Name	Size
Top left coordinates and bottom-right coordinates to identify the grid	32 bytes (8 bytes $\times 4$)
Pointers to 4 children	32 bytes (8 bytes $\times 4$)
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 $=\sim \frac{200 \text{ million}}{100} =\sim 2 \text{ million}$
- Number of internal nodes $= 2 \text{ million} \times \frac{1}{3} =\sim 0.67 \text{ million}$. 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 $= 2 \text{ million} \times 832 \text{ bytes} + 0.67 \text{ million} \times 64 \text{ bytes} = \sim 1.71\text{GB}$. 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.

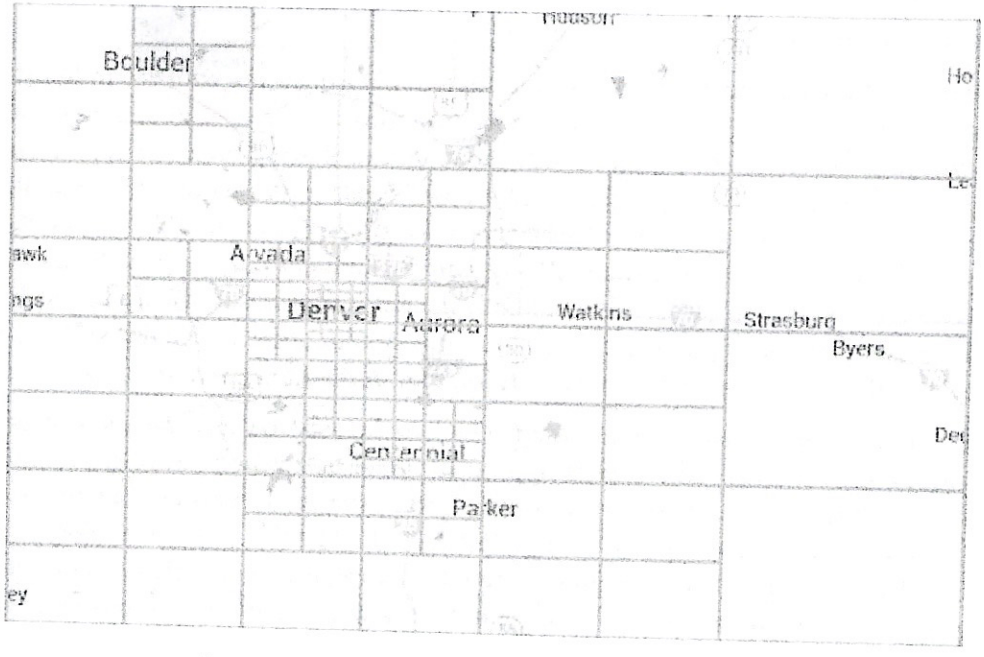

Figure 1.15: Real-world example of a quadtree

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.

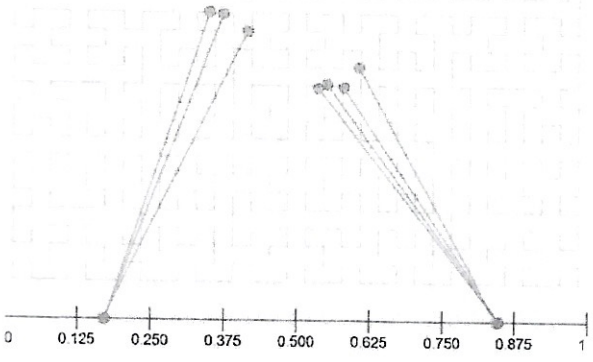

Figure 1.16: Hilbert curve (source: [24])

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.

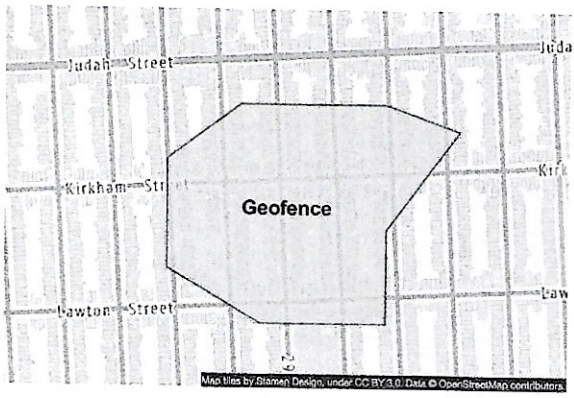

Figure 1.17: Geofence

- 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.

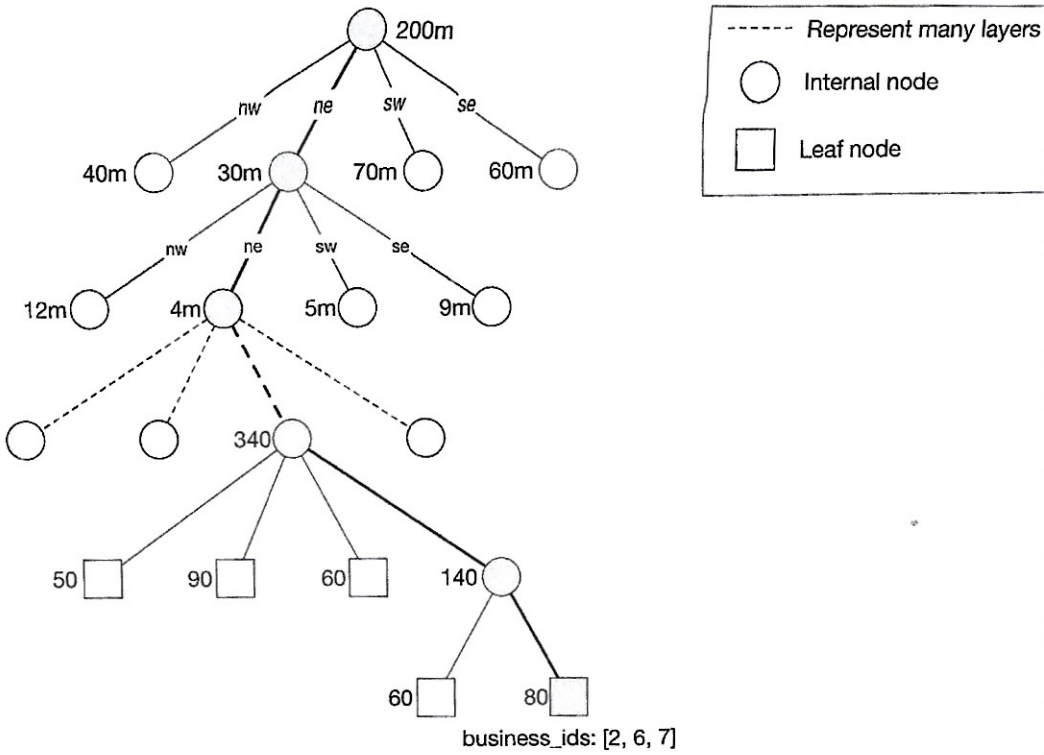

Figure 1.19: Update quadtree

Step 3 - Design Deep Dive

By now you should have a good picture of what the overall system looks like. Now let' 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 th geospatial index table.

Business table

The data for the business table may not all fit in one server, so it is a good candidat for sharding. The easiest approach is to shard everything by business ID. This shardin scheme ensures that load is evenly distributed among all the shards, and operationally i 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 solution without considering the actual data size of the table. In our case, the full dataset