NEWSFEED SYSTEM

M T W T F S S
Page No.:
Date:
YOUVA

STEP1 : CLARIFICATION OF REQUIREMENTS, ESTABLISH DESIGN SCOPE.

Important Features :-

User: Publish post
See his/her friends posts.
Follow/Unfollow
Like & Comment.

Clarifications :-

Posts sorted by reverse chronological order? Yes.
Posts has media? Yes.

* Functional Requirements :

→ Social Media Post - text, video, image
→ Follow/ Unfollow.
→ Newsfeed.
→ Like & Comment
→ Notifications

* Non-functional Requirements :

→ Availability 99.999%
→ Eventual Consistency
→ Latency
→ Scalability 500M DAU & 2B MAU
→ Extensibility
→ Usability
→ Fault tolerance

STEP 2.5 CAPACITY ESTIMATION ← for no of servers
cost management
decide type & specifications of all hardware.

→ DAU & MAU
→ Throughput
→ Storage
→ Memory → No. of servers.
→ Network

→ DAU & MAU

1B users | 500M DAU
2B MAU.

(Each user has 300 friends & follows 250 pages on average.)

→ Throughput

① write throughput
→ comments / likes
→ create posts
→ follow / unfollow

comments/likes activities : each user performs 3 activities/day

create posts : 10% of DAU post in a day.

follow/unfollow : 1 user follows 1 user per week.

② read throughput
a user reads.
→ 100 posts in a day.

500M users × 100
= 50B users read/day.

∴ 50B read requests/day.

→ storage

- posts
- follows
- activities (likes / comments)
- user metadata

① user metadata

50 KB / user.

1B × 50KB = 50 TB

② posts

50M posts / day

20% text — text – 100 KB

20% video — video – 20 MB

60% images — image – 0.5 MB.

Text storage = 50M × 0.2 × 100KB / day

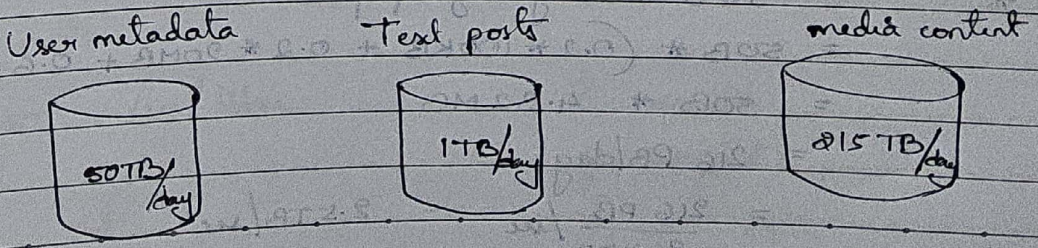
= 1 TB / day.

Image storage = 50M × 0.6 × 0.5 MB / day

= 15 TB / day.

Video storage = 50M × 0.2 × 20 MB / day

= 200 TB / day.



③ follows.

1 follow - 10B, 1 user follows 1 user per week.

(500M x 10B) / 7 per day.

$\left(\frac{500M \times 10B}{7} \times 365 \times 10\right)$ Bytes → for 10 years.

④ activities.

1 activity - 216 B, 3 activities per user per day.

(500M x 3 x 216 B x 365 x 10) Bytes → for 10 years.

→ Memory (cache)

cache = 1% of storage

0.01 x 216 TB/day.

= 2.16 TB/day.

→ Network & Bandwidth Estimation

216 TB/day - storage 50B read requests/day.

* $\frac{216TB}{86400}$ /sec. → ingress

= 2.5 GB/sec.

* 50B * avg. size of post

= 50B * (0.2 * 100KB + 0.2 * 20MB + 0.6 * 0.5MB)

= 50B * 4.32 MB

= 216 PB/day.

= $\frac{216\ PB}{86400}$ /sec = 2.5 TB/sec.

→ No. of servers.

Peak load 500M/sec.
1 server capacity 64000 RPS.
No. of servers = $\frac{500M}{64000}$
= 7812.5
≈ 8K servers.

STEP 3: API & HIGH LEVEL DESIGN.

* Databases
* Cache
* Blob storage.
* CDN.
* Load Balancers.

POST /v1/posts { userId:
mediaUrl:
text:
hashtags:
}

POST /v1/comments { userId:
postId:
comment:
}

POST /v1/likes { userId:
postId:
}

To update like/comment:

PATCH: /v1/posts/{postId} } extendable

body: like or comment data.

(or)

POST: /v1/comments and /v1/likes } scalable.

body: { }.

POST: /v1/follow.

body: {

"followerId":

"followeeId":

}

Read newsfeed:-

GET: /v1/feed {authToken}

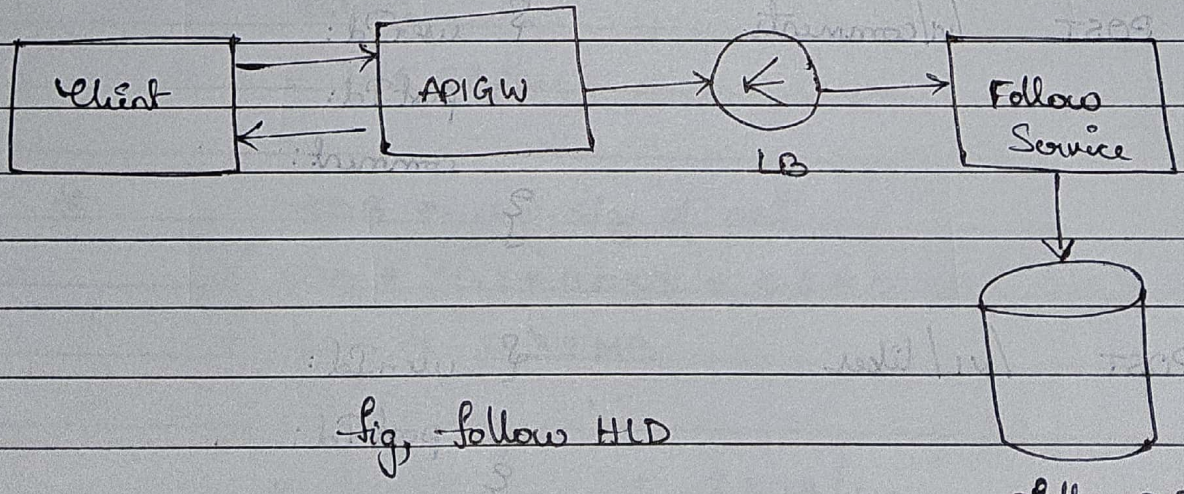
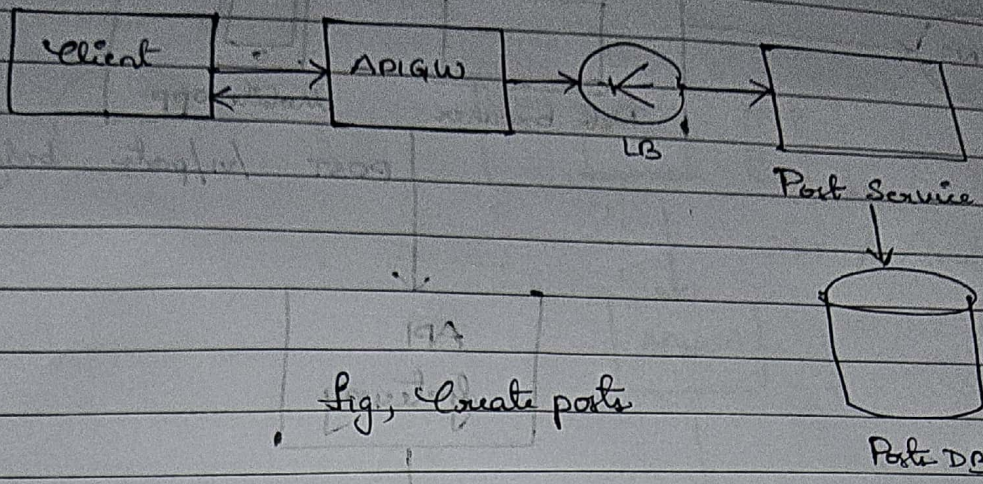
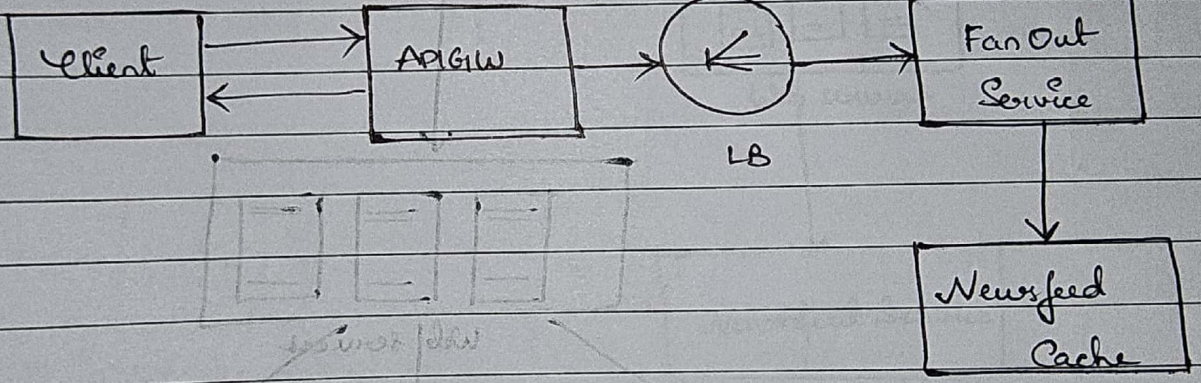


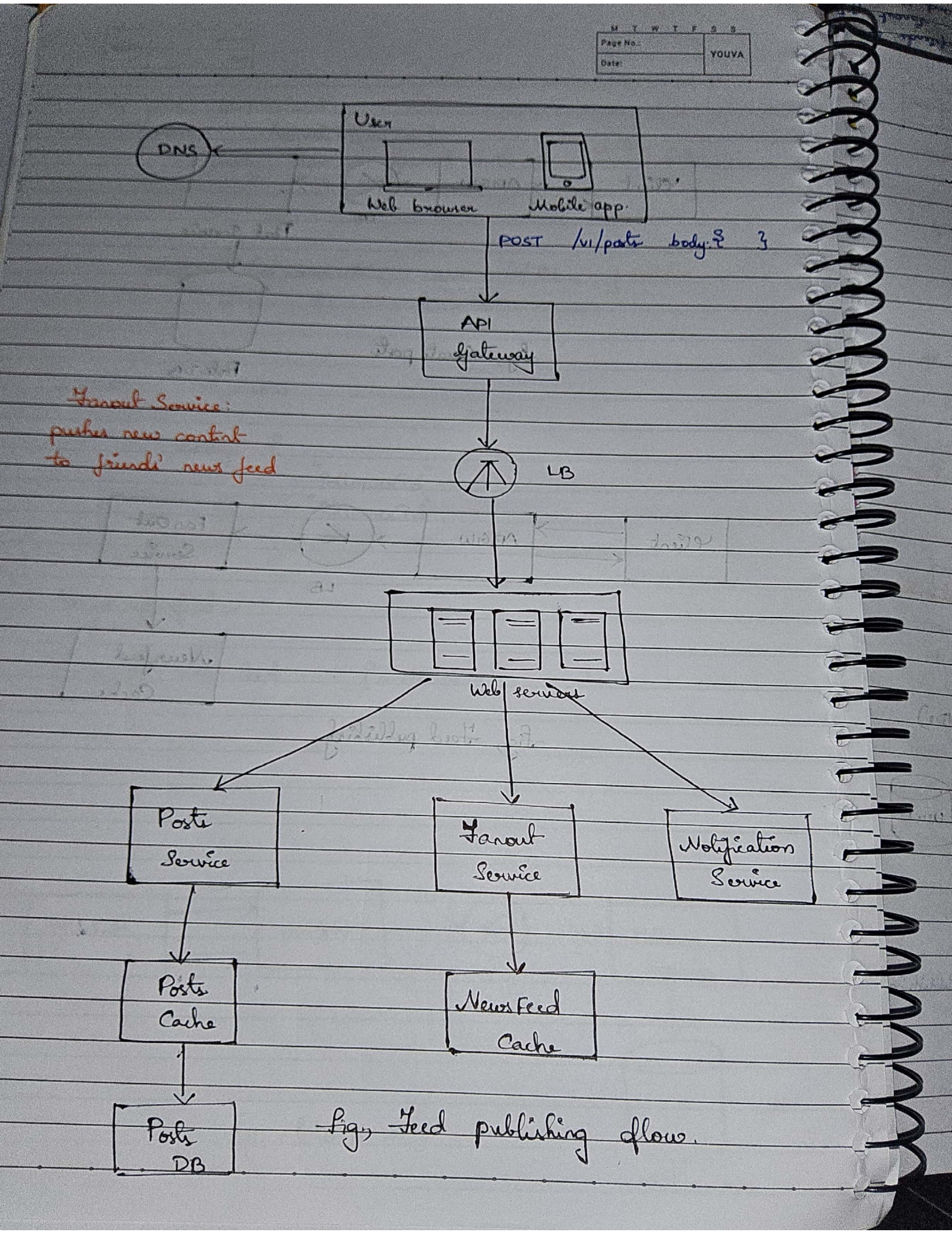
fig. follow HLD



Fig, Create posts



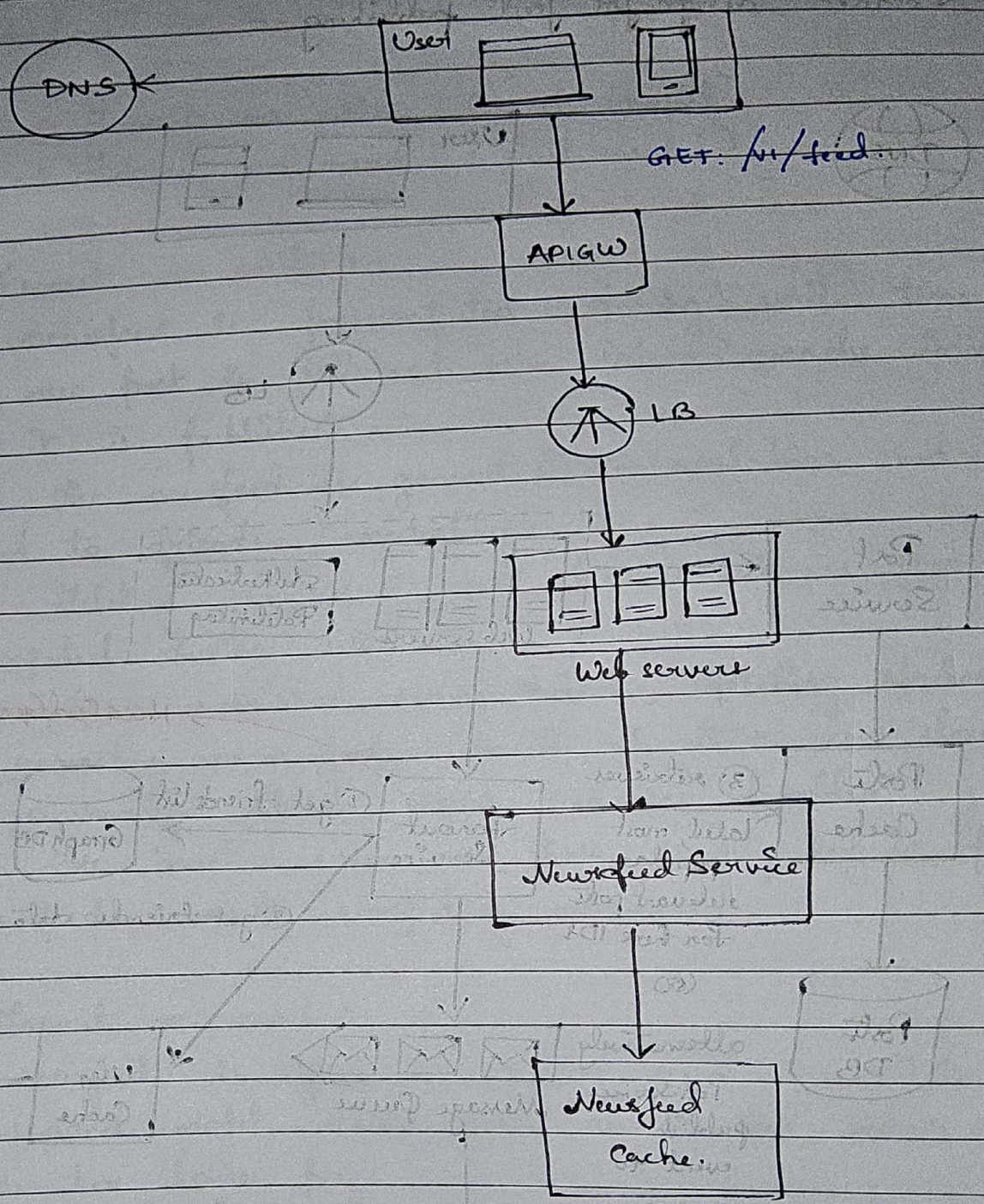
Fig, Feed publishing.



Fanout Service:
pushes new content to friends' news feed

Fig, Feed publishing flow.

Newsfeed Building :-



Fig, Newsfeed building

STEP 4 :- DESIGN DEEP DIVE

Detailed design for feed publishing :-

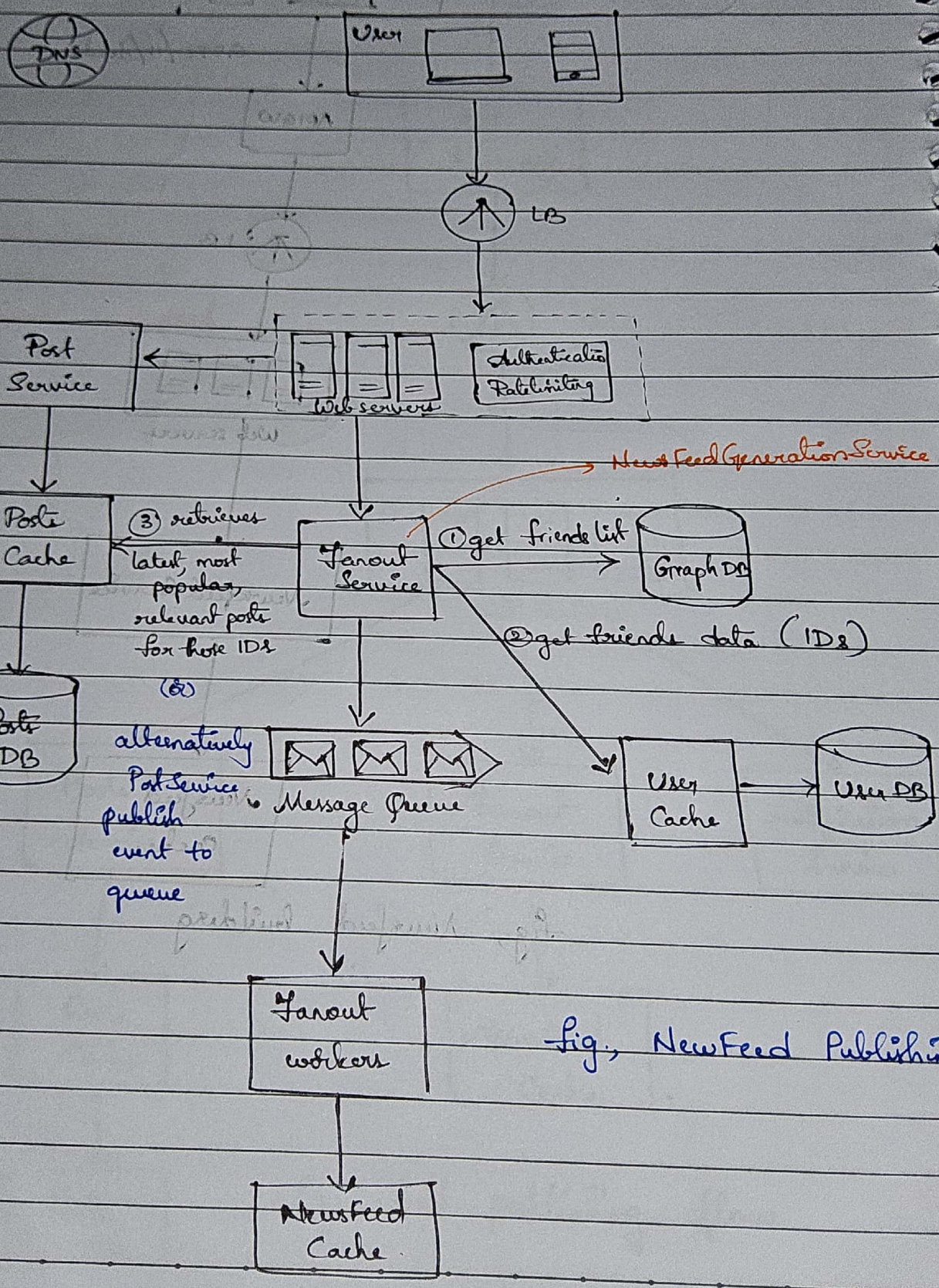


fig., Newsfeed Publishing

Fanout :-

Fanout is the process of delivering a post to all friends.

-> Fanout on write (push model)
-> Fanout on read (pull model)

* Fanout on write :-

newsfeed is precomputed during write time.
A new post is delivered to friends' cache immediately after it is published.

Pros: The newsfeed is generated in real-time and can be pushed to friends immediately.

Fetching newsfeed is fast because the newsfeed is precomputed during write time.

Cons: If a user has many friends, fetching friends list and generating news feeds for all of them are slow and time consuming. It is called hot-key problem.

For inactive users or those rarely log in, pre-computing news feeds waste computing resources.

* Fanout on read :-

newsfeed is generated during read time. This is an on-demand model. Recent posts are pulled when a user loads her home page.

Pros: For inactive users or those who rarely log in, fanout on read works better because it will not waste computing resources on them.

Data is not pushed to friends so there is no hot key problem.

Cons: Fetching the news feed is slow

We adopt a hybrid approach.

When a celebrity User Posts: Fanout-on-Read.

→ Post is stored in Posts DB.
→ Post is added to 'Global Cache' or 'CDN', so it can be fetched on demand.

→ Follower opens their feed - get normal posts from their newsfeed cache.
→ check if they follow any celebrity - fetch their posts from 'Global Cache'.
→ Merge both sets of posts dynamically.

When a normal user posts: Fanout-on-Write.

→ Post is stored in Posts DB.
→ Fetch the list of followers from Followers DB.
→ push post_id, user_id into newsfeedcache.
follower_id ~~fetched~~ from DB.
get from queue where post service publishes event
& get from posts cache.

→ Follower opens their feed - directly fetches posts from news feed cache.

Another option is while pushing we only push to online followers after a "normal" user publishes a post.

Flow of Newsfeed Publishing (see in fig.) :-

User posts. /v1/posts (POST)
Request flows through load balancers, webservers & goes to post service.
PostService saves the post in PostDB and updates the Post Cache.
post_id is generated.

{post_id, author_id} data flows to FanOutService (or Newsfeed Generator service)

Newsfeed Generator (or Fanout) service works as follows:

1. Fetch friend IDs from the graph database (FollowsDB).
2. Get friends info from the user cache. The system then filters out friends based on user settings.
For example, if you mute someone, his posts will not show up on your news feed even though you are still friends.
Another reason why posts may not show is that a user could selectively share information with specific friends or hide it from other people.
3. Send friends list and new post ID to the message queue.
4. Fanout workers fetch data from the message queue and store news feed data in the news feed cache.
5. Append each <post_id, user_id> to news feed cache.

post_id	user_id
post_id	user_id
post_id	user_id

Methods for sending post_id to Newsfeed Generator :-

① Synchronous API Call (Direct Approach)

Post Service — REST API call : or RPC (gRPC, Thrift) call → NFG Service

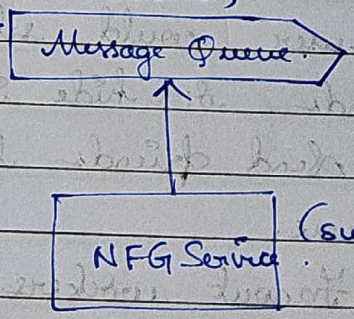
After post service stores the post in Post DB, returns post_id

Post service calls POST /newsfeed/fanout
body: {"post_id": "456", "author_id": "123"}

NFG service fetches followers by author_id & pushes {post_id, follower_id} to queue.

Cons:-
↳ Increases latency
↳ If Newsfeed generator fails, retries are needed.

② Asynchronous Event-Driven (Better for Scalability).



Pros:- Decoupled & Scalable.
Can handle failures via retries.

③ Database Polling (from Posts Cache).

Creates post:
user → POST /feed/posts : content
Post Service saves post in Post DB & generates post_id

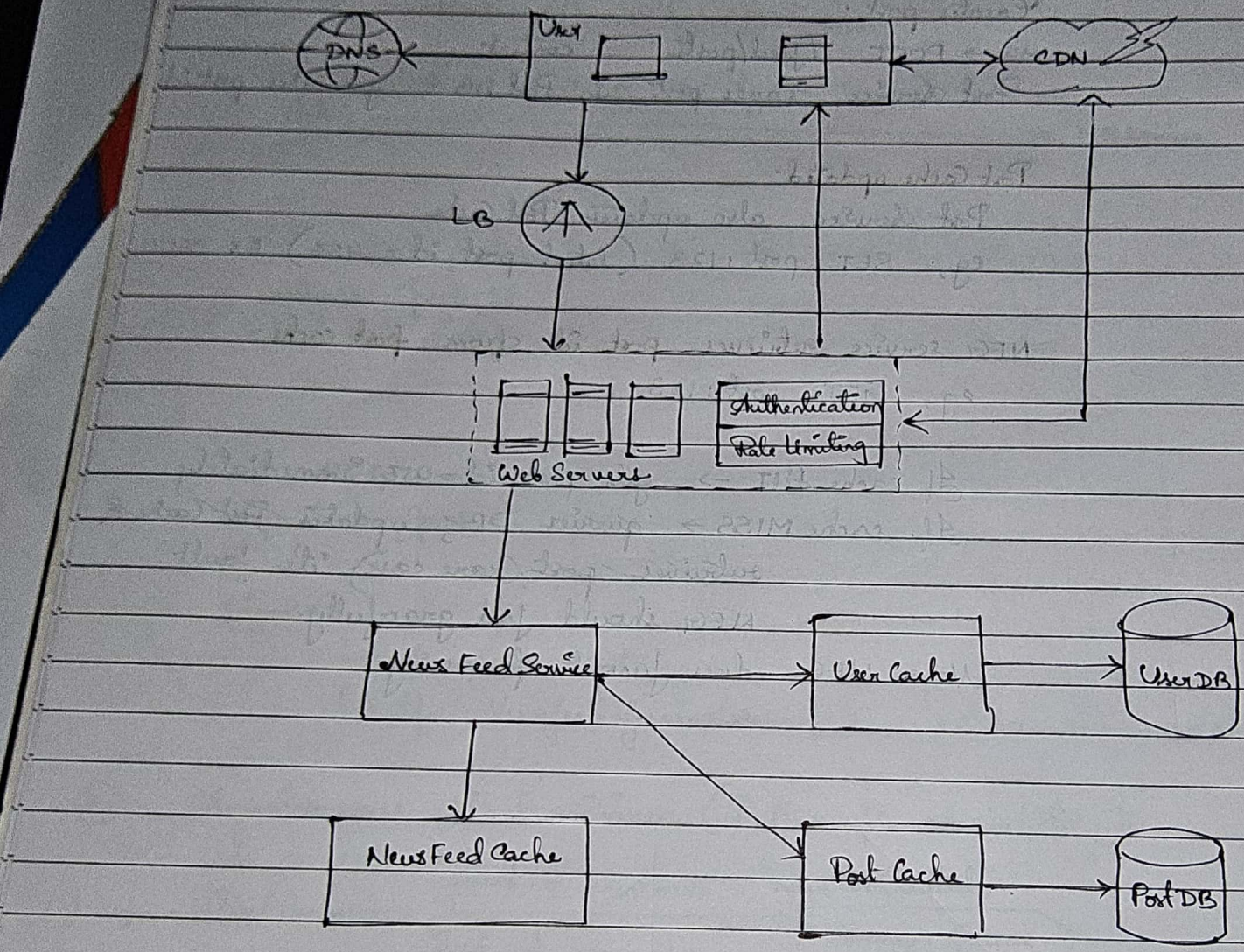
Post Cache updated:
Post Service also updates Post Cache
eg, SET post:123 (latest_post_id = 456) EX 3600

NFG service retrieves post_id from post cache:
eg, GET post:123

If cache HIT → gets post_id = 456 immediately
If cache MISS → queries DB, updates Post Cache & retrieves post (rare case) it's fault.
→ NFG should fail gracefully.
Now NFG does fanout processing.

Page No.: YOUVA
Date:

Detailed Design for Newsfeed retrieval:-



1. A user sends a request to retrieve her news feed. /v1/feed.
2. Load Balancer redistributes requests to web servers.
3. Web servers call the newsfeed service to fetch news feeds.
4. News feed service gets a list of postIds from news feed cache.
5. News feed service fetches posts related content & user content from User Cache, Post Cache to construct fully hydrated newsfeed.

6. The fully hydrated newsfeed is returned in JSON format back to the client for rendering.

What is the problem with generating a newsfeed upon a user's request? (also called live updates)

Suppose if our service receives billions of requests at once and the system starts generating live newsfeeds.

This situation would put an enormous number of users on hold and could crash servers.

To avoid this, we can assign dedicated servers that will continuously rank and generate newsfeeds and store them in the newsfeed database and memory.

Therefore, upon a request for new posts from a user, our system will provide the pre-generated feeds.

* The newsfeed ranking service :-

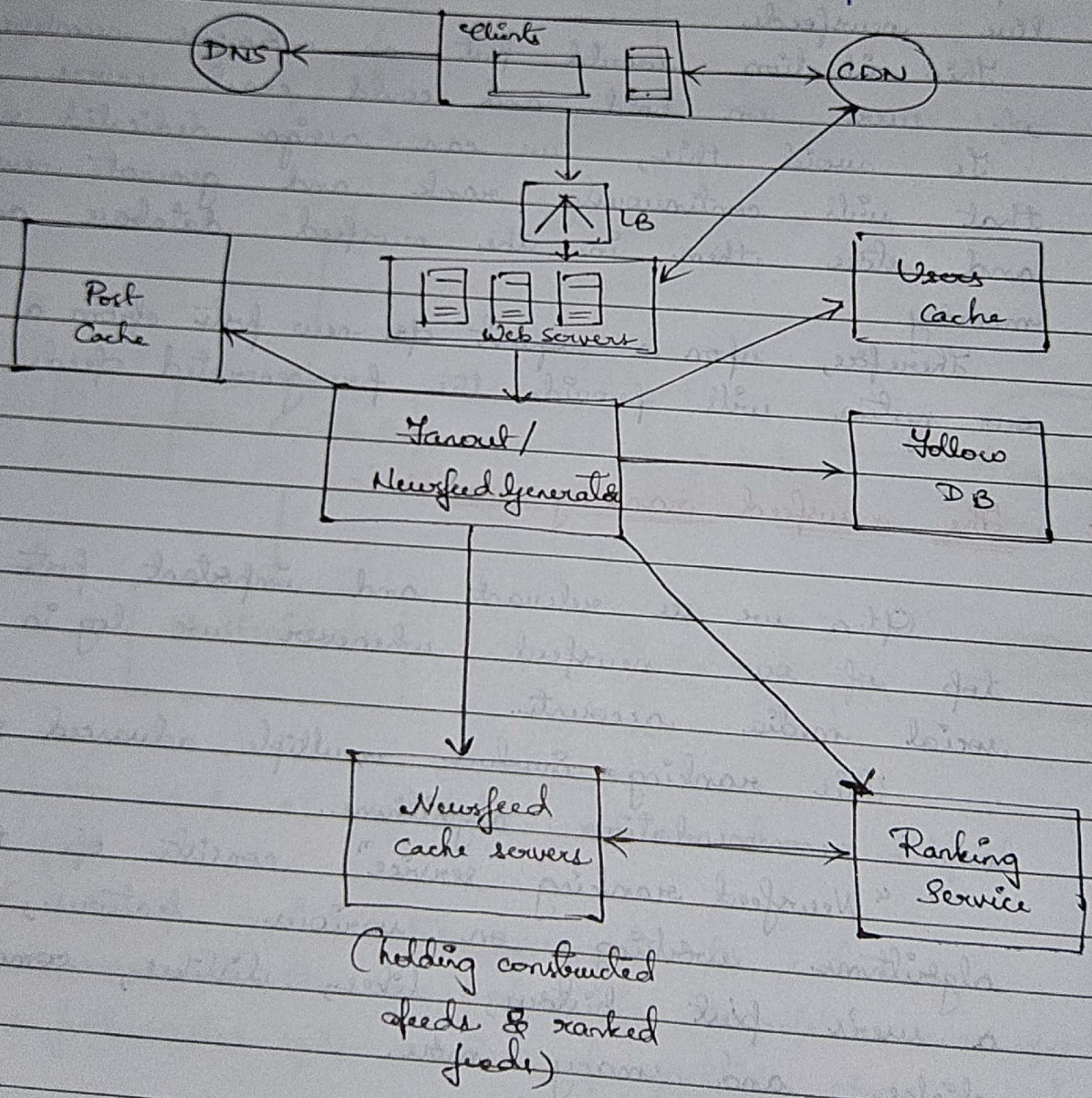
Often we see relevant and important posts on the top of our newsfeed whenever we log in to our social media accounts.

This ranking involves multiple advanced ranking and recommendation algorithms.

"Newsfeed ranking service" consists of these algorithms working on various features, such as, a user's past history, likes, dislikes, comments, clicks, and many more.

These algorithms also perform the following functions:

1. Select "candidate" posts to show in a newsfeed.
2. Eliminate posts including misinformation and clickbait from the candidate posts.
3. Create a list of friends a user frequently interacts with.
4. Choose topics on which a user spent more time.



Fig, Feed publishing with Ranking

* Posts ranking & Newsfeed Construction :-

Assume there are 10 posts in the database. published by 5 users whom Bob follows.

We aim to rank only 4 posts out of 10.

We follow below steps for ranking and creating a newsfeed for Bob:

(1) Various features such as likes, comments, shares, category, duration, etc and so on, are extracted from each post.

(2) Based on Bob's previous history, stored in user DB the relevance is calculated for each post via different ranking and machine learning algorithms.

(3) A relevance score (1 to 5) is assigned to each post.

(4) Top 4 out of 10 are selected based on the assigned scores.

(5) Top 4 posts are combined and presented on Bob's timeline in decreasing order of the score assigned.

* Uploading media :-

It is not efficient to send image/video in a request.

First client uploads an image to application, application stores it in an object storage and returns file location.

Client uses that location and sends in his request.

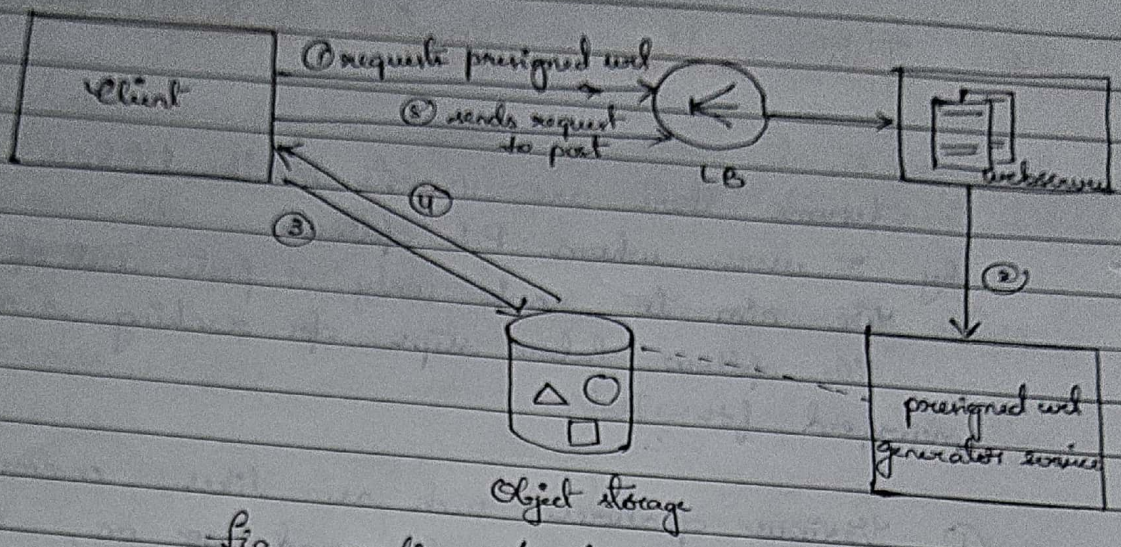
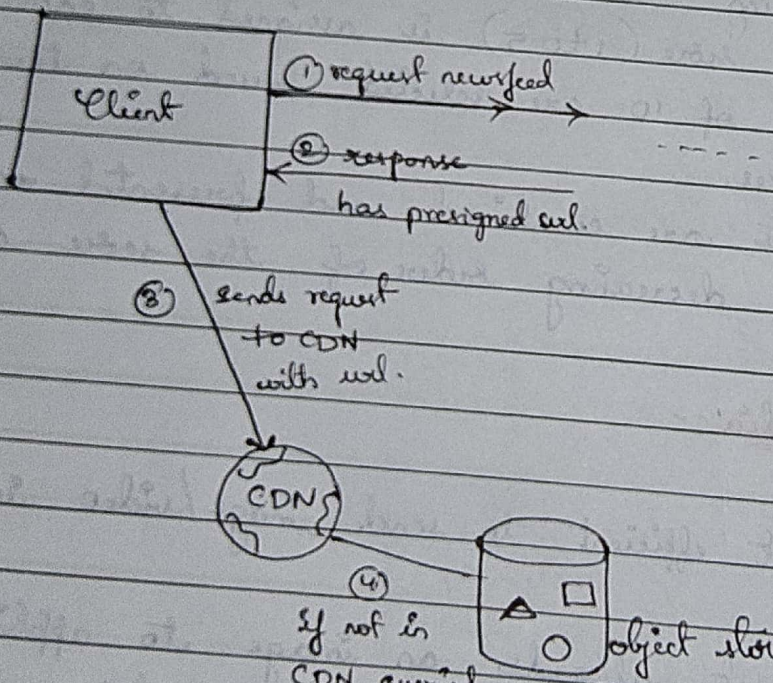


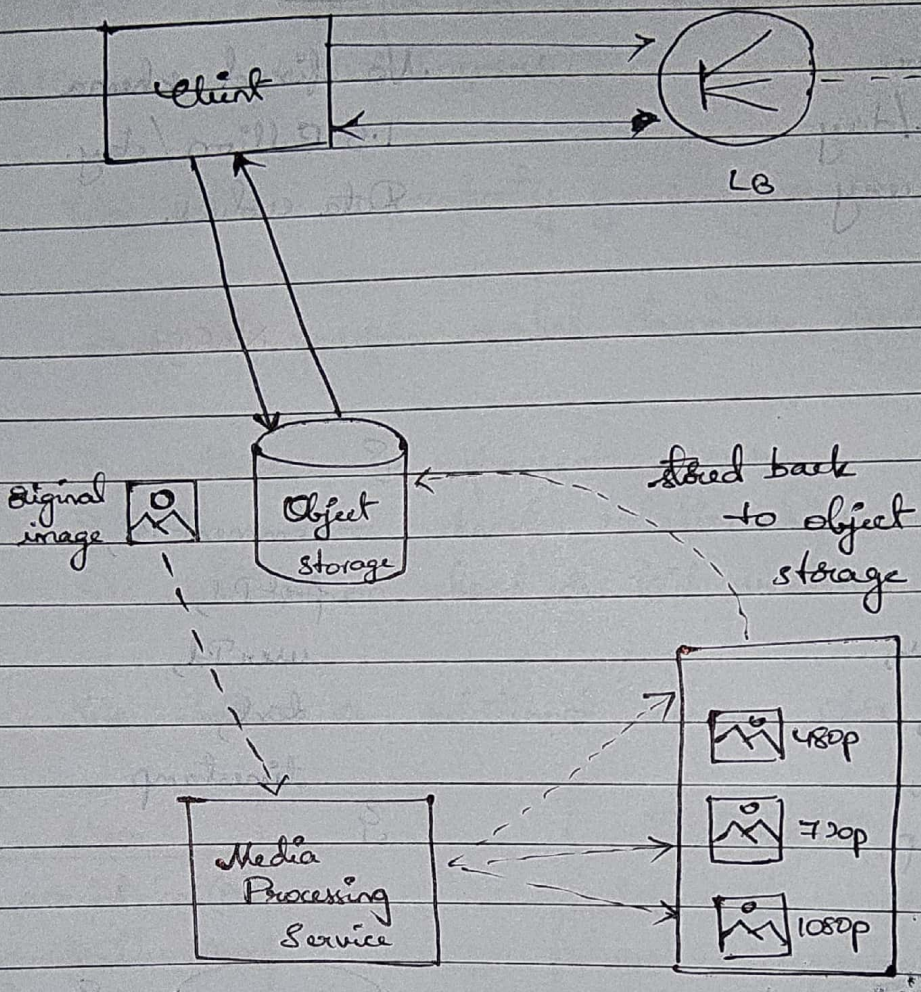
Fig., media upload.

Presigned url is a special url that allows client to upload file.



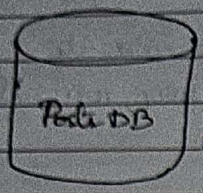
* Media Processing :-

Different formats are required mp4, mov, etc
Different resolutions are required 480p, 1080p etc.



* Database Selection :-

	SQL	NoSQL
(1) Fast Data Access		✓
(2) Scale is too large.		✓
(3) Data is in fixed structure	✓	
(4) Complex Queries	✓	
(5) Data changes frequently / evolves over time		✓



Text
Metadata
media urls etc.

No structure
50M posts/day
Simple query

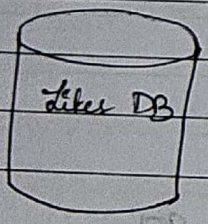
NoSQL

{
postId,
userId,
mediaUrls,
hashtags,
postText,
timestamp
}

No fixed schema
1.5 Billion/day.
Data evolves.

NoSQL

{
commentId,
postId,
userId,
text,
timestamp
}

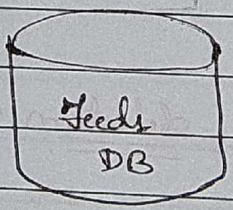


NoSQL

No fixed schema
1.5 Billion/day.
Data evolves.

{
likeId,
userId,
postId,
timestamp
}

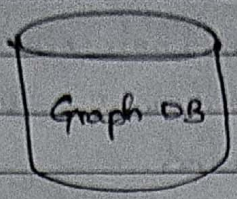
NoSQL



No structure
50M posts/day
Simple query pattern

{
~~userId~~,
feedItems: [
{
postId,
userId
}
...
]
}

Follow DB



We use graph DB to store relationships between followees and followers.

We follow property graph model

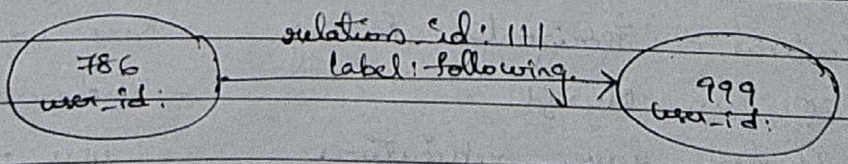
Property graph model describes graph by its

1) vertices — represent users.
2) edges — denote relationships between them.
3) indices on head & tail vertices

We follow a relational schema for the graph store

User
user_id: int (PK)
properties: JSON

Relationship
relation_id: int (PK)
from: int REFERENCES User (User_ID)
to: "
label: varchar (32)
properties: JSON.



relation_id	relation_from	relation_to	label	properties
111	786	999		& JSON formatted data

STEP 4: EVALUATION AND WRAP UP.

① Scalability:- loadbalancers, webservers and other relevant servers are added/removed on demand.

② Fault tolerance :- replication of data
redundant resources to handle failures
monitoring service to enhance reliability by continuously observing system health, detecting issues early, providing insights for optimization and assisting in timely incident response.

③ Availability :- highly available by
redundant servers
& replicating data
when a user gets disconnected due to some fault in the server, the session is re-created via a load balancer with a different server.
data is stored on different and redundant database clusters, which provides high availability and durability.

④ Low Latency :- we can minimize system's latency at various levels by:
- Geographically distributed servers & cache associated with them.
- Use CDN for frequently accessed news feeds and media content.

⑤ we can talk about
- Vertical & Horizontal Scaling
- SQL vs NoSQL
- Master-slave replication
- Read replicas
- Consistency models.
- Database sharding
- Keep web tier stateless
- Cache data as much as you can
- Support multiple DCs.
- Lose couple with msg q's
- Monitor key metrics - QPS at peak, latency