Proximity Service

xxx

March 2023

1 Step1

Understand Problem and Establish Design Scope
Functional Requirements:

- return all businesses based on user lat log and radius
- update/add/delete business by business owner
- customer views detail about a business

Nonfunctional requirements:

- low latency
- high availability and scalability
- data privacy

QPS estimation:

- 100 million user per day
- 5 times per user
- business owner operation times negligible
- $QPS = \frac{100million*5}{86400} = 5000$

2 Step2

2.1 Propose High-level Design

- API Design
  RESTful API

- High-level Design
  users call location-based service
  business owners call business service

- algorithm for finding nearby services
  (expanded later)

- Data model
  it is read heavy model, so we can use relational db (mysql etc).
  using business_id as key

2.2 Expand on location-based service (LBS)

stateless, easy to scale

- two_dimensional search
  use SQL WHERE to query for both lat and long
  WHERE Lat BETWEEN LatRange AND Long BETWEEN LongRange

  - Good: Easy to implement and understand
  - Bad: query involves too many data from db

- evenly_devided grid:
  devide all the world into fixed small grids

  - Good: easy to understand and find the grid
  - Bad: waste of memory, some grid in the sea, never used

- GeoHash:
  divide using different precision
  biggest grid: 00, 01, 10, 11
  if a grid has too many business, divide it further:
  0101, 0100, 0110, 0111

  - Good: easy to scale with precision (length of binary string)
  - Bad: zoom level is fixed, has boundary issue:
    - 0100 and 0001 are adjacent but do not share prefix
    - if there is not enough business in the grid, we need to expand to the adjacent grid

        - A solution: while in a grid, mathematically compute the adjacent 8 grids GeoHash value and add them all to the API, so won't miss anything

- QuadTree:
  in-memory structure, need to create a tree
  GeoHash is just a structless table
  Memory estimation:

  - if 200 million business in our db

  - each leaf node contains an area with 100 businesses

  - each leaf node size 800 bytes

  - each internal node size 100 bytes

  - number of internal node is $\frac{1}{3}$ of leaf nodes

  - in-memory storage is: $\frac{200million}{100} * 800 + \frac{200million}{100} * \frac{1}{3} * 100 = 2GB$
    it can fit in memory well.

  - Good: fit it in the table and query
  - Bad: Need to build at server start-up time, might take minutes
    if business add or delete, need to go down the tree and fix the node

- Google S2
  Some mapping from map to 1D space

3 Step3

Design deep dive

3.1 scale the database

the db is small, do not need to shard
just use replica to replicate thru machines to ease read traffic

3.2 use caching

it can fit in memory
can use redis cache: $geoHash, business_id$ to get business detail for a specific user request