Functional Requirements:

1. Show map tiles surrounding a given location
2. Search for entities(restaurants, doctors etc) given a location
3. Plan route between locations
4. Real-time traffic information

Design Constraints:

1. Static Dataset except real-time traffic
2. Read-only System except real-time traffic and subsequent updates due to changing traffic
3. Number of map-views : thousands/sec
4. Number of search: thousands/sec
5. Number of route planning requests: thousands/sec

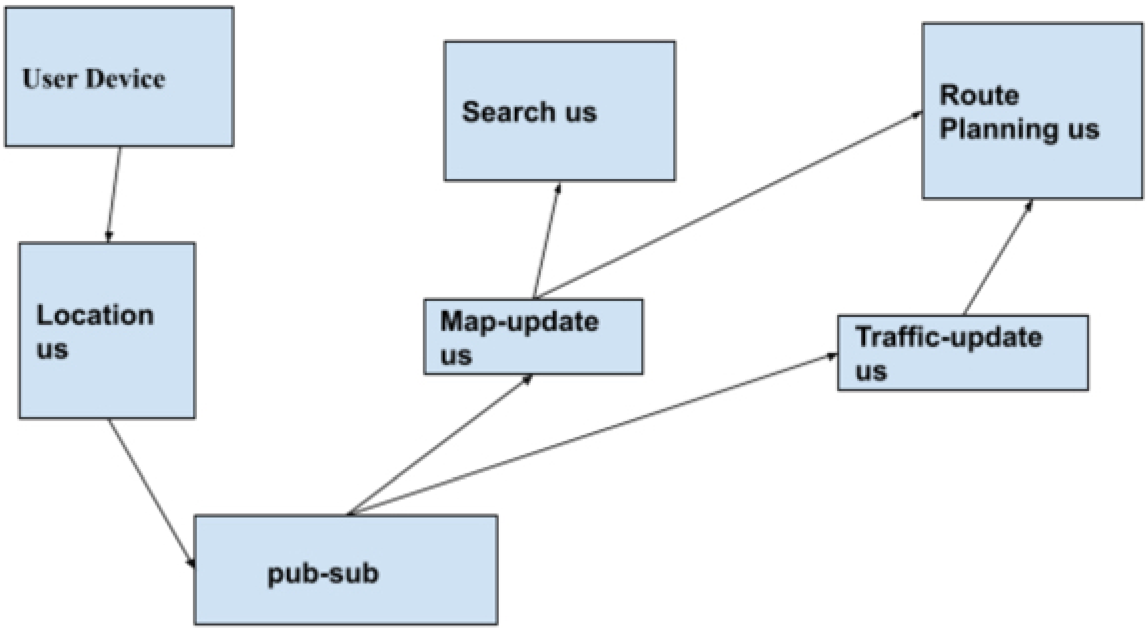
Not like social media or google search where qps is in several-tens-of-thousands/sec.

Real-time traffic information can be received from
1. Traffic sensors and collaboration with government transport department
2. Crowd-sourcing with GPS enabled sensors

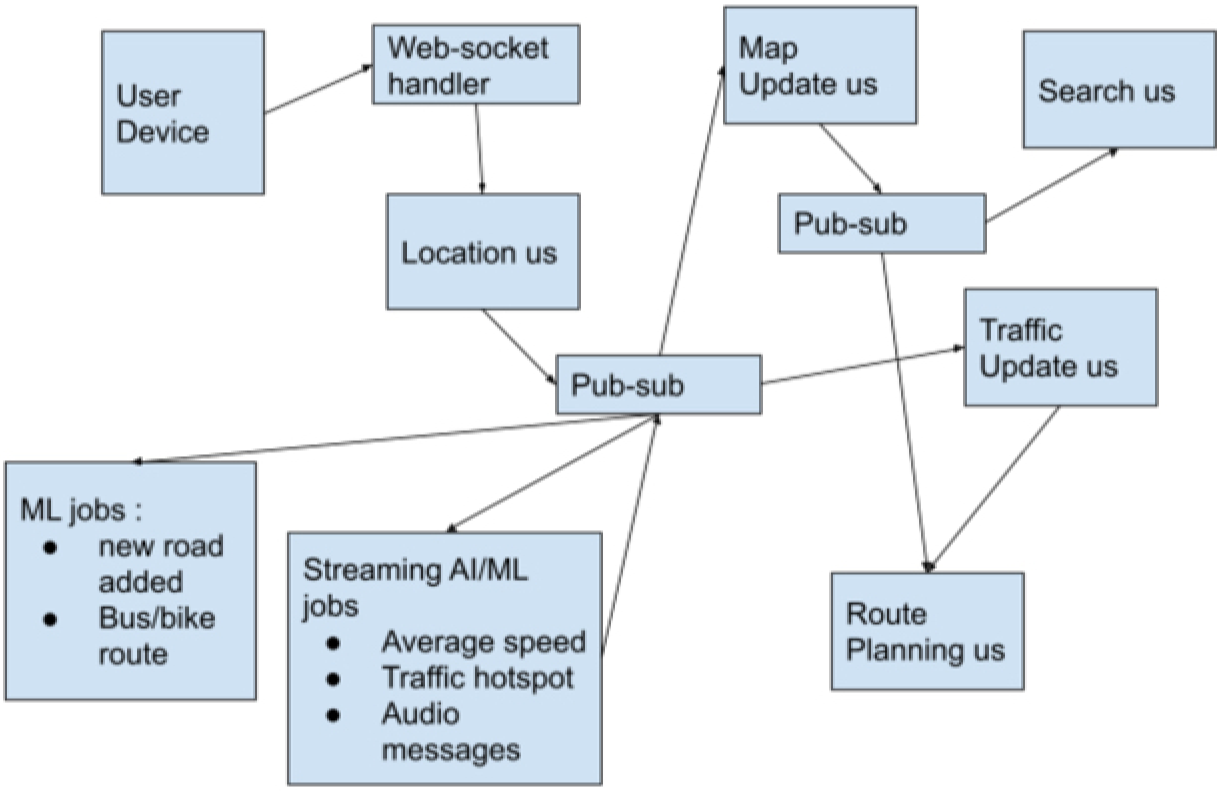
Micro-services

1. Location microservice
2. Search microservice
3. Route Planning microservice

Logical Architecture Block diagram



Location updates communicated via the pub-sub



1. Path recommendation versus user's actual path selection can provide hints on learning new paths for the Route planning microservice : accuracy of prediction and correction
2. User of Google maps include companies like Uber etc apart from individual users
3. Third-party traffic management inputs for accident information, weather updates, live traffic information
4. Historical data on routes can be input for route-planning
5. Analytics on the search microservice data
6. Third party data validation against real traffic flow
7. Location data can be used to identify popular points of congregation and user-profile
8. Show different maps to people in different regions : for instance, people in the India-China border, Google maps will show a different map depending on the user's location either in India / China.
9. To provide recommendation from the search microservice, a recommendation system can be built around historical information about the user and rankings of places around a location.

Location microservice

App

1. Location metadata
    a. {key : location-id, value: offset inside tile image}
    b. hierarchical in nature : allow zoom-in/zoom-out
2. Fully cached access from memory

Cache (distributed Key-Value store)

1. Complete in-memory access to metadata via R-trees and Quad trees
2. R-trees : to find nearest neighbor or overlapping regions
3. Quad-trees :
    a. keep tile image information.
    b. are not balanced since similar regions (ocean) or unrecognized regions (North Korea) can be treated as single node higher up in the tree.

Storage (distributed file system

1. Tile image data :
    a. { key : location-id, value: tile image file, tile offset byte stream}
2. R-trees and Quad trees are serialized and stored on disk as files for recovery purposes only

API : CRUD API

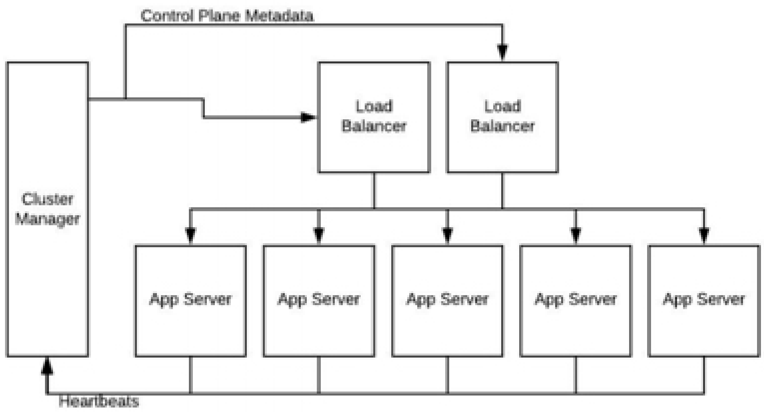
Need to scale

1. Throughput : NO for data or metadata;
    a. thousands of req/sec,
    b. average 1MB size images
    c. (a) and (b) translates to 1GB/sec or few GB/sec which can be served by a single server
2. API parallel : NO
    a. CRUD key-value API
    b. All data are served from in-memory cache and so constant/logarithmic lookup
3. Availability : Yes
4. Geo-location : NO
    a. The map and tile image of the entire world is replicated across the world
    b. The data is static
5. Hotspot : NO
6. Storage: YES

a. Tile data for locations are saved at different zoom levels: say 4 levels of resolution
b. Land area : 5 X (10 pow 7) square-km
c. 10m X 10m block each of 1MB
d. (a) (b) and © implies 350 PB X 4 zoom levels i.e 1500 PB

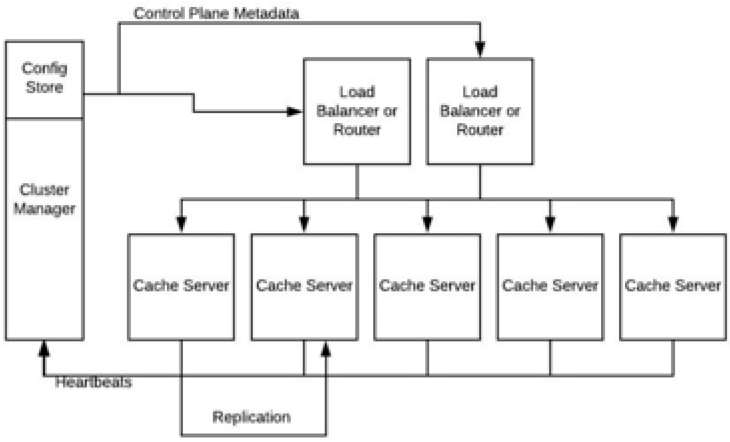
Location Microservice: Dist. Architecture

Architectural layout for app tier



Location Microservice: Dist. Architecture

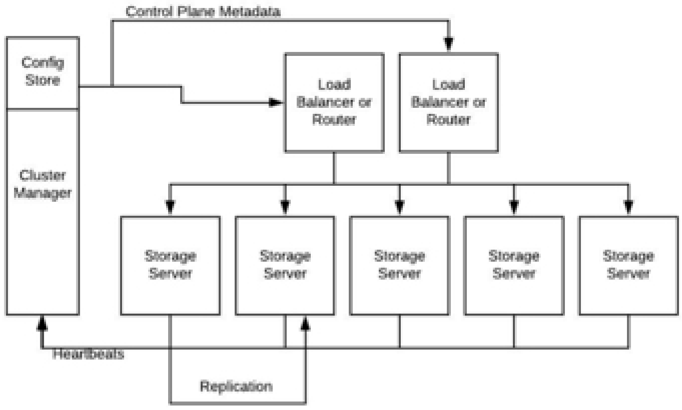
Architectural layout for cache tier



Key location mapped to specific cache server. Distributed Key-Value store.

Location Microservice: Dist. Architecture

Architectural layout for storage tier: both for distributed K-V store as well as distributed file system



Distributed file system :location to tile image.

1. Sharding :
    a. Horizontal sharding based on location-id
    b. Tile data : horizontal sharding of file system
2. Placement of shard in servers
    a. Consistent hashing
3. CP or AP
    a. AP: read-only system

Search Microservice

Types of searches :

1. Give all restaurants in SF south bay : location provided
2. Or, All Indian restaurants near me : location not provided

Crux is a Document Search problem :

1. Inverted indexes of entities (key) to location id (value) :
    a. {dentist : location idx}
    b. {office: location idy, location idz}
    c. {restaurant: location idp}
2. Inverted indexes are saved in Geo-spatial indexes stored in DB (example mongo DB)

App

1. Geospatial queries : Join/Intersection (all indian restaurants around location)
2. Query processing
   a. Location not provided : All indian restaurants near me
      i. Determine user location with GeoInterest (tile location)
      ii. Show number of restaurants in that location with GeoWithin
   b. Location provided: All indian restaurants in NY city
      i. Find restaurants within 5/10 miles specified by user using GeoNearby

Cache

1. All search from cache

Storage

1. Db to store Inverted index / Geospatial Indexes

Complexity of search us API (unival tree from document search inverted index) =
Total number of elements X (O(log(number of search attributes) + O(number of search attributes))

Total number of elements = number of search attributes X size of the list for inverted index of each search attribute

Need to scale

1. Storage : YES
   a. Indexes cannot fit into single server
2. Throughput : NO
   a. Thousands of requests/sec
3. API parallelization : YES
   a. joins/intersection need API parallelization
4. Availability : YES
5. Geo-location based distribution : NO
   a. Static data and the queries could be served from any location

Search Microservice : Distributed Architecture

1. Sharding : vertical sharding by location-id : all entities in locationID in one shard
   a. If location is not provided in query, may need scatter-gather in load balancer if entities spread across multiple locations so results from multiple shards would need to be processed

b. If location is provided in query, more local requests in a shard

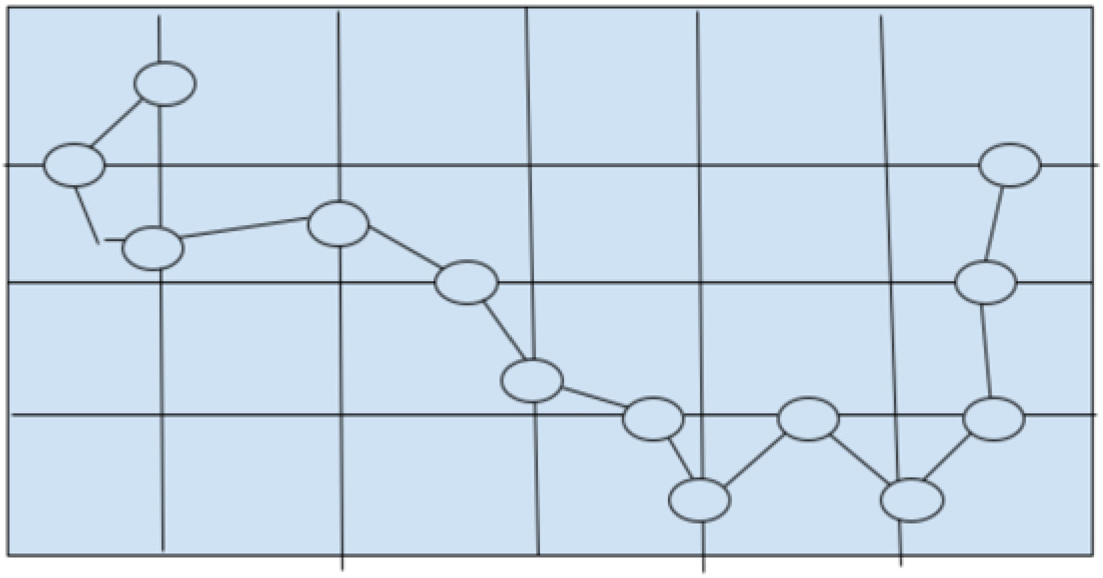
2. Replication : for availability and throughput
3. AP system : read-only
4. Placement of shards in servers : Consistent hashing

Route Planning Microservice

Path finding algorithm

1. Mark milestones in every quad
2. Cache route data from source milestone to target milestone
3. Compute route from {source -> source-milestone} and {target-> target-milestone}
4. Run path-finding on the entry-exit points of every quad

Entry/Exit points in each Quad



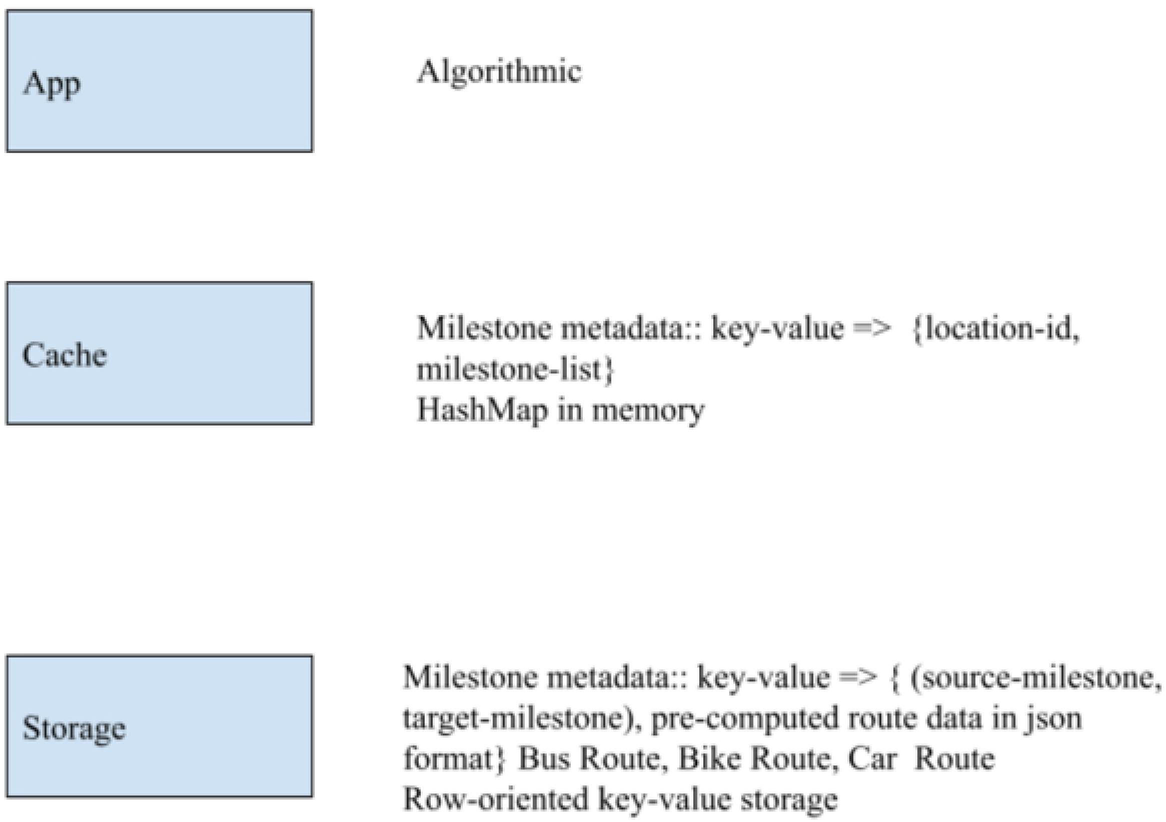
External data

1. Traffic :
    a. Low
    b. High
    c. Med
2. Weather :
    a. Rain

Possible algorithms in APP tier :

1. A* path finding algorithm : computes the best heuristic (say distance by {latitude, longitude}) from current node + motorized road network distance between nodes. Memory footprint is heavily reduced since paths which are not near target (by euclidean distance) are discarded
    a. Other heuristics can include country roads, Expected time of Arrival, Average Speed, Traffic Information
2. Or Floyd Warshall all-pair shortest paths and cache results in storage/cache tier;
    a. Every quad can have milestones
    b. For every {quad-1 milestone, quad-2-milestone} pairs

Key- Value CRUD API

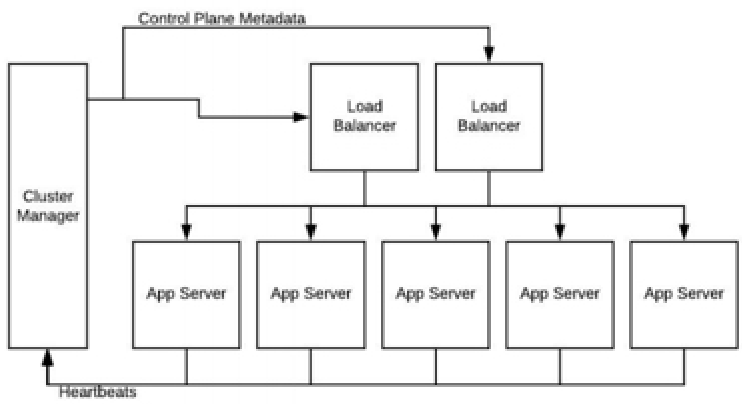


Need to scale

1. Storage : Yes.
    a. Depends on how many ordered pairs of (source-milestone, destination-milestone) need to be stored and cached
    b. Bus Route, Bike Route, Car Route
2. Throughput : No

a. Requests are thousands/sec
3. API parallel : Yes
    a. A* or Greedy or DP : O(nlogn) algorithm can be API parallel and results scatter-gathered from (source->source-milestone) , (source-milestone->target-milestone), (target-milestone, target)
4. Geo-location : NA
5. Hotspot: NA
6. Availability: Yes

Architectural layout for app tier, needs to scale for API parallelization



1. Sharding :
    a. Logic sharding in App tier
    b. Horizontal sharding in cache/storage tier
2. Replication :
    a. For availability and throughput
    b. Scatter-gather API
3. Placement of shards in servers: Consistent hashing
4. AP : read-only system mostly

Use of Graph DB or Not :

For Google Maps route planning
1. Not too many relationships between locations and milestones
2. The location information does not change often
3. AP system and there is no strict consistency requirements : very few writes/updates on the route planning

So , a graph DB may not be needed. [What are the scenarios where graph DB may help?]

If functional requirements include recommendation based on locations where relationship between user + location + attributes of restaurants + other attributes like users' friends choices and so on are included, a Graph DB may be considered.