Feature Requirements

Primary features:

1. Addition, deletion or updating the place can be done by the user.
2. Based the location given, users should be able to find all the nearby places within the given radius.
3. User should be given option to add feedback and also add a review about the place.
4. Reviews or feedback might not just be limited to text, consider pictures and should have ratings as well.

Secondary features:

1. The design must be able to refresh instantaneously, limit the delay to be as minimum as possible.
2. Your design should be able to accommodate huge search load. Consider huge requests for search compared to adding new place.

Capacity and Estimation

Scale Estimation

Total Places= 600M

Total Queries per second = 120000

Total Growth pe year = 25%

Capacity

For each Place, if we cache only LocationID and Lat/Long, we would need 12GB to store all places.

24 * 600M => 12.4 GB

Since each grid can have a maximum of 500 places, and we have 600M locations, how many total grids we will have?

600M / 500 => 800K grids

Which means we will have 800K leaf nodes and they will be holding 12.4 GB of location data. A QuadTree with 800K leaf nodes will have 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:

800K * 1/3 * 4 * 8 = 10 MB

So, total memory required to hold the whole QuadTree would be 12.01GB.

Database Schema

Business ‘

id’: a unique identifier for this business (VARCHAR),
‘name’: the full business name (VARCHAR),
‘full address’: localized address (VARCHAR),
‘city’: city (VARCHAR),
‘state’: state (VARCHAR),
‘latitude’: latitude (REAL),
‘longitude’: longitude (REAL),
‘stars’: star rating, rounded to half-stars (REAL),
‘review count’: review count (INT), ‘
open’: is the business still open for business? (INT),
‘photo_url’: photo url (VARCHAR)

Users

‘id’: unique user identifier (VARCHAR),
‘name’: first name, last initial, like ‘Matt J.’ (VARCHAR),
‘review_count’: review count (INT),
‘useful_votes’: count of useful votes across all reviews (INT),
‘funny_votes’: count of funny votes across all reviews (INT),
‘cool_votes’: count of cool votes across all reviews (INT)

Review

‘business_id’: the identifier of the reviewed business (VARCHAR),
‘user_id’: the identifier of the authoring user (VARCHAR),
‘stars’: star rating, integer 1-5 (INT),
‘text’: review text (TEXT),
‘useful_votes’: count of useful votes (INT),
‘funny_votes’: count of funny votes (INT),
‘cool_votes’: count of cool votes (INT)

Assuming we are predicting 25 % growth every year, we propose to utilize NoSQL datastore suchs as Cassandra . **Cassandra** database is the right choice when you need scalability and high availability without compromising performance

System APIs

search(search_terms, user_location, radius_filter, maximum_results_to_return, category_filter, sort, page_token)

Basic System Design

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 the nearby places users expect to see the results in real-time. The business location does not get updated frequently

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 specific range of longitude and latitude. This scheme would enable us to query only a few grids to find nearby places. Based on a given location and radius, we can find all the neighboring grids and then query these grids to find nearby places.

However our grids will be of fixed size making query the places slower reducing system performance . Instead we can utilized dynamic grids to improve the performance of system

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.

Quad Tree

Quadtrees are trees used to efficiently store data of points on a two-dimensional space. In this tree, each node has at most four children.
We can construct a quadtree from a two-dimensional area using the following steps:

1. Divide the current two dimensional space into four boxes.
2. If a box contains one or more points in it, create a child object, storing in it the two dimensional space of the box
3. If a box does not contain any points, do not create a child for it
4. Recurse for each of the children.

Quadtrees are used in image compression, where each node contains the average colour of each of its children. The deeper you traverse in the tree, the more the detail of the image. Quadtrees are also used in searching for nodes in a two-dimensional area. For instance, if you wanted to find the closest point to given coordinates, you can do it using quadtrees.

Insert Function

The insert functions is used to insert a node into an existing Quad Tree. This function first checks whether the given node is within the boundaries of the current quad. If it is not, then we immediately cease the insertion. If it is within the boundaries, we select the appropriate child

to contain this node based on its location.
This function is O(Log N) where N is the size of distance.

Search Function

The search function is used to locate a node in the given quad. It can also be modified to return the closest node to the given point. This function is implemented by taking the given point, comparing with the boundaries of the child quads and recursing.
This function is O(Log N) where N is size of distance.

Data Partitioning

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. To store a place we will find the server through its region and, similarly, while querying for nearby places we will ask the region server that contains user's location. However it doesn't take consideration into the overall rating, popularity of places. A region/location can have more queries than other location/region making huge performance issues on our servers/service.

To solve above issue, we can use Consistent Hashing to evenly distribute the data

b.) Sharding based on Location: Our hash function will map each Location to a server where we will store that place. While building our Quadtree, we will iterate through all the places and calculate the hash of each Location to find a server where it would be stored. To find places near 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.

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; all write traffic will first go to 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.

Cache

Since we are generating huge amount of data every day ,Distributed/In-Memory caching server can be of immense benefit for our system. These servers will be storing information for given user location and its corresponding quad tree. For continuous updating user location in Quad Tree, we need to put user and corresponding information .

For this ,we propose to utilize Redis for our system. It is used as in memory distributed cache. Redis has good support for all kind of data structures. For storing and retrieving ,we can use Location_Id key and quad object as value

Mapping between location and Quad Tree

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' is the Quadtree server number and the 'value' is a HashSet containing all the Places being kept on that Quadtree server. We need to store Location and Lat/Long with each place because information servers can build their Quadtrees through this. So now, whenever a Quadtree server goes down or needs to rebuild itself, it can simply ask the QuadTree Index server for all the Places it needs to store.

L**oad Balancing (LB)**

We can add LB layer at two places in our system

1) Clients and Application servers and

2) Application servers and Backend server.

3) Application and Cache Server

We can use simple round robin mechanism for load balancing ,it does not take the server load into consideration. So, we can use Agent-Based Adaptive Load Balancing approach for load balancing Each server in the pool has an agent that reports on its current load to the load balancer. This real-time information is used when deciding which server is best placed to handle a request. This is used in conjunction with other techniques such as Weighted Round Robin and Weighted Least Connection.