~~Ticketing~~

Ticket master

FR:-

- User can
  - view event
  - search for events
  - book tickets to events

NFR

- Consistency for booking events.
- Scalable & high throughput to handle Popular events (10 million users, one event)
- low latency search (<500ms)
- Read heavy & they needs to be able to support high read throughput (100:1)

Below lines:-

(a) User should be able to view booked events

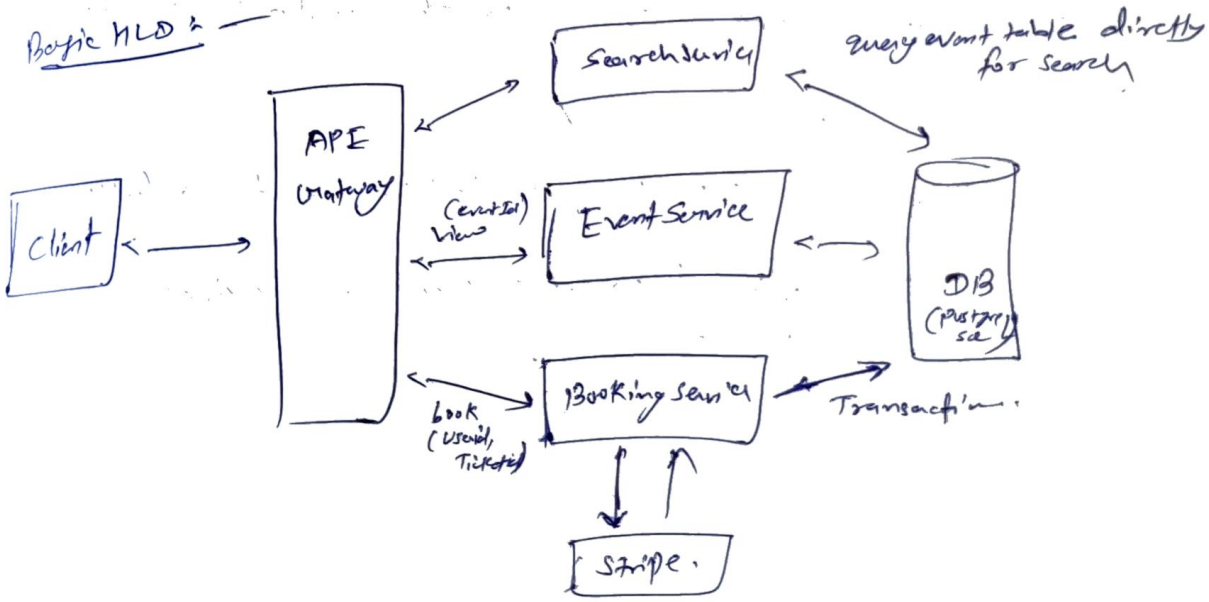
(b) Admin or Coordinator should be able to add events.

(c) Popular events should have dynamic pricing

Core Entities

Event, User, Performer, Venue, Ticket, Booking

Basic HLD :-



⇒ Potential deep dives :-

(1) How do we improve booking experience by reserving tickets: -

⇒ Bad soln - pessimistic Locking
⇒ Good soln - Status & Expiration time with Cron job
⇒ Great soln - Implicit status with Status & Expiration time
⇒ Great soln - Distributed Lock with TTL (Use Redis)

(acquire a lock in Redis using unique identifier i.e. TicketId) with Predefined TTL)

(2) How is the view API going to scale to support 10s of millions of concurrent requests during Popular events ?

⇒ Great soln - Caching, Load Balancing & Horizontal Scaling

(3) How will the system ensure a good User experience during high-demand events with millions simultaneously booking tickets?

⇒ Good soln - SSE for Real-Time seat updates

⇒ Great soln - Virtual waiting queue for extremely popular events

(4) How can you improve search to ensure we meet our low latency requirements?

⇒ Good soln - Indexing & SQL query optimization

⇒ Great soln - Full text indexes in the DB

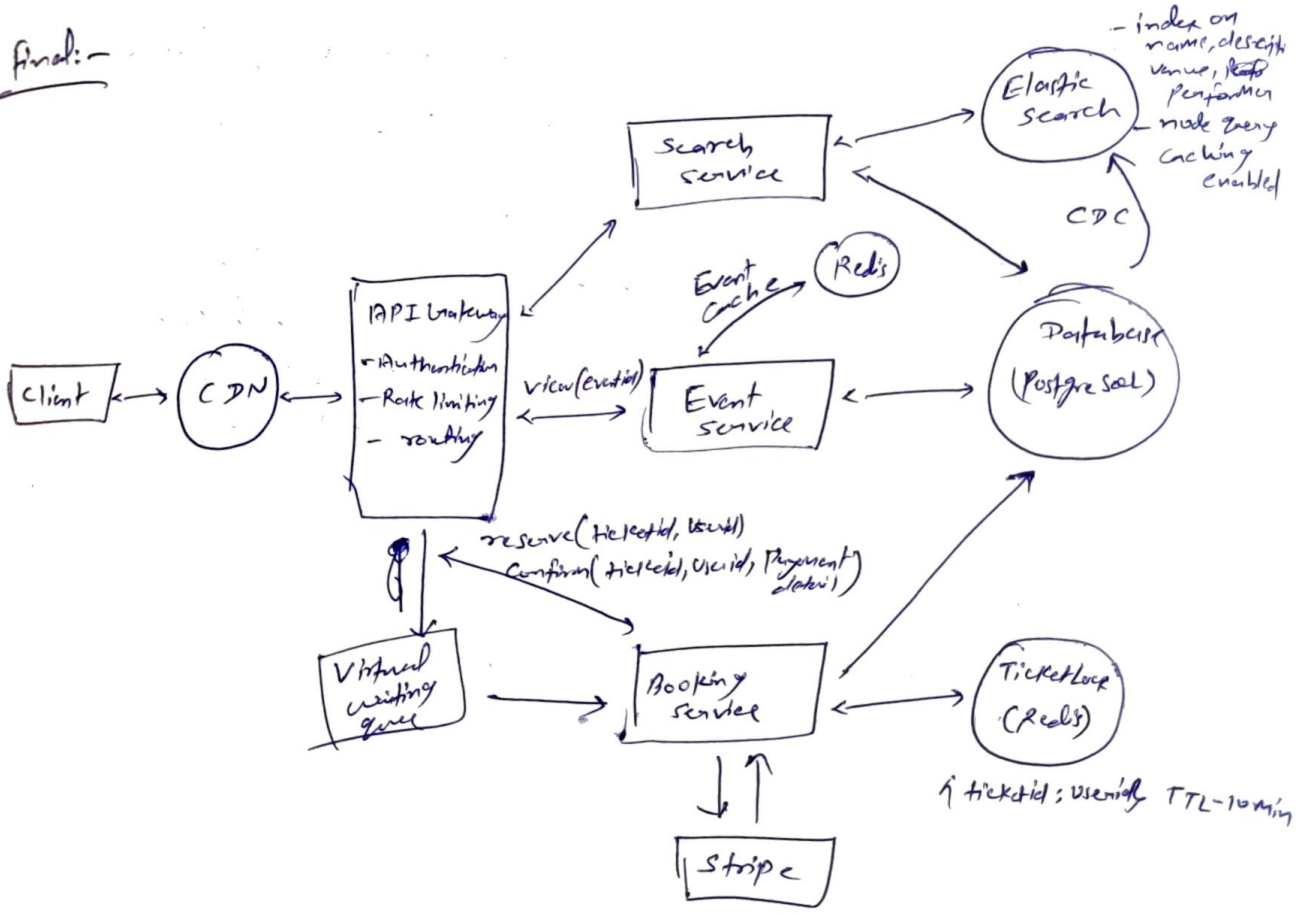
⇒ Great soln - Use a full-text search engine like Elastic Search

(5) How can you speed up frequently repeated search queries & reduce load on our search infrastructure?

⇒ Good soln - Implement Caching strategies using Redis or memcached

⇒ Great soln - Great answer:- Implement Query Result Caching and Edge Caching Techniques.

final:-



DropBox

(Design a file storage service like Dropbox)

Functional :-

1. User should be able to upload a file from any device
2. able to download file from any device
3. share file with other & view files shared with them
4. User can automatically sync files across devices.

Below the line..

1. Edit the file
2. view file without downloading them.

Non-functional

1. Highly available (availability >> Consistency)
2. support large files (<= 50 GB)
3. secure & reliable (should recover file if they are lost or corrupted)
4. Low Latency (upload, download, sync time as fast as possible)

Below line

1. storage limit per user
2. support file versioning
3. Scan files for viruses & malware.

Entity

- File
- FileMetadata
- User

HLD :-

(1) User should be able to upload a file from any device :-

⇒ Bad sol^{n} - Upload file to single server

⇒ Good sol^{n} - store file in Blob storage

⇒ Great sol^{n} - Upload file directly to blob storage

(2) User should be able to download a file from any device :-

⇒ Bad sol^{n} - Download through file server

⇒ Good sol^{n} - Download from blob storage

⇒ Great sol^{n} - Download from CDN

(3) Users should be able to share a file with other users :-

⇒ Bad sol^{n} - Add a sharelist to Metadata

⇒ Good sol^{n} - Caching to speed up fetching the sharelist

⇒ Great sol^{n} - Create a separate table for shares

UserId	FileId

(4) Users can automatically sync files across devices :-

⇒ two direction → (a) Local → Remote

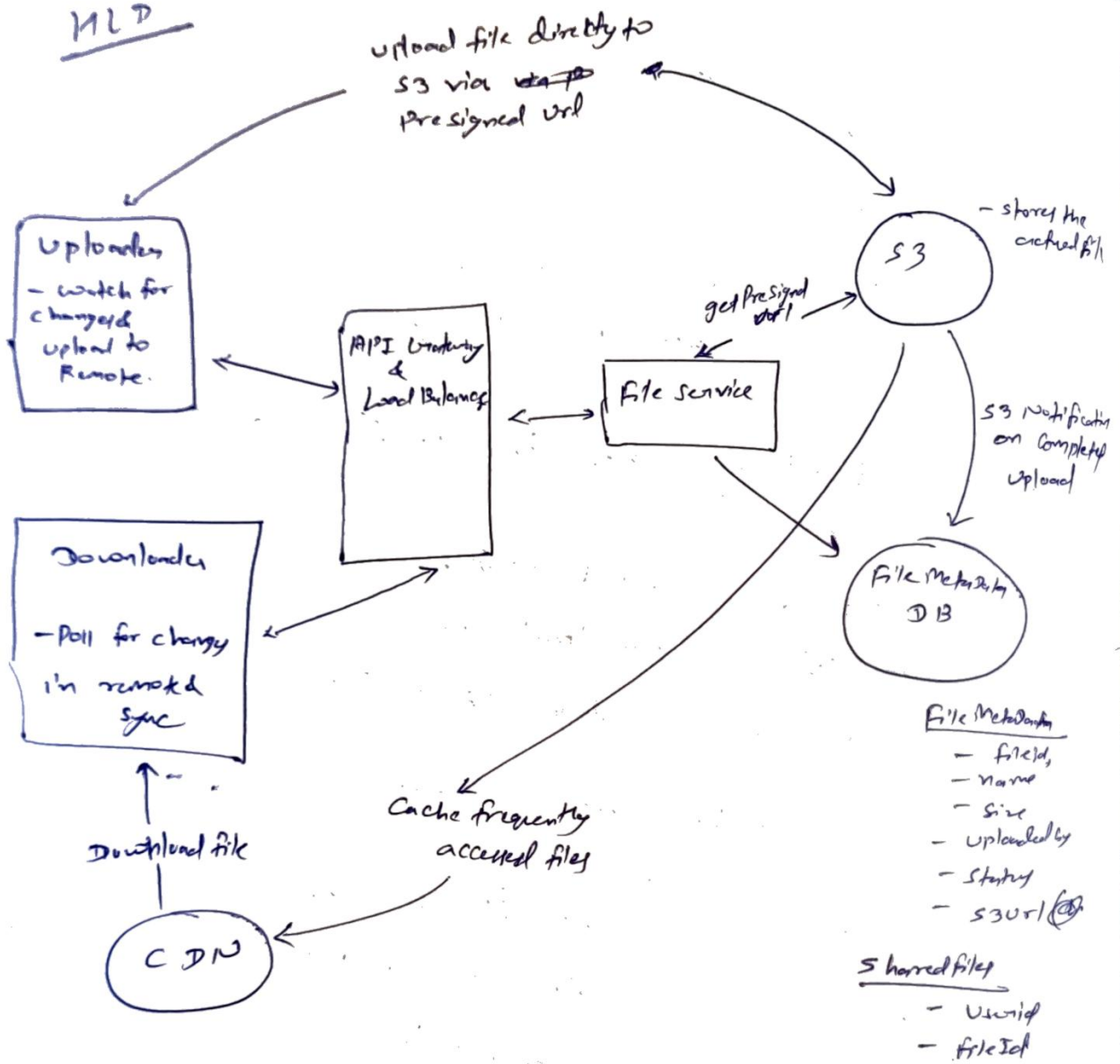
(b) Remote → Local → polling / websocket or SSE

for dropbox → hybrid approach

fresh files → websocket connection to ensure near Real-time sync.

Stale files (haven't modified for a while) → polling

HLD



⟹ Potential deep dives :-

(1) How can you support large files?

ⓐ Progress indicator
ⓑ Resumable uploads

] use "chunking"

→ divide file into chunks on client side

"chunks": [
{ id: chunk1, status: uploaded
},
...
]

but how should we ensure this 'chunks' field is kept in sync with the actual chunks that have been uploaded?

two approach
- (i) Good sol^n – Update based on client patch request
- (ii) Great sol^n – Rely on S3 Event notification

(2) How can we make uploads, downloads & syncing as fast as possible?

- Caching
- Chunking
- Compression to speed up both uploads & downloads.

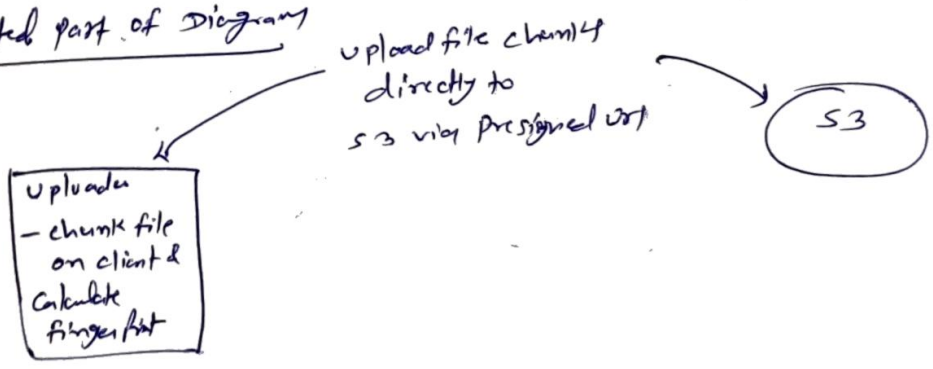
(3) How can you ensure file security?

(a) Encryption in transit (use HTTPS)

(b) Encryption at Rest

(c) Access control – shareList is our basic ACL. We make sure that we share download links only with authorized users.

Updated part of Diagram



Design a Url Shortener like Bit.ly

Functional :-

1. Users can Create short Urls from original urls
   - optional custom alias
   - optional expiration time
2. Users can access the original url by visiting short Url

Non-functional

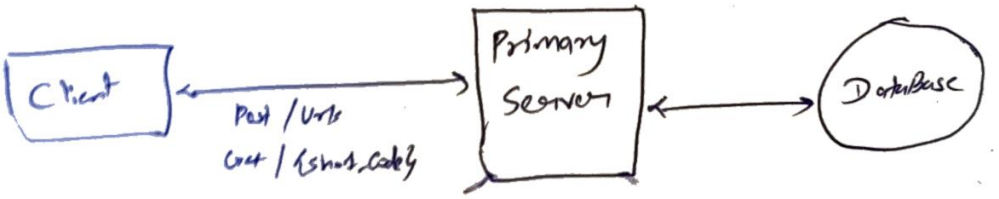
1. Ensure Uniqueness of short urls
2. Low latency redirection
3. Availability >> Consistency
4. Scalable to 1B urls & 100M DAU

Entities:-

- Original Url
- short Url
- User

Basic HLD:-

Write: 1) generate short url
2) save to DB



Urls:-
- short urlCode
- CreatedTime
- ExpirationTime
- CreatedBy

Read: 1) Lookup original url in DB
2) return with 302 redirect

⇒ Potential deep dives :-

(1) How can we ensure short urls are unique?

⇒ Bad sol^{n} – long url prefix (no uniqueness)

⇒ Good sol^{n} – Random number generation or Hash function

⇒ Great sol^{n} – Unique Counter with Base 62 encoding

↓

Using Redis → it's single threaded & support atomic operations

Challenge

↳ in distributed environment, maintaining a single global counter can be challenging due to synchronization issues.

(2) How can we ensure that redirects are fast?

– Full database scan is inefficient

⇒ Good sol^{n} – Add an index

⇒ Great sol^{n} – Implementing an In-memory cache (eg. Redis)

↳ Complex architecture as need to take care of invalidation strategies.

⇒ Great sol^{n} – Leveraging Content Delivery Network (CDNs) & Edge Computing

↳ benifit – for popular short urls, redirection can happen at the CDN (close to user) & it never even reaches our Primary server, Meaning reducing latency

(5) How can we scale to support 1B shortened url & 100M DAU?

Database size estimation:-

short code – 8 byte
long url – 100 byte
creation time – 6 byte
customalize – 100 byte

round up – 500 bytes

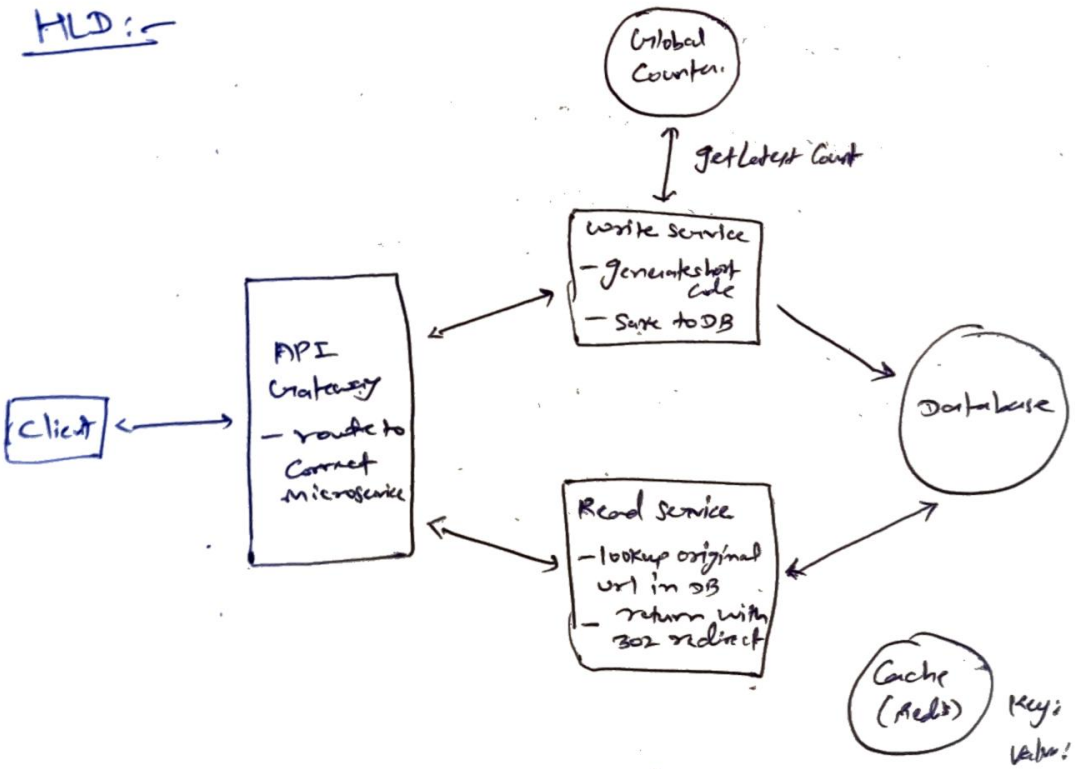
Assume 1L urls/day created
↓
1 per second
↓
any database will have.

total ⇒ 500 bytes × 1B rows = 500 GB

this application → Read heavy, write throughput is pretty low

what if DB goes down?
- Database Replication
- Database Backup

HLD:-



Design Facebook's News Feed

Functional :-

1. Create Posts
2. Follow People
3. View Feed
4. User able to Page through their feed

Below line

- liked & Comment on post
- Posts can be Private or have restricted visibility

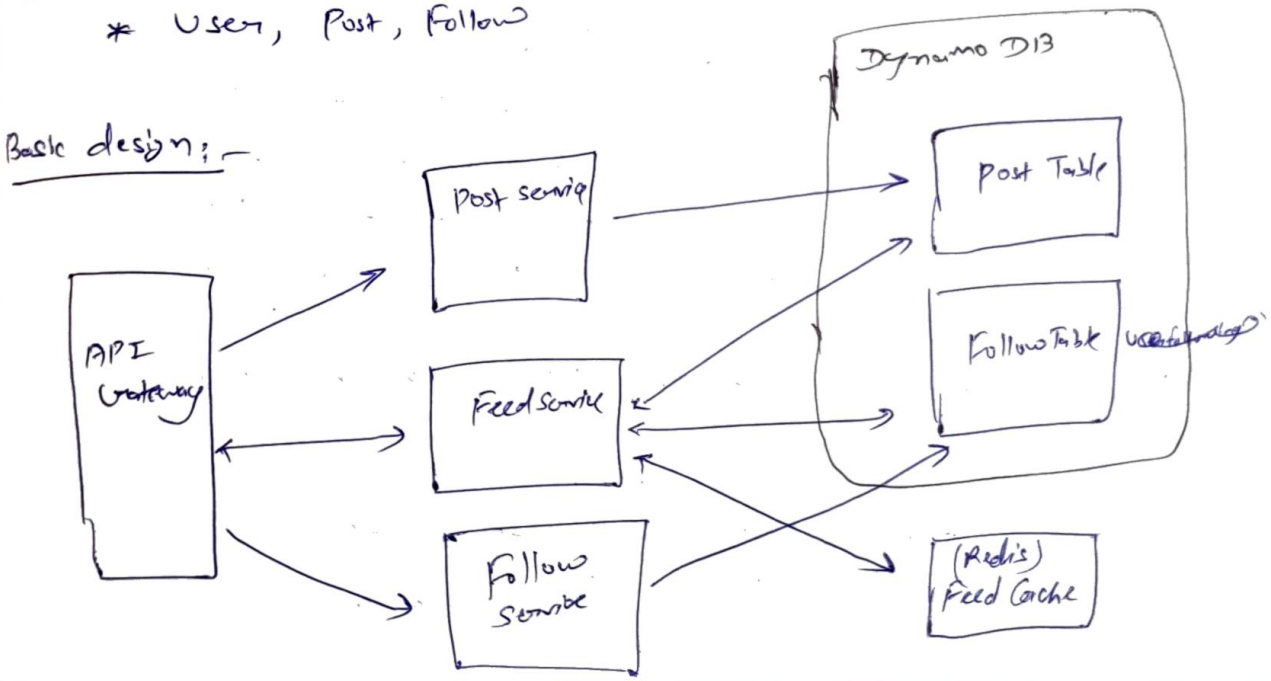
Non-functional :-

1. Posts visible in < 2 minutes (eventually Consistent)
2. Posting & viewing posts < 500 ms
3. Massive number of users (2B)
4. Unlimited followers / follows

Core Entities:-

* User, Post, Follow

Basic design:-



follows table { userfollowing: userfollowed }

⇒ Potential Deep Dives :-

(1) How do we handle users who are following a large number of users ?

⇒ issue :- user follows large no of users → follow query will take time
↓
for every follow, need to make query to 'Post' table
↓
Fan-out Problem

⇒ Bad soln :- Horizontal scaling via sharding

⇒ Great soln :- Adding a feed table ⇒ when a new post is created, we'll simply add to the relavant feeds.

- feed table itself is just a list of post ids, stored in chronological order & limited to a small no. of posts (200 or so)

issue with this soln → it improves read performance, but creats new issue, when a user with a lot of followers creates a post, we need to write to millions of feeds efficiently. so let's dive into the next.

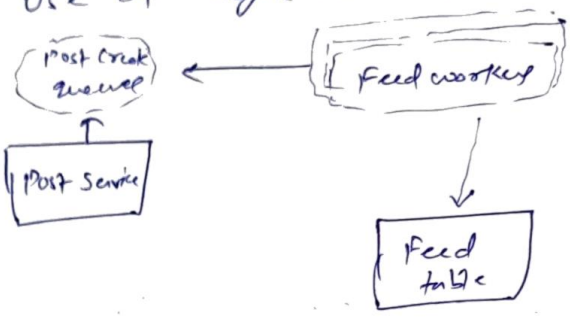
(2) How do we handle users with a large number of followers ?

↳ fan out problem there, when we create a post ; we need to write to million of Feed records.

⇒ Bad soln — Shotgun the Request (in worst case, we're trying to write to million of feeds with the new post entry)

⇒ Good soln — Async workers

↳ Use of async workers behind a queue.



⇒ Great soln — Async workers with hybrid feeds

↳ we can choose which accounts we'd like to Pre-Calculate into feeds and which we do not.

For justin bieber (& other high followed accounts), instead of writing to 90+ million followers, we can instead add a flag onto the follow table which indicates that this particular follow isn't precomputed. In async workers queue, we will ignore requests for these users.

on the read side, when users request their feed via the Feed service, we can grab their (Partially) precomputed feed from the Feed table & merge it with recent posts from these accounts which aren't precomputed.

- This hybrid approach allows us to choose whether we favour on read or write & for most users we will do a little of both.

(3) How can we handle uneven reads of Posts ?

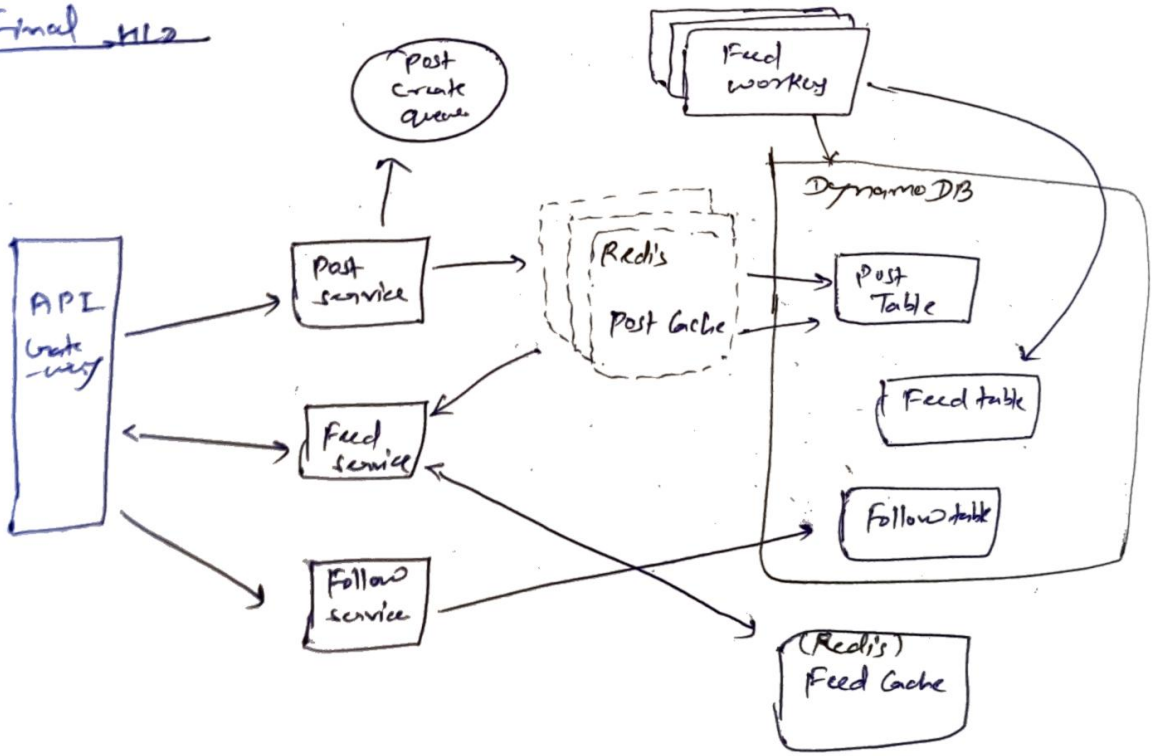
Problem- Posts get uneven reads
↓
Cause a "hot key"
↓
hurts DynamoDB performance
↓
need to spread load using
caching, replication or Precomputation

⇒ Good soln — Post Cache with Large keyspace

⇒ Great soln — Redundant Post Cache → insert a replicating cache b/w the readers of the Post table & the table itself.

— as long as our cache is big enough to house our most popular posts, the vast majority of requests to the Post table will instead hit our cache.

Final HLD



Design a Dating App like Tinder

Functional:-

1. Create Profile
2. User recommended based on filters / location
3. User can swipe left / right
4. 2 users swipe "Yes", match → notification

Below line

1. User should be able to upload pictures
2. chat via DM after meeting
3. Premium features

Non-functional

1. Consistency for swipes
2. Many concurrent users, with peaks in Traffic
3. Low latency feed / stack loading
4. Avoid showing repeat Profiles

Below line

1. system should protect against fake Profiles
2. monitoring | alerting

=> Core Entity

- User
- Swipe → expression of "yes" or "no" on a user Profile, belongs to a user (swiping_user) and is about another user (target_user)
- match

=> APIs

1. post /profile

```
{
  age_min : 20,
  age_max : 30,
  "distance" : 10
  "interested in" : Female/ Male
}
```

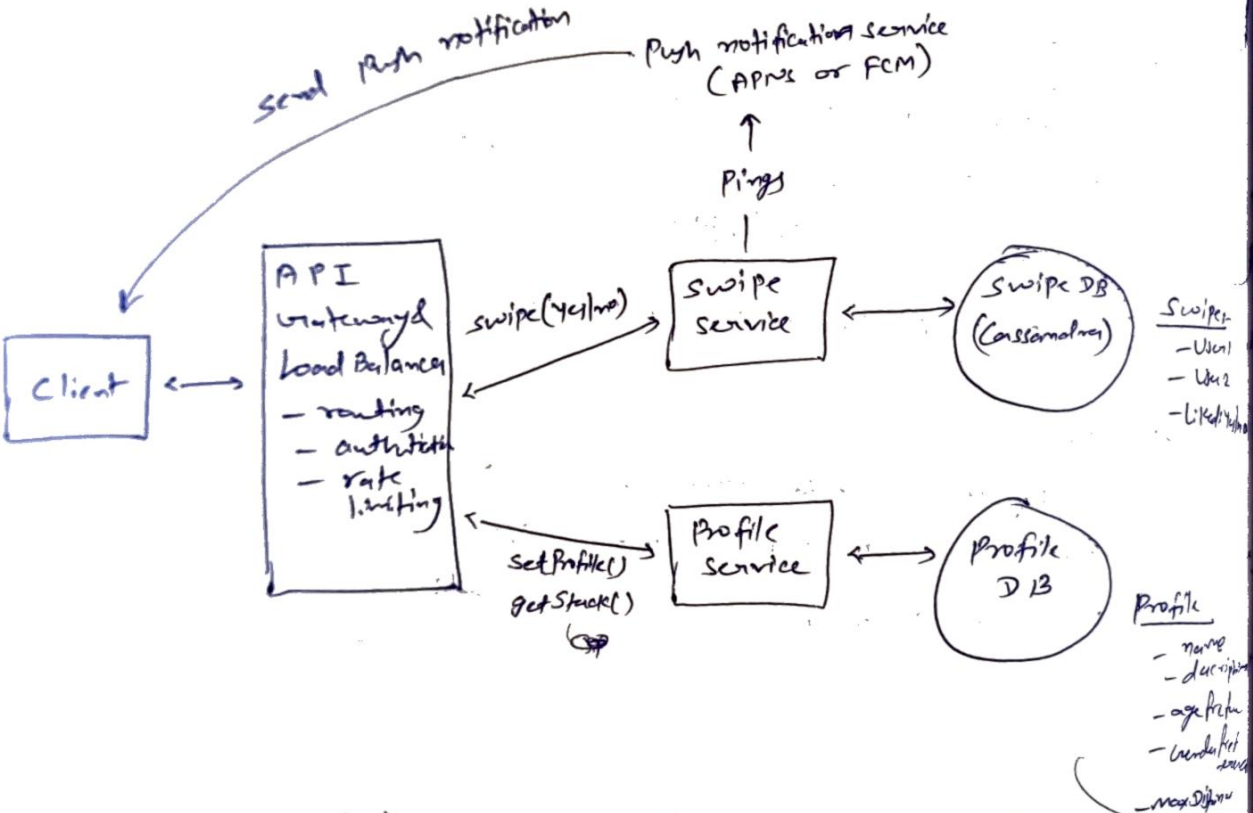
(2) GET /feed ? lat={} & long={} ⟶ User[]

(3) Post /swipe/{userId}

Request :

```
{
  decision : "yes"|"no"
}
```

Basic Design :



We assume 20M daily active users doing 200 swipes a day on average, that nets us 4B swipes a day. This certainly means we we'll need to partition the data.

Cassandra is a good fit as a database here; we can partition by swiping_user_id. This mean an access pattern to see if userA swiped on userB will be fast, because we can predictably query a single partition for that data.

Additionally, Cassandra is extremely capable of massive writes, due to its write-optimized storage engine (Commitlog + Memtables + SSTables).

A Con of Cassandra is the element of eventual Consistency of swipe data we inherit from using it. we will discuss ways to avoid this Con in later deep dives.

⇒ Potential Deep Dives :-

(1) How can we ensure that swiping is consistent and low Latency?

failure scenario :- Imagine Person A & Person B both swipe right (like) on each other at roughly at same time.
↓
They found nothing for each other & save swipe. but missed opportunity to notify.

⇒ Bad soln :- Database polling for matches

⇒ Good soln :- Transactions (database transactions)

⇒ Great soln :- sharded Cassandra with single-partition transactions.
↳ we can leverage Cassandra's single-partition transactions to atomically handle swipes. The key is to ensure that all swipes b/w two users are stored in same partition.

_This approach is effective because Cassandra's single-partition transactions provide the atomicity guarantees we need.

⇒ Great soln — Redis for atomic Operations

↓

We can use Redis to handle the atomic swipe operations while still using Cassandra as our durable storage layer.

↓

Key value structure:-

```
Key: "swipe:123:456"
value: {
        "123_swipe": "right",
        "456_swipe": "left"
       }
```

Challanges — we need to carefully handle node failures & rebalancing of the consistent hashing ring.

(2) How can we ensure low latency for feed/stack generation?

⇒ Good soln ⇒ Use of Indexed databases for Real-Time Querying

⇒ Bad soln — Pre-Computation & Caching

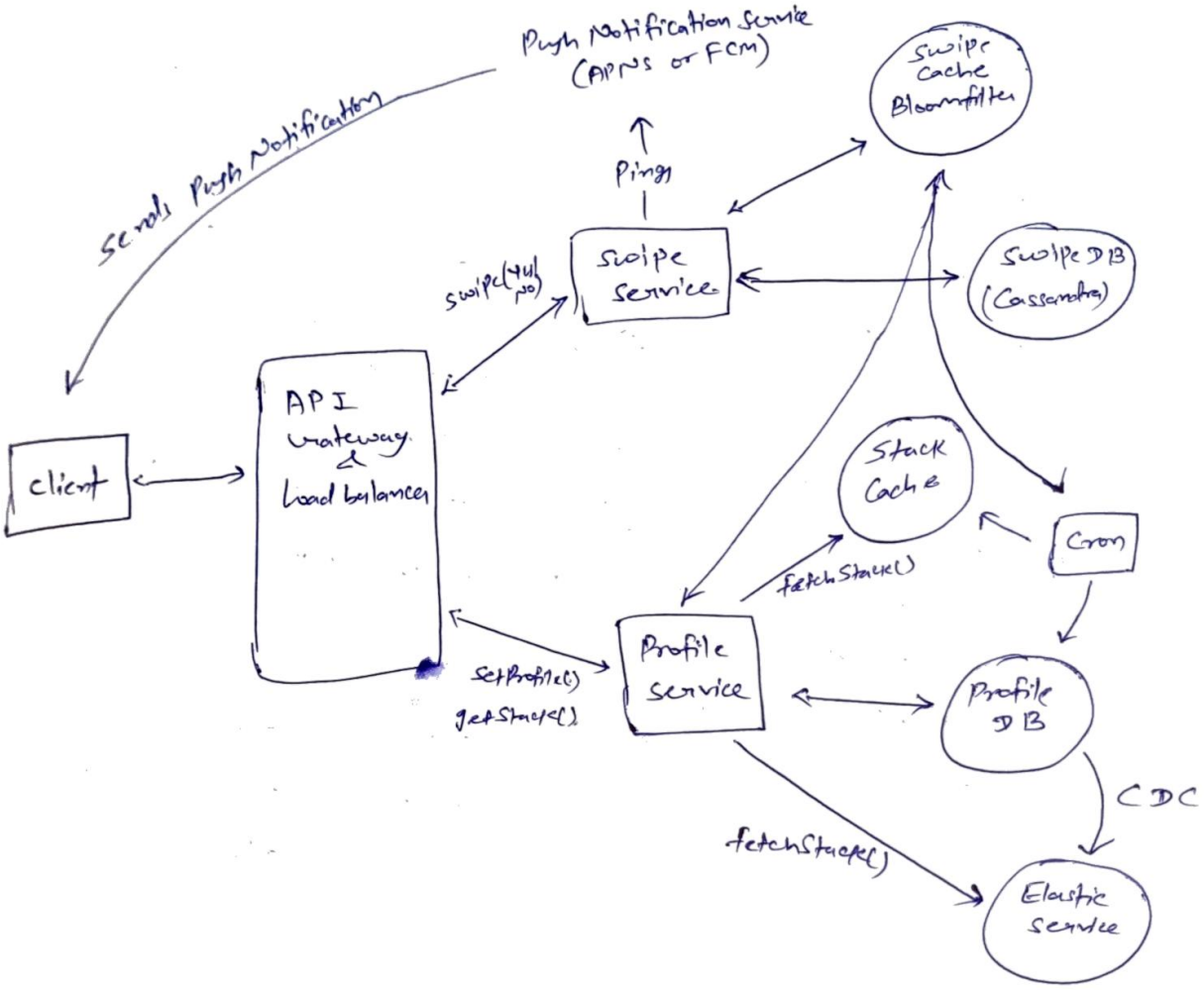
⇒ Great soln — Combination of Pre-Computation & Indexed database.

(3) How can the system avoid showing user profiles that the user has previously swiped on?

⇒ Bad soln — DB Query + Contains check

⇒ current soln – Cache + DB query + Containing check

⇒ current soln – Cache + Containing check + Bloom filter.



Design a Video Streaming Platform Like YouTube

Functional:-

1. Upload video
2. watch (stream) videos

Below the line

1. view info about a video
2. search for video
3. comment on video
4. see recommended video
5. make channel & manage it
6. subscribe to channels

Non-functional :-

1. Availability >> Consistency
2. Low latency streaming (even if low bandwidth)
3. High scale
4. Support large videos (10+ GB)
5. Resumable ~~down~~ uploads

Below line

1. Protect against bad content in video
2. Protect against bot or fake account
3. monitoring | alerting

Core Entity:-

- User
- Video
- VideoMetaData

API :-

1. post /upload
   Request: { video, VideoMetadata }
2. Get /video/{videoId} → Video & VideoMetadata

⇒ Background (Video Streaming) :-

① **Video Codec** – A video codec compresses & decompresses digital videos, making it more efficient for storage & transmission.

Codec – is abbreviation for "encoder/decoder"

② **Video Container** – It is a file format that stores video data (frame, audio) & metadata.

③ **Bitrate** :- The bitrate of a video is the number of bits transmitted over a period of time.

↳ The size & quality of the video affect the bitrate.

④ **manifest file**:- Manifest files are text-based documents that give details about video streams.

↳ two type → Primary file
→ Media file

functional –

① **User can upload videos** :-

⇒ Bad soln – store the raw video in S3 (because diff. device need diff. formats)

⇒ Good soln – store different video formats

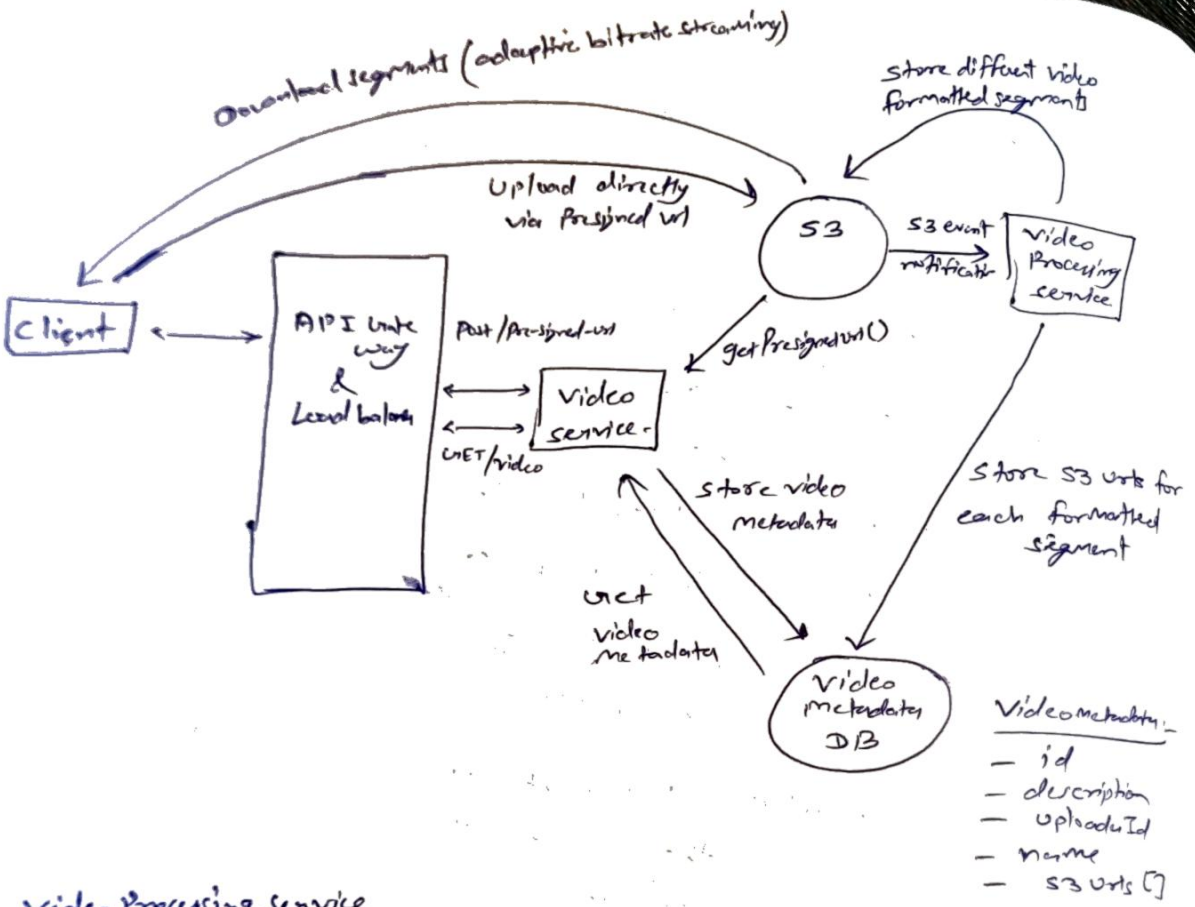
⇒ Great soln – Store different video formats as segments

② **User can watch videos** :-

⇒ Bad soln :- Download the video file

⇒ Good soln :- Downloads segments incrementally

⇒ Great soln :- Adaptive bitrate streaming



Video Processing service

→ pipeline that → 1) splits up a video
→ 2) Convert each segment into various formats

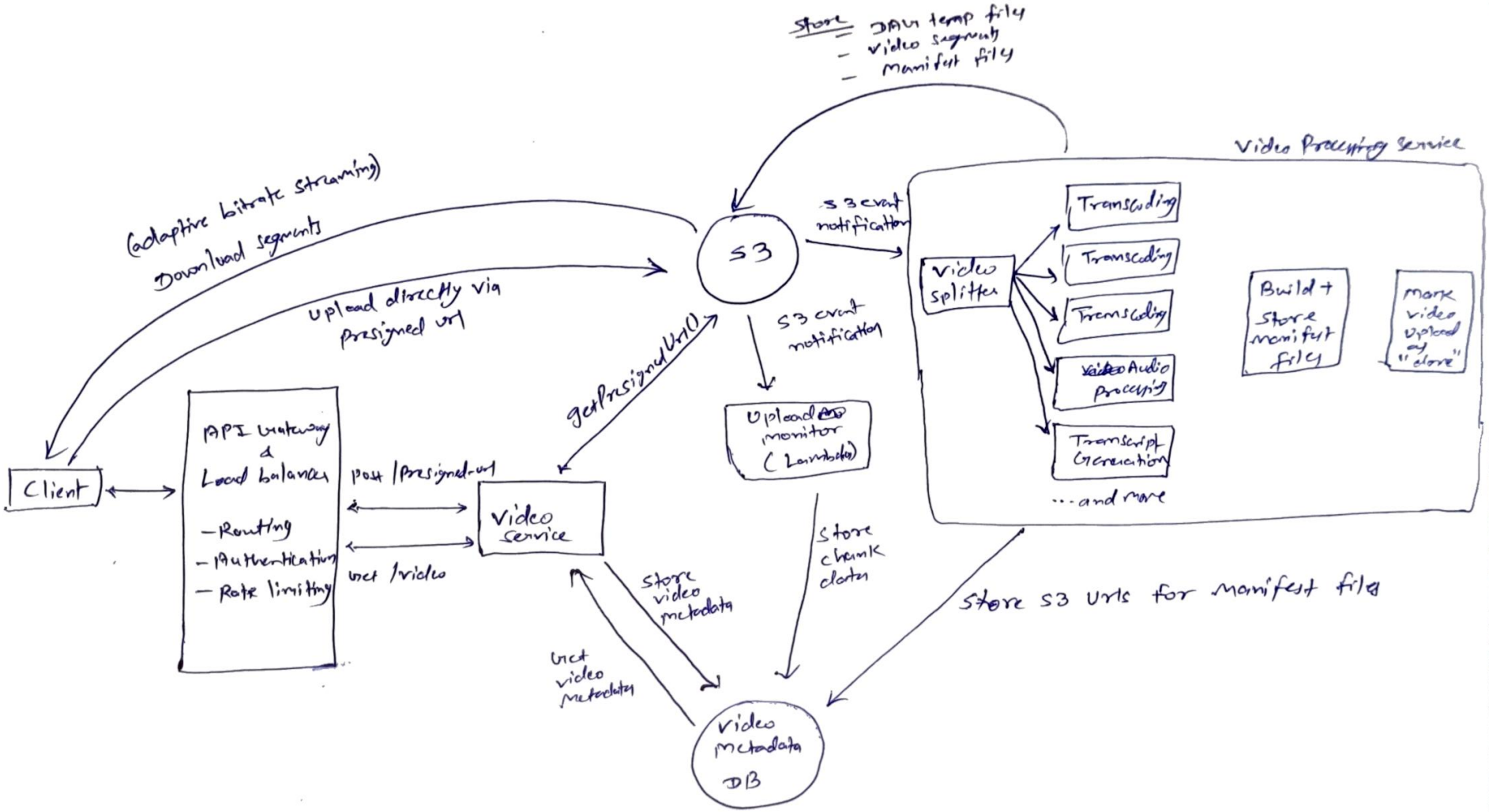
Deep dive —

(1) How do we support resumeable uploads?

↳ strong overlap with the Dropbox deep dive involving
- How to support large file uploads.

(Chunking, fingerprint, S3 event notification)

Full design:—


Store
DAW temp files
- video segments
- manifest files
Video Processing Service
(adaptive bitrate streaming)
Download segments
Upload directly via
Presigned url
S3
S3 event notification
Transcoding
Transcoding
Transcoding
Audio Processing
Transcript Generation
Video splitter
...and more
Build + store manifest file
mark video upload as "done"
getPresignedUrl()
S3 event notification
Upload monitor (Lambda)
Client
API Gateway & Load balancer
- Routing
- Authentication
- Rate limiting
post /presigned-url
get /video
Video service
Store video metadata
Get video metadata
Store chunk data
Store S3 urls for manifest file
Video metadata DB

Design Facebook's Live Comments System

Functional :-

1. viewers can post comments.
2. " " see all comments posted in near realtime.
3. " " " all comments posted before they joined.

Below the line

A. viewers can reply to comments.
B. " " react to comments.

Non-functional Requirements :-

1. Scale to millions of viewers per video.
2. Availability >> Consistency
3. Low latency comment broadcast (~200 ms)

Below the line

A. Security
B. Integrity (not spam, hate speech etc)

Core Entity :-

1) User
2) Comment
3) Live video

API :-

1. Post /comment/:liveVideoId

```
{
  message: "Cool video!"
}
```

② GET /comments/:liveVideoId ? cursor=<last_comment_id> & pageSize=10 & sort=desc

Basic design –

① viewers can see new comments while watching video

↳ Naive soln — polling

GET /comments/:liveVideoId ? since=<last_comment_id>

② viewers can see comments made before they joined the live feed

⇒ Bad soln – offset pagination
⇒ Correct soln – cursor pagination

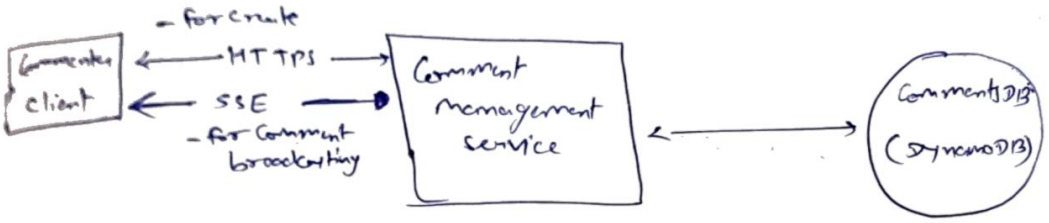
⇒ Potential Deep Dive :–

① How can we ensure comments are broadcasted to viewers in real-time?

⇒ Bad soln – polling
⇒ Good soln – WebSockets → Good for more-balanced read/write ratio
→ our use case → read/write ratio is not balanced. ↓ why?
readers >> writers
↓
doesn't make sense to open a two-way communication channel for each viewer.

— What is? — Server Sent Events (SSE)

↳ Server can push data to client in real-time.



- when a new comment is posted, persist in DB & sent over SSE to connected clients

(2) How will the system scale to support millions of concurrent viewers?

Horizontal Scaling

challenge.

* UserA is watching live video1 & connected to server1,
* UserB " " video1 & " " server2.

now, imagine a new comment is posted on live video1, If this comment request hit server1 :-

* server1 can easily send it to UserA since they're directly connected.
* But server1 has no way to send it to UserB, who is connected to server2.

This is our key challenge: How do we ensure all viewers see new comments, regardless of which server they are connected to?

⇒ Bad Soln :- Horizontal Scaling with Load balancer and Pub/Sub.

```
{ "liveVideoId1" : ["sseConnection1", "sseConnection2" ],
  "liveVideoId2" : ["sseConnection3", "sseConnection4"]
}
```

⇒ Good Soln :- Pub/Sub Partitioning into channels per live video.

⇒ Great Soln :- Partitioned Pub/Sub with Layer 7 Load Balancer :-

To address the issue of servers handling viewers from many different live videos, we need a more intelligent allocation strategy.

The goal is to have each server primarily handle viewers of the same live video, making it easier to limit the number of topics or channels each server needs to subscribe to.

There are two ways we can achieve this allocation, both require some scripting or configuration.

Option 1 Intelligent Routing via scripts or Configuration

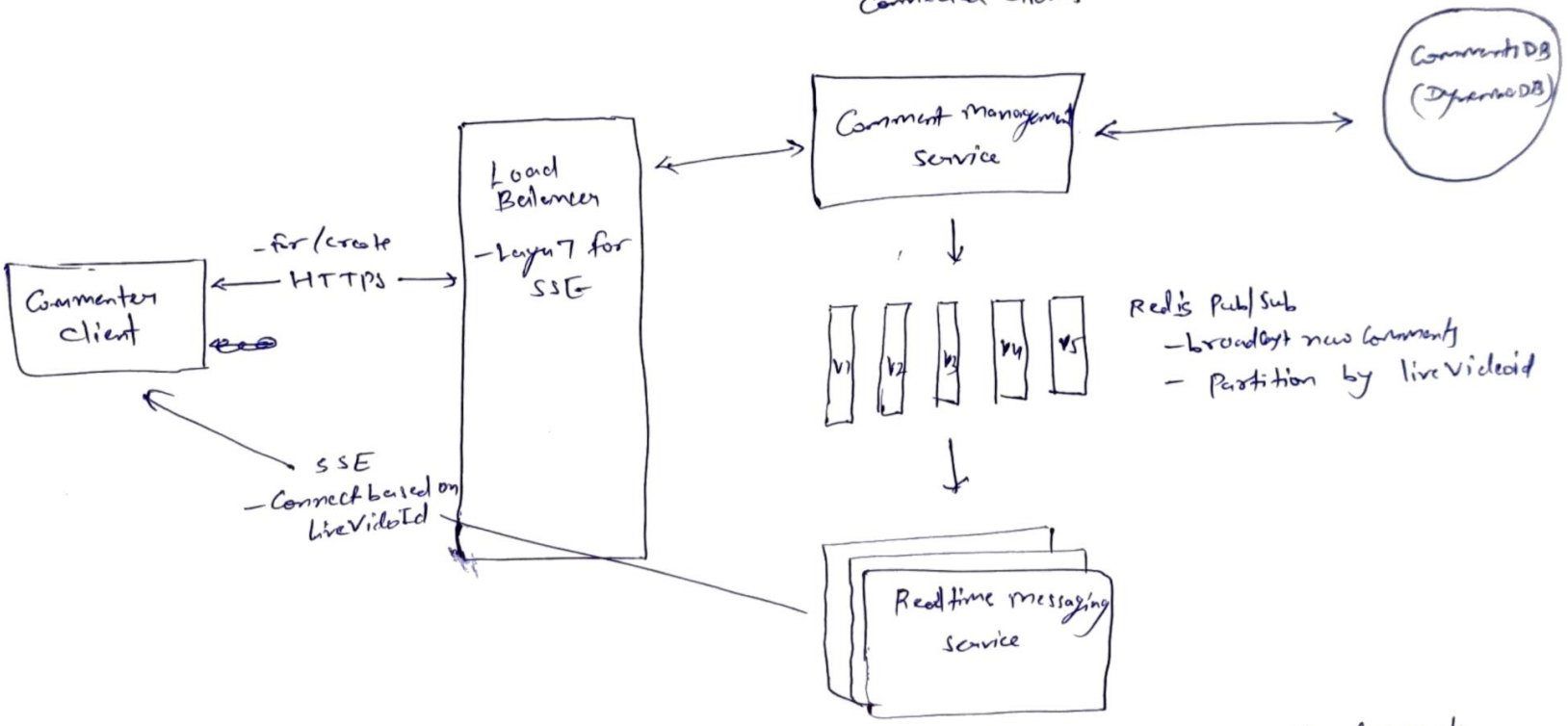
↓

A Layer 7 LB can leverage Consistent hashing based on "liveVideoId". Load balancer can apply a hashing function on liveVideoId that always routes viewers of same video to same server.

Option 2 Dynamic Lookup via a Coordination service (e.g. ZooKeeper)

HLD :-

- when a new comment is posted, persist in DB, & sent over SSE to connected clients



- subscribe only to the topics with comments from live videos that your client are watching.
- Viewers of same video end up on same box based on layer 7 LB. Leads to fewer topics to subscribe to per server.

Design a Messaging App like Whatsapp

Functional :-

1. Start group chats (limit 100 users)
2. send-receive messages
3. send/receive media
4. Access messages after they've been offline.

Non-Functional:-

1. Delivered with low-latency < 500ms
2. Guarantee delivery of messages
3. Billions of users, high throughput
4. messages not stored unnecessarily fault-tolerant

Below the line

1. Audio/video calling
2. Interactions with businesses
3. Registration & Profile Management

Below the line

1. Exhaustive treatment of security concerns
2. Spam & Scraping Prevention systems

Core Entities:-

1. Users
2. Chat (2-100 users)
3. Messages
4. Clients (a user might have multiple devices)

API:-

1. // → CreateChat

```
{ "participants": [],
  "name": ""
} → {
    "chatId": ""
  }
```

2. // → SendMessage

```
{ "chatId": "",
  "message": "",
  "attachments": []
} → "SUCCESS" | "FAILURE"
```

(3) createAttachment

{ "body": --,
"hash": -
} → { "attachmentId": ""
}

(4) modifyChatParticipants

{ "chatId": " ",
"userId": " ",
"Operation": "ADD" | "REMOVE"
} → "SUCCESS" | "FAILURE"

---

(5) when a chat is created or updated ---

// → chatUpdate
{ "chatId": " ",
"participant": [],
} → RECEIVED

(6) when a message is received...

// → newMessage
{ "chatId": "",
"userId": " ",
"message": " ",
"attachments": []
} → RECEIVED

on interview page:-

Commands Sent

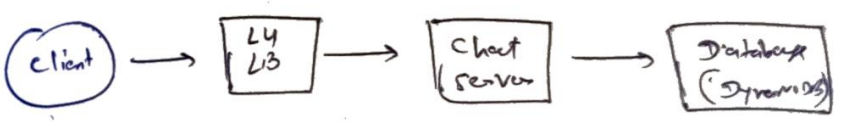
* createChat
* sendMessage
* createAttachment
* modifyParticipants

Commands Received

* newMessage
* ChatUpdate

Basic HLD:-

(1) Users should be able to start group chats with multiple participants (limit 100).



Chat:-
- id
- name
- metadata

ChatParticipant:-
- chatId
- participantId

Steps ⇒ User connect to service & sends a "createChat" message ⟹ service, inside a transaction, create a "chat" record in the database & create a "chatParticipant" record for each user in the chat.

(a) chat Table ⇒ id will be primary key

chatParticipant Table ⇒ Composite primary key on 'chatId' & 'participantId' field.

(b) we will need a Global Secondary Index (GSI) with "participantId" as the partition key and "chatId" as the sort key. This will allow us to efficiently query all chats for a given user. The GSI will automatically be kept in sync with the base table by DynamoDB.

(2) Users should be able to send/receive messages.

for simplicity ⟶ assuming (single chat server)

↓

Contains HashMap <UserId, WebSocket Connection>

Steps:-

(a) user send a "sendMessage" to chat server,

(b) chatServer look up all participants in chat via 'chatParticipant' table,

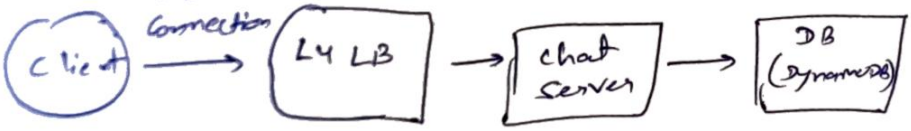
(c) chat server looks up the WebSocket Connection for each participant in its internal hash table & sends the message via each connection.

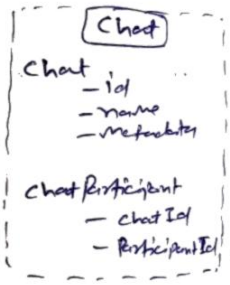
We are making some really strong assumptions here! We are assuming all users are online, connected to same chat server, & that we have a websocket connection for each of them. But under those conditions we are moving, so let's keep going

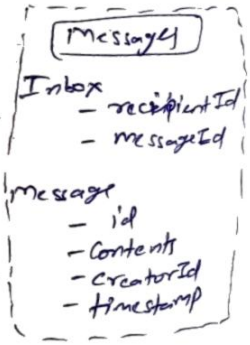
(3) Users should be able to receive messages sent while they are not online (up to 30 days).

=> we need ~~double to store~~ start storing messages in DB. so that we can deliver them to users even when they are offline. Let's add some robustness to system.

Let's keep an "Inbox" for each user which will contain all undelivered messages. when messages are sent, we'll write them to the inbox of each recipient user. If they are already online, we can go ahead & try to deliver the message immediately. If they are not online, we will store messages & wait for them to come back later.







(a) User sends 'SendMessage' to chat server.

(b) Chat server check for participants

(c) Chat server creates a transaction

(i) write message to messages table.

(ii) Create an entry in our Inbox table

(d) Chat server return a SUCCESS or FAILURE to the user with final message id.

(e) Chat server looks up the websocket connection for each participant & attempts to deliver message to each of them via "newMessage"

(f) (For connected client) Upon receipt, client will send 'ack' message to chat server to indicate they have received message. The chat server will then ~~delete~~ delete message from 'Inbox' table.

For clients who are not connected, we will keep the messages to inbox table. Once the client connects to our service later, we will:-

(i) look up user's 'inbox' & find any undelivered messages.

(ii) For each message id, look up the message in "message" table.

(iii) write those messages to client's connection via 'newMessage' message.

(iv) Upon reciept, client send 'ack' to server.

(v) chat server will then delete message from 'Inbox' table.

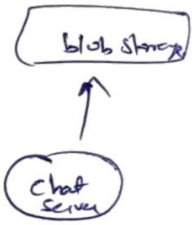
Finally, we will need to periodically clean up the old messages in the 'Inbox' & messages table. We can do this with a simple cron job which will delete messages older than 30 days.

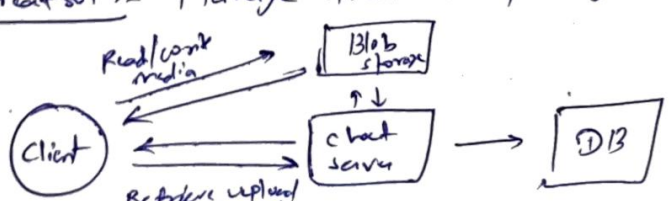
(4) Users should be able to send/receive media in their messages.

⇒ Bad soln :- Keep attachments in DB

⇒ Good soln :- Send attachments via chat server

⇒ Great soln :- manage attachments separately.





⟹ Potential Deep Dives :-

(1) How can we handle billions of simultaneous users?

⇒ Bad soln - Naively horizontally scale.

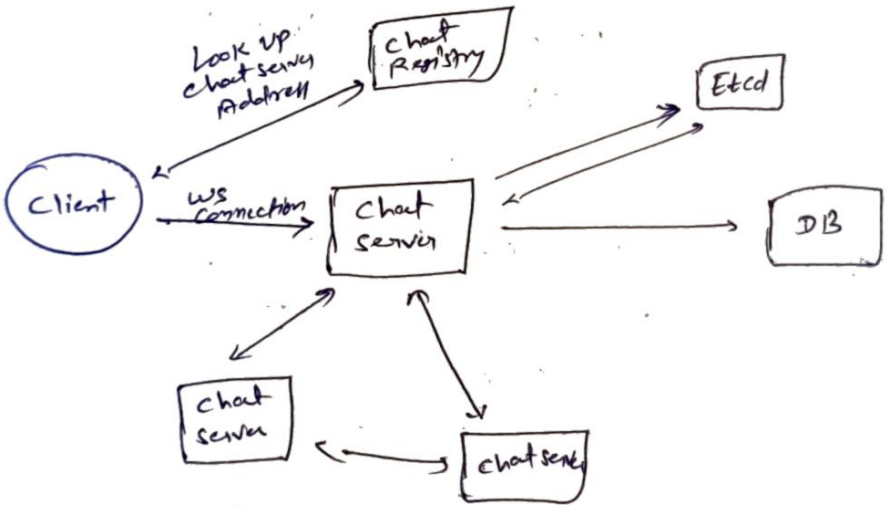
⇒ Bad soln - Keep a kafka topic per user.

↳ Idea is keep 'Inbox' table ~~for~~ at kafka topic but kafka is not built for billions of topics

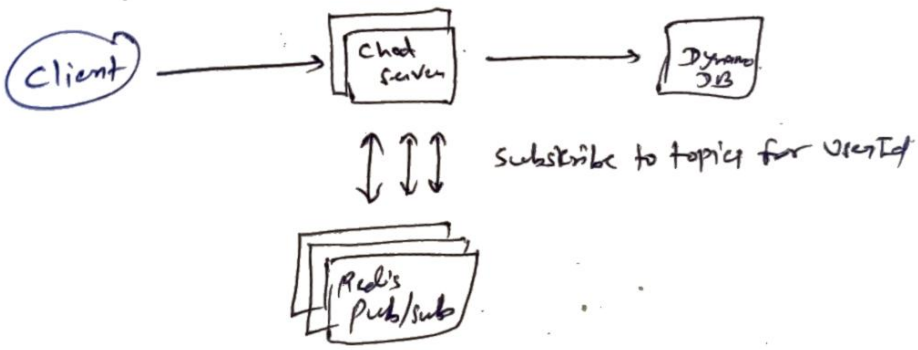
~~⇒ Consistent H~~

⇒ Good soln - Consistent Hashing of chat servers.

↳ Always assign users to a specific chat server based on userId. We need central registry for chat servers. (zookeeper)



⇒ Great soln - offload to Pub/sub



(2) what do we do to handle multiple clients for a given user?

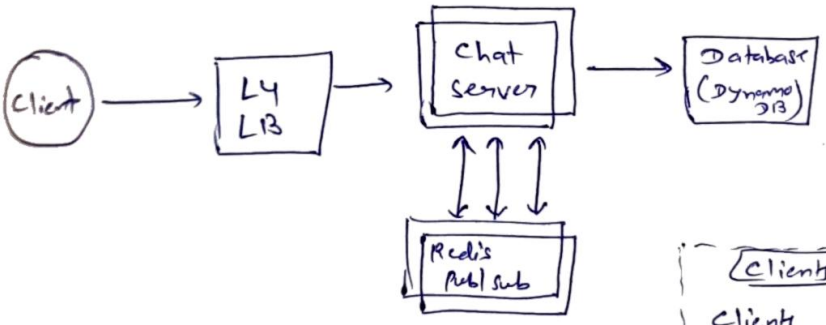
(a) we will need to create a new "clients" table to keep track of clients by userId.

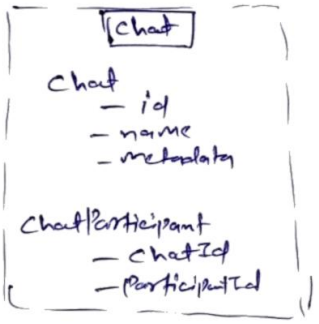
(b) when we look up participants for a chat, we will need to look up all of the clients for that user.

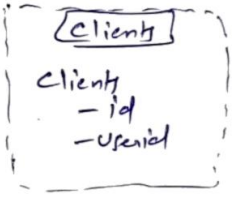
(c) Chat servers will subscribe to a topic with the "clientId" vs the "userId".

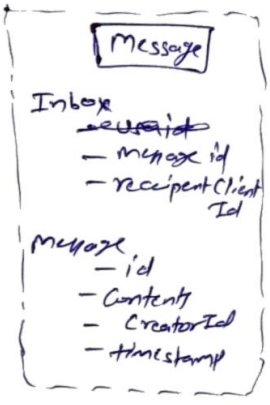
(d) when we send a message; we will need to send it to all of the clients for that user.

(e) we will need to update our "Inbox" table to be per-client rather than per user.









Design a Ride-sharing Service like Uber

Functional:-

1. Riders request estimated fare (input start & destination location).
2. Riders accept fare to get matched with a driver.
3. Driver can accept/deny & navigate to pickup/dropoff.

Below line

A. Rating drivers/riders
B. Scheduling rides in advance
C. Support different ride types (X, XL, etc)

Non-Functional :-

1. Low-latency ride matching (<1 minute)
2. Strong consistency for matching.
3. Handle surges (100k from same location @ same time)

Below the line

A. GDPR
B. Backups
C. CI/CD
D. Fault Tolerance
E. Monitoring

B.

Core Entities

- Rider
- Driver
- Location
- Fare
- Ride

⇒ APIs

① POST /fare → Fare

Body: { PickUpLocation, destination }

② Request Ride endpoint

POST /rides → Ride

Body: { fareId }

③ Update drivers location endpoint

POST /drivers/location → Success/Error

Body: { lat, long }

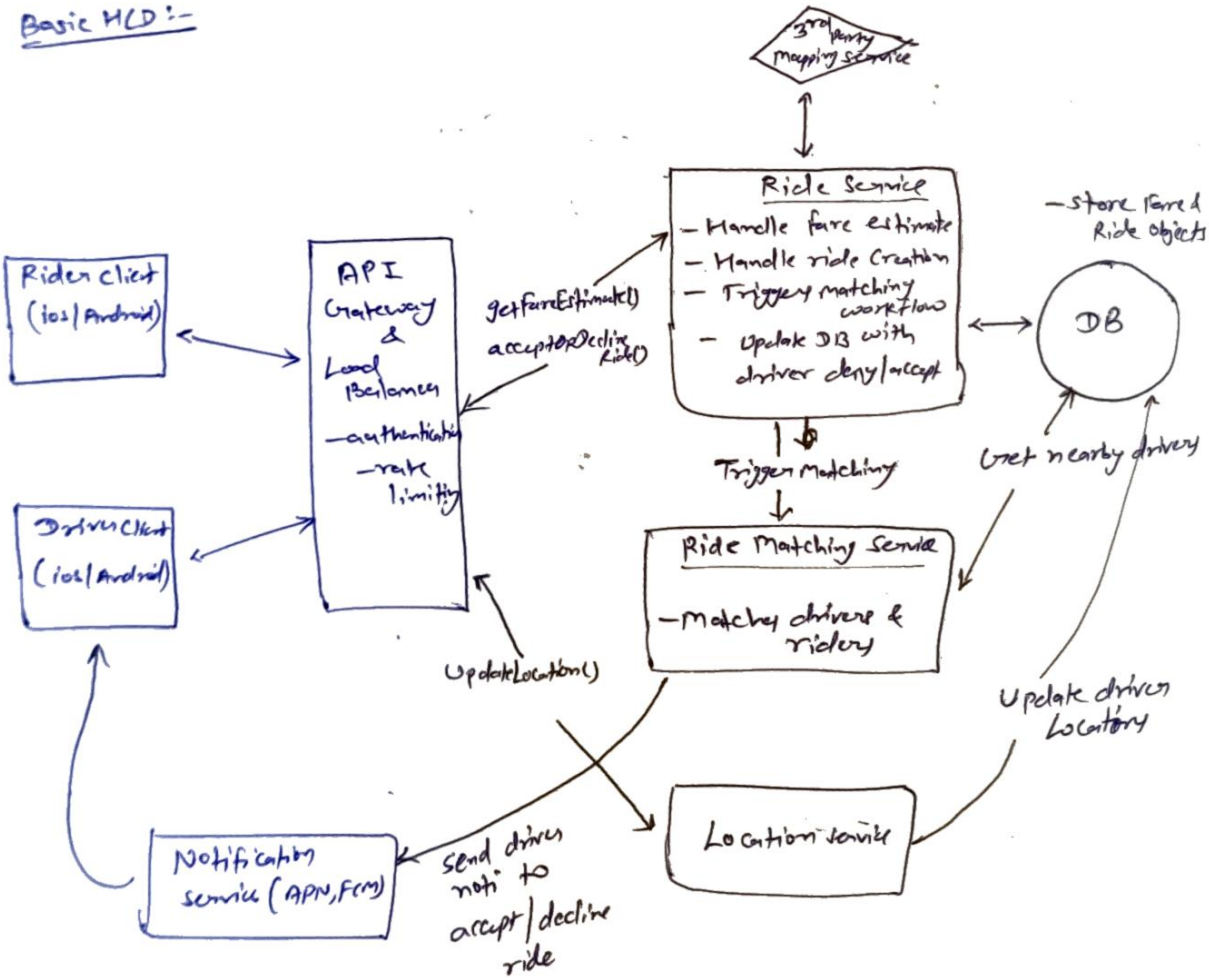
④ Accept ride request endpoint

POST /rides/:rideId → Ride

Body: { accept/deny }

- Ride object contains info about pickup, droploc .. so client can display this info to driver.

Basic HLD :-



Ride
- rideId
- riderId
- driverId ?
- fareId
- source
- destination
- status
- ...

Fare
- id
- userid
- source
- destination
- Price
- etc
- ...

Rider
- id
- ...

Driver
- id
- vehicle
- name
- location

⇒ Potential Deep Dives :-

(1) How do we handle frequent driver location updates & efficient proximity searches on location data?

Issue :- there are two main problems with our current design that we need to solve :-

(A) High Frequency of writes :-
* total drivers = 10 million
* sending locations roughly every 5 seconds
- that's about 2 million updates a second.

(B) Query Efficiency :-
- without any optimization, to query a table based on lat/long we would need to perform a full table scan.

⇒ Bad soln - Direct database writes & Proximity queries
↳ Inefficient, slow, system overload, high latency

⇒ Good soln - Batch Processing & specialized Geospatial Database. (quadtrees)

⇒ Great soln - Real-time In-memory Geospatial Data Store. (Redis) ↓

↓

Redis provides 'geospatial commands' like 'GEOADD' for adding location data & 'GEOSEARCH' for querying nearby locations within a given radius or bounding box.

(2) How can we manage system overload from frequent driver location updates while ensuring location accuracy?

⇒ Solution - Adaptive location update intervals.

- (based on factors as speed, ride request, driver's activity) moving slowly or fast —.

(3) How do we prevent multiple ride requests from being sent to the same driver simultaneously?

- we defined Consistency in ride matching as a key non-functional requirement. This means that we only request one driver at a time for a given ride request AND that each driver only receives one ride request at a time.
- That driver would then have 10 seconds to accept or deny the request before we move on to the next driver if necessary.
- If you have solved TicketMaster before, you know this problem as well - as it's ~~same~~ almost exactly the same as ensuring that a ticket is only sold once while being reserved ~~amount~~ for a ~~set~~ specific amount of time at checkout.

⇒ Bad soln – Application-level locking with manual Timeout checks.

⇒ Good soln – Database status update with Timeout handling.

⇒ Great soln – Distributed Lock with TTL.

(4) How can we ensure no ride requests are dropped during peak demand periods? (Ride matching service can be crashed or is restarted, leading to dropped rides)

⇒ Bad soln – First-come, First served with No Queue

⇒ Great soln – Queue with Dynamic Scaling

⇓ Introduce queuing system with dynamic scaling.

(a) Add ride request to queue.

(b) Ride matching service will process requests from queue.

(c) If queue grows too large, the system scales horizontally by adding more instances of Ride Matching service to handle increased load.

Challenges:

- Manage Complexity of Queuing System
- Need to ensure that the queue is scalable, fault-tolerant & high ~~scalable~~ available.
- Use Amazon SQS or Kafka, which provides these capabilities out of the box.

(5) How can you further scale the system to reduce latency & improve throughput?

⇒ Bad soln – Vertical scaling (increase capacity of existing server)

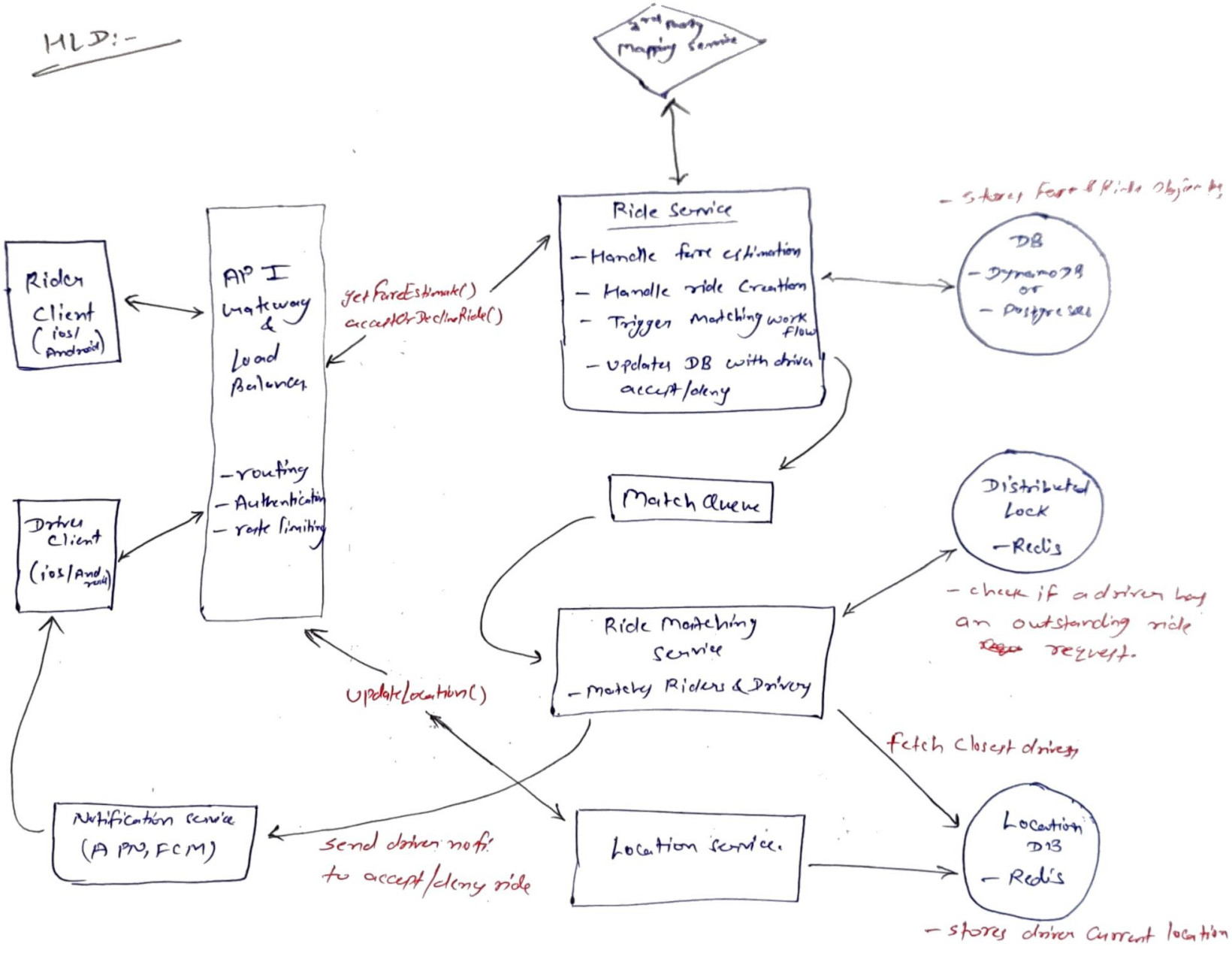
⇒ Correct soln – Geo sharding with Read Replicas

↳ services, message queue, our database can be sharded geographically.

↳ (Consistent hashing usage)

(

HLD:-


3rd party Mapping Service
Rider Client (ios/ Android)
API Gateway & Load Balancer
- routing
- Authentication
- rate limiting
getFareEstimate()
acceptOrDeclineRide()
Ride Service
- Handle fare estimation
- Handle ride Creation
- Trigger Matching work flow
- updates DB with driver accept/deny
- stores Fare & Ride Objects
DB
- DynamoDB or
- PostgreSQL
Driver Client (ios/Android)
Match Queue
Distributed Lock
- Redis
- check if a driver has an outstanding ride request.
Ride Matching Service
- matches Riders & Drivers
updateLocation()
fetch Closest drivers
Notification service (APN, FCM)
send driver notif to accept/deny ride
Location service
Location DB
- Redis
- stores driver current location as lat/long

Design a Coding platform like LeetCode

Functional:-

1. View a list of Coding problems
2. View a given problem & code a solution in multiple languages
3. Submit their solution & get instant feedback
4. View a live leaderboard for competitions.

out of scope

a. Auth & Profiles
b. Payment
c. analytics
d. social features

Non-functional:-

1. Availability >> Consistency
2. Isolation & Security when running User Code.
3. Low latency - return submission results < 5 seconds
4. Scale to support Competitions with 100,000 Users

out of scope

a. fault tolerant
b. Secure Purchases
c. CI/CD pipelines
d. Backups

Core Entity:-

1) Problem
2) Solution
3) Leaderboard

APIS:-

(1) GET /Problems ? Page=1 & limit=100 → Partial<Problem>[].

(2) GET /Problems/:id ? language = {language} → Problem

(3) POST /Problems/:id/submit → Submission

```
{ code: string,
  language: string
}
```

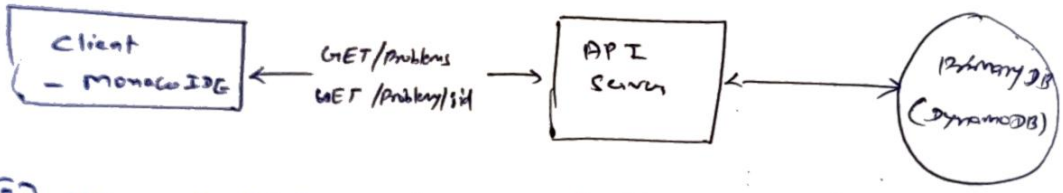
(4) GET /leaderboard /: CompetitionId ? Page = 1 && limit=100 → Leaderboard

Problem Schema-

```
{ id: string,
  title: string,
  Question: string,
  level: string,
  tags : string[]
  CodeStubs : {
        Python: string,
        Javascript: string,
        typescript: string,
        ---
  },
  testCases: {
        type : string,
        input: string,
        output string
  }[]
}
```

Design:

(1) User can view list of problems,
(2) Can view single problem & Code a solution :-



(3) Users should be able to submit their solution & get instant feedback.

⇒ Bad soln - Run Code in the API Server.

↳ Security issues
↳ Performance issue
↳ Isolation issue. (if code crash - it will take down server)

⇒ Good soln - Run Code in a virtual machine (VM)

Challenge → VMs are resource intensive & Can be slow to start up
→ Manage lifecycle of VM to ensure that You are not running out of resources or leaving VMs running when they are not needed which Can Prove Costly.

⇒ Great soln - Run Code in a Container (Docker)

↳ Similar to VMs but much more lightweight & faster to start up.

Challenge - Need to Properly Configure & Manage.

⇒ Great soln - Run Code in a Serverless function

↳ Serverless functions are small, Stateless, event-driven functions that run in response to triggers (eg. an HTTP request).

- serverless functions are managed by cloud Providers & can be automatically scale up & down based on demand.

- In this approach, we would create a serverless function for each runtime that we support (eg. python & Java, ...) that installs the necessary dependency & runs the code in a sandboxed environment.
when a User submits their code, we would trigger the appropriate serverless function to run the code & return the result.

Challenge → These functions have a cold start time → introduce latency for first request to a function.
→ Manage resource limits

(4) Users should be able to view a live leaderboard for Competitions.

Defining Competition as :-
* 90 min long
* 10 Problems
* Up to 100k Users
* Scoring is number of problems solved in 90 min. In case of tie, we will rank by the time it took to complete all problems. (starting from Competition start time)

In SQL DB Query -

```
Select userId, Count(*) as numSuccessfulSubmissions
from Submissions
where CompetitionId = : CompetitionId AND Passed = true
Group By UserId
ORDER By numSuccessfulSubmissions DESC
```

In a noSql db like DynamoDB, You would need to have the Partition Key be the CompetitionId. Then You would pull all items into memory & Group & Sort.

⇒ Potential Deep Dives :-

(1) How will the system support isolation & security when running User Code ?

- By running our code in an isolated container, we have already taken a big step towards ensuring security & isolation. But there are few things we will want to include in our container setup to further enhance security :-

  * Read only FileSystem
  * CPU & Memory Bound
  * Explicit Timeout
  * Limit Network Access
  * No system calls (seccomp)

(2) How would you make fetching the leaderboard more efficient ?

⇒ Bad soln - Polling with Database queries

⇒ Good soln - Caching with Periodic updates

⇒ Great soln - Redis sorted set with Periodic Polling

  - When a submission is processed, both the database and Redis sorted set are updated : clients poll the server every 5 seconds for leaderboard updates, & the server return the top N users from the Redis sorted set which is wicked fast & requires no expensive db queries.

(3) How would the system scale to support competitions with 100,000 users?

- The main concern here is that we get a sudden spike in Traffic, say from a competition or a popular problem, that could overwhelm the containers running user code. The reality is 100k is still not a lot of users, & our API server, via horizontal scaling should be able to handle this load without any issues.

⇒ Bad soln — vertical scaling

⇒ Great soln — Dynamic Horizontal scaling → we can horizontally scale each of the language specific containers to handle more submissions.

⇒ Great soln:- Horizontal scaling with Queue

↳ we can take same approach as above but add a queue b/w the API server & the containers. This will allow us to buffer submissions during peak times & ensure that we don't overwhelm the containers. We can use a managed queue service like SQS to handle this for us.

(4) How would the systems handle running test cases?

- "How would you take ~~test~~ test cases & run them against user code of any language?"

↓

You don't want to have to write a set of test cases for each problem in each language. That would be a nightmare to maintain. Instead you would write a single set of test cases per ~~problem~~ Problem which can be run against any language

⇒ To do this, you will need a standard way to serialize the input & output of each test case and a test harness for each language which can deserialize these inputs, pass them to the user's code & compare the output to the deserialized expected output.

Final HLD:-