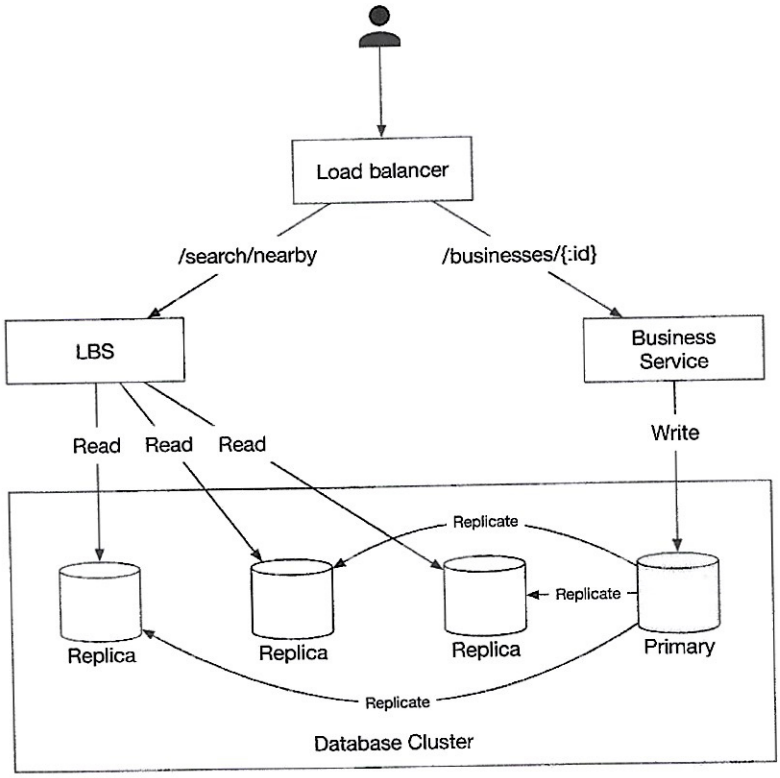

Figure 1.2: High-level design

Load balancer

The load balancer automatically distributes incoming traffic across multiple service Normally, a company provides a single DNS entry point and internally routes the AP 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 give 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 wri operations, and the QPS is not high.
- Customers view detailed information about a business. QPS is high during pe 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.

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.

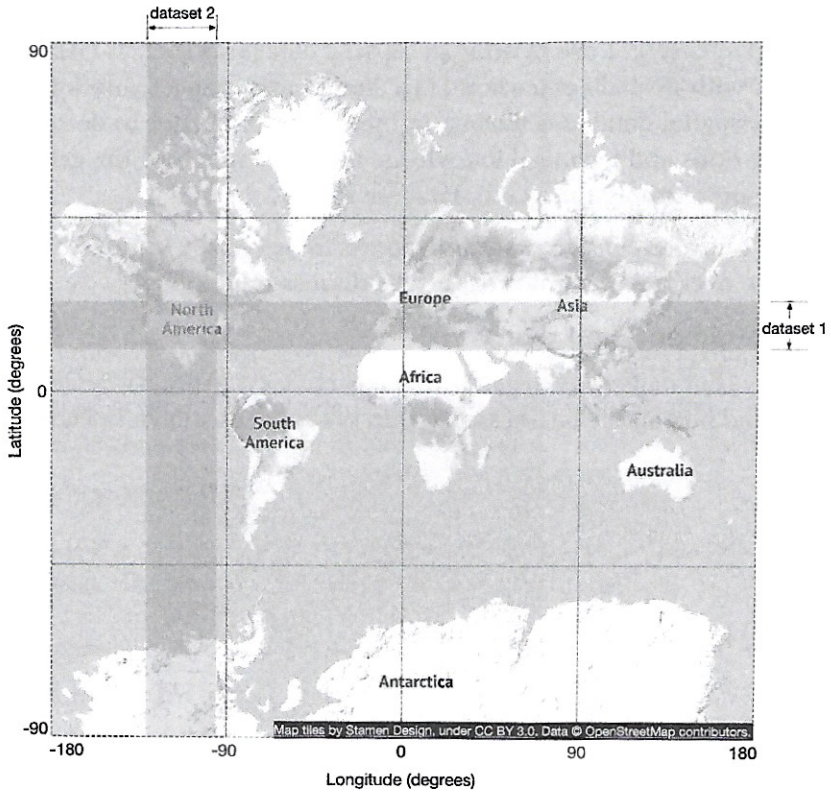

Figure 1.4: Intersect two datasets

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.

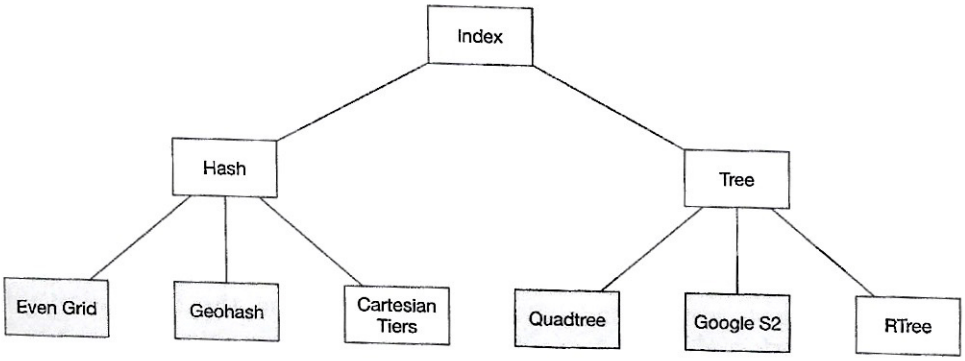

Figure 1.5: Different types of geospatial indexes

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.

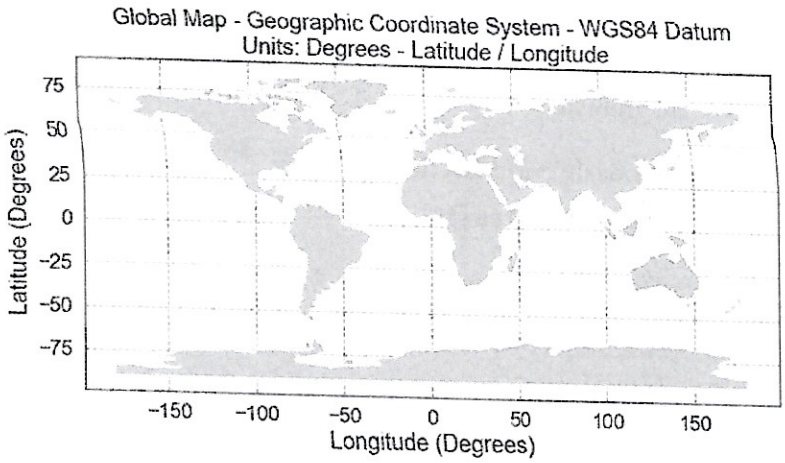

Figure 1.6: Global map (source: [14])

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.

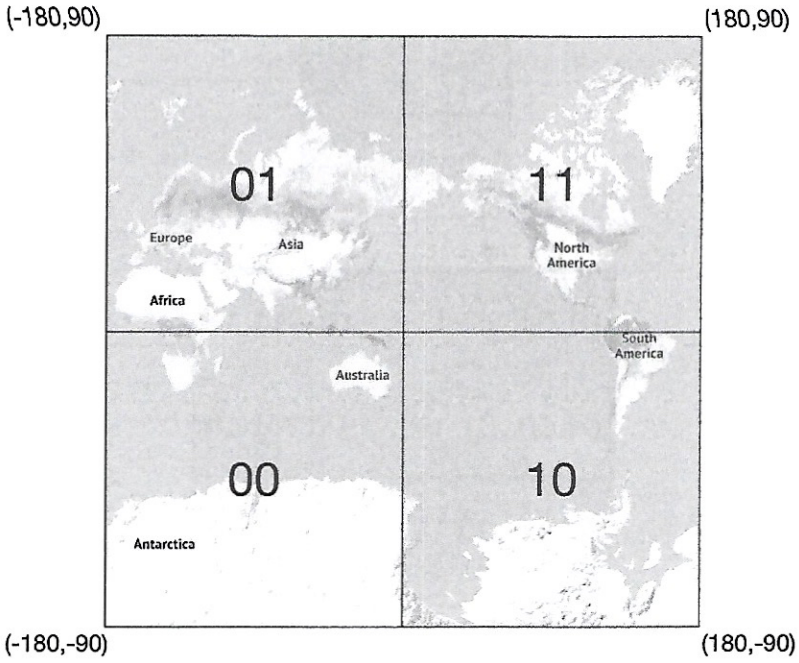

Figure 1.7: Geohash

- 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.

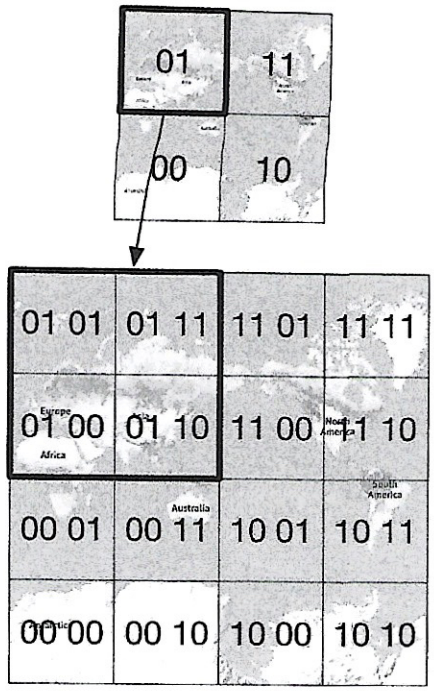

Figure 1.8: Divide grid

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.

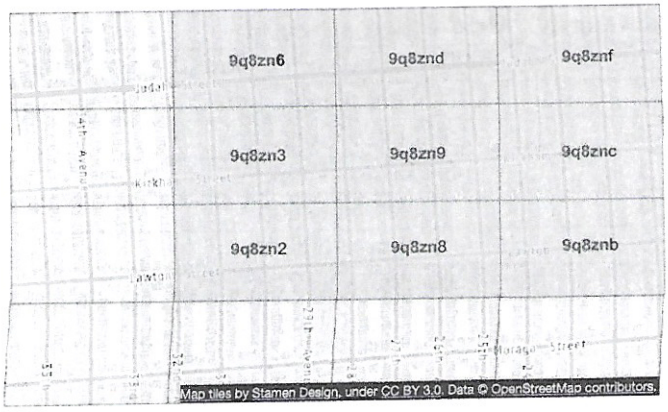

Figure 1.9: Shared prefix

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].

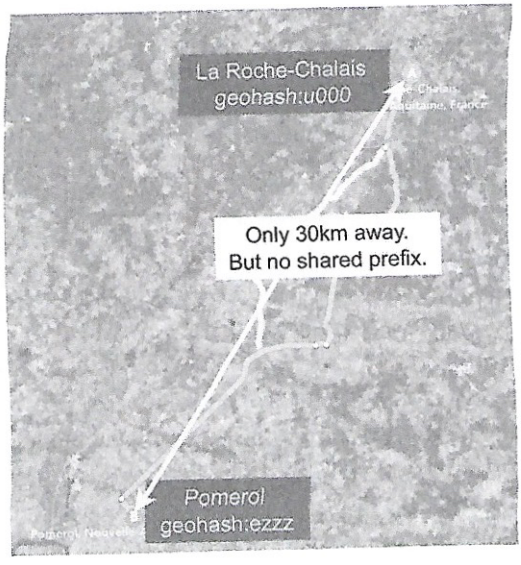

Figure 1.10: No shared prefix

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.

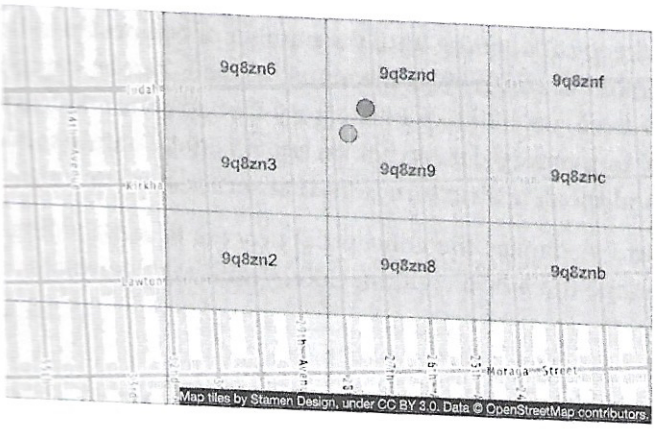

Figure 1.11: Boundary issue

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.

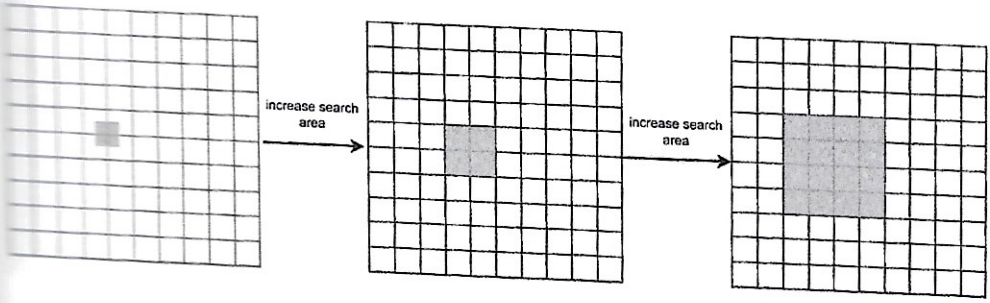

Figure 1.12: Expand the search process

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.

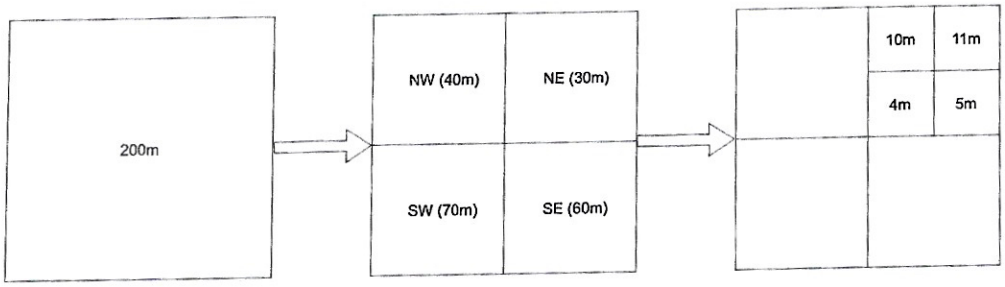

Figure 1.13: Quadtree

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.

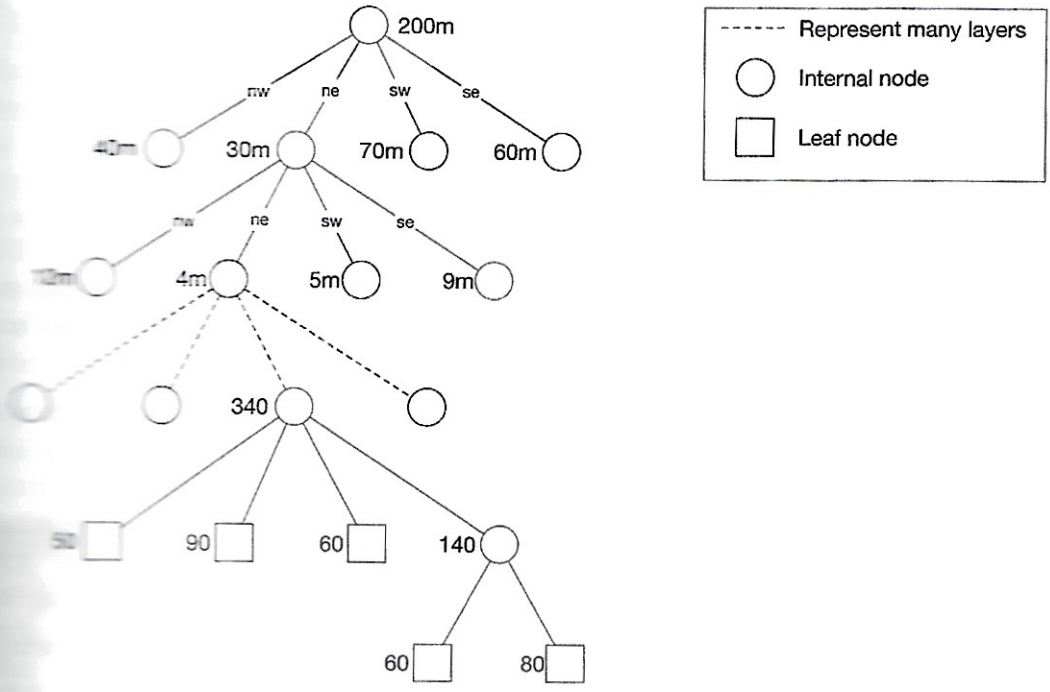

Figure 1.14: Build quadtree

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 $\times 4$)
List of business IDs in the grid	8 bytes per ID $\times$ 100 (maximal number of businesses allowed in one grid)
Total	832 bytes

Table 1.6: Leaf node

Data on internal node