Design Yelp (Nearby Friends Service)

What is Yelp?

Yelp is a Proximity server useful for discovering nearby places such as restaurants, theatres, temples, or recreational areas. It allows users to view and add comments, photos and reviews for these events and places. In this blog, we will discuss proximity server system design.

Key Requirements

Our proximity server will store information about various locations so that users can search for them based on their current location. When a user makes a query, the proximity server will deliver a list of places that are nearby. For example, if a user searches for "restaurants" near their current location, the proximity server will retrieve a list of nearby restaurants.

Our system should meet the following requirements:

Functional Requirements

- A proximity server's fundamental operation is searching. Users should be able to search all nearby locations within a specified radius for any particular location (latitude and longitude).
- Users can create new places or change existing ones by adding or editing basic features such as photos, a brief description, and so on.
- Users can give places ratings (from one to five stars) and reviews (written and photographs).

Non-Functional Requirements

1. The system should be extremely reliable, with real-time search capabilities and low latency.
2. We may anticipate this application being read-heavy due to a large

number of search requests compared to the frequent inclusion of new locations.

Capacity Estimation

With 100K queries per second, the total number of spots in the system is estimated to reach 200 million. We should develop our system at a size of at least 5 years, assuming a future scale of 5 years with 20% annual growth.

- Our system should be capable of handling a population of 400 million people.
- A load of 200K requests per second should be no problem for our system.

High Level Design

We have three high-level working API's for this service:

Search API

Search API is responsible for searching any user query and returns information regarding the searched query.

```
searchRequest: query_param = {
   search_terms: "paratha",
   radius: 10,
   category_filter: "indian",
   filter: "5 start rated restaurants",
   maximun_no_of_results: 50
}
Response: A JSON containing information about a list of popular places matc
```

Add places

Add places API is responsible for adding places and returns the response of newly created place with its place_id.

```
placesRequest: body = {
   place_name: "Baba ka Dhaba",
   latitude: 115,
   longitude: 100,
   category: "Restaurant",
   description: "Best local shop",
   photos: ["url1", "url2"]
}
Response: Response of the newly created place with place_id
```

Add reviews

Add reviews API is responsible for adding reviews about the places.

```
reviewsRequest: body = {
   user_id: "user12",
   place_id: "place12",
   rating: 5,
   review: "Food was awesome",
   photos: ["url1", "url2"]
}
Response: Response of the newly created review with it's id.
```

Database Schema

The basic schema of the database for the system is described below.

Each dataset given above must be stored and indexed (places, reviews, etc.). Users want to see results in real-time while searching for local places; therefore, the indexing must be read-efficient to query this enormous database. We don't need to worry about frequent data updates because the position of a place doesn't change too often. In contrast, if we want to construct a service where items, such as people or cabs, change their location regularly, we might develop a radically different design.

Let's look at the various options for storing this information and determine which technique is appropriate for our needs:

Simple SQL based storage

Places, reviews, and user details can be stored in an SQL database and indexed to optimize search using latitude and longitude. To further improve search efficiency, we can use the concept of 2D grids to divide the world map into smaller squares. For example, if we assume the earth's surface area is 100 million square kilometers and the search radius is fixed at 10 kilometers, we can create 10 million squares with a grid size of 10 kilometers. By fixing the grid size to the query radius, we can limit the search to the target grid and its eight neighbouring grids.

Every place with a location will belong to a specific grid. Each grid will have a unique id that can be indexed and stored in the places table. Let's see how our basic search flow works :)

We can find the grid id for every location and its eight nearby grids because our grids are statically created with a search radius equal to the grid size. As a result, the query's overall runtime will be lowered and improved, as the search query execution scope has been narrowed to just nine grids instead of the brute force strategy, which requires us to search for the entire map.

We can make it even faster by storing the grid's number and a list of its locations in memory. As previously stated, we will have 10 million grids,

with each grid id being 5 bytes and the place id being 10 bytes, similar to the gigantic scale of 100 million locations. As a result, the total amount of memory required to cache grid and place ids is 2 GB. (10M * 5) + (100M * 10) ~ 2GB.

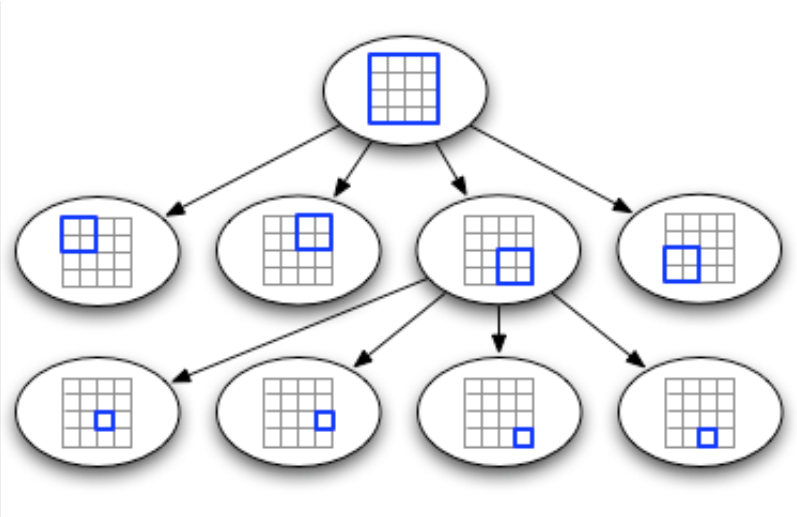
Problems with this approach

- For popular locations, this approach may be slow to implement due to an imbalanced distribution of locations within the grids. This can lead to some grids being heavily populated and others being sparsely populated, such as in coastal regions or on islands.
- An alternative approach is to dynamically adjust grid sizes by maintaining a maximum number of locations in each grid. However, this strategy can be challenging to implement and adds complexity to the system.

However, we can solve this issue using QuadTrees.

QuadTrees

A quadtree is a tree data structure that has zero or four child nodes for each parent node. It is designed to efficiently divide a flat two-dimensional space and store location data in its nodes. This makes it useful for spatial indexing and collision detection in applications such as games.



In our example, we can use each node in the quadtree to represent a grid

and store information about the locations within that grid. When a node reaches a maximum capacity, or "bucket size," of X locations, it is divided into four child nodes and the data is distributed among them recursively.

Initially, all of the locations are stored in a single root node. However, as our system is designed to handle a scale of 100 million locations over five years, the root node will not be able to hold them all. The root node will therefore be recursively split into child nodes until no nodes contain more than 100 locations. This results in a quadtree with leaf nodes that store all of the locations.

Let’s see how our basic search flow works.

To search for a location in our quadtree, we start at the root node and work our way down the tree until we reach the appropriate leaf node. This is because locations are only stored in leaf nodes.

Our quadtree creation approach ensures that locations in neighbouring nodes are geographically close to each other. As a result, when searching for nearby locations, we also consider the data in neighbouring nodes.

To improve search performance, we can cache the quadtree information. With a total of 1 million nodes (100 million locations divided by a bucket size of 100), we can estimate that the node id will be 5 bytes long and each node will contain four child pointers. In addition, we need 10 bytes each for the location id, latitude, and longitude. This means that the total storage requirement is approximately 4GB (3* 10 * 100 M + 4*5*1 M ~= 4 GB).

Database sharding

Given the scale of our system, we cannot rely on a single server to handle all of the traffic. This could create a single point of failure, which is unacceptable in today's distributed systems. One solution to this issue is to partition the quadtree data using a method such as region-based sharding or sharding based on place id.

1. **Region-based sharding:** This approach involves partitioning the data based on regions, but this can lead to a non-uniform distribution of data because some regions are more densely populated than others. This can make it difficult to achieve a uniform distribution of data.
2. **Sharding based on Place ID:** An alternative approach is to shard the data based on the place id. We can use a hash function to calculate the hash of each place id as we build the quadtree and map each place id to a specific server where the location information is stored. This can help to evenly distribute the data across servers.

The second method of sharding based on place id appears to be simpler and more effective. Even if we end up with multiple quadtrees, this is not a major issue because the data will be uniformly distributed across all servers. This ensures that the system can handle the expected traffic and maintain high availability.

Data Replication

In a distributed system of this scale, it is essential to have the ability to operate correctly even in the event of failures. To ensure high availability, we cannot rely on a single machine as this creates a single point of failure.

One solution is to use a master-slave architecture, where only masters can write data and slaves can only read. If a master server goes down, any slave servers can take over as the master and handle writes. There may be a small delay of a few milliseconds in updating newly changed data, resulting in eventual consistency, but this should not be a significant issue for this application.

To further improve fault tolerance, we can also have a replica of the quadtree index server. If a quadtree server fails, it can be quickly rebuilt by querying the index server instead of the database, which can help to reduce the load on the database.

Load Balancing

To improve the efficiency of our system, we can use load balancers in two places:

- Between clients and application servers
- Between application servers and backend servers

A simple round robin technique can evenly distribute incoming requests among the backend servers. This type of load balancer is easy to set up and does not add significant overhead. It also has the advantage of automatically removing a server from the rotation if it goes down, which helps to prevent traffic from being sent to an unavailable server. However, the round robin method does not take server load into account, so a more intelligent load balancer that regularly polls the backend servers and adjusts traffic based on their load may be necessary in some cases.

Conclusion

In this blog, we discussed the design and scalability of a system similar to Yelp. Designing and scaling such a system is a challenging task that requires careful planning and consideration of various factors. If you have any thoughts or feedback on the topic, please feel free to share them.

Thanks to Suyash Namdeo for his contribution in creating the first version of this content. If you have any queries/doubts/feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy system design, Enjoy algorithms!