DynamoDB

- Fully managed, highly scalable, key-value service provided by AWS.

- Not an open-source.

⇒ The Data Model :-

In dynamodb, data is organized into tables, where each table has multiple items that represent individual records.

Ex:- Users table in DynamoDB :-

```
{ "personId": 101,
  "name": "Paul"
},
{ "personId": 102,
  "name": "Mary",
  "address": { "city": "Town",
               }
}
```

- DynamoDB uses JSON for data transmission, it's merely a transport format.

Partition Key & Sort Key

DynamoDB tables are defined by a primary key; which can consist of one or two attributes :-

(1) **Partition Key :-** A single attribute that, along with the sort key (if present), uniquely identifies each item in table.

- DynamoDB uses the partition key's value to determine the physical location of item within the database.
- This value is hashed to determine the partition where the item is stored.

(2) Sort Key (optional) - An additional attribute that, when combined with the Partition key, forms a Composite primary key. The sort key is used to order items with the same partition key value, enabling efficient range queries & sorting within a partition.

Primary key = {Partition key} : {Sort key}
optional

⇒ But what is actually happening under the hood?

DynamoDB uses a Combination of Consistent hashing and B-trees, to efficiently manage data distribution & retrival.
for Partition key
for sort keys

* Composite key operation - when querying with both keys, DynamoDB first uses the partition key's hash to find the right node, then uses the sort key to traverse the B-tree & find the specific items

This two tier approach allows DynamoDB to achieve both → horizontally scalability
→ Efficient querying within partition.

Secondary Indexes

But what if you need to query your data by an attribute that isn't the Partition key? This is where secondary indexes come in. DynamoDB supports two types of secondary indexes:-

(i) Global secondary Index (GSI) –

↳ An index with a partition key & optional sort key that differs from the table's Partition key. GSIs allow you to query items based on attributes stored on entirely different physical partitions from the base table & is replicated seperately.

(ii) Local secondary Index (LSI):–

An index with the same Partition key as the table's Partition key but a different sort key.

↳ LSIs enable range queries & sorting within a Partition. Since LSIs use the same Partition key as the base table, they are stored on the same physical Partitions as the items they are indexing.

Accessing data in dynamodb-

1. Scan operation
2. Query operation,

Redis

↳ It is single threaded making it very fast & easy to reason about.

⇒ some of the most fundamental data structures supported by Redis:-

(1) strings
(2) Hashes (objects)
(3) Lists
(4) sets
(5) Sorted sets (Priority queues)
(6) Bloom filters
(7) Geospatial Indexes
(8) Time series

In addition to simple data structures, Redis also supports different communication patterns like Pub|sub & streams, partially standing in for more complex setups like Apache kafka etc.

Capability

(1) Redis as a cache
(2) Redis as a Distributed Lock (Used in TicketMaster, Uber)
(3) Redis for leaderboards
↳ Redis's sorted sets maintains ordered data which can be queried in log time which make them appropriate for leaderboard applications.

The high write throughput & low read latency make this especially useful for scaled applications where something like a SQL DB will start to struggle.

- In "Post search Facebook" system design problem, we have a need to find the posts which contain a given keyword (e.g. "Tiger") which have the most likes.
- we can use Redis's sorted set to maintain a list of the top liked posts for a given keyword. Periodically, we can remove low-ranked posts to save space.

(4) Redis for rate limiting

(5) Redis for Proximity Search — (GEOADD, GEOSEARCH commands)

(6) Redis for Event-Sourcing

#1 Facebook's Post Search

For this problem, we're zooming in on the search experience for Facebook. This question is primed for infrastructure-style interviews where your interviewer wants to understand how deeply you understand data layout, indexing & scaling.

Functional Requirements —

Core Req:-

1) User should be able to create & like posts
2) Users should be able to search posts by keyword.
3) Users should be able to get search results by recency or like count.

Out of scope

- support fuzzy matching on terms (e.g. search for "bird" matches "ostrich")
- Personalization in search results
- Privacy rules & filters
- Images & media
- Realtime updates to the search page as new posts come in.

Non-Functional Req:—

1) Fast queries <500ms
2) High volume of requests
3) New Post Must be searchable in short amount of time <1 Minute

4) All posts must be discoverable, including old or unpopular posts. (we can take more time for these)

5) The system should be highly ~~scalable~~ available.

note => If you have a lot of data in your system, there's a good chance you'll have "hot" and "cold" data. Hot data is readily accessed and needs to be served quickly, often from memory.
Cold data is infrequently accessed & can be served from disk, a remote database or even tape.

Scale Estimation —

volume of work:-

Post Created = 1B * 1 post/day / (100K sec/day) = 10^4 = 10K post per sec.

Likes Created = 1B * 10 Likes/day / (100K sec/day) = 100K likes per second.

Let's look at reads

Searches: 1B * 1 searches/day / (100K sec/day) = 10K searches/second.

- while this is a lot (& may burst the 10x this value or more), our system is write-heavy or Read-heavy, another thing to note.

Storage Req

Posts Searchable: 1B posts/day * 365 day/year * 10 year = 3.6T posts

Raw Size: 3.6 posts * 1 kb/post = 3.6 PB

Core Entities

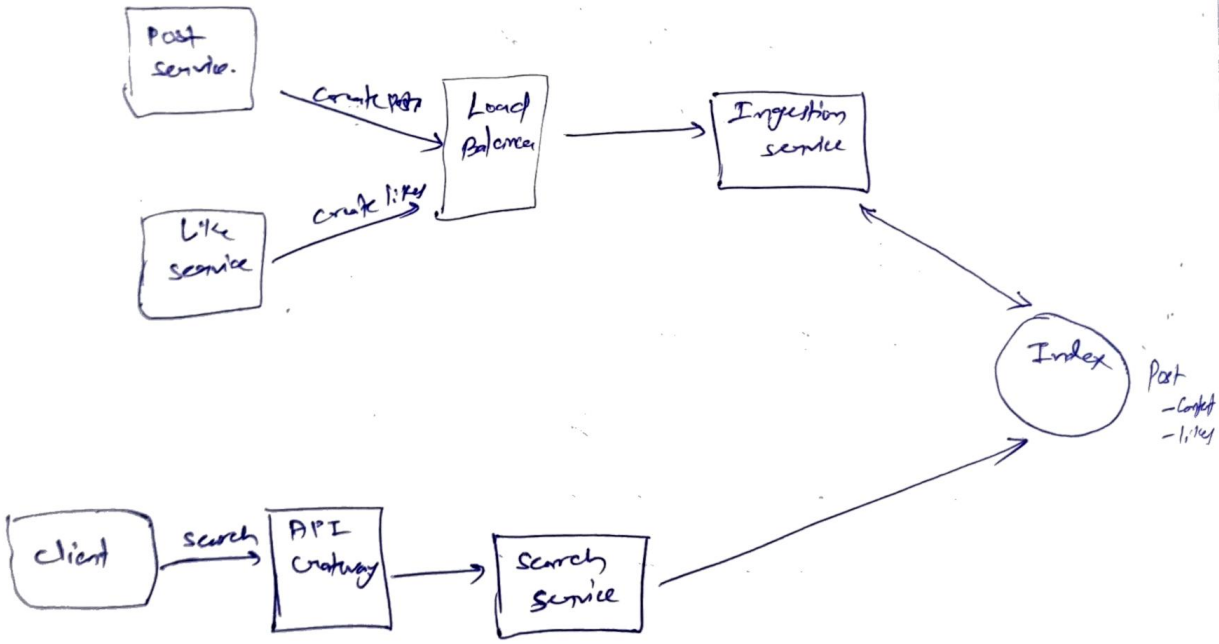
1. User
2. Post
3. Like

API:-

1. // search
   GET /search ?query=QUERY & sortorder=Like/Time
2. // Create a post
   POST /post
3. // Like a post
   POST /post/{id}/like

High level design

1. Users should be able to Create & like posts.
2. Users should be able to search posts by keyword.



~~search post by keyword~~ — ways —

- ⇒ Bad soln — scale an unindexed database
  ↳ (Go through each content in DB & search)

→ Normal Soln — Create an Inverted Index

Create dictionary, map keyword to document ids

Key	Value (Post ids)
Taylor	1, 5, 10
Beach	2, 5, 13, 15

– we can use Redis for this (in-memory)

↓

There are durability concerns but can be handled then with a durable alternative like Memory DB or in a deep-dive.

Challanges — * these post ids list are going to get very large, especially of common keywords. We will have to address this later.

* we also need to write to many keys for every post since a post might have 10–1,000 keywords. We'll need to handle some of scaling challanges associated with this.

③ User should be able to get search results sorted by recency or like count.

⇒ Brute force — Request time sorting

↳ grab all post ids for a given keyword from index & then look up all the posts for timestamp or like count, & then sort those in memory.

↳ challanges – there could be millions of posts for a common keyword – add latency to system.

⇒ (more solution) - Multiple Indexes

↳ Different approach would be to have two separate indexes :-

- one ~~for~~ sorted by Creation (Creation Index)
- one sorted by likes Count (likes Index)

* For 'Creation index' - we can use standard Redis list. We are always going to appending to this list & our query will only be taking from last element

* for the 'Likes Index' - each key can use a Redis sorted list :(keep a list of items ordered by a score)

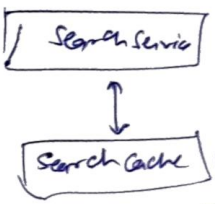
Challenges - (a) doubled the storage for Indexes.

(b) introduced a new problem - likes are happening quite frequently. Each like event require us to update many scores so that the like indexes are up-to-date. This put a lot of stress on our system, which we'll both note for our interviewer & plan to address late.

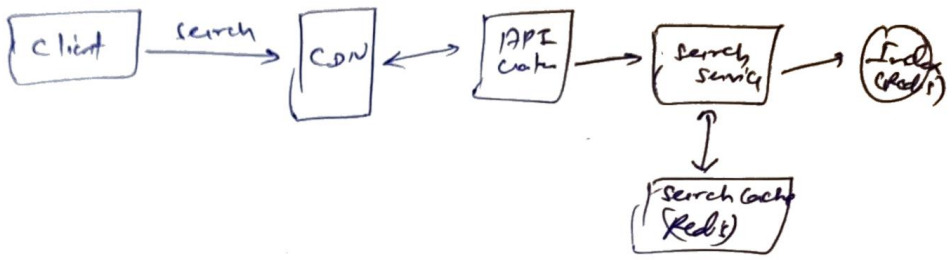
Potential deep dives

(1) How can we handle large volume of requests from users?

⇒ Solution - Use a distributed cache alongside our search service.



⇒ (optimal soln) — Use a CDN to cache at the edge.



(2) How can we handle multi-keyword, phrase queries?

⇒ (brute soln) — Intersection & filter

↳ Challenge ⇒ Millions of post for each keyword
↓
transferring megabytes of data which need to put into a hash table & intersected. This can be hard to do while meeting our 500ms SLA.
— quite computationally extensive (in worst case)

⇒ (smart soln) — Bigrams or Shingles

↳ Create token for each pair of word & index.

Ex:- I saw taylor swift at concert.

Inserted this pairs to indexes.

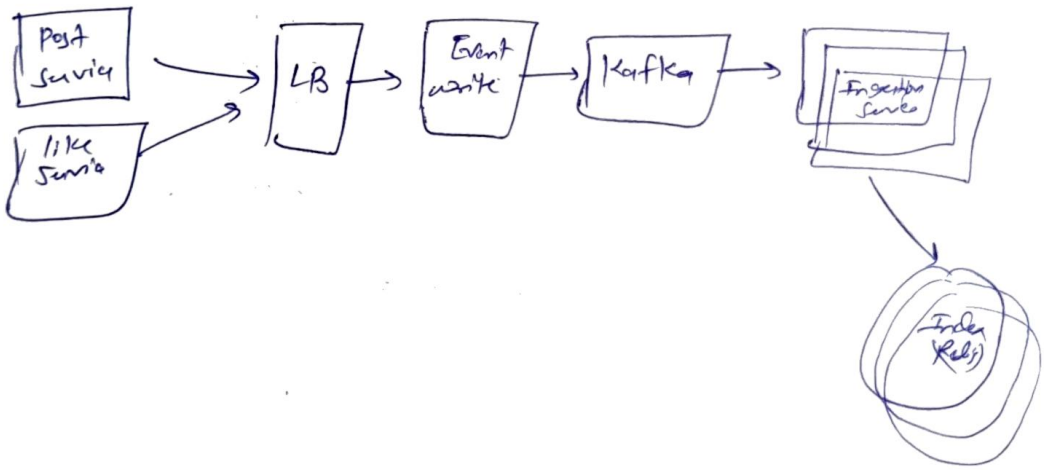
challenge ⇒ * dramatically increases the size of index, so instead of 10M single-words keyword, we might end up 100M+ indexed keys.

Potential remediation — only extract bigrams that are likely to be searched & fallback to intersection approach when there is no entry in our index.

(3) How can we address large volume of writes?

(A) Post Creation –

* Add more capacity to ingestion service & Partitioning the incoming request.
* use log/stream like Kafka → Partitioning load across ingestion service.
* Finally we can scale out our index by sharding the indexes by keyword. This way writes to the indexes are spread across many indexes.



(B) Like Event –

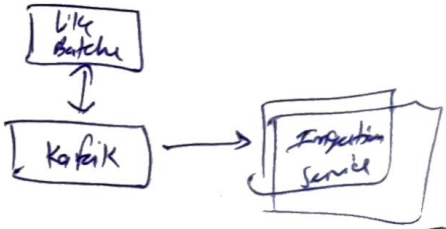
⇒ Viral post – Batch likes before writing to the database.

↳ We can batch likes for a given postid over a period (like 30 second). Then instead of needing to write 500 times for particular viral post, we can make 1 update with an increment of 500.

Challenge –

↳ won't drastically reduce volume of writes since most posts aren't viral

↳ overhead added,

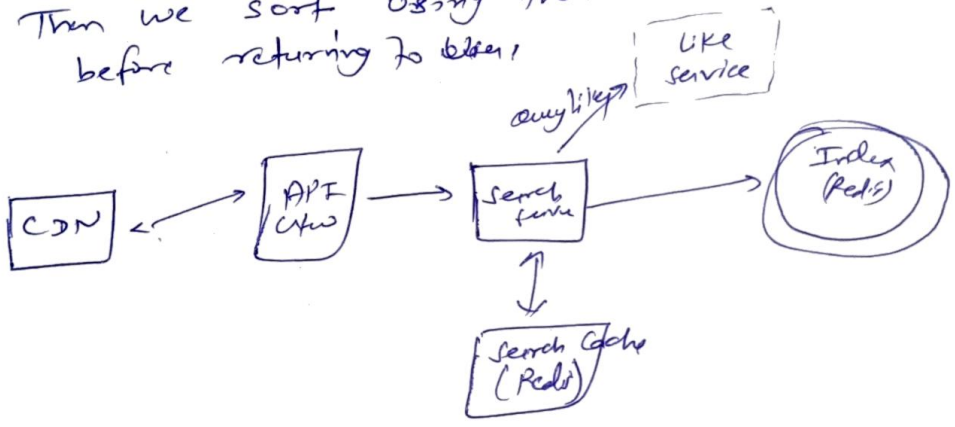


=> (social sol") : Two stage architecture

↳ we can do even better by only updating the like count in our indexes when it passes specific milestone like power of 2 or 10. So we would write to the index when the post has 1 like, 2 likes, 4 likes etc. This reduces the number of write exponentially.

↳ but this mean our index is inherently stale.

- If we want to retrieve N posts for a given query, we grab the top N*2 posts from like index. For each post in that list, we query the "LikeService" for up-to-date count. Then we sort using that fresh like count before returning to client.



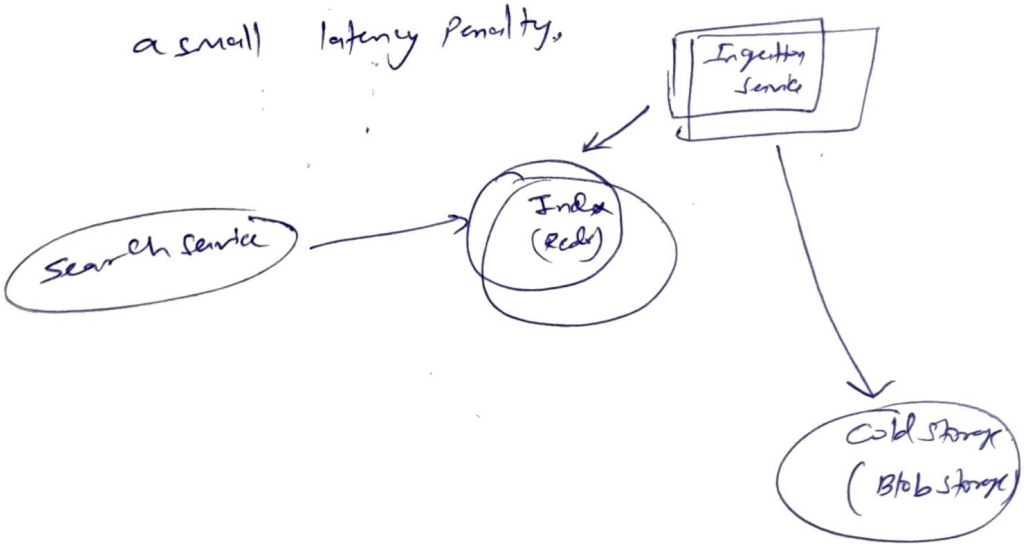
Challenges
↳ more complexity

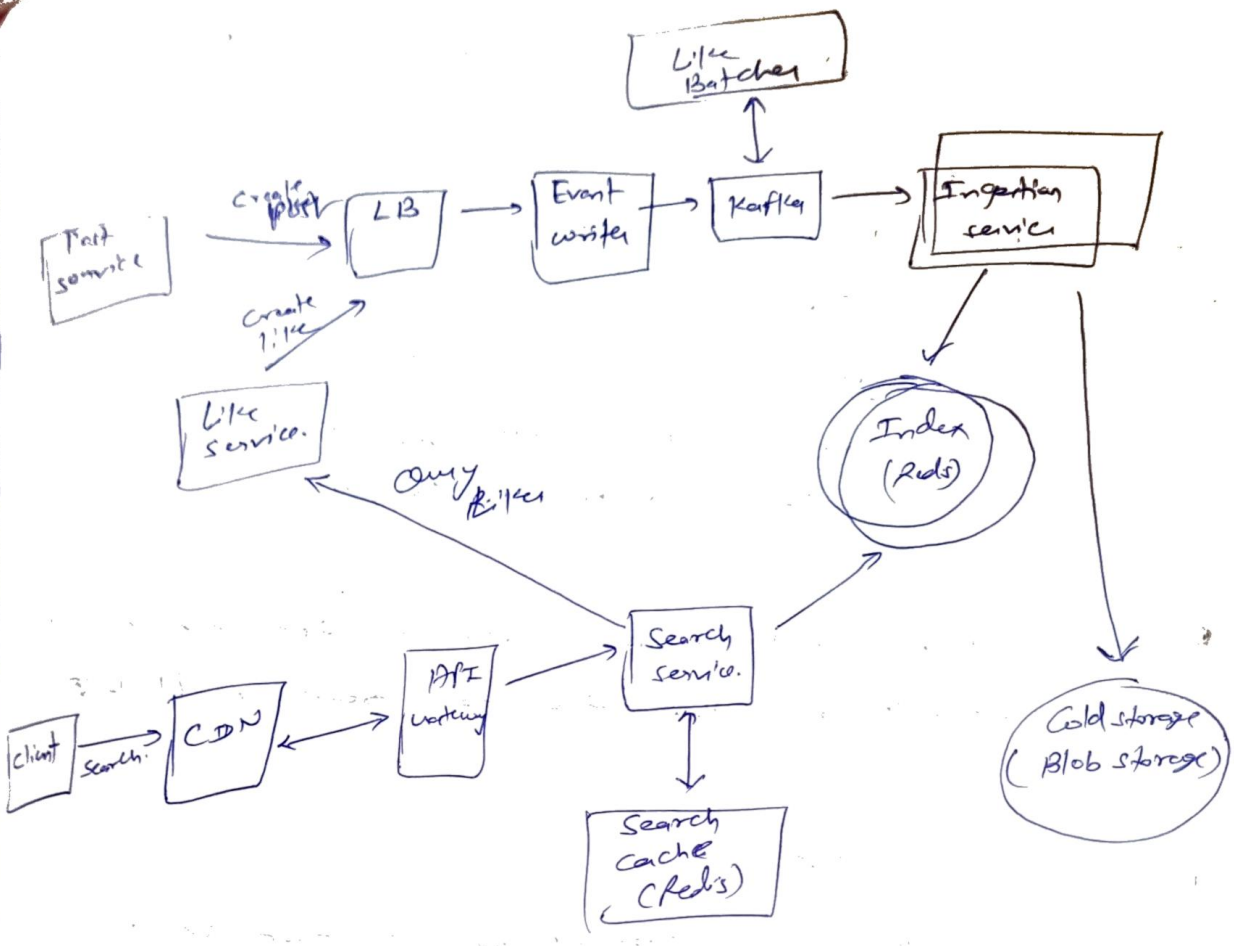
(4) How can we optimize storage of our system?

- First, we can put caps & limits on each of our inverted indexes. we ~~could~~ probably won't need all 10M post with "Mark" contained in their content. By keeping our indexes to 1K - 10K items, we can reduce the necessary storage by orders of magnitude.

Next, most keywords won't be searched often or even at all. Based on our search analytics, we can run a batch job to move rarely used keywords to a less frequently accessed but ultimately cheaper storage. one way to do this to move those keyword indexes to cold, blob storage like S3 or R2

- on a regular basis we'll determine which keywords were rarely ~~used~~ (or not at all) accessed in past month. we'll move these indexed from our in-memory Redis instance to a blob in our blob storage.
- when the index need to be queried, we'll first try to query Redis. If we don't get our keyword then, we can query the index from our blob storage with a small latency penalty.




Like Batcher
Post service
create post
LB
Event writer
Kafka
Ingestion service
create like
Like service
query likes
Index (Redis)
Search service
client
search
CDN
API routing
Search cache (Redis)
Cold storage (Blob storage)

Design a Web Crawler

Web Crawler → Program which traverse web pages → download web pages & following link from one page to another.

- It is used to index the web for search engines, collect data for research or monitor websites for changes.

For our purpose, we'll design a web crawler whose goal is to extract from the web to train an LLM.

Functional Req.

1. Crawl the web starting from a given set of seed urls.
2. Extract text data from each web page & store the text for later processing.

out of Scope

- Actual Processing of text data (LLM etc.)
- Handling non-text data (Audio, video)
- Handling of dynamic content
- Handling of Authentication. (e.g. Login required Pages)

Non-functional

1. Fault tolerance to handle failures gracefully & resume Crawling without losing progress
2. Politeness to adhere to robots.txt & not overloaded website servers inappropriately

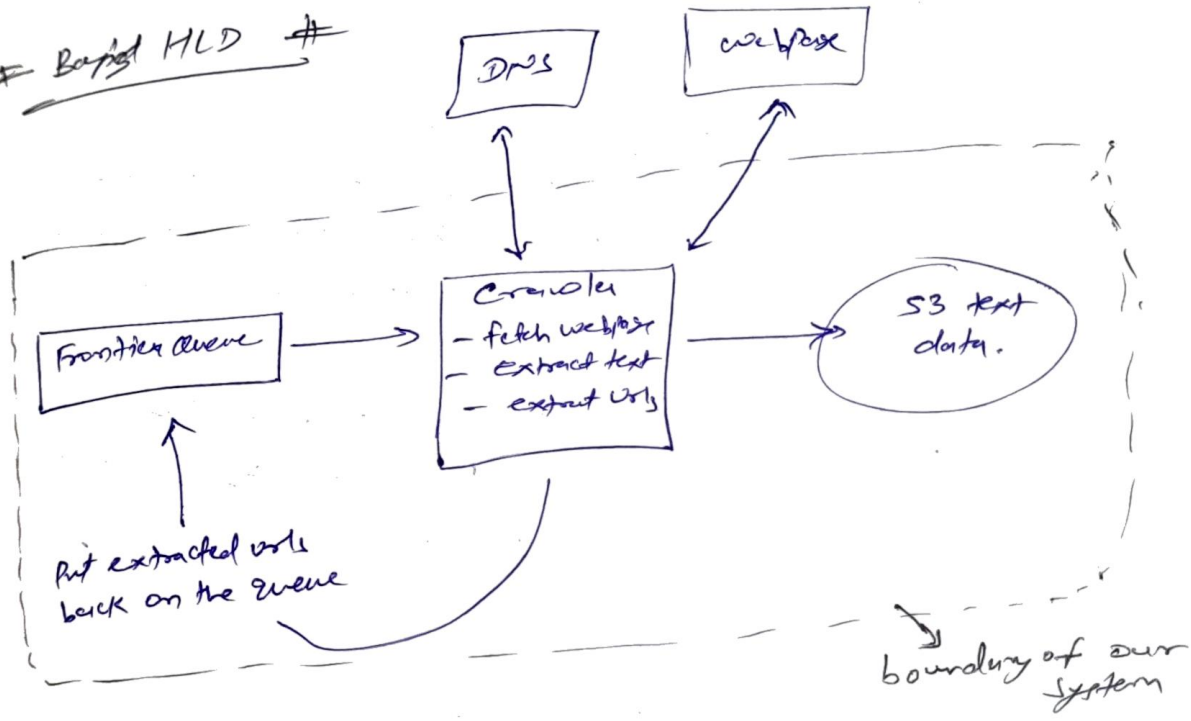
(3) Efficiency to crawl the web in under 5 day.

(4) Scalability to handle 10B pages.

Out of Scope

- Security to protect system from malicious actor
- Cost to operate the system within budget constraint
- Compliance to adhere to legal requirement & privacy regulations.

Basic HLD



Potential Deep Dives

(1) How can we ensure we are fault tolerant & don't lose progress?

Currently Crawler Service is doing many things
- hitting DNS
- Fetching web Pages ⇒ (most likely to fail)
- extracting text data
- extracting new urls to add to the frontier queue.

⇓

This is not fault tolerance. If there is a failure in any single task, all Progress will be lost.

— we should break Crawler service into smaller, pipelined stages:—

(a) URL fetcher → fetch HTML of web pages from external server. If there is failure, we can retry without losing Progress on the rest of data. (storing data in blob storage)

(b) Text & URL Extraction → Extract text data from HTML, any linked url to add to frontier queue.

⇒ what about if we fail to fetch a URL?

(i) Bad soln ⇒ In memory Timer (This isn't robust. if crawler goes down, we will lose timers. Also it will not likely to work in few more seconds need some sort of exponential backoff

(ii) Good soln ⇒ Kafka with manual exponential backoff

(Complex to implement & maintain)

↳ Separate topic for failed urls & separate service that reads from this topic & retries the fetch with exponential backoff.

(iii) Unreachable - SQS with exponential backoff

↳ SQS support retries with configurable exponential backoff out of the box. (No need to implement own logic)

↳ Retry attempt - 30sec, 2 min, 5 min, up to 15 min.

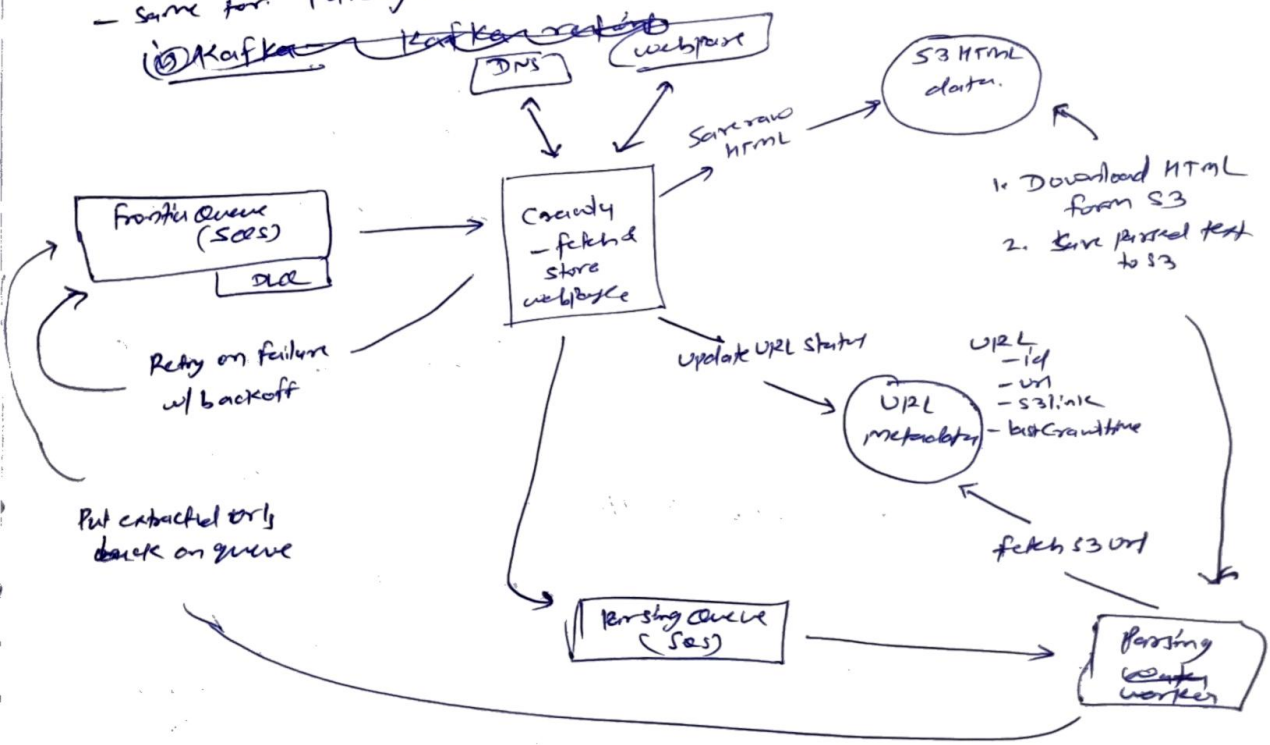
⇒ What if Crawler goes down?

Answer is simple :- we spin up a new one. We'll just have to make sure that the half-finished URL is not lost.

Good news is the URL will stay in queue until it is confirmed to have been fetch by a crawler & the HTML is stored in blob storage. This way if a crawler goes down, the URL will be picked up by another crawler & process will continue.

We can accomplish this in Apache Kafka & SQS both,

- same for Parsing workers goes down.



(2) How can we ensure politeness & adhere to robots.txt.

robots.txt → file that website use to communicate with web crawlers.

↳ similar to

```
User-agent: *
Disallow: /private/
Crawl-Delay: 10
```

→ Specifies how many seconds the crawler should wait b/w requests.

To ensure politeness & adhere to robots.txt, we will need to do two things:

(i) Respect robots.txt — Before crawling a page, we will need to check the robots.txt file to see if we are allowed to crawl the page.

(ii) Rate limiting — (implement a global, domain-specific rate limiting mechanism using a centralized data store (like Redis) to track request count per domain per second).

(3) How to scale to 10B pages & efficiently crawl them in under 5 days.

In AWS ecosystem, a network optimized instance can handle about 400 MBPS.

* 400 MBPS / 8 bits/byte / 2MB/page = 2500 pages/second
* utilizing 30% => 7500 pages/second
* 10B pages / 7500 pages/second = 1,333,333 second = 15.4 days for single machine

=> (4 machine needed)

for DNS → 1) DNS Caching
2) Multiple DNS Providers

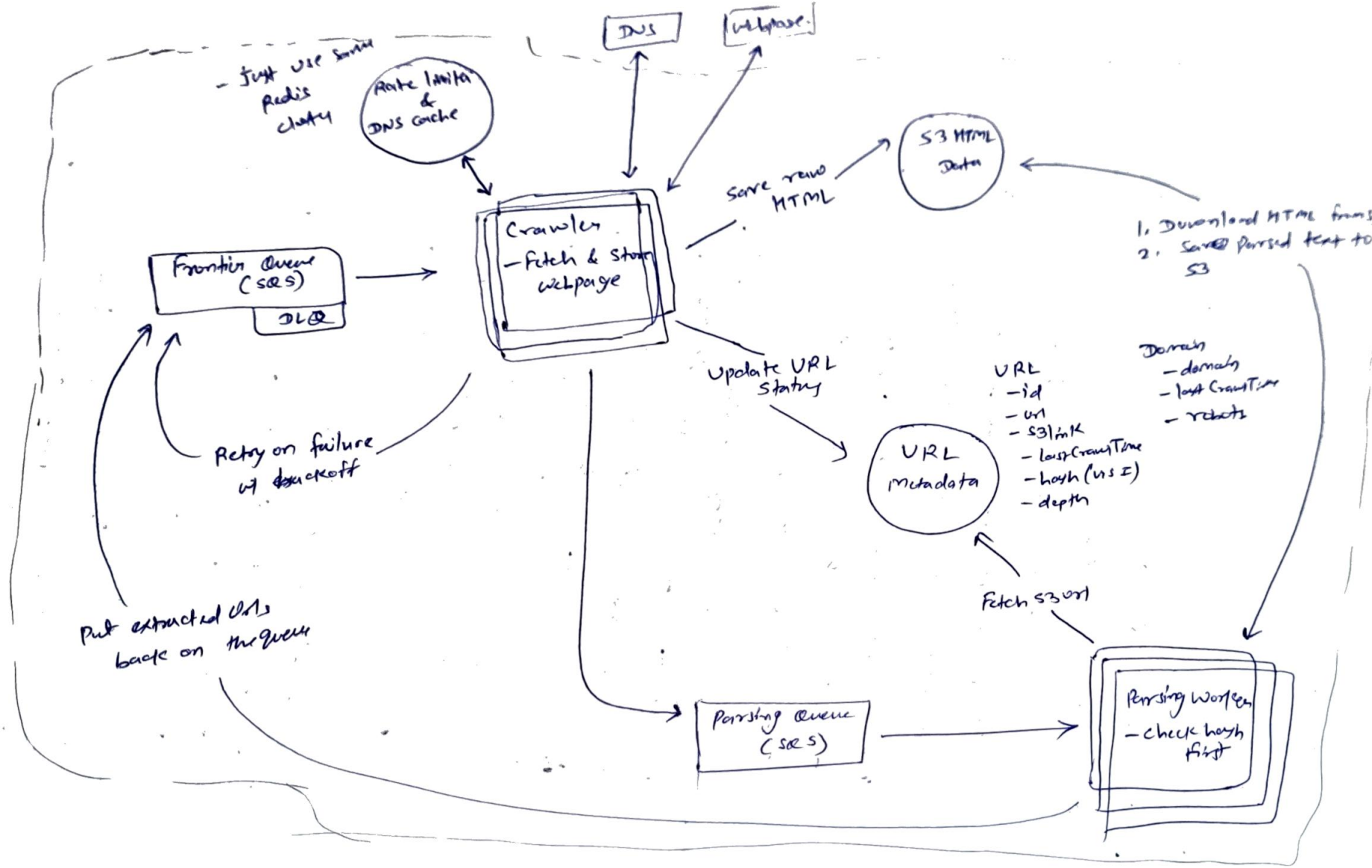
⇒ Last thing! watch out for Crawler traps —

* Crawler traps are pages that are designed to keep crawlers on the site indefinitely. They can be created by having a page that links to itself many times or by having a page that links to many other pages on site. If we're not careful, we could end up crawling the same site over & over again & never finish.

* Solution is pretty straightforwarded, we can implement a maximum depth for our crawler. we can add a 'depth' field to our URL table in Metadata DB & increment this field each time we follow a link. If the depth exceeds a certain threshold, we can stop crawling the page. This will prevent us from getting stuck in a crawler trap.

⇒ How to handle dynamic content?
↳ we'll need to use a headless browser like puppeteer to render the page & extract the component.

⇒ How to handle large files?
↳ we can use 'Content-Length' header to determine size of the file before downloading it & skip file that are too large


DNS
Webpage
- just use built Redis cluster
Rate limiter & DNS cache
S3 HTML Data
Save raw HTML
Crawler
- Fetch & store webpage
1. Download HTML from S3
2. Save parsed text to S3
Frontier Queue (SQS)
DLQ
Retry on failure w/ backoff
Update URL status
URL
- id
- url
- S3 link
- lastCrawlTime
- hash (use I)
- depth
URL Metadata
Domain
- domain
- last CrawlTime
- robots
Put extracted URLs back on the queue
Fetch S3 url
Parsing Queue (SQS)
Parsing Workers
- check hash first

Design an Online Auction Platform Like eBay

Intro :- An online auction service lets users list items for sale while others compete to purchase them by placing increasingly higher bids until the auction ends, with the highest bidder winning the item.

Functional Req. :-

1. Users should be able to post/put an item up for auction with a starting price & end date.
2. Bid on auctions by user
3. view an auction & current highest bid (/ max bid)

Out of scope

- Search for items
- Filter items by Category
- Sort items by Price
- view the auction history of an item.

Non-Functional Req :-

1. System should maintain strong consistency for bids to ensure all users see the same highest bid.
2. Fault tolerant & durable. We can't drop any bids

(3) Real-time max bid display on client.
(4) scale to 10M concurrent auctions.

out of scope

- system should have proper observability & monitoring.
- secure & protect user data.
- system should be well tested & easy to deploy (CI/CD pipeling)

Entity :-

(1) Auction :- start price, end date, item being auctioned
(2) Item :- name, description, image
(3) Bid :- amount bid, user placing the bid, the auction being bid on
(4) User - who start auction or bids on an auction.

API / System Interface

(1) POST /auctions → Auction & Item

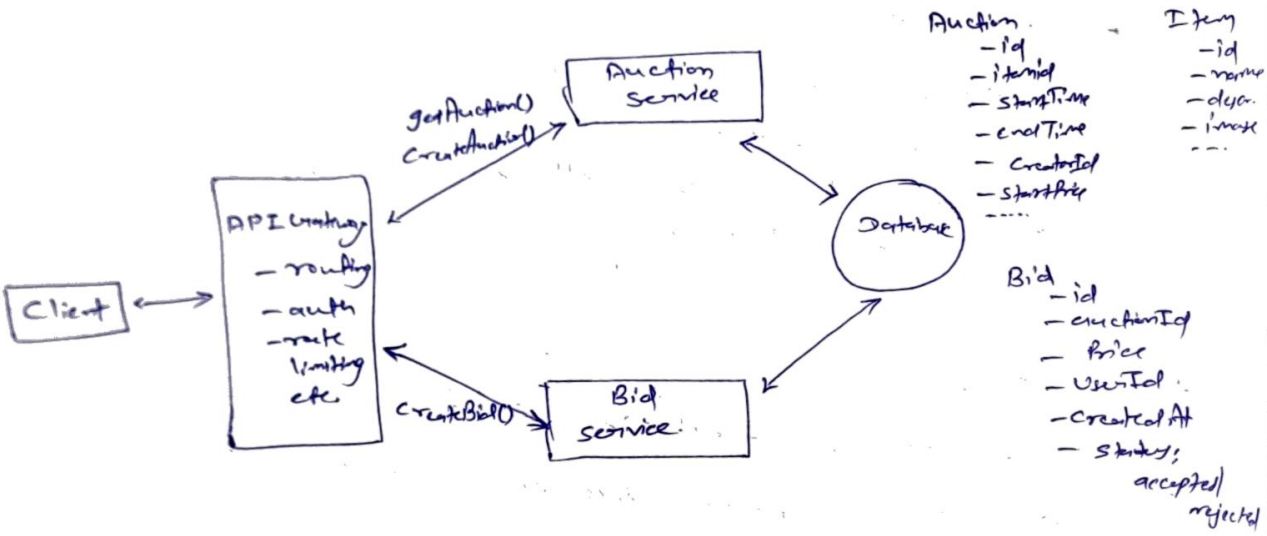
```
{ item: Item,
  startDate: Date,
  endDate: Date,
  startingPrice: number
}
```

(2) POST /auction/:auctionId/bid → Bid

```
{ Bid
}
```

(3) GET /auctions/:auctionId → Auction & Item.

High Level Design



* why two services:-
  - Bidding Traffic is typically much higher volume than auction creation - we expect ~100x more bids than auction. Having a separate service allows us to scale the bidding infrastructure independently.

* Bidding service will:-
  - validate incoming bid (Ex:- check bid amount is higher than current max)
  - Update the auction with new highest bid.
  - Store bid history in the database
  - Notify relevant parties of bid updates.

=> Let's assume user request for data for an auction with current max bid amount. what happens next is more interesting:-

- If we never refrat maximum bid price, then the user will bid based on a stale amount & be confused (& frustrated) when they are

told their bid was not accepted. Especially in an auction with a lot of activity, this is a problem.

To solve this, we can simply poll for the latest maximum bid price every few seconds. While imperfect, this ensures at least some degree of consistency & reduces the likelihood of a user being told their bid was not accepted when it actually was.

Potentially deep dives

(1) How can we ensure strong consistency for bids? ?

(a) Bad soln — Row Level Locking with Bids query :-

*approach:
- Begin a transaction
- Lock all bids for auction using select... for update
- query the current maximum bid from the bids table
- Compare the new bid against it
- write the new bid if accepted
- commit the transaction

*challenges:
- Performance & scalability issue.
  - ↳ Locking all bid rows for an auction serializes bid processing, creating a major bottleneck
- Poor user Experience
  - ↳ Delay introduced by lock contention & serialized processing result in slow response time or timeouts when placing bids.

(b) Greatsoil - Cache max bid externally.

↳ Problem with above soln (a) is lock on large no. of bid rows.

Solution

↳ we can cache the max bid in memory in something like Redis.

Flow :-
* Read cache (for max bid)
* Update cache (with new bid)
* write bid (write bid to db with status of accepted/rejected).

Issues
- Cache-DB inconsistency (Cache update succeeds but db write fails, leading to inconsistent state.)
- Rollback Complexity - (Rolling back cache after DB failure adds operational complexity & potential race conditions)

(c) Greatsoil - Cache max bid in database

↳ storing max bid in the database.

New flow :-
- lock auction row for given auction (just one row)
- Read max bid for given auction from Auction table.
- write bid to database with status of accepted or rejected
- Update max bid in Auction table if new bid is higher
- Commit transaction.

↳ we now only lock a single row & for a short duration

(2) How can we ensure that system is fault-tolerant & durable?

↳ we need to guarantee durability & ensure that all bids are recorded & processed, even in the face of system failure.

The best approach here is to introduce a durable message queue & get bids into it as soon as possible. This offers several benifits:—

a) Durable storage

b) Buffering Against Load Spikes

↳ Imagine popular auction

↓

thousands of bids per second

↓

Without queue → Drop bids → unacceptable,
→ Crash under the load
→ massively overprovision our servers (expensive)

c) Guaranteed ordering

↳ This is important for fairness - if two users bid the same amount, we want to ensure the first bid wins, The queue gives us this ordering for free.

⇒ Kafka is suitable here because -

- High throughput
- Durability
- Partitioning (by auctionId)

(3) How can we ensure that the system displays the current highest bid in real-time?

⇒ Bad sol^n – Polling (too slow, inefficient)

⇒ Good sol^n – long polling for max bid

⇒ Great sol^n – Server-Sent Events (SSE)

↳ when a user views an auction, their browser establishes a single SSE connection. The server can then push new maximum bid values through this connection whenever they change. This creates a true real-time experience.

↳ on server side, we maintain a set of active SSE connections for each auction. when a new bid is accepted, we push the updated maximum bid to all connected clients for that auction.

Challenge – Scaling issue when multiple servers come to picture.

∴ we will talk about solution in next deep dive

(4) How can we ensure that system scales to support 10M concurrent auctions?

– while it comes to discussing scale, you will typically want to follow a similar process for every system design question, working left to right, evaluate each component in your design asking the following questions: –

a) what are the resource requirement at peak? Consider storage, compute & bandwidth.

b) Does the current design satisfy the requirement at peak?

c) If not, how can we scale the component to meet the new requirement?

Start with same assumptions => 10M concurrent auctions, 100 bids.

↓

1 Billion bids/day

↓

10K bids per second

⇒ Message Queue

↳ decent hardware can handle 10K request per second.

↳ No issues here.

⇒ Bid service

↳ In this case with almost all stateless service, we can horizontally scale.

⇒ Database :-

- Auction = 1KB
- Bid = 500 bytes
- Avg auction runs for a week

10M × 52 = 520M auctions per year

520M * (1KB + (0.5KB + 100))

= 25 TB of storage per year

↓

modern SSDs can handle 100 TB of storage, no issue here.

⇒ SSE :-

↳ issue - All connections won't fit on same server. so we need a way for Bid service servers to coordinate with one another.

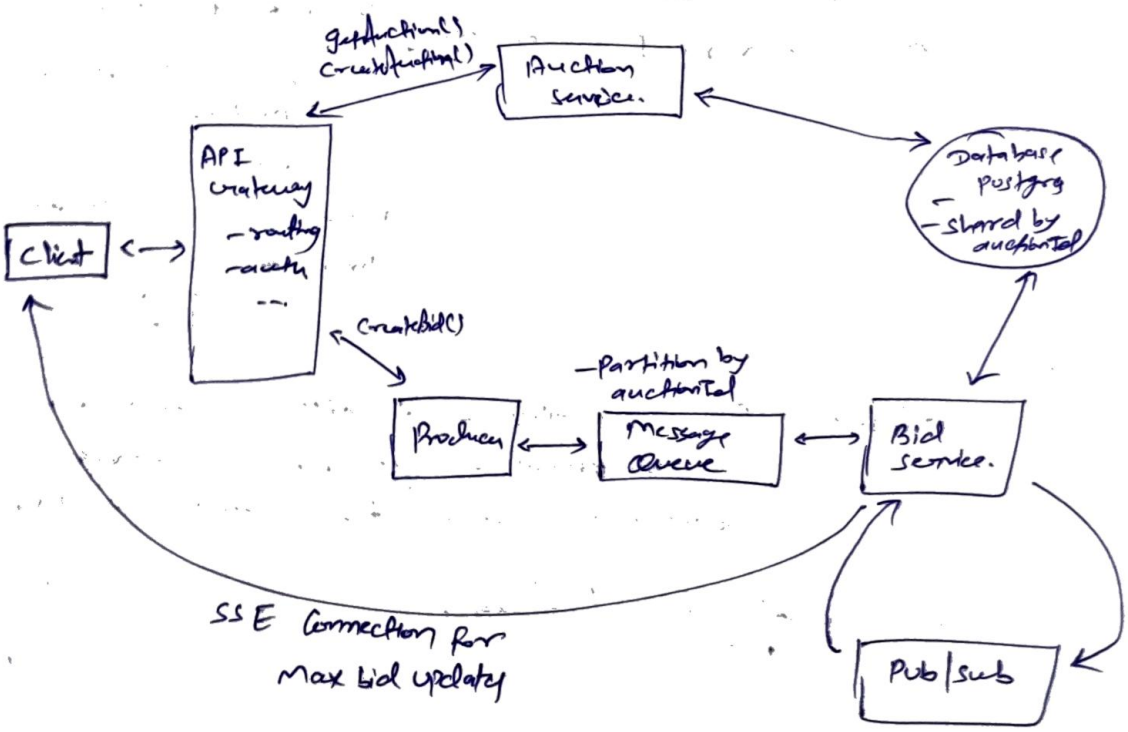
The solution is Pub/Sub.

↓

↓

whether using a technology like Redis Pub/sub, Kafka, or even a custom solution, we need to broadcast the new bid to all relevant servers so they can push the update to their connected client.

The way this works is simple: when server A receives a new bid, it publishes that bid to the Pub/sub system, essentially putting it on a queue. All other instances of Bid service are subscribed to same pub/sub channel & receive the message. If the bid for an auction that one of their current client connections is subscribed to, they send that connection the new bid data.



AutoComplete | Word Completion

— AutoComplete is involved in many search features ~~in many search~~ features on various platform like facebook, Instagram, Google search, etc.

For facebook — one usecase which involves keeping user's friends, mutual connections in mind while giving suggestions in the typeahead.

Functional Req. :-

(1) ~~user~~ suggestions should be displayed on type of user.

(2) It is supposed to be real-time @. Let's say if currently there is an India vs Pakistan match going on, then ~~should~~ our autocomplete should include this as a suggestion in the typeahead if searching "India vs Australia" is a trending query.

if search query is in trending then add it to db so that include it in autocomplete suggestions

(3) Rank our suggestions (based on frequency) ↓ no of times searched by our users

~~of 4 billion Qps~~

— 4 Billion ~~search~~ query per day.

— we need 7 autocomplete suggestion for each character we enter in the search box.

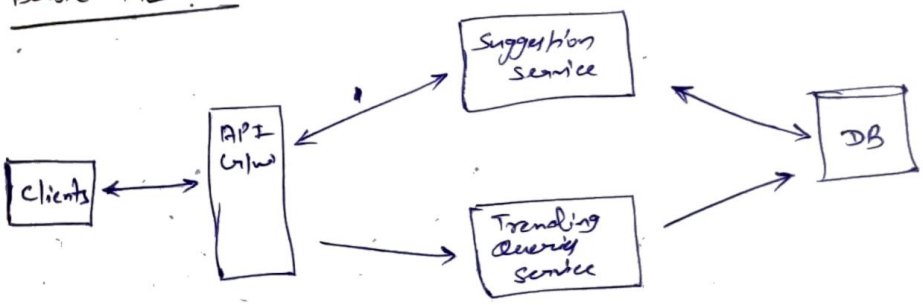
Non-functional Req :-

1. Latency of suggestions must be very less.
2. Handle large amount of data. (Partition tolerance)
3. Highly available
4. Eventual consistency is fine.
5. search results must be ordered by some ranking scores
6. The results must be relevant to whatever prefix the user has entered.

System APIs :-

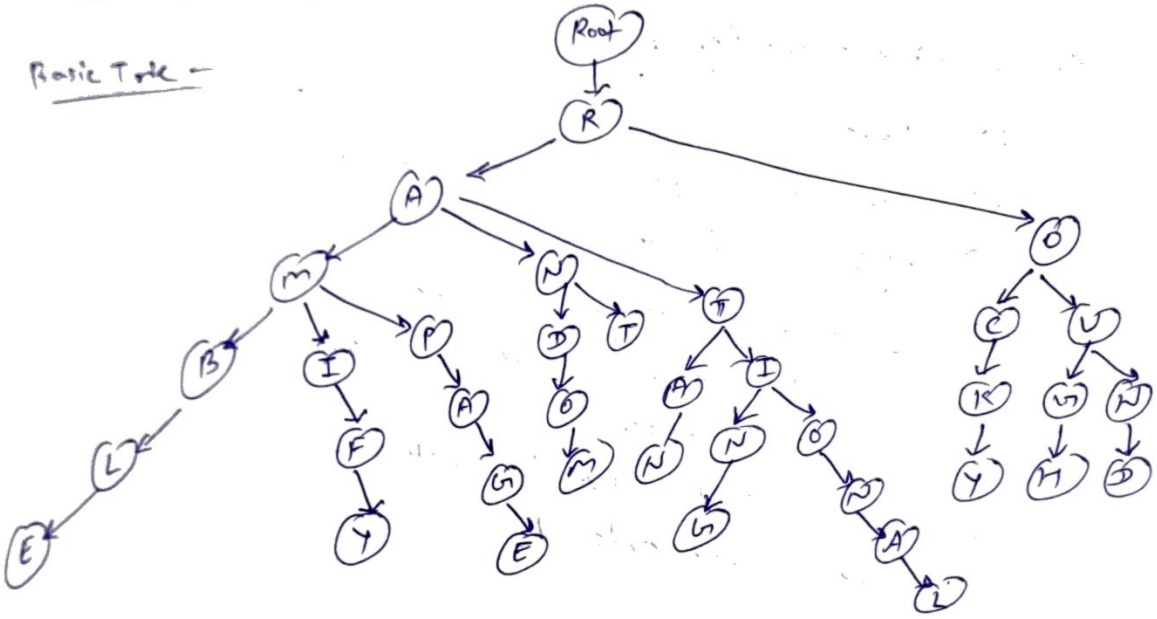
1. get_suggestions (Prefix)
2. add_to_db (query) — a new unique trending query which has been searched above a certain threshold will be stored in our DB

Basic HLD



⇒ Data structure to store search queries :-

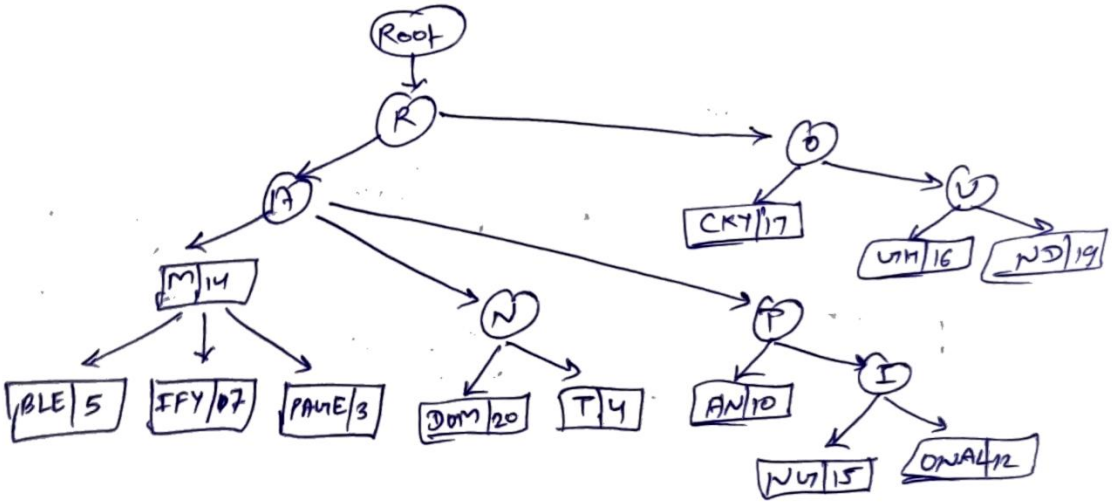
Basic Trie –



In order to optimize space, we can merge nodes that have only one branch. Now the trie will be stored like: —

This is called a suffix tree. (Compressed trie of all the suffixes)

Further, we can store, the frequency of each search results in the trie itself as follows :-



Time Complexity to go to the Prefix node – O(L) → L → length of Prefix

Let's say we entered 'RAT' – all possible suggestions would be found by going through the entire subtree under RAT & all nodes which have frequency, i.e. which are complete words will be fetched

from RAT we get -- RATAN-10, RATIONAL-12, RATINUM-15

- Time Complexity to find all possible nodes under the Prefix node i.e. Scanning the entire subtree - $O(N)$
  - N → number of nodes under the Prefix node.
- Then we have to sort it in descending order so that the word with the highest frequency is shown at the top.
- then we get suggestion in the following order for RAT - RATINUM-15, RATIONAL-12, RATAN-10
  - Time Complexity to sort output - $O(K \log K)$
    - K → no. of top suggestions.

⇒ total Time Complexity = $O(L) + O(N) + O(K \log K)$

⇒ with this time Complexity → latency would be big.

⇓

we need to do better

⇓

we can precompute all the possible top suggestions for each & every prefix node in trie data structure.

↓

Now the time Complexity to find suggestions for a given prefix would be - $O(L)$, where L is length of the prefix entered.

How can we do this? → Store in Hash table

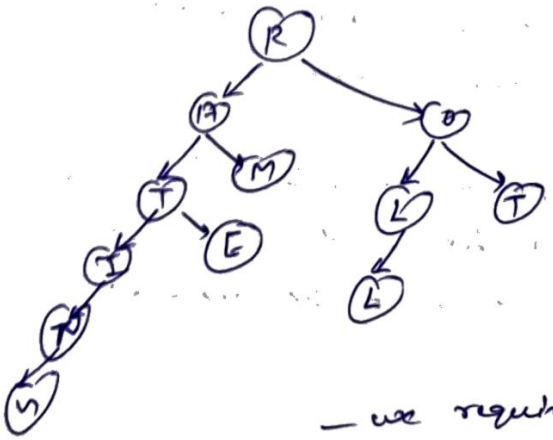
⇓

Key → Prefix
Value → Suggestions

↘ (Redis)

Prefix	Suggestions
R	{RANDOM, ROUND, ROCKY} 20 19 17
RA	{RANDOM, RATING, RAM} 20 15 14
RAM	{RAM, RAMIFY, RAMBLE} 14 7 5
RAN	{RANDOM, RANT} 20 4
RAT	{RATING, RATIONAL, RATAN} 15 12 10
RO	{ROCKY {ROUND, ROCKY, ROUND} 19 17 16
ROC	{ROCKY 17
ROUN	{ROUND 16

Storage → Storing entire trie in database can be done by taking snapshot of trie.



⇒ "R 2, A 2, T 2, I, N, M, E, M, O 2, L, L, T"

↓

from this we can easily rebuild our trie.

- we require partition tolerance, Availability & scability, hence, noSql databases like Cassandra can be used.

⇒ Traffic & storage estimation —

(1) Traffic estimation —

* 4B queries/day
* each query consists 3 words (words → 5 letters)

total read requests in one query = 3 * 5 request/query

total read request in one day = 3 * 5 * 4B = 60B queries

Read Req Per Second = $60 \times 10^9 / 10^5$ (sec.) = 60K req./sec.

(2) Storage estimation —

* 2 byte to store a character, total 30 byte for 15 char

* 20% of 4B queries to be unique.
  ↓
  800M * 30 byte = 24 TB need to store.

- Assume 10% new search queries generated every day.
- 10% of 24 TB = 2.4 TB
- For a year = 2.4 * 365 = 876 TB
- 10 year => 8.8 TB of data in 10 years.

(1)

⇒ Scaling trie database —

(2) * will the trie data fit in one server?,
  ↓
  initially 24 TB can fit in a server. but later may not because of increasing new search queries
  ↓
  need more db servers with trie data stored in them.
  Lets call these db servers "trie servers".

=> Sharding - sharding trie data, but how?

(approach 1) - dedicate trie for every alphabet

↓

A trie server for queries starting with A,
another " " " " " B,

Problem - May lead to an imbalance in load. Many query may start with A but very few will start from x.

(approach 2) - we can add another server when one fills & the split the one trie. whenever space on the server gets occupied, we start using the next trie server.

for eg:- one server gets filled from a-abd, then next server can store from abe-cdr. & so on.

If user type A → it will hit two servers

↓

aggregate results & return top 7 results to user.

Problem:- Can still lead to an unbalanced load.

(approach 3) - Hash the words & (hash%, number of shards) will give server number. This way queries will be stored randomly. We can introduce Consistent hashing

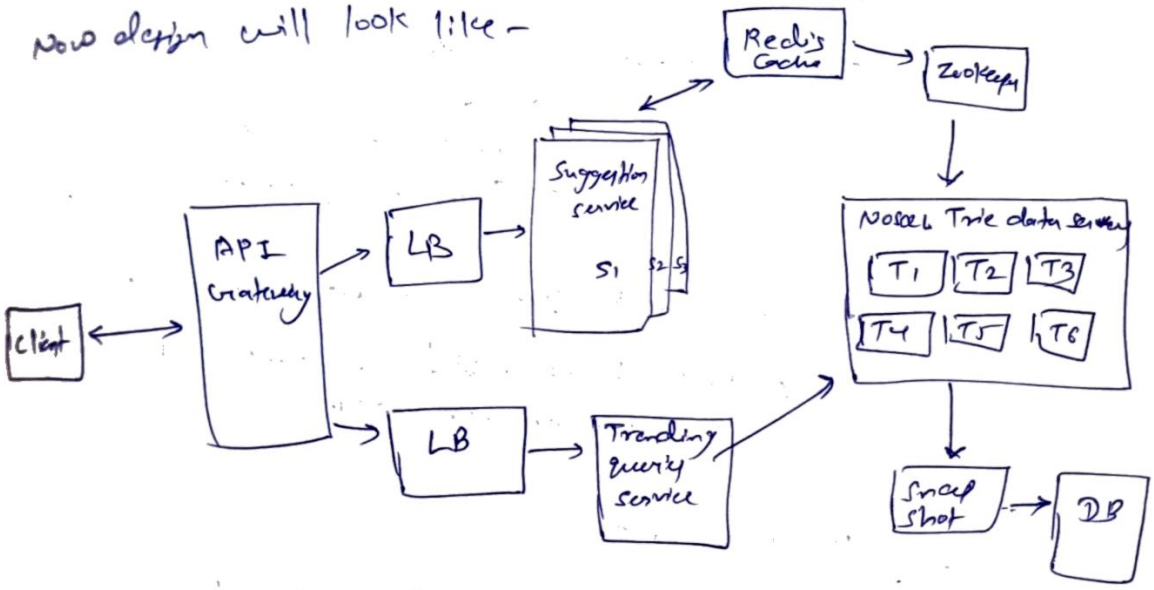
Problem - We have balanced trie server, but we will need to aggregate the search results from each trie server. & then again sort results & provide it to client.

so here, solution 2 seems better. Aggregating from a few servers is better than aggregating results from each server. Hence we will use a mapping to see which server stores which range of prefix.

- we can use (Zookeeper).
  - Zookeeper is very highly available & it's very good with lots of reads & some writes.

⇒ Also we can cache popular searched prefixes.

Now design will look like -



② How will the Trending Queries service work?

It collects popular or recent queries & if the frequency of these new queries is more than threeshold, it is stored in our trie servers.

⇒ log will look like

Query	Time	weight
RAM	6 July 12 PM to 1 AM	173848
RAMIFY	6 July 1 PM to 2 PM	34869
RATION	6 July 2 to 3 PM	256488
ROUND	5th July	8234

workflow -

(Step 1) Collect Query logs

* Queries are logged hourly with weights based on frequency & recency.
* Older & low-frequency queries are discarded.
* Daily aggregates are created for long-term efficiency.

(Step 2) MapReduce + Aggregator

* "MapReduce jobs" run hourly to compute frequency scores.
* Aggregator service combines:-
  - Old rank score
  - New frequency score
  - Time decay factor (recent queries favored)
* Queries with low score are discarded.
* Output: "Rank table" containing trending queries with scores.

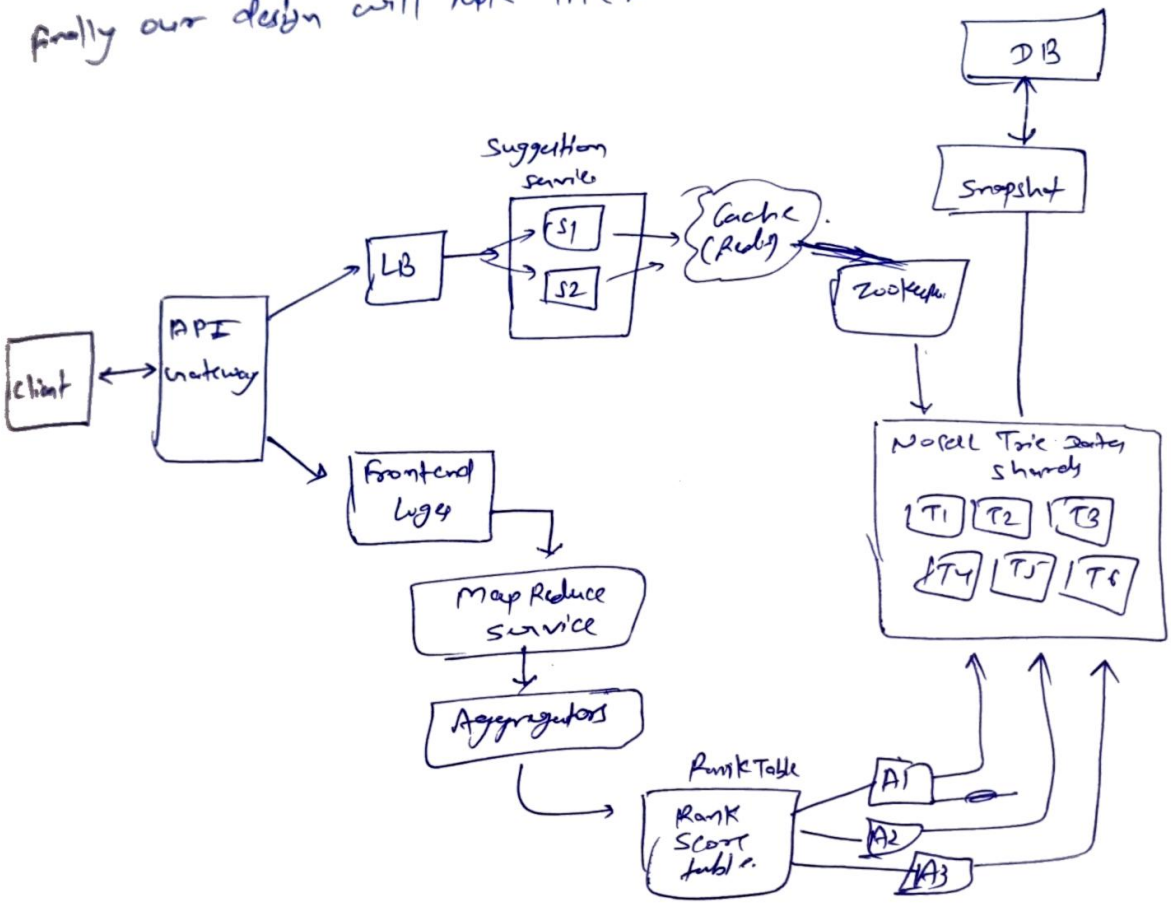
(Step 3) Apply to Trie

* Specialized components called 'Appliers' update trie structures:-
  - A1 → queries from [a-f]
  - A2 → [fs-o1]
  - A3 → [om-z]
* Each Appliers:-
  - Uses Rank table to assign weights
  - Builds internal trie with top-k suggestions per node.
  - Dumps into corresponding trie servers.

Step 4) Update trie servers -

* Done Offline using master-slave configuration -
  - Slave is updated while master serves traffic.
  - After update, slave becomes the new master.

finally our design will look like -



(3) Incorporating user's historical search queries :-

while recommending two queries are fired

one to our server

another one locally to get recent search queries stored in our local system cache.

Combination

4 + 3 => 7 top 7

Design a Distributed Job Scheduler like Airflow

What is a job scheduler :-

A job scheduler is a program that automatically schedules & executes jobs at specified times or intervals. It is used to automate repetitive tasks, run scheduled maintenance, or execute batch processes.

Key Terms :-

(A) **Task :-** A task is the abstract concept of work to be done. For example, "send an email". Tasks are reusable & can be executed multiple times by different jobs.

(B) **Job :-** A job is an instance of a task. It is made up of task to be executed, the schedule for when the task should be executed, & parameters needed to execute the task.

The main responsibility of a job scheduler is to take a set of jobs & execute them according to the schedule.

Functional Req :-

(1) User should be able to schedule jobs to be executed immediately, at a future date, or on a recurring schedule (i.e. every day at 10:00AM).

(2) Users should be able to monitor the status of their jobs.

Out of scope

- Cancel or reschedule jobs

Non Functional Req :-

Core Requirements:-

1. Availability > Consistency
2. System should execute jobs within 2s of their scheduled time.
3. Scalable to support up to 10K jobs per second
4. The system ensure at-least-once execution of jobs.

out of scope

- enforce security policy
- CI/CD pipeline.

Core Entities:-

1. Task
2. Job
3. Schedule
4. User

APIs :-

1. POST /jobs

```
{ "task_id": "send_email",
  "schedule": '0 10 * * *',
  "parameter": {
      "to": "john@example.com",
      "subject": "Daily report"
  }
}
```

2. GET /jobs ? userId={id} & status={status} & startTime={startTime} & endTime={endTime}

Data Flow

1. A user schedules a job by providing the task to be executed, schedule for when the task should be executed, & the parameters needed to execute the task
2. The job is persisted in the system.
3. The job is picked up by a worker & executed at the scheduled time.
   - if the job fails, it is retried with exponential backoff
4. Update the job status in the status

High level design

① Users should be able to schedule jobs to be executed immediately, at a future date, or on a recurring schedule. :-

we will have 2 tables:-

(a) Job table

```
{ "job_id": "123e4567-e89b"   // Partition key for easy lookup by job_id
  "user_id": "user_123",
  "task_id": "sendemail",
  "schedule": {
        "type": "CRON" | "DATE"
        "expression": "0 10 * * *"   // Every date at 10:00AM for cron, specific date for DATE
  },
  "parameters": {
        "to": "john@example.com",
        "subject": "Daily Report"
  }
}
```

(b) Executions table track each individual time a job should run:-

```
{ "time_bucket": 1715547600,   // Partition key (Unix timestamp rounded down to hour)
  "execution_time": "1715548800-123e4567-e89b",
  "job_id": "user_123",
  "status": "PENDING",
  "attempt": 0
}
```

By using time_bucket as Partition key → achieved efficient Querying while avoiding hot Partition issues

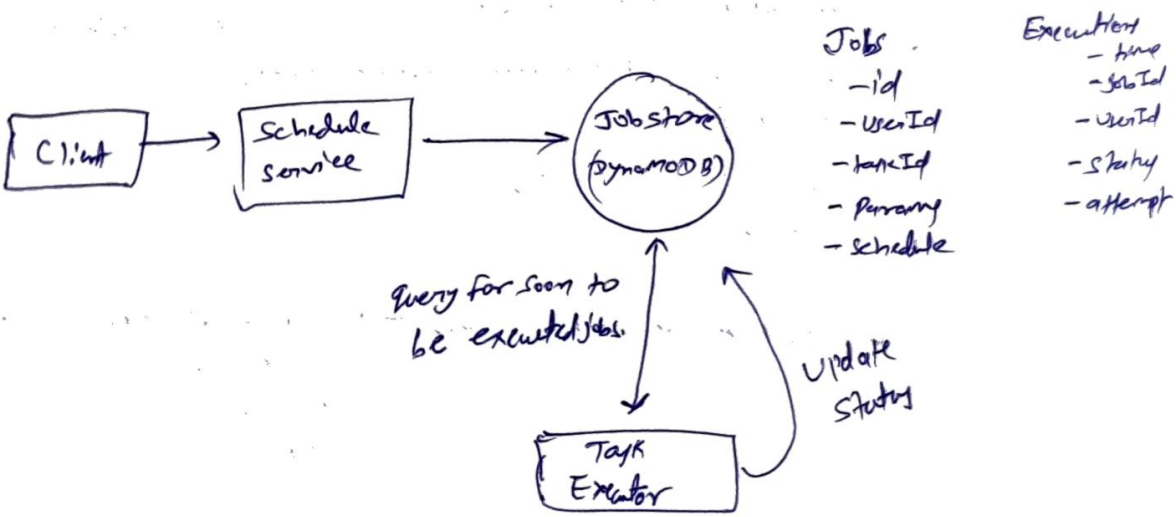
for example, to find jobs that need to run in the next few minutes, we only need to query the current hour's bucket & possibly the next hour's bucket. The time bucket can be easily calculated:-

time_bucket = (executionTime // 3600) * 3600

Round down to nearest hour

* First, by querying only 1-2 partitions to find upcoming jobs, we maintain efficient read operations. The hourly bucket ensure jobs are well-distributed across partitions, preventing hotspots.
* Second, when a recurring job completes, we can easily schedule its next occurence by calculating the next execution time & creating a new entry in the "Executions" table. The job definition stays the same, but we keep creating new execution instances.

⇒ when a worker node is ready to execute jobs, it simply queries the Executions table for entries where:-
* 'executionTime' is within the next few minutes,
* 'Status' is PENDING.



(2) Users should be able to monitor the status of their jobs! :-

when a job is executed → need to update status on execution table.

COMPLETED, FAILED, IN_PROGRESS, RETRYING

But how can we query for status of all jobs for a given user?

with current design ⇒ two-step process

1) find all jobs id for user from Jobs table.
2) query Execution table to find status of each job.

To solve this, we'll add a Global Secondary Index (GSI) on the Execution table:-

* Partition key : user_id
* Sort key: execution_time + job_id

This GSI allows us to efficiently find all executions for a user, sort them by execution time, support pagination, & filter by status if needed.

Potential Deep Dive

(1) How can we ensure the system executes jobs within 2 sec. of their scheduled time?

⇒ To ensure jobs are executed with 2 seconds of their scheduled time, we must move away from our current model of querying the database every few minutes. Frequent querying (e.g. every 2 seconds) to

meet this timing precision would cause excessive database load & latency due to large payloads, making it impractical.

Instead, a more efficient solution is a two-layered scheduler architecture: (1) periodically query the database (e.g. every 5 minutes) for upcoming jobs and (2) push these jobs into a message queue ordered by execution time. workers then pull jobs from this queue for timely execution.

⇒ This design reduces database strain while leveraging the message queue's ordering & throughput to achieve sub-second execution precision. However, handling newly created jobs scheduled within the next few minutes presents a challenge, as they could be missed by the periodic database query.

~~Direct~~ Directly inserting these jobs into the queue doesn't help either due to FIFO constraints in system like kafka. To address this, the queue system must support dynamic prioritization based on execution time, allowing late-arriving jobs to be inserted in the correct position & executed on schedule.

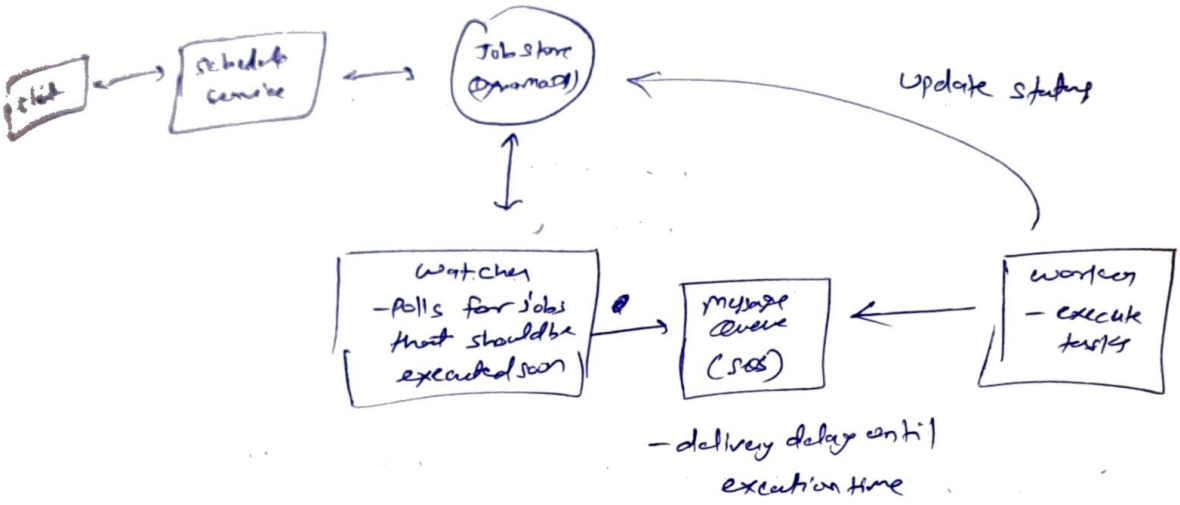
⇒ Usual soln:- Redis Sorted set (issue → operation complexity in distributed system
→ implement own Retry logic

⇒ Usual soln - RabbitMQ [issue → less reliable at scale.

⇒ Usual soln:- Amazon SQS ⇒ (native support for delayed message delivery.

For eg schedule a job in 10 sec. ⇒ Send message to SQS with a delay of 10 sec ⇒ SQS will deliver msg to worker after delay. Easy as that.

Here we implement two layered scheduler architecture -



(2) How can we ensure the system is ~~support~~ scalable to support up to 10K jobs per second?

- we can go left to right in design then look any bottlenecks & addressing them one-by-one.

(a) Job creation — to handle high job create rate
↓
introduce message queue. (allow to horizontally scale to scheduler service)

(b) Jobs DB -
↳ used DynamoDB with partition key optimally
↳ All set here.

(10K jobs/second 300 sec)

(c) Message Queue Capacity -
for each 5 min window = process around ~~3~~ 3 million jobs

- SQS ~~message~~ message can be upto 256 KB. (our message will be less 200 byte)
- total 600 MB data per 5 minute window
- default ~~limit~~ limit = 3000 message/second (Amazon SQS)
↕ AWS can increase quota
~~Can increase after speci~~

(d) workers —

two main choice of compute options:—

- Containers
- Lambda functions

→ Containers with ECS & auto-scaling groups, can be used as containers give us the best balance of cost efficiency & operational simplicity while meeting our performance needs.

(3) How do we ensure at-least-once execution of jobs?

Jobs can fail within a worker for one of two reasons:—

* Visible failure:— most likely bug in code or incorrect input parameter.

After retried 3 times again failed then status mark as FAILED

* Invisible failure:— most likely worker itself went down.

Bad way ⇒ Health check points

Good way ⇒ Job leasing

correct way ⇒ SQS visibility timeout (built-in mechanism)

when a worker receives a message from the queue, SQS automatically make that message invisible to other workers for a configurable period. The worker processes the message & delete it upon successful completion. If worker crashes or fails to process the message within the visibility timeout period, SQS automatically makes the message visible again for other workers to process.

— we can have relatively short visibility timeout (~30 sec).

⇒ Lastly, one consequence of at-least-once execution is that we need to ensure our task code is idempotent. In other words, running the task multiple times should have the same outcome as running it just once.

* ways to handle this:-

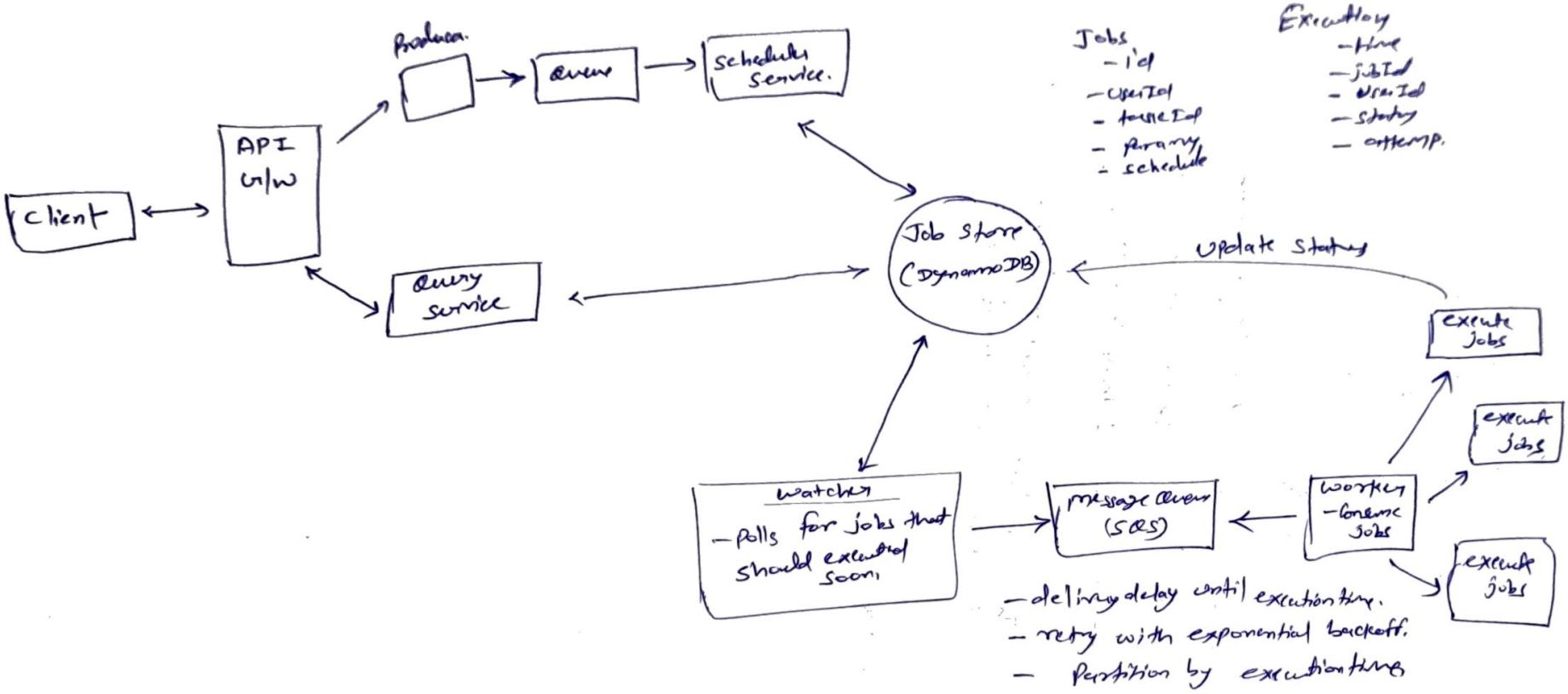
⇒ Bad soln - No idempotency controls

⇒ Good soln - Deduplication table

⇒ Great soln - Idempotent Job Design

This involves incorporating idempotency keys & using conditional or state-aware operations.

⇒ HLD is on next Page------


Jobs
- Id
- UserId
- targetId
- Priority
- schedule
Executions
- time
- JobId
- UserId
- Status
- attempt
Client
API G/W
Producer
Queue
Scheduler Service
Query service
Job store (DynamoDB)
Update Status
Execute Jobs
Execute Jobs
Execute Jobs
Watchers
- polls for jobs that should execute soon
Message Queue (SQS)
Workers
- Consume Jobs
- delivery delay until execution time.
- retry with exponential backoff.
- Partition by execution time

Design a Distributed Cache like Redis

A Distributed Cache is a system that stores data as key-value pairs in-memory across multiple machines in a network.

Functional Req :-

1. 'set, get, delete' key-value pairs
2. User should be able to configure expiration time for key-value pairs.
3. Data should be evicted according to LRU policy.

Below the line

- Users should be able to configure the Cache size

Non-Functional Req :-

1. highly available. Eventual consistency is acceptable.
2. Low latency operations (< 10ms for get/set operation)
3. Scalable to support the expected 1TB of data & 100K requests per second.

Out of Scope

- Durability
- Strong consistency guarantees
- Complex querying capabilities
- Transaction support

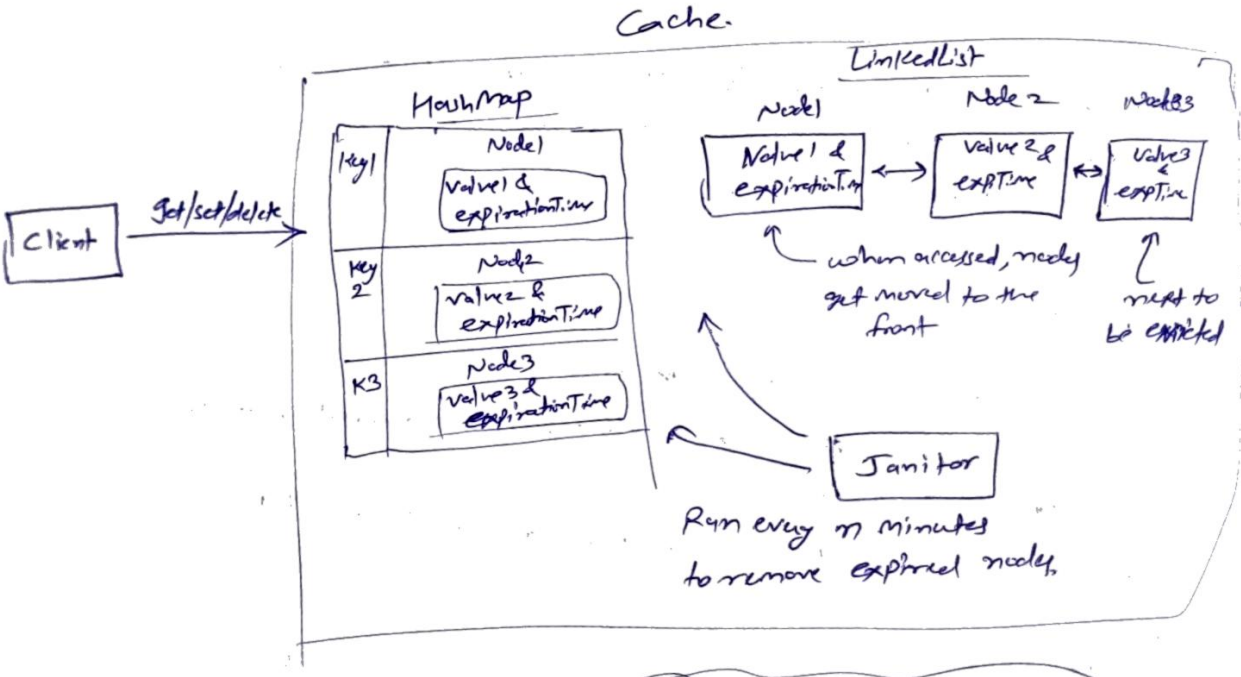
Entities

- Keys
- Value

APIs

1. POST /:key
   { "value": "..." }
2. GET /:key → { value: "..." }
3. DELETE /:key

Basic HLD



Data structure :- (Hashmap + Doubly Linked list)

Potential Deep Dive

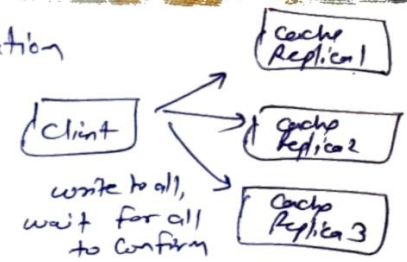
① How do we ensure our cache is highly available & fault-tolerant?

* Single node creak issue when node fails.

We need multiple copies of our data spread across different nodes. But this opens up a whole new set of operations:-

- How many copies should we keep?
- which nodes should store the copies?
- How do we keep the copies in sync?
- What happens when node fail or exit Communication?

⇒ Bad solⁿ — Synchronous replication

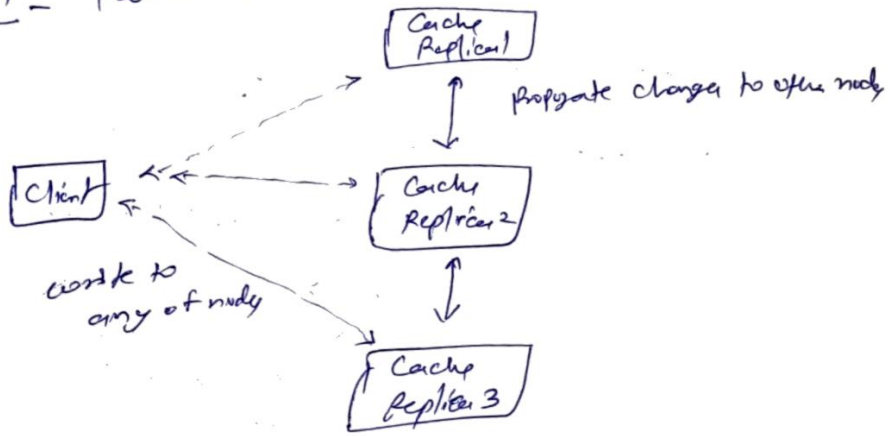


⇒ Good solⁿ — Asynchronous replication

(New primary node need to select if primary fails)

(Replicas may temporarily have stale data until changes fully propagate through system)

⇒ Great solⁿ — Peer-to-Peer replication (Using Gossip Protocol)



- maximum scalability / availability
- any node can read/write request sure

② How do we ensure our Cache is scalable?

↳ Need sharding

↓

How we ensure even distribution of keys across our nodes

↓

Consistent Hashing — hash(key) % n

(3) what happens if you have a hot key that is being read from a lot?

there are two distinct types of hot key problems we need to handle:-

a) Hot Reads - key that receive an extremely high volume of read requests

b) Hot writes - keys that receive many concurrent write requests

ways to handle Hot Reads -

a) Vertical Scalling all nodes (Bad soln)

b) Good soln :- Dedicated Hot key Cache

issue → Manage Consistency b/w main Cache & hot key Cache,
→ need monitoring to identify hot keys & mechanism to promote/demote keys b/w tiers.
→ operation overhead

c) Good soln - Read Replicas => create multiple copies of the same data across diff nodes.

issue → overhead in terms of storage & network bandwidth since entire nodes need to be replicated.

d) Great soln - Copies of Hot Keys :-

↳ unlike read replicas which copy entire nodes, this approach selectively copies only the specific key that are experiencing high read traffic

Here how it works :-

↓

* First, System monitors key access pattern to detect hot keys that are frequently read.

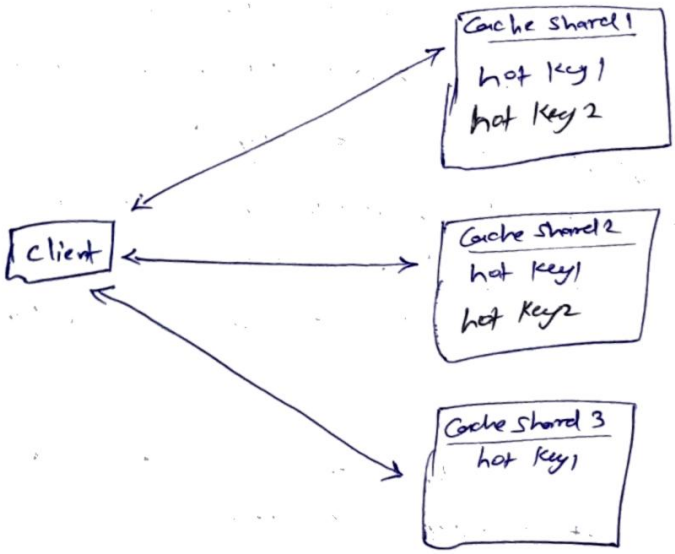
* When a key becomes "hot", instead of having just one copy as User:123, the system creates multiple copies with different suffixes:-

User:123#1 → Node A stores a copy
User:123#2 → Node B " " "
User:123#3 → Node C " " "

* These copies get distributed to different nodes via Consistent hashing.

* For reads, clients randomly choose one of the suffix keys, spreading read load across multiple keys.

* For writes, the system must update all copies to maintain consistency.



- overhead in monitoring to detect hot keys & managing lifecycle of copies - when to create them & when to remove them if a key is no longer hot. The approach works best when hot keys are primarily read-heavy with minimal writes.

(4) what happens if you have a hot key that is being written to a lot?

⇒ Great soln – write Batching

↳ Consider a viral video receiving 10,000 view update per second. Rather than executing 10,000 separate operation to set new value (eg views=1, views=2, etc), write batching might collect these updates for 100ms, then execute a single operation to set the final value 1,000 hits.

↓

it reduces the write pressure on cache node.

⇒ Great soln – Sharding Hot Key with suffixes.

- Instead of having a single counter or value that receives all writes, the system spreads write across multiple shards using a suffix strategy.

For eg., a hot counter key "views:video123" might be split into 10 shards: "views:video123:1" through "views:video123:10", when a write arrives, the system randomly selects one of these shards to update.

- this approach distributes write load across multiple nodes in cluster.

- while read, need to sum values from all shards

⇒ Trying it all together:-

Two data structures → hash table
→ linked list

when it comes to scaling, it depends on which of the above approaches you choose, as you can end up with a slightly different design. Assuming you opted for:-

1) Asynchronous replication for high availability & handling hot key reads.
2) Consistent hashing for sharding & routing.
3) Random suffixes for distributing hot key writes across nodes.
4) write batching & connection pooling for decreasing network latency/overhead.

Final design:-