Designing Yelp or Nearby Friends

Let's design a Yelp like service, where users can search for nearby places like restaurants, theaters or shopping malls, etc., and can also add/view reviews of places.

Similar Services: Proximity server.
Difficulty Level: Hard

1. Why Yelp or Proximity Server?

Proximity servers are used to discover nearby attractions like places, events, etc. If you haven't used yelp.com before, please try it before proceeding. You can search for nearby restaurants, theaters, etc., and spend some time understanding different options the website offers. This will help you a lot in understanding this chapter better.

that users can perform a search on them. Upon querying, our service will return a list of places around the user.

Our Yelp-like service should meet the following requirements:

Functional Requirements:

1. Users should be able to add/delete/update Places.
2. Given their location (longitude/latitude), users should be able to find all nearby places within a given radius.
3. Users should be able to add feedback/review about a place. The feedback can have pictures, text, and a rating.

Non-functional Requirements:

1. Users should have real-time search experience with minimum latency.
2. Our service should support a heavy search load. There will be a lot of search requests compared to adding a new place.

3. Scale Estimation

Let's build our system assuming that we have 500M places and 100K queries per second (QPS). Let's also assume a 20% growth in the number of places and QPS each year.

Each location can have following fields:

1. LocationID (8 bytes): Uniquely identifies a location.
2. Name (256 bytes)
3. Latitude (8 bytes)
4. Longitude (8 bytes)
5. Description (512 bytes)
6. Category (1 byte): E.g., coffee shop, restaurant, theater, etc.

Although a four bytes number can uniquely identify 500M locations, with future growth in mind, we will go with 8 bytes for LocationID.

Total size: 8 + 256 + 8 + 8 + 512 + 1 = 793 bytes

We also need to store reviews, photos, and ratings of a Place. We can have a separate table to store reviews for Places:

1. LocationID (8 bytes)
2. ReviewID (4 bytes): Uniquely identifies a review, assuming any location will not have more than 2^32 reviews.

Similarly, we can have a separate table to store photos for Places and Reviews.

5. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. Following could be the definition of the API for searching:

```
search(api_dev_key, search_term, user_location, radius_filter, maximum_results_to_return,
       category_filter, sort, page_token)
```

Parameters:
api_dev_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.
search_term (string): A string containing the search terms.
user_location (string): Location of the user performing the search.
radius_filter (number): Optional search radius in meters.
maximum_results_to_return (number): Number of business results to return.
category_filter (string): Optional category to filter search results with, e.g., Restaurants, Shopping Centers. etc.

Returns: (JSON)
A JSON containing information about a list of businesses matching the search query. Each result entry will have the business name, address, category, rating, and thumbnail.

6. Basic System Design and Algorithm

At a high level, we need to store and index each dataset described above (places, reviews, etc.). For users to query this massive database, the indexing should be read efficient, since while searching for nearby places users expect to see the results in real-time.

Given that the location of a place doesn't change that often, we don't need to worry about frequent updates of the data. As a contrast, if we intend to build a service where objects do change their location frequently, e.g., people or taxis, then we might come up with a very different design.

Let's see what are different ways to store this data and find out which method will suit best for our use cases:

a. SQL solution

columns, and to perform a fast search, we should have indexes on both these tests.

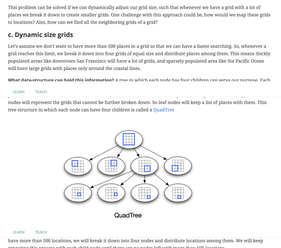
To find all the nearby places of a given location (X, Y) within a radius 'D', we can query like this:

```
Select * from Places where Latitude between X-D and X+D and Longitude between Y-D and Y+D
```

How efficient this query would be? We have estimated 500M places to be stored in our service. Since we have two separate indexes, each index can return a huge list of places, and performing an intersection on these two lists won't be efficient. Another way to look at this problem is that there could be too many locations between 'X-D' and 'X+D', and similarly between 'Y-D' and 'Y+D'. If we can somehow shorten these lists, it can improve the performance of our query.

b. Grids

We can divide the whole map into smaller grids to group locations into smaller sets. Each grid will store all the Places residing within a certain range of longitude and latitude. This scheme would enable us to query only a few grids to find nearby places. Based on given location and radius, we can find all the nearby grids and then only query these grids to find nearby places.



Given a location and a radius, we can first find the grid that contains the given location, and then we can query neighboring grids to find nearby places. Since places can be very close to the grid boundary, there may be some places in the neighboring grids which are closer than places in the given grid. So to make our search results accurate, we will have to find all such places which are within the given radius. To do so, we will have to find the neighboring grids of the given grid.

to store our grids.

To reduce the number of grids, we can divide the world into equal size grids, such that the total number of places in any grid does not exceed a certain limit. In this case, we should also be able to quickly know the location of any place, so that we can find the grid that contains the given location. To uniquely identify any location, we can assign a unique number or name to each grid.

In the database, we can store the GridID with each location and have an index on it too for faster searching. Now, our query will look like:

```
Select * from Places where Latitude between X-D and X+D and Longitude between Y-D and Y+D and GridID in (GridID, GridID1, GridID2, ..., GridID8)
```

This will undoubtedly improve the runtime of our query.

Should we keep our index in memory? Maintaining the index in memory will improve the performance of our service. We can keep our index in a hash table, where 'key' would be the grid number and 'value' would be the list of places contained in that grid.

How much memory will we need to store the index? Let's assume our search radius is 10 miles, given that total area of the

to store our grids.

$$(4 * 2\pi R) / (2 * 100SQ \text{ miles}) = 4 \text{ GB}$$

This solution can still run slow for those grids that have a lot of places since our places are not uniformly distributed among grids. We can have a thickly dense area with a lot of places, and on the other side, we can have areas which are sparsely populated.

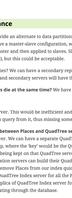
This problem can be solved if we can dynamically adjust our grid size, such that whenever we have a grid with a lot of places we break it down to create smaller grids. One challenge with this approach could be, how would we map these grids to locations? Also, how can we find all the neighboring grids of a grid?

c. Dynamic size grids

Let's assume we don't want to have more than 500 places in a grid so that we can have a faster searching. So, whenever a grid reaches this limit, we break it down into four grids of equal size and distribute places among them. This means thickly populated areas like downtown San Francisco will have a lot of grids, and sparsely populated area like the Pacific Ocean will have large grids with places only around the coastal lines.

What data structure can hold this information? A tree in which each node has four children can serve our purpose. Each

nodes will represent the grids that cannot be further broken down. So leaf nodes will keep a list of places with them. This tree structure in which each node can have four children is called a QuadTree



have more than 500 locations, we will break it down into four nodes and distribute locations among them. We will keep repeating this process with each child node until there are no nodes left with more than 500 locations.

How will we find the grid for a given location? We will start with the root node and search downward to find our required nodegrid. At each step, we will see if the current node we are visiting has children. If it has we will move to the child node that contains our desired location and repeat this process. If the node does not have any children, then that is our desired node.

How will we find neighboring grids of a given grid? Since only leaf nodes contain a list of locations, we can connect all leaf nodes with a doubly linked list. This way we can move forward or backward among the neighboring leaf nodes to find our desired locations. Another approach for finding adjacent grids would be through parent nodes. We can keep a pointer in each node to access its parent, and since each parent node has pointers to all of its children, we can easily find siblings of a node. We can keep expanding our search for neighboring grids by going up through the parent pointers.

Once we have nearby LocationIDs, we can query backend database to find details about these places.

What will be the search workflow? We will first find the node that contains the user's location. If that node has enough desired places, we can return them to the user. If not, we will keep expanding to the neighboring nodes (either through the parent pointers or doubly linked list, until either we find the required number of places or reached the maximum allowed search area.

How much memory will be needed to store the QuadTree? For each Place, if we cache only LocationID and LatLong, we would need 12GB to store all places.

$$24 * 500M = 12 \text{ GB}$$

Since each grid can have maximum 500 places and we have 500M locations, how many total grids we will have?

$$500M / 500 = 1M \text{ grids}$$

Which means we will have 1M leaf nodes and they will be holding 12GB of location data. A QuadTree with 1M leaf nodes will approximately 1/3rd internal nodes, and each internal node will have 4 pointers (for its children). If each pointer is 8 bytes, then the memory we need to store all internal nodes would be:

$$1M * 1/3 * 4 * 8 = 10 \text{ MB}$$

So, total memory required to hold the whole QuadTree would be 12.01GB. This can easily fit into a modern-day server.

How would we insert a new Place into our system? Whenever a new Place is added by a user, we need to insert it into the database, as well as, in the QuadTree. If our tree resides on one server, it is easy to add a new Place, but if the QuadTree is distributed among different servers, first we need to find the whereabouts of the new Place and then add it there (discussed

7. Data Partitioning

What if we have a huge number of places such that, our index does not fit into a single machine's memory? With 20% growth, each year, we will reach the memory limit of the server in the future. Also, what if one server cannot serve the desired read traffic? To resolve these issues, we must partition our QuadTree.

We will explore two solutions here (both of these partitioning schemes can be applied to databases too):

a. Sharding based on regions: We can divide our places into regions (like zip codes), such that all places belonging to a region will be stored on a fixed node. While storing, we will find the region of each place to find the server and store the place there. Similarly, while querying for nearby places, we can ask the region server that contains user's location. This approach has a couple of issues:

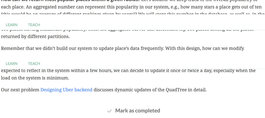
1. What if a region becomes hot? There would be a lot of queries on the server holding that region, making it perform slow. This will affect the performance of our service.
2. Over time some regions can end up storing a lot of places compared to others. Hence maintaining a uniform distribution of places, while regions are growing, is quite difficult.

To recover from these situations either we have to repartition our data or use consistent hashing

While building our QuadTree, we will iterate through all the places and calculate the hash of each LocationID to find a server where it would be stored. To find nearby places of a location we have to query all servers, and each server will return a set of nearby places. A centralized server will aggregate these results to return them to the user.

Will we have different QuadTree structure on different partitions? Yes, this can happen, since it is not guaranteed that we will have an equal number of places in any given grid on all partitions. Though, we do make sure that all servers have approximately equal number of Places. This different tree structure on different servers will not cause any issue though, as we will be searching all the neighboring grids within the given radius on all partitions.

Remaining part of this chapter assumes that we have partitioned our data based on LocationID.



8. Replication and Fault Tolerance

Having replicas of QuadTree servers can provide an alternate to data partitioning. To distribute read traffic, we can have replicas of each QuadTree server. We can have a master-slave configuration, where replicas (slaves) will only serve read traffic and all write traffic will first go the master and then applied to slaves. Slaves might not have some recently inserted places (a few milliseconds delay will be there), but this could be acceptable.

What will happen when a QuadTree server dies? We can have a secondary replica of each server, and if primary dies, it can take control after the failover. Both primary and secondary servers will have the same QuadTree structure.

What if both primary and secondary servers die at the same time? We have to allocate a new server and rebuild the

required places that will be stored on this server. This would be inefficient and slow, also during the time when the server is being rebuilt; we will not be able to serve any query from it, thus missing some places that should have been seen by users.

How can we efficiently retrieve a mapping between Places and QuadTree server? We have to build a reverse index that will map all the Places to their QuadTree server. We can have a separate QuadTree Index server that will hold this information. We will need to build a HashMap, where the 'key' would be the QuadTree server number and the 'value' would be a HashSet containing all the Places being kept on that QuadTree server. We need to store LocationID and LatLong with each place because through this information servers can build their QuadTrees. Notice that we are keeping Places' data in a HashSet, this will enable us to add/remove Places from our index quickly. So now whenever a QuadTree server needs to rebuild itself, it can simply ask the QuadTree Index server for all the Places it needs to store. This approach will surely be quite fast. We should also have a replica of QuadTree Index server for fault tolerance. If a QuadTree Index server dies, it can always rebuild its index from iterating through the database.

9. Cache

To deal with hot Places, we can introduce a cache in front of our database. We can use an off-the-shelf solution like Memcache, which can store all data about hot places. Application servers before hitting backend database can quickly check if the cache has that Place. Based on clients' usage pattern, we can adjust how many cache servers we need. For cache

10. Load Balancing (LB)

We can add LB layer at two places in our system 1) Between Clients and Application servers and 2) Between Application servers and Backend server. Initially, a simple Round Robin approach can be adopted; that will distribute all incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is if a server is dead, the load balancer will take it out of the rotation and will stop sending any traffic to it.

A problem with Round Robin LB is, it won't take server load into consideration. If a server is overloaded or slow, the load balancer will not stop sending new requests to that server. To handle this, a more intelligent LB solution would be needed that periodically queries backend server about their load and adjusts traffic based on that.

11. Ranking

How about if we want to rank the search results not just by proximity but also by popularity or relevance?

How can return most popular places within a given radius? Let's assume we keep track of the overall popularity of each place. An aggregated number can represent this popularity in our system, e.g., how many stars a place gets out of ten.

returned by different partitions.

Remember that we didn't build our system to update place's data frequently. With this design, how can we modify

expected to reflect in the system within a few hours, we can decide to update it once or twice a day, especially when the load on the system is minimum.

Our next problem Designing Uber backend discusses dynamic updates of the QuadTree in detail.

Mark as completed

Prev
Designing Facebook's Newsfeed

Next
Designing Uber backend

Send feedback or ask a question · 20 recommendations

Home Featured Team Blog FAQ Terms of Service Contact Us