1. What is Instagram?

Instagram is a social networking service that enables its users to upload and share their photos and videos with other users. Instagram users can choose to share information either publicly or privately. Anything shared publicly can be seen by any other user, whereas privately shared content can only be accessed by the specified set of people.

We plan to design a simpler version of Instagram for this design problem, where a user can share photos and follow other users. The 'News Feed' for each user will consist of top photos of all the people the user follows.

2. Requirements and Goals of the System

We'll focus on the following set of requirements while designing Instagram:

Functional Requirements

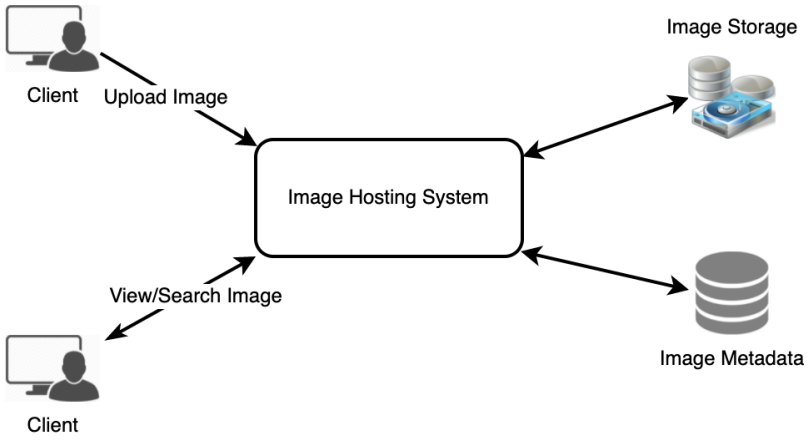
1. Users should be able to upload/download/view photos.
2. Users can perform searches based on photo/video titles.
3. Users can follow other users.
4. The system should generate and display a user's News Feed consisting of top photos from all the people the user follows.

Non-functional Requirements

1. Our service needs to be highly available.
2. The acceptable latency of the system is 200ms for News Feed generation.
3. Consistency can take a hit (in the interest of availability) if a user doesn't see a photo for a while; it should be fine.
4. The system should be highly reliable; any uploaded photo or video should never be lost.

3. High Level System Design

At a high-level, we need to support two scenarios, one to upload photos and the other to view/search photos. Our service would need some object storage servers to store photos and some database servers to store metadata information about the photos.



4. Database Schema

We need to store data about users, their uploaded photos, and the people they follow. The Photo table will store all data related to a photo; we need to have an index on (PhotoID, CreationDate) since we need to fetch recent photos first.

Photo	
PK	**PhotoID: int**
	UserID: int
	PhotoPath: varchar(256)
	PhotoLatitude: int
	PhotoLongitude: int
	UserLatitude: int
	UserLongitude: int
	CreationDate: datetime

User	
PK	**UserID: int**
	Name: varchar(20)
	Email: varchar(32)
	DateOfBirth: datetime
	CreationDate: datetime
	LastLogin: datetime

UserFollow	
PK	**UserID1: int** **UserID2: int**

A straightforward approach for storing the above schema would be to use an RDBMS like MySQL since we require joins. But relational databases come with their challenges, especially when we need to scale them.

We can store photos in a distributed file storage like HDFS or S3.

We can store the above schema in a distributed key-value store to enjoy the benefits offered by NoSQL. All the metadata related to photos can go to a table where the 'key' would be the 'PhotoID' and the 'value' would be an object containing PhotoLocation, UserLocation, CreationTimestamp, etc.

If we go with a NoSQL database, we need an additional table to store the relationships between users and photos to know who owns which photo. Let's call this table 'UserPhoto'. We also need to store the list of people a user follows. Let's call it 'UserFollow'. For both of these tables, we can use a wide-column datastore like Cassandra. For the 'UserPhoto' table, the 'key' would be 'UserID', and the 'value' would be the list of 'PhotoIDs' the user owns, stored in different columns. We will have a similar scheme for the 'UserFollow' table.

Cassandra or key-value stores, in general, always maintain a certain number of replicas to offer reliability. Also, in such data stores, deletes don't get applied instantly; data is retained for certain days (to support undeleting) before getting removed from the system permanently.

5. Data Size Estimation

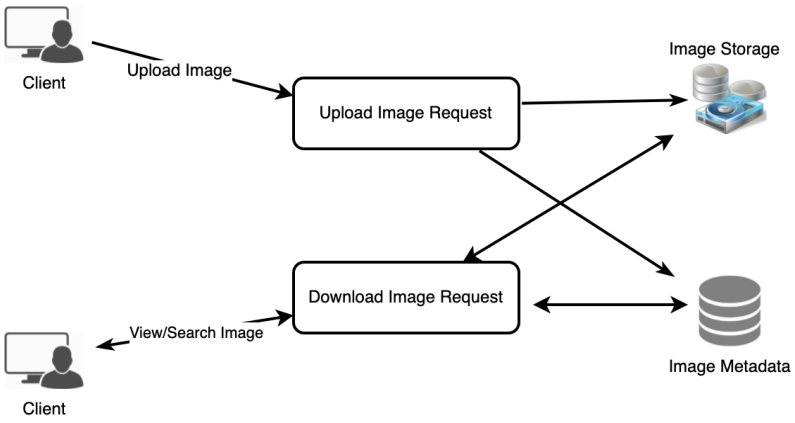
Let's estimate how much data will be going into each table and how much total storage we will need for 10 years. Total space required for all tables for 10 years will be 3.7TB:

6. Component Design

Photo uploads (or writes) can be slow as they have to go to the disk, whereas reads will be faster, especially if they are being served from cache.

Uploading users can consume all the available connections, as uploading is a slow process. This means that 'reads' cannot be served if the system gets busy with all the 'write' requests. We should keep in mind that web servers have a connection limit before designing our system. If we assume that a web server can have a maximum of 500 connections at any time, then it can't have more than 500 concurrent uploads or reads. 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 don't hog the system.

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



7. Reliability and Redundancy

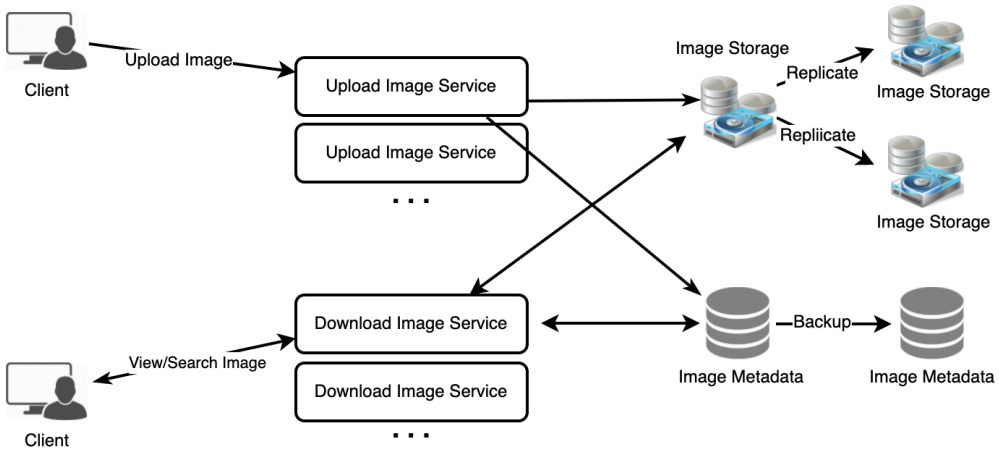
Losing files is not an option for our service. Therefore, we will store multiple copies of each file so that if one storage server dies, we can retrieve the photo from the other copy present on a different storage server.

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 even if a few services die down, the

system remains available and running. Redundancy removes the single point of failure in the system.

If only one instance of a service is required to run at any point, we can run a redundant secondary copy of the service that is not serving any traffic, but it can take control after the failover when the primary has a problem.

Creating redundancy in a system can remove single points of failure and provide a backup or spare functionality if needed in a crisis. For example, if there are two instances of the same service running in production and one fails or degrades, the system can failover to the healthy copy. Failover can happen automatically or require manual intervention.



8. Data Sharding

Let's discuss different schemes for metadata sharding:

a. Partitioning based on UserID Let's assume we shard based on the 'UserID' so that we can keep all photos of a user on the same shard. If one DB shard is 1TB, we will need four shards to store 3.7TB of data. Let's assume, for better performance and scalability, we keep 10 shards.

So we'll find the shard number by **UserID % 10** and then store the data there. To uniquely identify any photo in our system, we can append the shard number with each PhotoID.

How can we generate PhotoIDs? Each DB shard can have its own auto-increment sequence for PhotoIDs, and since we will append ShardID with each PhotoID, it will make it unique throughout our system.

What are the different issues with this partitioning scheme?

1. Some users will have a lot of photos compared to others, thus making a non-uniform distribution of storage.
2. What if we cannot store all pictures of a user on one shard? If we distribute photos of a user onto multiple shards, will it cause higher latencies?
3. Storing all photos of a user on one shard can cause issues like unavailability of all of the user's data if that shard is down or higher latency if it is serving high load etc.

b. Partitioning based on PhotoID If we can generate unique PhotoIDs first and then find a shard number through "PhotoID % 10", the above problems will have been solved. We would not need to append ShardID with PhotoID in this case, as PhotoID will itself be unique throughout the system.

How can we generate PhotoIDs? Here, we cannot have an auto-incrementing sequence in each shard to define PhotoID because we need to know PhotoID first to find the shard where it will be stored. One solution could be that we dedicate a separate database instance to generate auto-incrementing IDs. If our PhotoID can fit into 64 bits, we can define a table containing only a 64 bit ID field. So whenever we would like to add a photo in our system, we can insert a new row in this table and take that ID to be our PhotoID of the new photo.

Wouldn't this key generating DB be a single point of failure? Yes, it would be. A workaround for that could be to define two such databases, one generating even-numbered IDs and the other odd-numbered. For MySQL, the following script can define such sequences:

```
KeyGeneratingServer1:
auto-increment-increment = 2
auto-increment-offset = 1

KeyGeneratingServer2:
auto-increment-increment = 2
auto-increment-offset = 2
```

We can put a load balancer in front of both of these databases to round-robin between them and to deal with downtime. Both these servers could be out of sync, with one generating more keys than the other, but this will not cause any issue in our system.

Alternately, we can implement a 'key' generation scheme similar to what we have discussed in Designing a URL Shortening service like TinyURL.

9. Ranking and News Feed Generation

To create the News Feed for any given user, we need to fetch the latest, most popular, and relevant photos of the people the user follows.

For simplicity, let's assume we need to fetch the top 100 photos for a user's News Feed. Our application server will first get a list of people the user follows and then fetch metadata info of each user's latest 100 photos. In the final step, the server will submit all these photos to our ranking algorithm, which will determine the top 100 photos (based on recency, likeness, etc.) and return them to the user. A possible problem with this approach would be higher latency as we have to query multiple tables and perform sorting/merging/ranking on the results. To improve the efficiency, we can pre-generate the News Feed and store it in a separate table.

Pre-generating the News Feed: We can have dedicated servers that are continuously generating users' News Feeds and storing them in a 'UserNewsFeed' table. So whenever any user needs the latest photos for their News-Feed, we will simply query this table and return the results to the user.

Whenever these servers need to generate the News Feed of a user, they will first query the UserNewsFeed table to find the last time the News Feed was generated for that user. Then, new News-Feed data will be generated from that time onwards (following the steps mentioned above).

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 at a regular interval 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. 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 followers to a pull-based model and only push data to those who have a few hundred (or thousand) follows.

10. News Feed Creation with Sharded Data

One of the most important requirements to create the News Feed for any given user is to fetch the latest photos from all people the user follows. For this, we need to have a mechanism to sort photos on their time of creation. To efficiently do this, we can make photo creation time part of the PhotoID. As we

will have a primary index on PhotoID, it will be quite quick to find the latest PhotoIDs.

We can use epoch time for this. Let's say our PhotoID will have two parts; the first part will be representing epoch time, and the second part will be an auto-incrementing sequence. So to make a new PhotoID, we can take the current epoch time and append an auto-incrementing ID from our key-generating DB. We can figure out the shard number from this PhotoID (PhotoID % 10) and store the photo there.

11. Cache and Load balancing

Our service would need a massive-scale photo delivery system to serve globally distributed users. Our service should push its content closer to the user using a large number of geographically distributed photo cache servers and use CDNs (for details, see Caching).

We can introduce a cache for metadata servers to cache hot database rows. We can use Memcache to cache the data, and Application servers before hitting the database can quickly check if the cache has desired rows. Least Recently Used (LRU) can be a reasonable cache eviction policy for our system. Under this policy, we discard the least recently viewed row first.

How can we build a more intelligent cache? If we go with the eighty-twenty rule, i.e., 20% of daily read volume for photos is generating 80% of the traffic, which means that certain photos are so popular that most people read them. This dictates that we can try caching 20% of the daily read volume of photos and metadata.