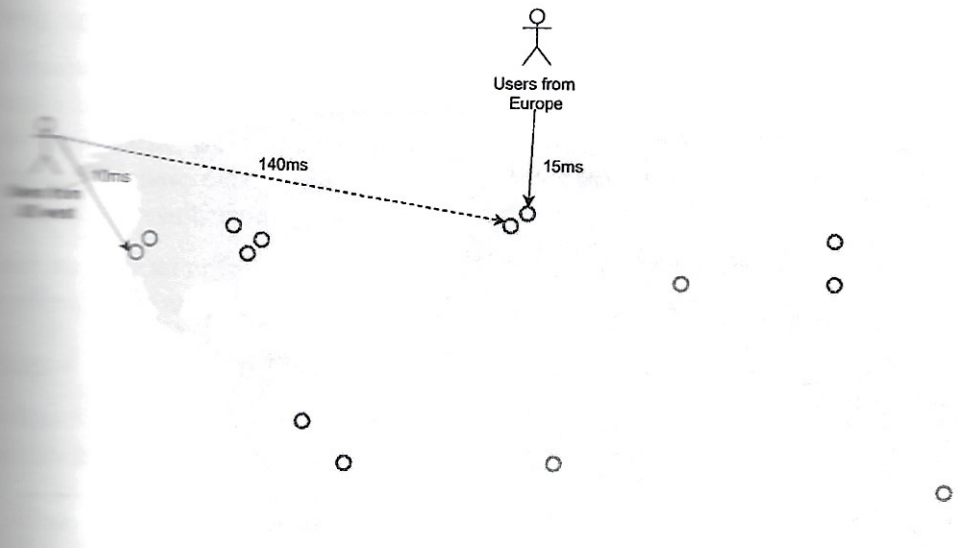


Figure 1.20: Deploy LBS "closer" to the user

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.

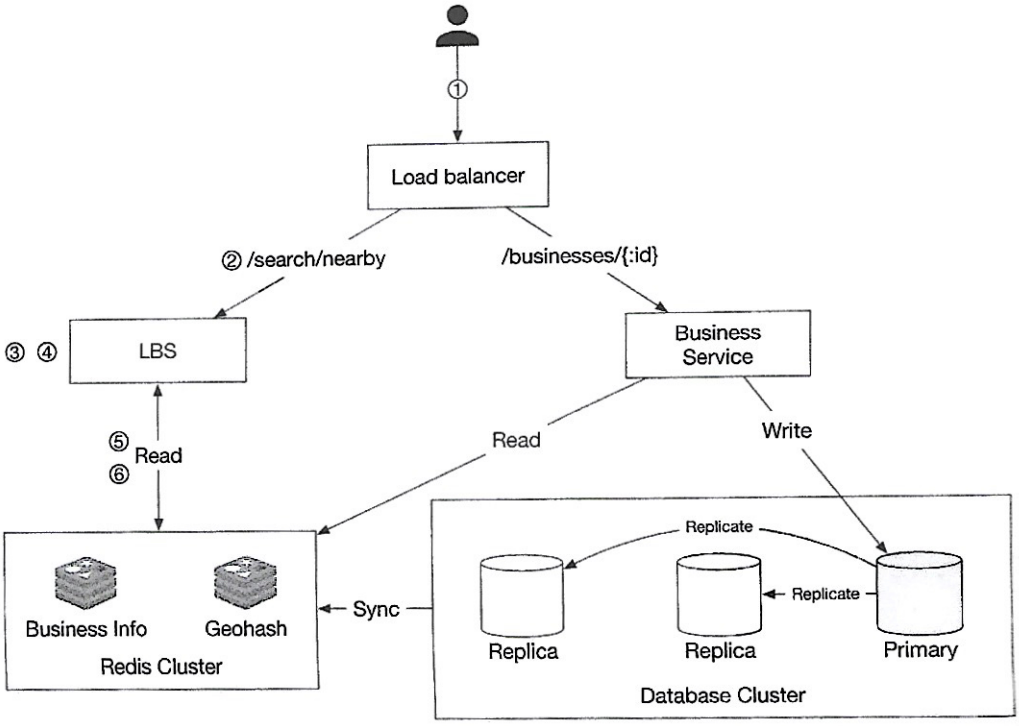


Figure 1.21: Design diagram

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.

Here 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

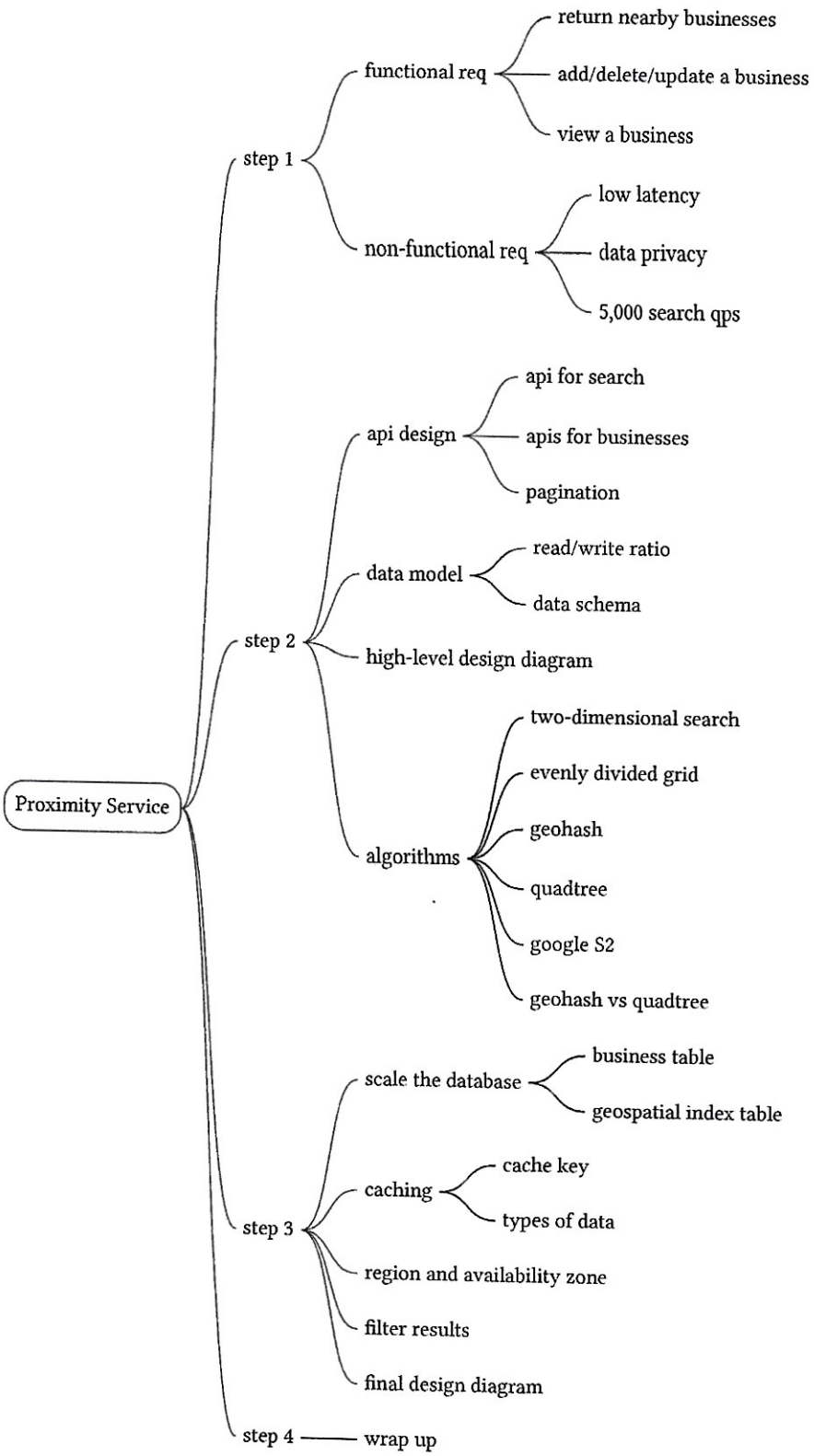
Geohash, quadtree, and S2 are widely used by different tech companies. We choose geohash as an example to show how a geospatial index works.

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.