System Design: Instagram

1. Requirements

1. Functional Requirements

1. User should be able to upload/download/view the photos.
2. User can search based on photos.
3. User can follow other users
4. Users can see a newsfeed based on the photos uploaded by users they follow.

2. Non-functional Requirements

1. The system should be highly available.
2. The acceptable latency for photos generation is 200ms.
3. The system should be highly reliable, the uploaded photos should be always visible, they should never be lost

2. Capacity Estimation and Constraints

Instagram will be a read-heavy system, here we will focus on building a system that can retrieve photos quickly.

1. Traffic estimates:

Suppose we have 500M users with 100M active daily users. Let's assume 2 photos per user per day are uploaded on an average.
Hence, photos per second = (200 M * 2) / (24 * 60 * 60) = 200 * 10^6 / 86400 ~ = 200 * 10^6 / 10^5 = 2 * 10^3 photos per second
Let's assume that we show 20 pictures on the newsfeed for active users. Hence, the read traffic will be = 20 * 100 * 10^6 / (24 * 60 * 60) = 2 * 10^9 / 10^5 = 20 * 10^3 photos per second.

2. Disk Space:

200M photos are getting uploaded every day. Let's assume photo size is 500kb.
Hence, for each day we will need = 200 * 10^6 * 500 * 10^3 = 10^{14} bytes = 100 TB
For 10 years = 100 * 365 * 10 TB = 365 PB.
Hence, we will need 365 PB storage.

3. Bandwidth:

Incoming data (write): 500 * 10^3 bytes * 2000 = 10^9 bps = 1 gbps
Outgoing data (read): 500 * 10^3 bytes * 20000 = 10^{10} bps = 10 gbps

4. Memory:

For newsfeed cache: Using 80-20 principle, (20% users use 80% space),
Reading, 100M active users, 20% of them need caching.
= 100 * 10^6 * 0.2 * 20 (images shown per newsfeed per day) * 500 * 10^3 bytes (image size)
= 200 TB/ day is needed for cache

3. System APIs

We can have SOAP / REST APIs to expose functionality of a service.
Following can be the APIs for creating and deleting short-links.

1. viewPhoto(photoId, userId):

Parameters:
1. photoId (String): unique identifier of the photograph
2. userId (String): unique Id of the user wishing to view the photo

Returns: Photo if success else, error code

2. deletePhoto(photoId, userId):

Parameters:
1. photoId (String): unique identifier of the photograph
2. userId (String): unique Id of the user wishing to delete the photo

Returns: (String) Success or Failure message

3. uploadPhoto(userId, location, caption, cratedAt):

Parameters:
1. userId (String): unique Id of the user wishing to upload the photo
2. location (String): geolocation of the picture
3. caption (String): caption entered by the user
4. createdAt (timestamp): automatically entered by the system

Returns: (String) Success or Failure message

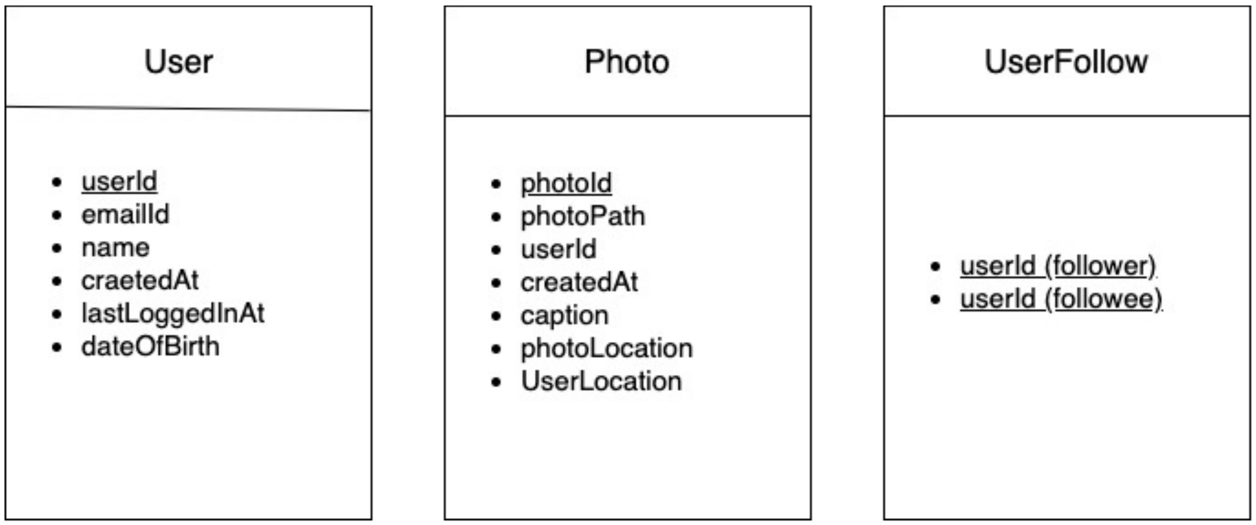
4. generateNewsFeed(userId):

Parameters:
1. userId (String): unique Id of the user wishing to generate the newsfeed

Returns: (String) Success or Failure message

4. Database Design:

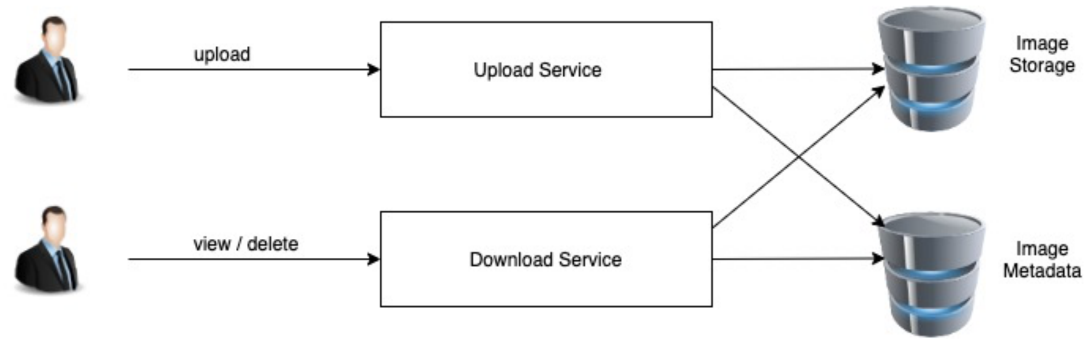
We need to store data about users, photos they have uploaded and people they follow. Looking at the data, we can use RDBMS here, but scaling will be a big issue here. Hence, we will choose a NoSQL database like dynamoDB here. Additionally, we can store photos in distributed storage such as S3. All the other data can be placed in a key-value pair in dynamoDB. The advantages of wide-column database should be taken in consideration.



Database size estimation:
For User table, for 500 M users, let the size of each row be 100 bytes, so total, $500 * 10^6 * 100$ bytes $= 5 * 10^{10}$ bytes = 50 GB
For Photo table, for each day, 200M photos are uploaded, assume 200 bytes of metadata per photo, hence, per day $= 200 * 10^6 * 200 = 40000 * 10^6 =$ 40 GB per day, for 10 years, we will need $= 40 * 10^9 *10 * 365 = 14600 * 10^{10} = 146 * 10^{12}$ bytes = 146 TB
For UserFollow table, each row will be, say, 8 bytes. Let's assume that each user follows 500 users, hence, for 500M $* 500 * 8 = 25 * 8 * 10^{10} = 2 * 10^{12}$ = 2 TB.

5. High Level Design

Photo uploads in our system will be slower as the photo will first have to go to disk, reads will be much faster (more if they are served from cache).
Upload can consume all the available connections as upload is a slow process. As the web servers have limited connections, if we want to remove this bottleneck, we can split this system into separate services. Hence, upload will not eat up entire system.



6. Low Level Design: Reliability and Redundancy

Our system should be highly reliable. Losing files is not an option for us. Hence, we need to store multiple copies of uploaded images on different servers. This is called as replication. Hence, if a server dies, we can still retrieve images from other servers.
This also applies to other services as well. We do not want a single point of failure in our system; hence, we replicate all servers, load balancers, services and databases. Redundancy, here, removes single point of failures in our systems.
If only one service is required at appoint of time, we can put duplicated services as standby for backup. i.e. if the primary services fail, these can take over.

7. Low Level Design: Data Sharding

1. Partitions based on userId: Suppose, we keep all users photos in same shard. If a partition of size 1 GB each, to store say 4.7 GB data, we will need 10 GB and with future scope, we dedicate 10 GB to a particular use. We can find shard number by doing userId % 10. To uniquely identify photos stored, we can generate auto-incrementing photo-id and appending shard-id along with it which will make it more unique.
   Problems:
   1. Hot-users: the data will not be balanced. non-uniform distribution of data
   2. Higher Latency
   3. In high-server load, for example if a user is a celebrity, will cause un-availability of the system.
2. Partitioning based on photoId: We can use autogenerating number service, can replicate it and also use load balancer in round-robin fashion to reduce response time and handle downtime. Or can go with Key Generating Service. We can save photos in shards by doing photoId % 10. By this we will achieve uniform distribution.

8. Low Level Design: Ranking and news feed generation

To generate newsfeed, we can use priorityQueue and store certain number, say, 20 images based upon users the user follows, the recency and likelihood. But generating this on the go (by using upload/ download service) will make this service latent and hence, we need another service which will take care of this periodically. This is the ranking algorithm for this system. We can save this pre-generated newsfeed for users in cache / newsfeed table and update as new pictures come in. Once user requests this, we can directly return the pre-generated newsfeed and if user asks for ore, the steps given above can be followed and with some latency, results can be returned.

9. Low Level Design: Sending notifications to users

1. Pull: Users request for new data manually or can be done periodically by server. Problem is: user will not be getting new data until requested or response will be empty if there is no new data.

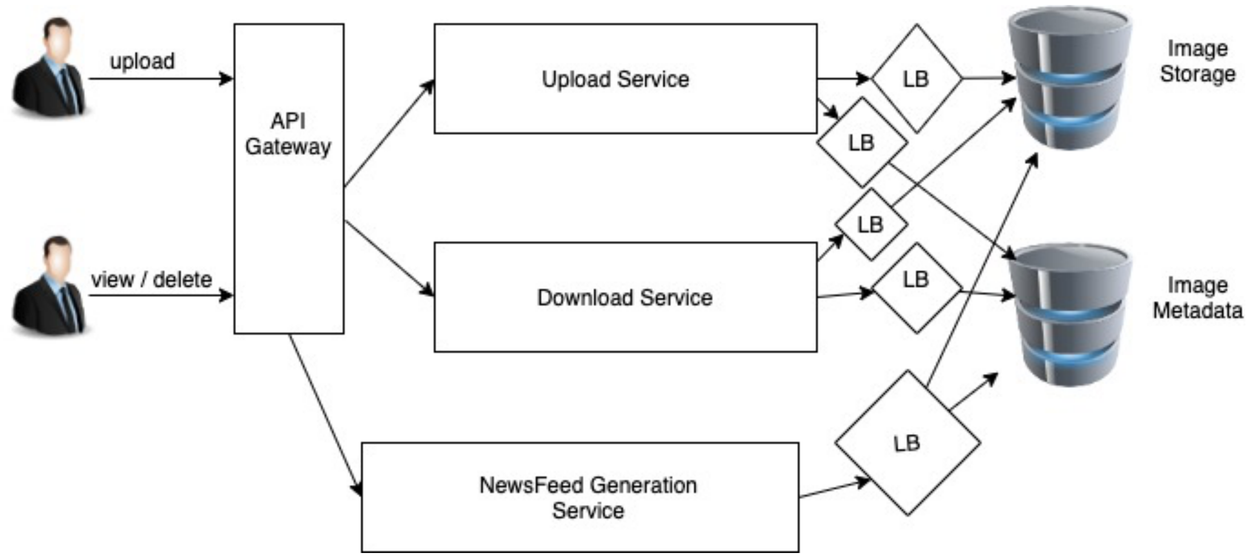
2. Push: Server sends notifications as soon as he sees new data for particular user. Inefficient if the user follows many people, then server will have to push requests quite frequently.
3. Hybrid: Most of the systems today use this technique. For example, we can send some part of users notifications, hence, server will not die and others can request info on the go.

10. Low Level Design: Cache and Load Balancing

Our system needs geographically distributed photo delivery system. Hence, we can give geographic specific servers called availability zones and have local cache with them. For metadata servers, LRU system along with caching hot database rows. Moreover, we can cache trending data daily in order to improve caching. Hence, we will also be balancing load based on geographic location.

11. Low Level Design: API Gateway

Rather than using load balancers, we can use API gateway here as we are separating upload and download service. API Gateway will not only balance the load but also route user requests to right service. It is also duplicated in order to avoid single point of failure.


upload
view / delete
API Gateway
Upload Service
Download Service
NewsFeed Generation Service
LB
LB
LB
LB
LB
Image Storage
Image Metadata