Design Considerations of Yelp

Learn about the different design aspects of the Yelp system.

Introduction

We discussed the design, building blocks, components, and the entire workflow of our system in the previous lesson, which brought up a number of interesting questions that now need answering. For example, what approach do we use to find the places, or how do we rank places based on their popularity? This lesson addresses important design concerns like the ones we just mentioned. The task given below summarizes the goals of this lesson.

Summary of the Lesson

Section/Sub-section	Purpose
Searching	This process divides the world into segments to optimi... nearby sites in a given location and radius ca...
Storing Indexes	We index the places to improve query performance a... required for all of the indexes.
Searching Using Segments	We search all the desired places by combinin...
Dynamic Segments	We solve the static segment limitations using dy...
Searching Using a QuadTree	We explore the searching functionality using...
Space Estimations for a QuadTree	We estimate the storage required for a C...
Data Partitioning	We look into the options we can use to pa...
Ensuring Availability	We look into how we'll ensure the availability...
Inserting a New Place	We look into how we'll insert the place in a...
Ranking Popular Places	We rank the places of the syste...
Evaluation	We evaluate how our system fulfills the non-funct...

Searching

From Google Maps, we were able to connect segments and meet the scalability challenge to process a large graph efficiently. The graph of the world had numerous nodes and vertices, and traversing them was time-consuming. Therefore, we divided the whole world into smaller segments/subgraphs to process and query them simultaneously. The segmentation helps us improve the scalability of the system.

Each segment will be of the size $5 \times 5 \ miles$ and will contain a list of places that exist within it. We only search a few segments to locate destinations that are close by. We can use a given location and defined radius to find all the nearby segments and identify sites that are close.
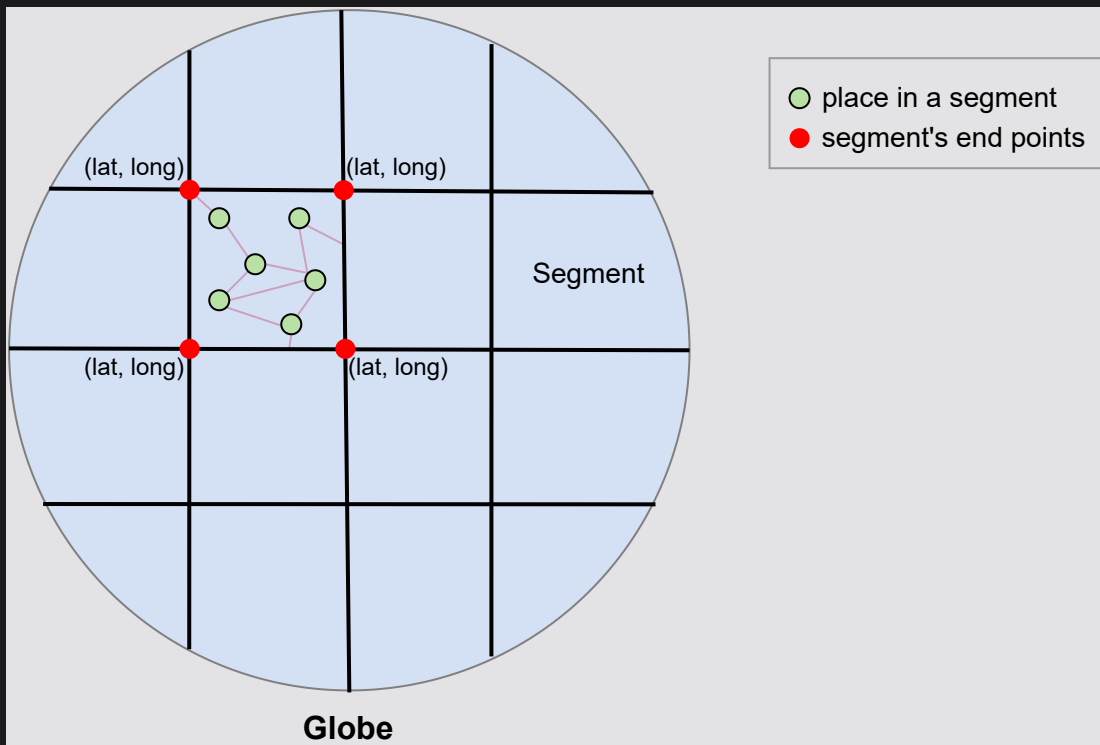
Globe

Partitioning the globe into small segments, where each segment has four coordinates (lat, long) and all segments hold the coordinates of different places

Points to Ponder

Question 1

How will we find nearby places if we create a table for storing all places?

Show Answer ⌄

We can store all the places in a table and uniquely identify a segment by having a `segment_ID`. We can index each segment in the database. Now, we have limited the number of segments we need to search, so the query will be optimized and return results quickly.

Improve data fetching

We use a key-value store for quick access to places in the segments. The key is the `segment_ID`, while the `value` contains the list of places in that segment. Let's estimate how much space we need to store the indexes.

> We can usually store a few MBs in the value of the key-value store. If we assume that each segment has 500 places then it will take up
> $500 \times 1296 \ Bytes = 0.648MB$, and we can easily store it as a value.

The total area of the earth is around 200 million square miles, and the land surface area is about 60 million square miles. If we consider our radius to be ten miles, then we'll have 6 million segments. An 8-Byte identifier will identify each segment.

Let's calculate how much memory we need.

Total Area of the Earth (Million Square Miles)	60
Search Radius (Miles)	10
Number of Segments (Millions)	6
Segment_ID (Bytes)	8
Place_ID (Bytes)	8

| Number of Places (Millions)	500
Total Space (GB) f	4.048

Search using segments

A user may select a radius for searching places that aren't present in a single segment. So, we need to combine multiple segments by underline connecting the segments to find locations within the specified radius—say, five miles.

First, we constrain the number of segments. This reduces the graph size and makes the searching process optimizable. Then, we identify all the relevant locations, compute the distance from the searching point, and show it to the user.
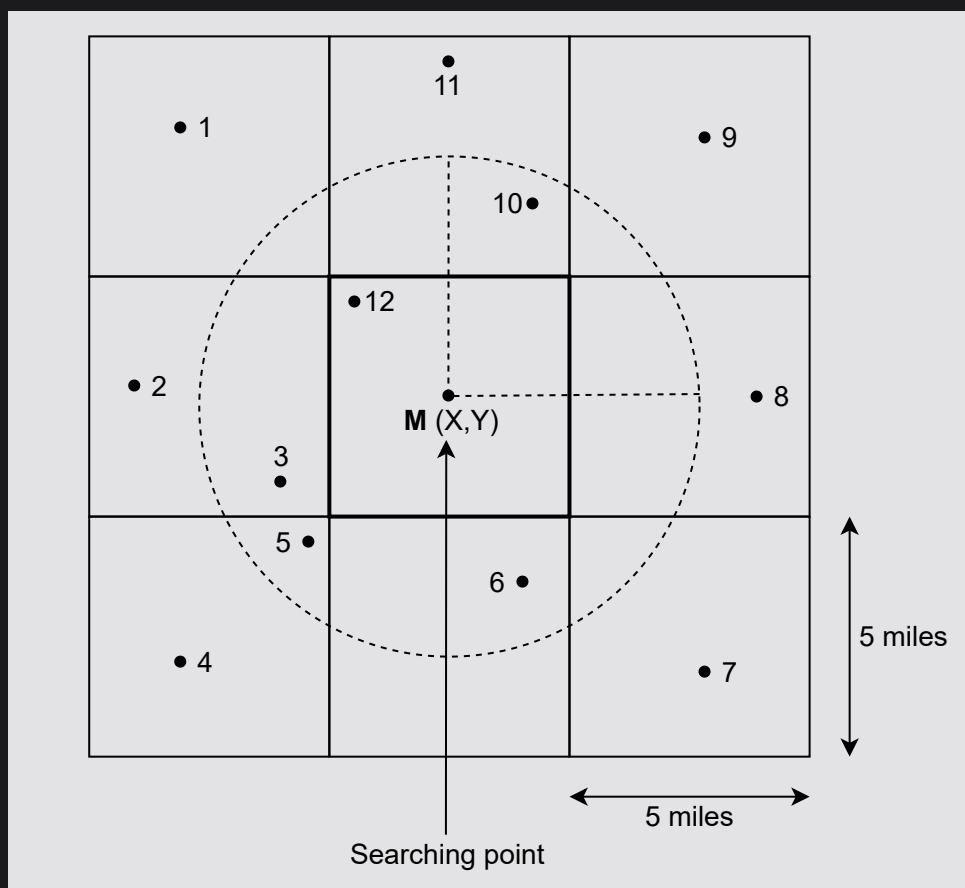
Searching for locations using segments

Points to Ponder

Dynamic segments

We solve the problem of uneven distribution of places in a segment by dynamically sizing the segments. We do this by focusing on the number of places. We split a segment into four more segments if the number of places reaches a certain limit. We assume 500 places as our limit. While using this approach, we need to decide on the following questions:

1. How will we map the segments?
2. How will we connect to other segments?

We use a **QuadTree** to manage our segments. Each node contains the information of a segment. If the number of places exceeds 500, then we split that segment into four more child nodes and divide the places between them. In this way, the leaf nodes will be those segments that can't be broken down any further. Each leaf node will have a list of places in it too.

Search using a QuadTree

We start searching from the root node and continue to visit the nodes to find our desired segment. We check every node to see if it has more child nodes. If a node has no more children, then we stop our search because that node is the required one. We also connect each child node with its neighboring nodes with a doubly-linked list. All the child nodes of all the parents nodes are connected through the doubly-linked list. This list allows us to find the neighboring segments when we can move forward and backward as per our requirement.

After identifying the segments, we have the required `PlaceID` values of the places and we can search our database to find more details on them.

Question

Is there an alternative approach to find the neighboring segments?

Show Answer ⌄

The following slides show how the process of searching for a place works. If a node has the places we need, we stop there. Otherwise, we explore more nodes until we reach our search radius. After finding the node, we query the database for information related to the places and return the desired ones.
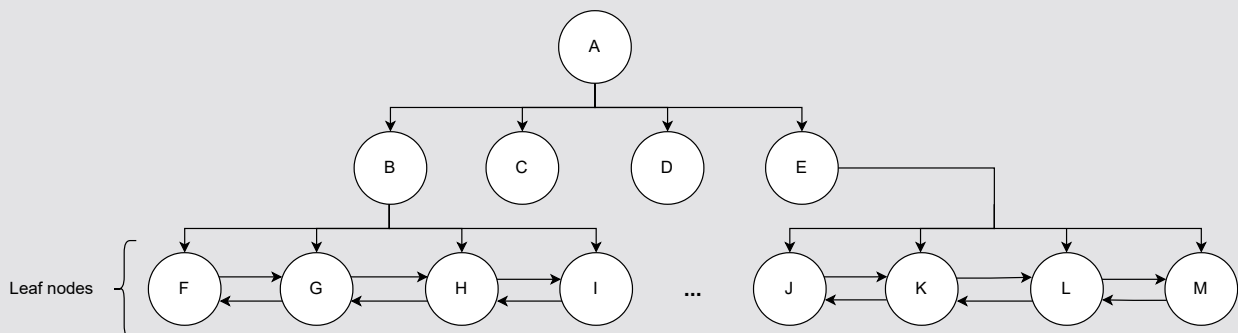
All the leaf nodes in the QuadTree are linked, just like in a doubly-linked list

Storage space estimation for QuadTrees

Let's calculate the storage we need for keeping QuadTrees:

- We store the `PlaceID`, `Latitude`, and `Longitude` for each place in the node. Each of these values is of the size 8 Bytes. For 500 million places, we need the following amount of storage for all places:
  - $(8 + 8 + 8) \times 10^6 \times 500 = 12GB$
- We'll have 500 places in a single node. So, for 500 million places, we'll need $\frac{500M}{500} = 1M$ leaf nodes.
- The QuadTree with 1 million leaf nodes has approximately 1/3rd of the leaf nodes of the internal nodes (all the nodes up the leaf level), and each internal node has four pointers to its child nodes. If we'll assume 8 Bytes for each pointer, then we'll need $1M * 1/3 * 4 * 8 = 10.67MB$ of space for the internal nodes.
- We can get the total space needed to store all the internal nodes by adding $12GB$ and $10.67MB$, that is, $12.01GB$ approximately.

We can easily store a QuadTree on a server.

Let's try the following calculator to calculate the storage needed for a QuadTree:

PlaceID, Latitude, and Longitude (Bytes)	24
Total Number of Places (Millions)	500
Total Space Needed to Store All Places (GB)	*f* 12
Limit of Places for Each Segment	500
Number of Segments (Millions)	*f* 1

Number of Pointers to Hold Children Pointers	4
Size of Each Pointer (Bytes)	8
Total Space to Store All Internal Nodes (MB)	*f* 10.67
Total Space to Store QuadTrees (GB)	*f* 12.01

Data partitioning

Keeping 20% growth per year in mind, the number of places will increase. We can partition data on the following basis:

1. **Regions**: We can split our places into regions on the basis of zip codes. This way, all the places that belong to a specific region are stored on a single node. We store information on the region along with the place, so that we can query on the basis of regions too. We can use the user's region to find the places in that specific region.

   This data partitioning comes with a few challenges. For example, if a region becomes popular during tourist season, it can affect the performance of our system. We might have numerous queries on the server that might result in slow responsiveness to user queries.

2. `PlaceID`: We can partition data on the basis of `PlaceID` instead of the region to avoid the query overload in popular seasons or rush hours. We can use a key-value store to store the places. In this case, the key is the `PlaceID` and the value contains the server in which that place is stored. This will make the process of fetching places more efficient.

So, we'll opt for partitioning on the basis of places. Moreover, we'll also use caches for popular places. The cache will have information about that particular place.

Ensure availability

Consider a scenario where multiple people in the same radius place a search request. If we have a single QuadTree, it'll affect the availability of the users. So, we can't rely on a single QuadTree. To cater to this problem, we replicate our QuadTrees on multiple servers to ensure availability. This allows us to distribute the read traffic and decrease the response time. QuadTrees are built on a server, so we can use the server ID as a key to identify the server on which the QuadTree is present. The value is the list of places that the QuadTree holds. The key-value store eases the rebuilding of the QuadTree in case we lose it.

Point to Ponder

Question

How will the leader-follower approach help us in replication?

Show Answer ⌄

Insert a new place

We insert a new place into the database as well as in our QuadTree. We find the segment of the new place if the QuadTree is distributed on different servers, and then we add it to that segment. We split the segment if required and update the QuadTree accordingly.

Rank popular places

We need a service, a **rating calculator**, which calculates the overall rating of a service. We can store the rating of a place in the database and also in the QuadTree, along with the ID, latitude, and longitude of the place. The QuadTree

returns the top 50 or 100 popular places within the given radius. The **aggregator service** determines the actual top places and returns them to the user.

> **Note:** We don't expect the rating to be updated within hours, as such frequent changes in QuadTrees or databases can be expensive. We

Finalized design

We added a few new components to our design. We introduced caches to store popular places. This allows us to fetch the places much faster. Moreover, we also added a rating calculator that sorts the places based on their ratings. This will enhance user experience, since the places with a good rating will be displayed first.

The updated design of our system is shown below:

Updated Yelp design

Evaluation

Let's see how our system design fulfills our requirements.

- **Availability**: We partitioned the data into smaller segments instead of having to deal with a huge dataset consisting of all the places on the world map. This made our system highly available. We also replicated the QuadTrees data using key-value stores to ensure availability.
- **Scalability**: We split the whole world into smaller dynamic segments. This allows us to search for a place within a specific radius and shorten our search area. We then used QuadTrees in which each child node holds a single segment. Upon adding or removing a place, we can restructure our QuadTrees. So, we were able to make our system scalable.
- **Performance**: We reduced the latency by using caches. We cached all the famous and popular places, so request time was minimized.
- **Consistency**: The users have a consistent view of the data regarding places, reviews, and photos because we used reliable and fault-tolerant databases like key-value stores and relational databases.

> (i) In the previous lesson, we chose a **SQL database** and a **key-value store** to design Yelp. How does replacing SQL with NoSQL impact the design? Discuss its impact on nonfunctional requirements like scalability and consistency. Also, discuss schema flexibility and the impact on the performance of complex queries.

<div align="center">💡 Want to know the correct answer?</div>

How does replacing SQL with NoSQL impact the design?

H₁ H₂ H₃ | **B** *I* | ☰ ☰ | ☰ ☰ ☰ | ✗

Provide your answer here!

Summary

The proximity-based servers allow the user to search for a specific place or places nearby. The map data of the world is huge and dividing it into segments and finding the specific segment was a challenge in itself. So, we used QuadTrees to optimize our search and provided the user with a list of places with minimum latency.

← **Back**   ✅ Completed   **Next** →