Design Instagram

1-Requirements and Goals of the System

Functional Requirements

Users should be able to upload/download/view photos

Search for images/videos using titles of the images/videos.

Users can follow other users.

User can generate newsfeed consisting of top photos from the people the user follow.

Non-functional Requirements

the application needs to be highly available

low lattency:200ms for News Feed generation.

favor (AP) so we can avoid a CP system, availability is preferred, so consistency can take a hit.

Reliability: our system cannot lose any photos. Highly reliable, videos/photos should never be lost.

2. Some Design Considerations

Read heavy because we store 1 phote and shared it multiple times.

100% reliable, no photo /video should be lost.

low latency viewing photos.

Eficient storage required.

3. Capacity Estimation and Constraints

Let's assume we have 500 millions users. with 1 million active users per day.

Average photo size = 200KB.

Traffic:

If we assume every user upload 2 photos, then we will have 2M photos per day, if we convert to seconds.

2 000 000/ 24 * 60 * 60= 2,000, 000/86400 = 2M/86400 approx. 23 photos per sec. (this is new photos write)

1M * 20(photos per newsfeed)= 20M per day = 230 per second (read)

Storage:

total space required per 1 day of photos, is 2 M x 200KB= 4M x 100= 400M KB. = 400 000 000 KB = 400 000 MB= 400GB.

now total space per 10 years is 400GB x 365 x 10 = aprox 1426 TB

Bandwidth:

incoming: 23 photos x 200kb = 4600KB = 4.6MBPS

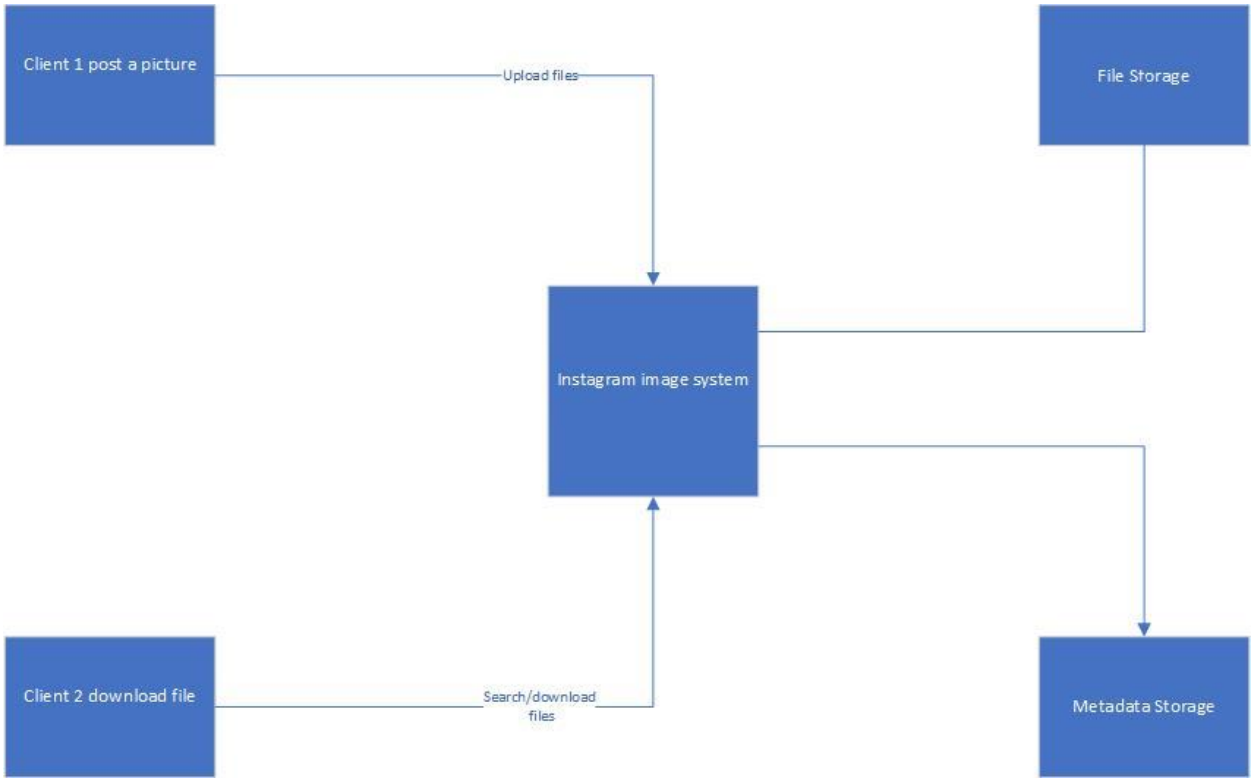
outgoing: 230 photos x 200kb = 46000KB=46MBPS

Cache for newsfeed (20% more viewed)= 0.2 * 20 M * 200KB approx. 800GB per day. GB/Day.

4. High Level System Design

At high level design we need storage service/block/object storage/S3 plus metadata storage(could be sql)

So we support two use cases upload images/videos and view images/videos



5. Database Schema

photos/videos Bytes stores in distributes file storage as HDFS or S3.

metadata can be store in Relational Databases such Oracle/MySql;however, there are problems with scalability so we will favor noSQL here:

tables: Photo/user/UserFollow.

User table:

UserID

Name

Email

DateOfBirth

CreationDate

LastLogin

primary key is UserID.

Photo table:

PhotoID

UserID

PhotoPath

PhotoLatitude

PhotLongitude

UserLatitude

UserLongitude

CreationDate

primary key is PhotoId and CreationDate because we need to retrieve photos per date, especially recent ones.

UserFollow Table:

UserID

FollowedID

7. Data Size Estimation

Estimate for USER table:

User: Assuming each “int” and “dateTime” is four bytes, each row in the User’s table will be of 68 bytes:

UserID (4 bytes) + Name (20 bytes) + Email (32 bytes) + DateOfBirth (4 bytes) + CreationDate (4 bytes) + LastLogin (4 bytes) = 68 bytes

If we have 500 million users, we will need 32GB of total storage.

500 million * 68 ~= 32GB

Estimation for Photo table:

Photo: Each row in Photo's table will be of 284 bytes:

PhotoID (4 bytes) + UserID (4 bytes) + PhotoPath (256 bytes) + PhotoLatitude (4 bytes) + PhotLongitude(4 bytes) + UserLatitude (4 bytes) + UserLongitude (4 bytes) + CreationDate (4 bytes) = 284 bytes

If 2M new photos get uploaded every day, we will need 0.5GB of storage for one day:

2M * 284 bytes ~= 0.5GB per day

For 10 years we will need 1.88TB of storage.

Estimation for UserFollow table:

Each row in the UserFollow table will consist of 8 bytes. If we have 500 million users and on average each user follows 500 users. We would need 1.82TB of storage for the UserFollow table:

500 million users * 500 followers * 8 bytes ~= 1.82TB

Total space required for all tables for 10 years will be 3.7TB:

32GB + 1.88TB + 1.82TB ~= 3.7TB

8-System APIs:

uploadImage(userID,apikey,urlToPhoto)

generateNewsfeed(userID)

viewImage(userID , photoId)

9. Component Design

if we create a monolithic application, we cannot scale the system correctly, especially in this case because we have writes that can overload the system blocking the reads,

So, we will need to separate writes from reads.

We need to consider that a server has a limit of connections, let us assume 250, so this means that we cannot server more than 250 users

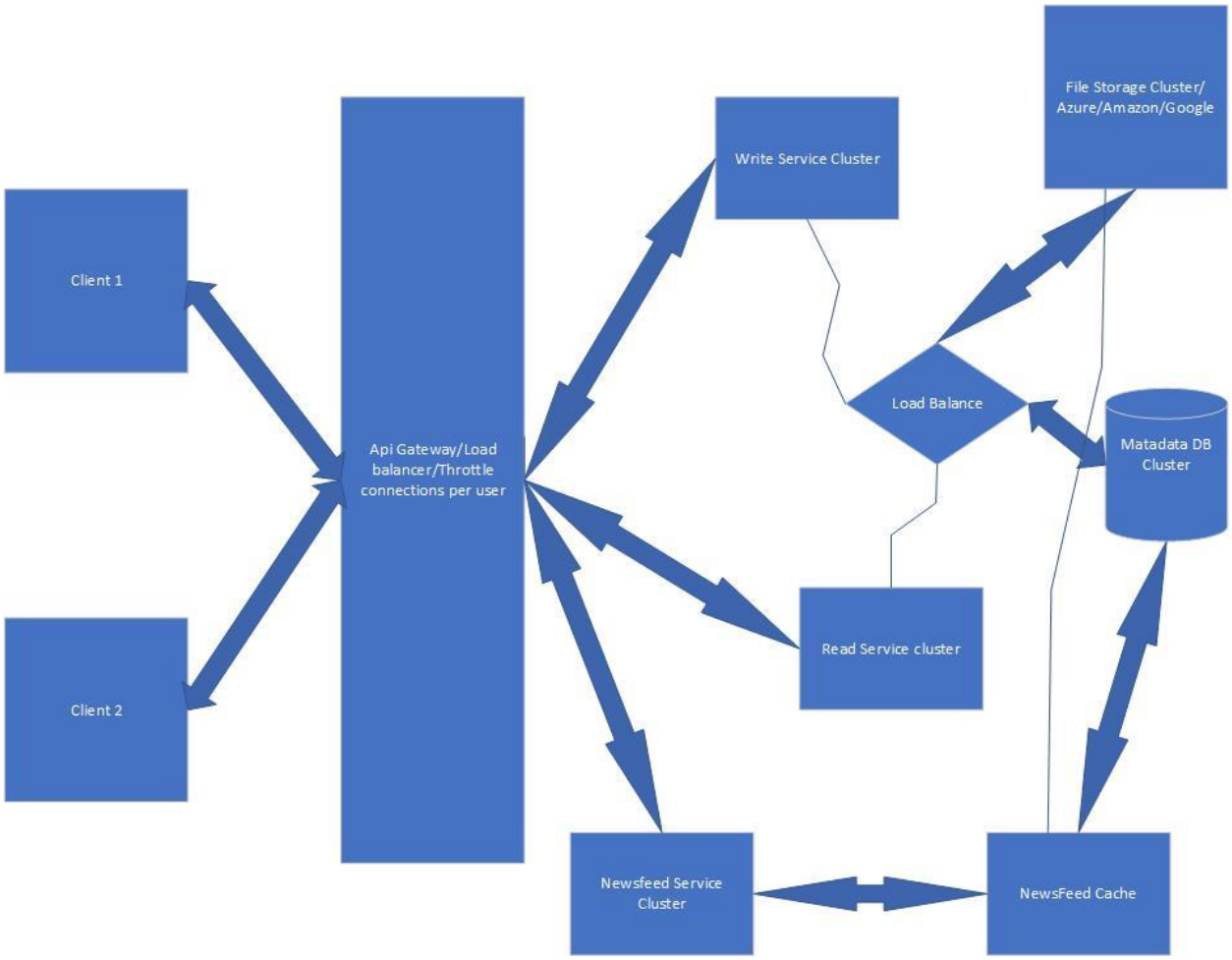
concurrently and we have 230 photos per sec for read and 23 for writes.

To handle this bottleneck, we can split reads and writes into separate services. We will have dedicated servers for reads

and different servers for writes to ensure that uploads do not hog the system.

Separating photos' read and write requests will also allow us to scale and optimize each of these operations independently.

Low level Design



10. Reliability and Redundancy

We will store multiple copies of each file so that if one storage server dies we can retrieve the photo from the other copy present on a different storage server.

This same principle also applies to other components of the system. If we want to have high availability of the system, we need to have multiple replicas of services running in the system, so that if a few services die down the system still remains available and running. Redundancy removes the single point of failure in the system.

Creating redundancy in a system can remove single points of failure and provide a backup or spare functionality if needed in a crisis

11. Trade off -- Data Sharding

If we partition based on email or user id, we will create an unbalance so it is better to generate a photoid and use it as a key for consistent hashing, we can use Keygeneration service or a sequence plus shard number to generate the key. With key generation service cluster we can avoid single point of failure.

12. **Ranking and News Feed Generation**

What are the different approaches for sending News Feed contents to the users?

1. Pull: Clients can pull the News Feed contents from the server on a regular basis or manually whenever they need it. Possible problems with this approach are a) New data might not be shown to the users until clients issue a pull request b) Most of the time pull requests will result in an empty response if there is no new data.

2. Push: Servers can push new data to the users as soon as it is available. To efficiently manage this, users have to maintain a Long Poll request with the server for receiving the updates. A possible problem with this approach is, a user who follows a lot of people or a celebrity user who has millions of followers; in this case, the server has to push updates quite frequently.

3. Hybrid: We can adopt a hybrid approach. We can move all the users who have a high number of follows to a pull-based model and only push data to those users who have a few hundred (or thousand) follows. Another approach could be that the server pushes updates to all the users not more than a certain frequency, letting users with a lot of follows/updates to regularly pull data.

13. **Cache and Load balancing**

We have a cache for metadata and for file storage, but in addition to that we should have a CDN service.

We will store 20% of the data per day using the 80/20 rule.