System Design — Nearby Places Recommender System

source: quanta magazine

Design a system to show recommendations for nearby places for an app like Yelp.

Similar Questions

- Design a system to find nearby restaurants
- Design a system to match drivers with riders for Uber
- Design a system to compute ETA for food delivery

Functional Requirements

- Show top recommendations for nearby places to visit.
- Users must be able to add new places.
- Users must be able to see and give star ratings to places.

**Since there are many other features such as check-in, search etc. we can omit those for now to focus the discussion within 40–45 minutes.

Non Functional Requirements

- **Durability** — Places added or ratings given by users should not get lost.
- **Availability** — It must always be possible to fetch recommendations, add new places or give ratings etc. through the app.
- **Latency** — Recommendations and search results must be fetched with minimal latency.
- **Eventual Consistency** — Places added need not be immediately available for recommendations.

Traffic and Throughput Requirements

- Number of registered users = 1 billion
- Number of daily active users (assuming 50% of total user base) = 500 million
- Average number of times each user opens the app in a day = 5
- Total number of places added = 3 billion
- Number of requests per second = Number of requests per day/86400 = 500 million*5/86400 = 30K QPS.
- Maximum QPS = 1 million

The “On My Computer” Approach

The recommendation algorithm works as follows:

1. Given the user’s latitude and longitude, fetch all places that are within a radius of say 10KM from the user.
2. With different attributes of the user and a place from step 1, call a ML model to get a score for the (user, place) pair.
3. Sort the places from step 1 using the scores obtained in step 2 (highest to lowest)

4. Send the results back to the client with pagination i.e. 50 recommendations at a time with an offset.

The scoring algorithm takes as input the following attributes of users and places:

- distance of the user from a place
- type of the place — “restaurant”, “historical”, “park” etc.
- average rating of the place
- number of ratings of the place
- has user visited this place earlier
- fraction of times user has been to this place type earlier.
- … and so on.

The model computes a score using the above features:

*score(user, place) = w1*distance + w2*type + w3*rating + …*

When a new place is added, it is inserted into a local Postgres database table “places” with the following fields:

(id, place, type, city, country, latitude, longitude)

We are using a Postgres database here because we **need ACID properties to handle simultaneous updates** to a place by two different processes. For now lets assume there is a service that can detect duplicate places.

The ratings are stored in an **in-memory HashMap: place_id -> (average_rating, num_ratings)**

For a new rating ‘ratingNew’, it is updated as:

average_rating = (num_ratings*average_rating + ratingNew)/(num_ratings+1) and then num_ratings += 1

To find the nearby places, we should be able to find places within a given radius from the user's current location efficiently.

One brute force solution is to use an SQL query as such (given user latitude and longitude is X, Y and search radius of R):

SELECT * FROM places WHERE (latitude-X)²+(longitude-Y)² ≤ R².

The formula is not as simple as shown above and the actual formula uses something known as the **Haversine distance**.

But the drawback here is that for a single query to fetch recommendations we have to scan all the records in the database.

Although we can filter by country and city as well, but still it could be that a particular region has a few million places.

Some geospatial data structures that we can use are:

- Quad Tree
- R-Tree
- KD-Tree

****Postgres and MySQL both support R-Tree based indexing.**

We will look into **QuadTrees** for this problem and would design a service that would use a QuadTree to fetch the nearby places.

Assuming that Earth is flat and is a 2D plane where each point is represented by (latitude, longitude) where 0 ≤ latitude ≤ 180 and 0 ≤ longitude ≤ 180.

We will divide the 2D plane into 4 equal parts using a horizontal split and a vertical split at the midway. We will continue to do the splits until we have a rectangle with either 0 or 1 points (leaf node) within it.

To add a new place to the QuadTree:

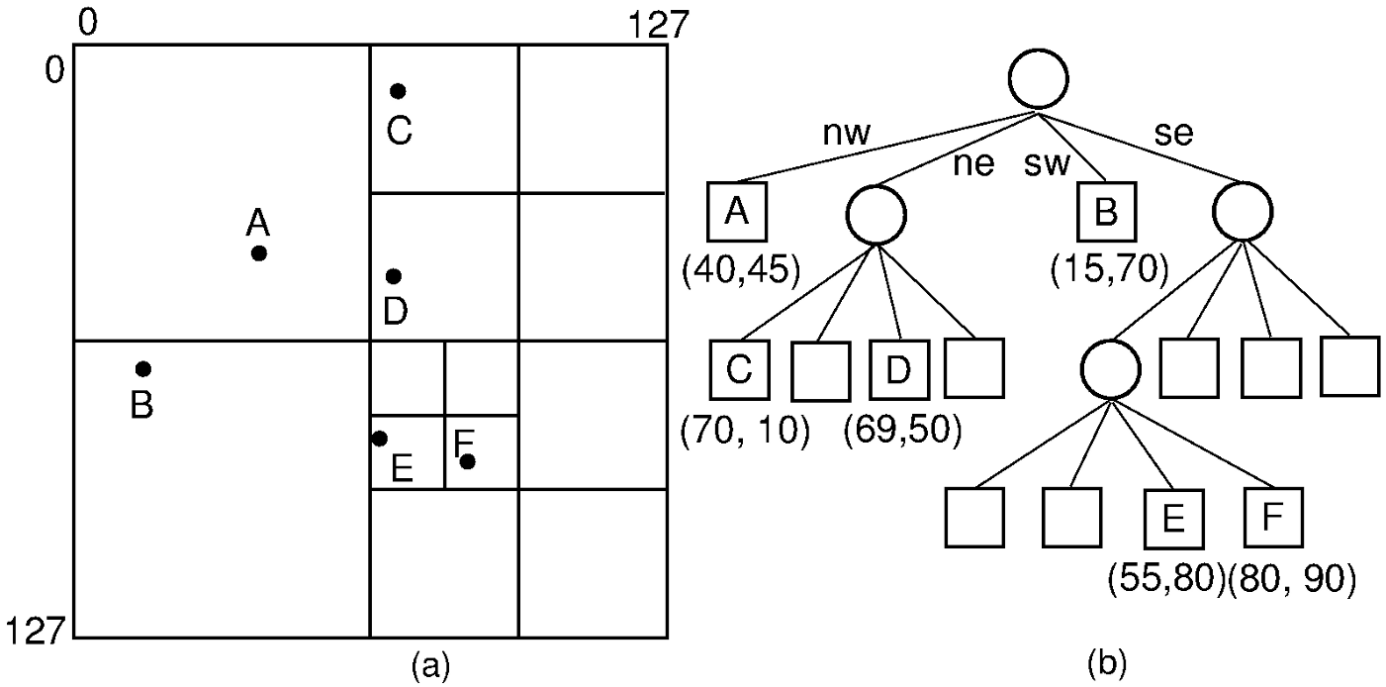
---

1. If we land on a leaf node with 0 places, then we simply add the new place in this node.
2. If we land on a leaf node with 1 place already in it, then we again continue to split the region into 4 equal parts until all the leaf nodes have either 0 or 1 place in it.
3. If its a non-leaf node, then depending on the latitude and longitude of the place it is recursively sent to one of the 4 sub-rectangles until it reaches a leaf node in which case either point 1 or 2 will hold true.

To find all places within radius R of a give coordinate (X, Y):

---

1. Find the leaf node in which the coordinate (X, Y) lies using the above algorithm.
2. To find which all sibling nodes of the current node will come inside radius R, find the vertical distance of (X, Y) from the horizontal split, horizontal distance from the vertical split and the distance from the intersection of the splits. If all 3 distances are ≥ R, then there could be places within all the other 3 quadrants which is within distance R. Do a linear search.
3. Repeat step 2 by moving to the parent node of the current node only if there is at-least 1 sibling node within radius R else exit.



QuadTree

There are obviously certain drawbacks with using QuadTree:

- **It is not balanced. R-Trees are balanced.**
- Partitioning is done at the level of the 2D plane and not based on the distribution of the places. One quadrant can have 1 million places and remaining 3 can have 10 places.

Since now we have all the basic components to build the service, lets take a look at whether we can run all these components on a local computer.

Assuming 3 billion places and each have the attributes: (id, place, type, city, country, latitude, longitude)

Number of bits for id = log2(3 billion) = 32

Number of bytes for place = 15 bytes = 120 bits

Number of bits for type = 6 bits (assuming there are 50 different types of places)

Number of bits for city = 14 bits (assuming number of cities in the world is 10K)

Number of bits for country = 8 bits (assuming there are 200 countries)

Number of bits for latitude and longitude = 32 bits

Total size of the “places” table = 3billion*(32+120+6+14+8+32) bits = 74GB

74GB can be easily accommodated on 500GB-1TB laptops. But we would want to have replication of the data so that we do not lost the data when hard disk crashes. **Thus it would violate Availability and Durability if laptop crashes.**

For the HashMap: place_id -> (average_rating, num_ratings), assuming id for place is 32 bits and average rating is 4 bytes and num_ratings is 4 bytes i.e. 32*3 bits, the total size of the HashMap is approximately 3billion*32*3 = 33GB. But most modern hardwares support only 16GB.

Also due to system crash, the HashMap would be lost thus violating Durability.

Assuming the QuadTree to be balanced, the height would be O(logN). Thus number of nodes in the tree is O(N*logN), where N=3billion.

Assuming that each **internal node stores an integer in the range [0,3] indicating NW, NE, SW, SE directions**, thus each internal node has size of 2 bits. Each leaf node also stores the place_id, thus 32 bits.

Total size = N*logN*2bits + N*32bits = 3billion*32*2 + 3billion*32 = 34GB. Thus it is not possible to store the entire QuadTree in memory of a single laptop.

Using Distributed Systems

To handle the durability of the ratings data, instead of a HashMap, we can use the same Postgres database because again we need **ACID properties as ratings will be updated often**.

(place_id, average_rating, num_ratings)

Whenever a new place is added, make a POST/PUT request to the API server. The API server does the following:

1. Generates a place_id.
2. Create a SQL query and insert the record into “places” table
3. Create a SQL query for average_rating and num_ratings and insert it into “ratings” table
4. Insert the place_id along with latitude and longitude into the QuadTree service.
5. Return success/failure message to client.

Steps 2 & 3 is a single transaction i.e. both insertions are successful or the data is not inserted and is retried again. If either only 1 of the insertion succeeds, then those changes are rolled back. For step 4, it is good to have it as part of the transaction but we can always recompute it with the “places” table.

Even after failing after several retries, the message would be queued into a Kafka topic to be retried after a day or so. The Kafka topic also needs to replicated so that no data

should be lost.

To handle fault tolerance and availability, we need replication. For replication, we can use either **Master-Slave** or **Multi-Master replication**.

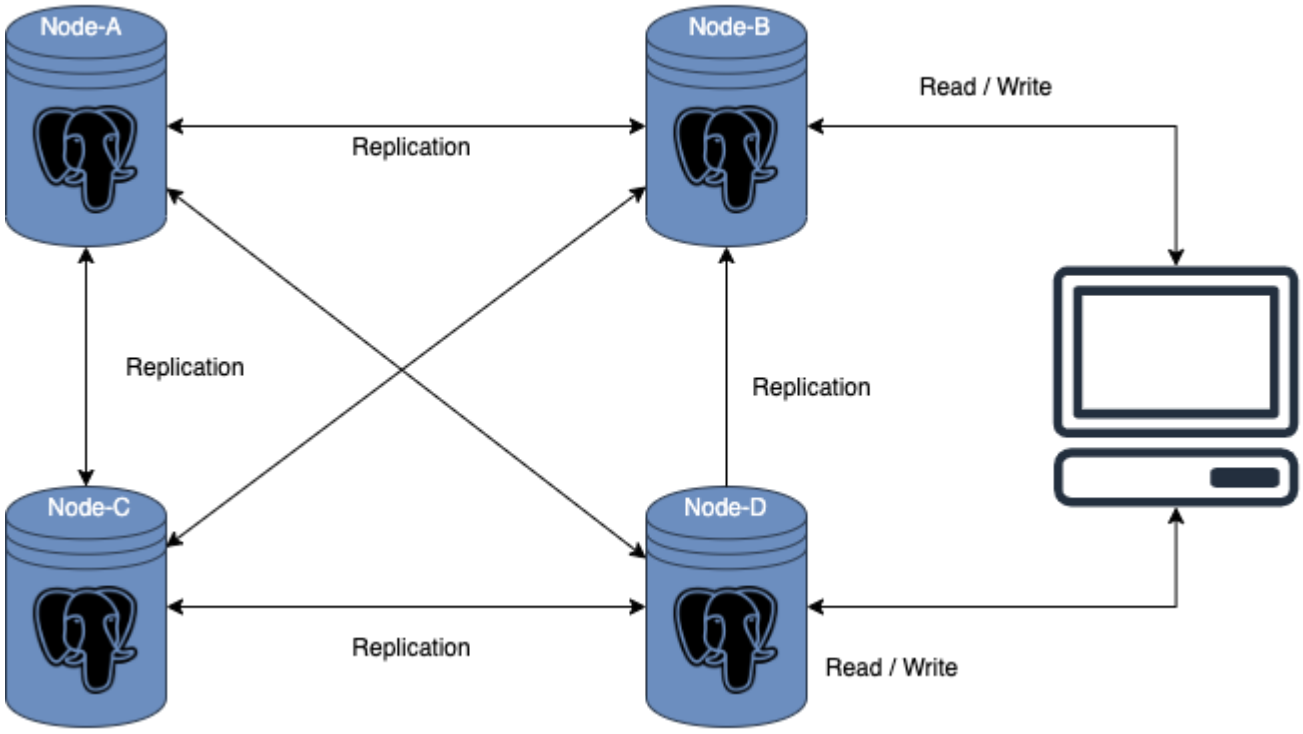
Given the following requirements:

- Eventual Consistency
- Globally distributed users
- High durability

*We can choose to go with **Multi-Master replication** since, there will be multiple Postgres master instances globally distributed. All writes will go to the closest master and from there asynchronously replicated to all other masters. This way writes will be fast as well as less prone for the TCP packet to get corrupted or lost due to network issues.*

But the drawbacks are obvious:

- **No ACID Transactions** — With a single master, if a record is being updated simultaneously by 2 users both request will come to the same master and thus we can use row based locking effectively. But with multiple masters, request for the same row simultaneously by 2 users can land up at different master nodes depending on how globally separated those users are. But chances of this happening are low as mostly users will add ratings for places nearby to them but there could be situations where someone travelled and back and then gave the ratings. We can also use Redis based locks to coordinate the updates but it will be slow.
- Master is always consistent and is the source of ground truth with master-slave replication but with multi-master, inconsistent updates might happen even for non simultaneous requests. For e.g. user A gave a rating to place X and after a few seconds user B also gave a rating to place X. But after A's rating, all the masters were not replicated before user B's rating and few of the masters will be in conflict. But we can live with slight inconsistencies in ratings as 1 or 2 inconsistent ratings will not drastically change the average rating for popular places but it could be a problem for new places or unpopular places.



Multi Master Replication

Since the size of the QuadTree is 34GB, how do we store it in memory ?

One solution is to partition the places data by city or country. Thus each QuadTree will work on only a certain range of (latitude, longitude). Assuming that we partition the QuadTree by country, since there are approximately 200 countries, thus each partition on average stores 175MB of QuadTree data.

But most often data is skewed across countries. For countries with lots of places added, we can further partition on cities.

How to handle 1 million peak QPS in QuadTree server ?

Assuming that the latency for fetching nearby places from the QuadTree is 50ms and the number of CPU cores in each instance is 8.

The maximum QPS from a single instance = 8*1000/50 = 160.

To handle 1 million QPS, we would require 1million/160 instances = 6250.

Assuming that we keep 3 replicas for each QuadTree server, thus the full data is partitioned across 2084 instances, with each instances holding approximately 17MB in-memory data.

How to handle 1 million peak QPS with DB ?

After fetching the nearby place_ids from the QuadTree, the Postgres DB is queried to get the places metadata.

Reading from disk is slow as compared to reading from memory, thus it cannot handle 1 million QPS at milliseconds latency. To handle the high read QPS we use a LRU Redis Cache to cache the places metadata.

The total size of the data in the DB is around 100GB (74GB for places table and 33 GB for ratings table).

Assuming each Redis instance can handle QPS of about 70K, number of Redis instances required = max(1 million/70K, 100GB/16GB) = 15 instances.

How to handle durability of the QuadTree ?

Storing the QuadTree in-memory is quite easy but if we have to persist the tree so that we do not lose the data in case of a crash, we have to think of different approaches:

1. Store the tree in a file where each line is one level of the tree: **node_id1:parent_node_id1, node_id2:parent_node_id2**, …. Then we can re-construct the tree by reading the file line by line and adding the nodes in the current line as children to nodes in the above line.
2. **Adjacency list representation**, each line is of the form — parent_node_id:child_node_id. The drawback here is that to construct a single level of the tree we might need to read multiple lines before that.

Approach 1 seems to be a good approach in order to deal with failures. If there is failure in reading a file, we have already constructed the tree upto certain level correctly and next time we can start reading from this line onwards again, but with approach 2, we have to re-read the entire file again.

The QuadTree is periodically written to file. In order to write to file, the **entire tree has to be written everytime** because new nodes are added everytime and internal nodes are also split.

How to deal with 1 million QPS with ML models in real-time ?

*One solution is to **pre-compute the nearby recommendations**. Run periodic jobs that will first fetch the nearby places within a radius of 10KM, then extract the features for user and all those places and call the ML model to get the scores for each place.*

Finally the background job will create a JSON object of the exact data that is going to be displayed to the user and store this JSON object in a Redis HashMap with the user_id as the key.

Assuming that each user payload has on average 500 recommendations and each recommendation has (place_name, url, type, distance etc.), size of each is 120+400+80+32 bits = 632 bits. Thus total size of the cached data = 1billion*500*632 = 36TB.

Number of Redis instances required = 36TB/16GB = 2304 (with replication)

The QPS these many instances can handle = 2304*70K = 160 million (assuming each instance has QPS of roughly 70K).

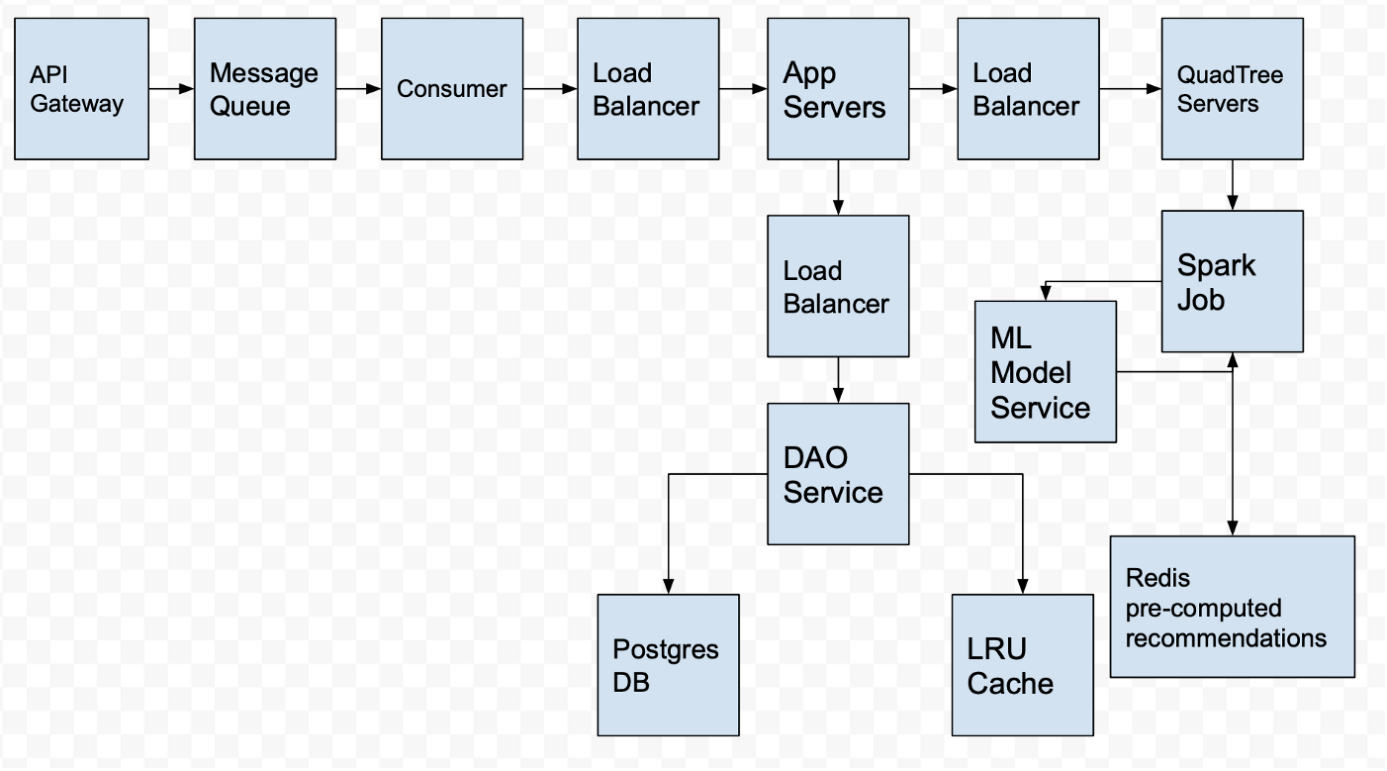
The data is partitioned on the user_id key.

Now all reads will first go to the Redis cache and if the user_id is a key, then fetch the results by pagination else compute the recommendations in real time and add it to the cache for future use.

What if the ML model service is not working ?

Return the nearby places ranked by ratings highest to lowest. But we need to store these results separately from the more accurate results when the model is working fine, because when the model is up again, we will not use these results anymore.

The Pipeline



Useful Resources

- https://redis.io/topics/benchmarks
- https://www.digitalocean.com/community/tutorials/how-to-perform-redis-benchmark-tests
- https://serverfault.com/questions/164980/how-many-selects-per-second-can-a-mysql-server-run
- https://stackoverflow.com/questions/1824772/how-many-requests-can-sql-server-handle-per-second
- https://severalnines.com/database-blog/how-measure-database-performance
- http://highscalability.com/blog/2015/5/18/how-mysql-is-able-to-scale-to-200-million-qps-mysql-cluster.html
- https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/quadtrees.pdf
- http://www.cs.cmu.edu/~christos/PUBLICATIONS.OLDER/vldb87.pdf
- https://www2.cs.sfu.ca/CourseCentral/454/jpei/slides/R-Tree.pdf

- https://stackoverflow.com/questions/4326332/what-is-the-difference-between-a-kd-tree-and-a-r-tree
- https://trac.osgeo.org/postgis/wiki/UsersWikiExamplesFindNearby