Let's design a photo-sharing service like Instagram, where users can upload photos to share them with other users. Similar Services: Flickr, Picasa
Difficulty Level: Medium

1. Why Instagram?

Instagram is a social networking service, which enables its users to upload and share their pictures and videos with other users. Users can share either publicly or privately, as well as through a number of other social networking platforms, such as Facebook, Twitter, Flickr, and Tumblr.

For the sake of this exercise, we plan to design a simpler version of Instagram, where a user can share photos and can also follow other users. Timeline for each user will consist of top photos from all the people the user follows.

2. Requirements and Goals of the System

We will 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 be able to generate and display a user's timeline 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 timeline 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 photo/video uploaded should not be lost.

Not in scope: Adding tags to photos, searching photos on tags, commenting on photos, tagging users to photos, who to follow, suggestions, etc.

3. Some Design Considerations

The system would be read-heavy, so we will focus on building a system that can retrieve photos quickly.

1. Practically users can upload as many photos as they like. Efficient management of storage should be a crucial factor while designing this system.
2. Low latency is expected while reading images.
3. Data should be 100% reliable. If a user uploads an image, the system will guarantee that it will never be lost.

4. Capacity Estimation and Constraints

- Let's assume we have 300M total users, with 1M daily active users.
- 2M new photos every day, 23 new photos every second.
- Average photo file size => 200KB
- Total space required for 1 day of photos

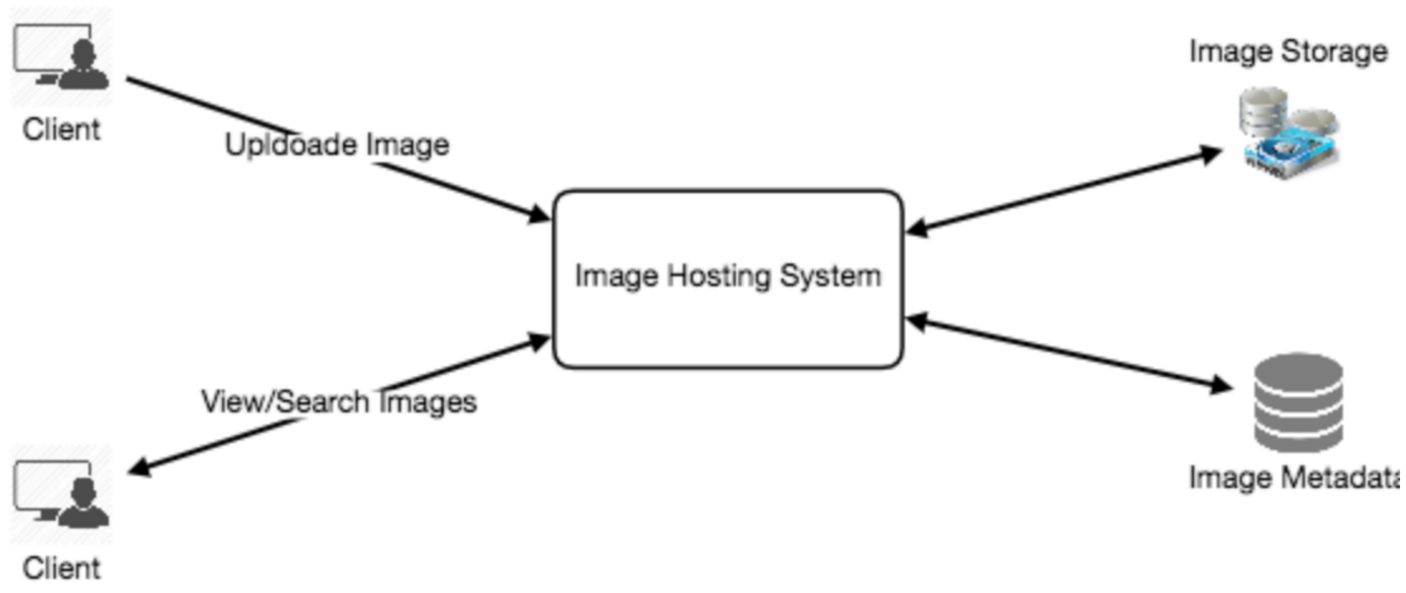
2M * 200KB => 400 GB

- Total space required for 5 years:

400GB * 365 (days a year) * 5 (years) ~= 712 TB

5. 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 block storage servers to store photos and also some database servers to store metadata information about the photos.



6. Database Schema

💡 ***Defining the DB schema in the early stages of the interview would help to understand the data flow among various components and later would guide towards the data partitioning.***

We need to store data about users, their uploaded photos, and people they follow. 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.

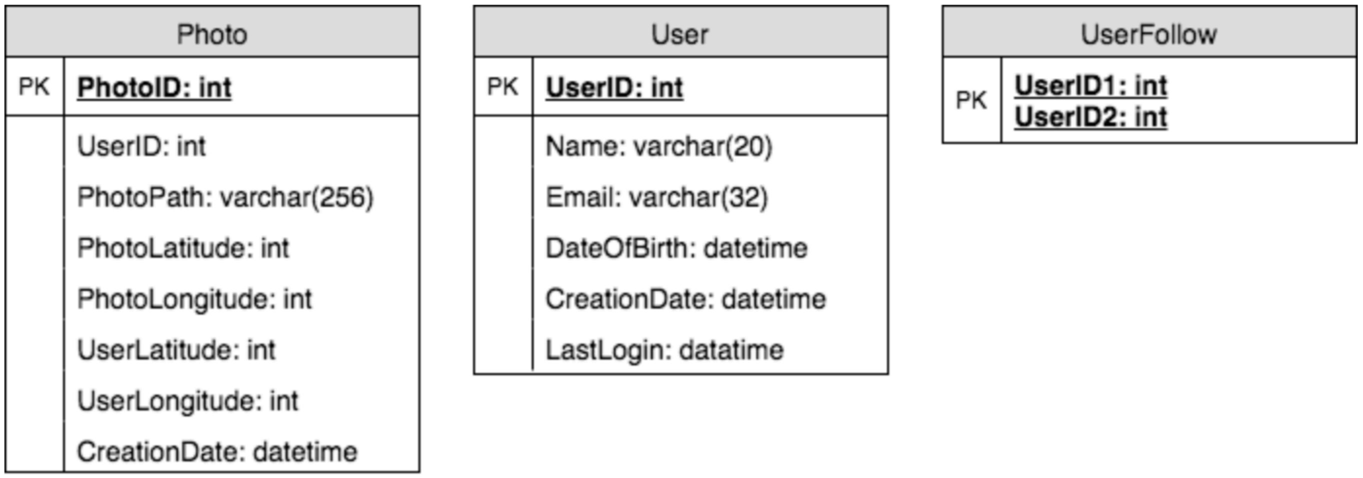


	Photo
PK	**PhotoID: int**
	UserID: int
	PhotoPath: varchar(256)
	PhotoLatitude: int
	PhotoLongitude: int
	UserLatitude: int
	UserLongitude: int
	CreationDate: datetime

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

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

One simple 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. For details, please take a look at SQL vs. NoSQL.

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 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.

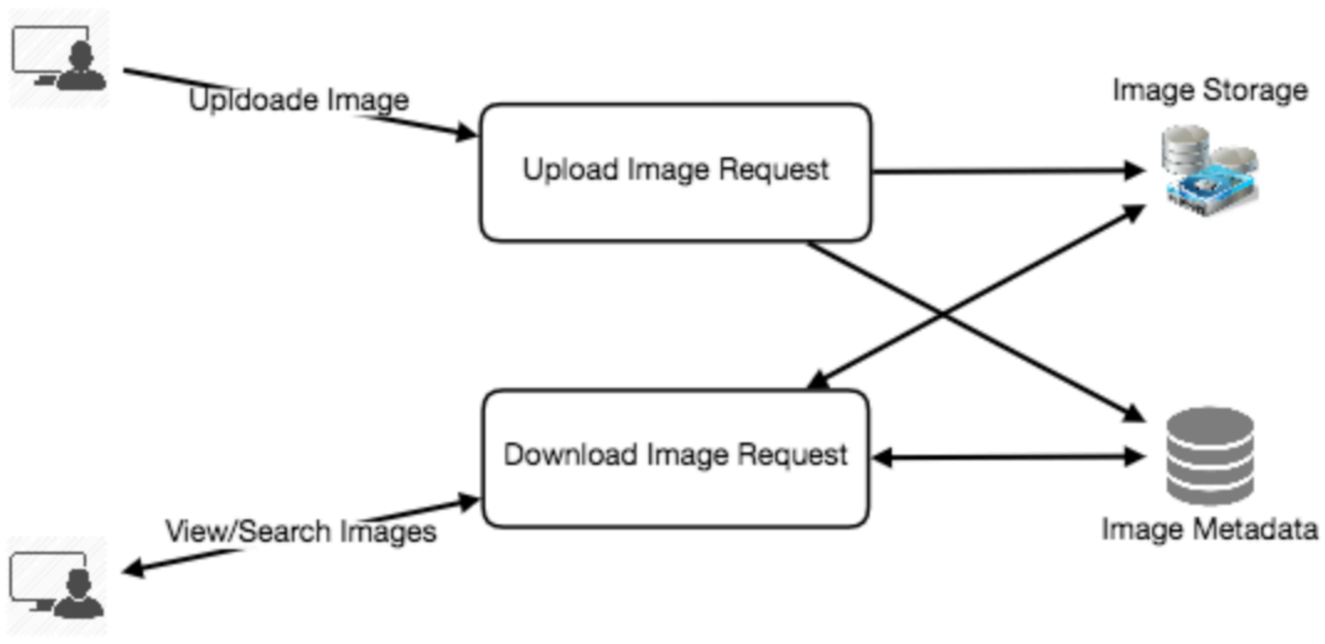
We also need to store relationships between users and photos, to know who owns which photo. Another relationship we would need to store is the list of people a user follows. For both of these tables, we can use a wide-column datastore like Cassandra. For the 'UserPhoto' table, the 'key' would be 'UserID' 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.

7. Component Design

Writes or photo uploads could be slow as they have to go to the disk, whereas reads could be faster, especially, if they are being served from cache.

Uploading users can consume all the connections, as uploading would be a slower process. This means reads cannot be served if the system gets busy with all the write requests. To handle this bottleneck we can split out read and writes into separate services. Since web servers have connection limit, we should keep this thing in mind before designing our system. Let's assume if a web server can have maximum 500 connections at any time, which means it can't have more than 500 concurrent uploads or reads. This guides us to have separate dedicated servers for reads and writes so that uploads don't hog the system.
Separating image read and write requests will also allow us to scale or optimize each of them independently.



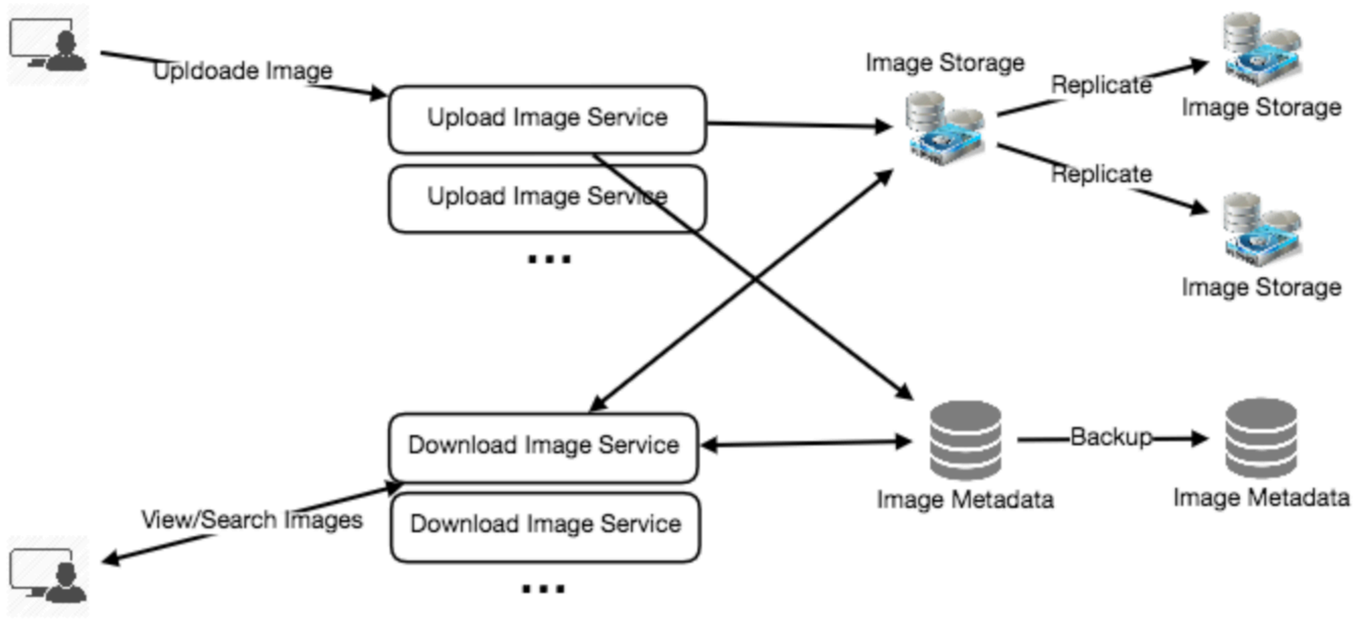
8. Reliability and Redundancy

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 image 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 is still available and serving. Redundancy removes the single point of failures in the system.

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

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.



9. 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 4TB, we will have 712/4 => 178 shards. Let’s assume for future growths we keep 200 shards.

So we will find the shard number by UserID % 200 and then store the data there. To uniquely identify any photo in our system, we can append 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.

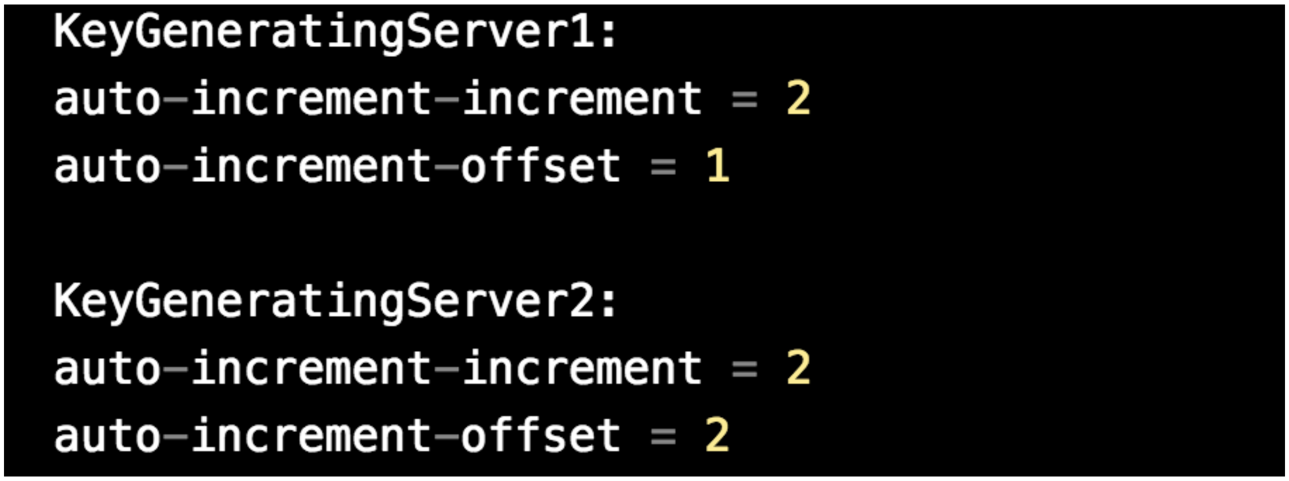
What are different issues with this partitioning scheme?

1. How would we handle hot users? Several people follow such hot users, and any photo they upload is seen by a lot of other people.
2. Some users will have a lot of photos compared to others, thus making a non-uniform distribution of storage.
3. 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?
4. Storing all pictures 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 shard number through PhotoID % 200, this can solve above problems. 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 since we need to have 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 will be. A workaround for that could be, we can define two such databases, with one generating even numbered IDs and the other odd numbered. For MySQL 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 down time. 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. We can extend this design by defining separate ID tables for Users, Photo-Comments or other objects present 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.

How can we plan for future growth of our system? We can have a large number of logical partitions to accommodate future data growth, such that, in the beginning, multiple logical partitions reside on a single physical database server. Since each database server can have multiple database instances on it, we can have separate databases for each logical partition on any server. So whenever we feel that a certain database server has a lot of data, we can migrate some logical partitions from it to another server. We can maintain a config file (or a separate database) that can map our logical partitions to database servers; this will enable us to move partitions around easily. Whenever we want to move a partition, we just have to update the config file to announce the change.

10. Ranking and Timeline Generation

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

For simplicity, let's assume we need to fetch top 100 photos for a user's timeline. Our application server will first get a list of people the user follows and then fetches metadata info of latest 100 photos from each user. 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.) to be returned 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 timeline and store it in a separate table.

Pre-generating the timeline: We can have dedicated servers that are continuously generating users' timelines and storing them in a 'UserTimeline' table. So whenever any user needs the latest photos for their timeline, we will simply query this table and return the results to the user.
Whenever these servers need to generate the timeline of a user, they will first query the UserTimeline table to see what was the last time the timeline was generated for that user. Then, new timeline data will be generated from that time onwards (following the abovementioned steps).

What are the different approaches for sending timeline data to the users?
1. Pull: Clients can pull the timeline data 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 when a user has a lot of follows 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 with high followings to 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.
For a detailed discussion about timeline generation, take a look at Designing Facebook's Newsfeed.

11. Timeline Creation with Sharded Data

To create the timeline for any given user, one of the most important requirements is to fetch latest photos from all people the user follows. For this, we need to have a mechanism to sort photos on their time of creation. This can be done efficiently if we can make photo creation time part of the PhotoID. Since we will have a primary index on PhotoID, it will be quite quick to find 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 seconds 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 shard number from this PhotoID (PhotoID % 200) and store the photo there.

What could be the size of our PhotoID? Let's say our epoch time starts today, how many bits we would need to store the number of seconds for next 50 years?
86400 sec/day * 365 (days a year) * 50 (years) => 1.6 billion seconds

We would need 31 bits to store this number. Since on the average, we are expecting 23 new photos per second; we can allocate 9 bits to store auto incremented sequence. So every second we can store (2^9 => 512) new photos. We can reset our auto incrementing sequence every second.
We will discuss more details about this technique under 'Data Sharding' in Designing Twitter.

12. Cache and Load balancing

To serve globally distributed users, our service needs a massive-scale photo delivery system. 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 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 more intelligent cache? If we go with 80-20 rule, i.e., 20% of daily read volume for photos is generating 80% of traffic which means that certain photos are so popular that the majority of people reads them. This dictates we can try caching 20% of daily read volume of photos and metadata.