Designing Instagram

Difficulty: Medium

Let's design a photo-sharing service like Instagram, where users can upload photos to share them with other users.

1. What is Instagram?

Instagram is a social networking service that enables its users to upload and shared 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.

Instagram also enables its users to share through many other social networking platforms, such as Facebook, Twitter, Flickr, and Tumblr.

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 that 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, and view photos
2. Users can perform searches based on photo and video titles
3. Users can follow other users
4. The system should generate and display a user's 'News Feed' consisting of the top photos from all the people that 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 the 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.

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

3. Some Design Considerations

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

- Practically, users can upload as many photos as they like; therefore, efficient management of storage should be a crucial factor in designing this system
- Low latency is expected while viewing photos
- Data should be 100% reliable — if a user uploads a photo, the system will guarantee that it will never be lost

4. Capacity Estimation and Constraints

Let's assume that we have 500 million total users, with 1 million daily active users.

Let's also assume that 2 million new photos are posted every day — this equates to 23 new photos per second.

The average photo file size is 200KB

So the total space required for 1 day of photos is:

$$2M \cdot 200 \text{ KB} = 400 \text{ GB}$$

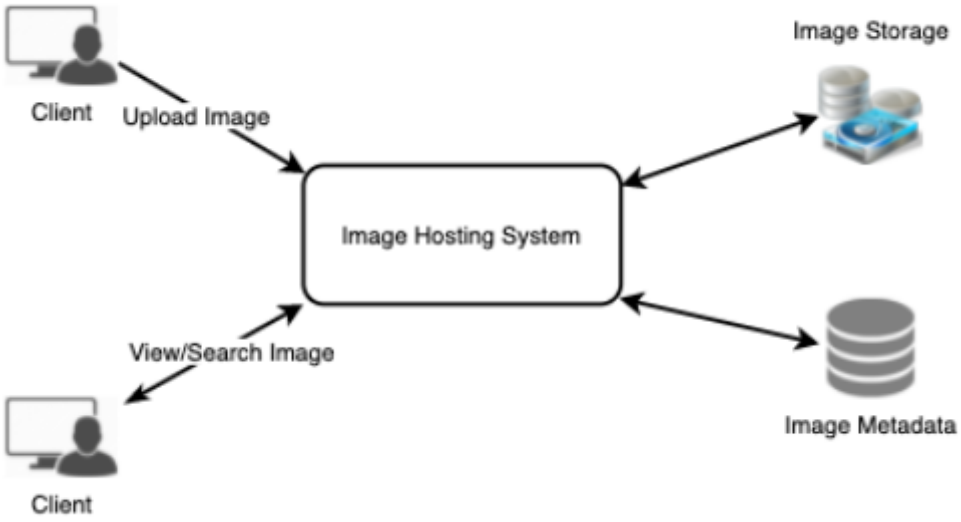
And the total space required for 10 years is then:

$$400 \text{ GB/day} \cdot 365 \text{ days/year} \cdot 10 \text{ years} \approx 1425 \text{ TB}$$

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 object storage servers to store photos, and some database servers to store metadata information about the photos.



6. Database Schema

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

- The Photo table will store all data related to a photo
  - We need to have an index on PhotoID and CreationDate since we need to fetch recent photos first
- The User table stores all the user data
  - It is indexed by UserID
- The UserFollow table stores all information about follower relationships
  - It is indexed by both FollowerID and FolloweeID together

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

User	
PK	**UserID: int**
	Name: varchar
	Email: varchar
	DateOfBirth: datetime
	CreationDate: datetime
	LastLogin: datatime

UserFollow	
PK	**FollowerID: int** **FolloweeID: int**

DB schema

A straightforward approach for storing the above schema would be to use an RDBMS like MySQL since we require joins.

But relational databases will be difficult to scale.

We can store our photo objects in a distributed file storage like HDFS or S3.

We can store the above scheme 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, CreationDate etc.

If we go with a NoSQL database, we will also need an additional table to store the relationships between users and photos to know who owns which photo. Let’s cal this UserPhoto.

We can also store the list of people a user follows in a table called 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 a list of PhotoIDs that the user owns, stored in different columns.

We 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 a certain number of days (to support undeleting), before getting removed from the system permanently.

7. 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.

User: Assuming each `int` and `dateTime` is 4 bytes, each row in the User's table will be 68 bytes:

$$\begin{aligned}&\text{UserID (4 bytes)} + \text{Name (20 bytes)} + \text{Email (32 bytes)} + \\ &\text{DateOfBirth (4 bytes)} + \text{CreationDate (4 bytes)} + \\ &\text{LastLogin (4 bytes)} = 68 \text{ bytes}\end{aligned}$$

Then, if we have 500 million users, we will need 32GB of total storage:

$$500M \cdot 68 \text{ bytes} \approx 32GB$$

Photo: Each row in the Photo table will be of 284 bytes:

$$\begin{aligned}&\text{PhotoID (4 bytes)} + \text{UserID (4 bytes)} + \text{PhotoPath (4 bytes)} + \\ &\text{PhotoLatitude (4 bytes)} + \text{PhotoLongitude (4 bytes)} + \\ &\text{UserLatitude (4 bytes)} + \text{UserLongitude (4 bytes)} + \\ &\text{CreationDate (4 bytes)} = 284 \text{ bytes}\end{aligned}$$

And if 2 million new photos get uploaded every day, we will need 0.5GB of storage for one day:

$$2M \cdot 284 \text{ bytes} \approx 0.5 \text{ GB/day}$$

And for 10 years this will be:

$$0.5 \text{ GB/day} \cdot 365 \text{ day/year} \cdot 10 \text{ year} \approx 1.88 \text{ TB}$$

UserFollow: Each row in the UserFollow table will consists of 8 bytes (a FollowerID and a FolloweeID). 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:

$$500 \text{ million users} \cdot 500 \text{ followers/user} \cdot 8 \text{ bytes} \approx 1.82 \text{ TB}$$

So the total space required for all tables for 10 years will be 3.7TB:

$$32 \text{ GB} + 1.88 \text{ TB} + 1.82 \text{ TB} \approx 3.7 \text{ TB}$$

8. Component Design

Photo uploads (or writes) can be slow as they have to go to 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.

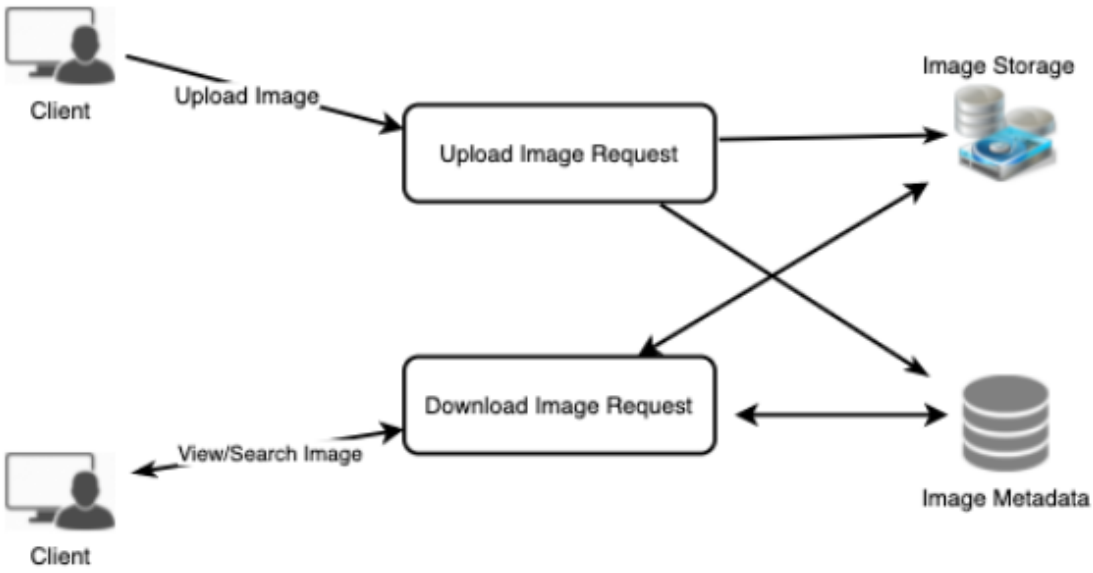
This means that reads cannot be served if the system gets busy with all of the write requests.

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 concurrent connections at any one time, then it cannot have more than 500 concurrent uploads and reads.

To handle this bottleneck, we can split reads and writes into separate services. We will have dedicated application server for reads and different servers for writes to ensure that uploads do not hog system resources.

Separating the photo read and write requests will also allow us to scale and optimise each of these operations independently.



9. 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 the other components of our system.

If we want the system to be highly available, 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.

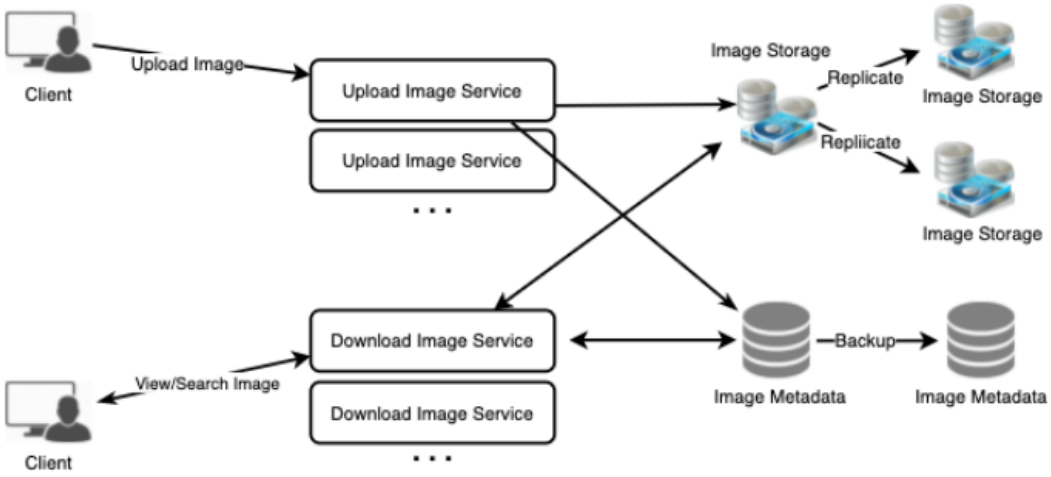
Redundancy removes the single point of failure in the system.

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 can remove single points of failure, and provide a backup or spare functionality if needed in a crisis.

For example, if there are two instance of the same service running in production and one fails or degrades, the system can failover to the healthy copy.

This failover can happen automatically or may require manual intervention.



10. Data Sharding

Let’s discuss different schemes for metadata sharding.

a. Partitioning based on UserID

Let’s assume that 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 that for better performance and scalability, we keep 10 shards.

So, we’ll find the shard number by $\text{UserID} \bmod 10$, 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 the ID unique throughout our system.

What are the different issues with this partitioning scheme?

1. How would we handle hot users? Several people follow such hot users, and a lot of other people see any photo that they upload.
2. Some users will have a lot of photos compared to others, thus leading to a non-uniform distribution of storage across shards.
3. What if we cannot store all the pictures of a user on one shard? If we distributed photos of a user onto multiple shards, will it cause higher latencies?
4. Storing all the 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 latencies if it is serving a high load etc.

b. Partitioning based on PhotoID

If we can generate unique PhotoIDs first, and then find a shard number through $\text{PhotoID} \bmod 10$, 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 (like a KGS) 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 to our system, we can insert a new row into 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 is.

A workaround for that could be to define two such databases, one that generates even-numbered IDs, and the other odd-numbered IDs.

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 of these servers could be out of sync, with one generating more keys than the other.

But this will not cause any issues in our system because the keys that they generate cannot cliide.

We can extend this design by defining separate ID tables for Users, Comments on Photos, or other objects present in our system.

Alternately, we can implement a key generation scheme (KGS) similarly to what we have discussed in *Designing a URL Shortening Service like TinyURL*.

How can we place for the 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 running on it, we can have separate databases for each logical partition on any server.

So whenever we feel that a particular 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 databases server, thereby enabling us to move partitions around easily.

Whenever we want to move a partitions, we only have to update the config file to announce the change.

11. 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 that the user follows.

For simplicity, let's assume that we need to fetch the top 100 photos for a user's News Feed.

Our application server will first get a list of people that 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 then return them to the user.

A large problem with this approach would be the high latency, as we have to query multiple tables and perform sorting, merging and ranking on the results.

To improve the efficiency, we could pre-generate the News Feed and store it in a separate table.

Pre-generating the News Feed

We can have dedicated server 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 that the News Feed was generated for that user.

Then, new News Feed data will be generated from that time onwards.

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.

   To efficiently manage this, users have to maintain a Long Poll request with the server for receiving the updates.

   A possible approach is a user who follows a lot of people, or a celebrity user that has millions of followers — in these cases, 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 that 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 and letting users with a lot of followers/updates to pull data regularly.

12. 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 that 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 a part of the PhotoID.

As we will have a primary index on PhotoID, this feature makes it quite quick to find the latest PhotoIDs.

We can use epoch time for this.

Let's say that 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 ($\text{PhotoID} \bmod 10$) and store the photo on that shard.

What could be the size of our PhotoID?

Let's say that our epoch time start today: how many bits would we need to store the number of seconds for the next 50 years?

$$86400 \text{ s/day} \cdot 365 \text{ days/year} \cdot 50 \text{ years} = 1.6 \text{ billion seconds}$$

$$\log_2 1.6 \text{ billion} \approx 31 \text{ bits}$$

We would need 31 bits to store this number.

Since, on average, we are expecting 23 new photos per second, we can allocate at least $\log_2 23 \approx 5 \text{ bits}$ for this auto-incrementing sequence.

We will allocate 9 bits to give us a full-byte number — this is more than we requires.

This means that every second we can store $2^9 = 512$ new photos.

Additionally, it means that we can reset our auto-incrementing sequence every second.

13. Cache and Load Balancing

Our service would need a massive-scale photo delivery system to server globally distributed users.

Our service should push its content closer to the user using a large number of geographically distributed photo cache serves and use CDNs.

We can introduce a cache for metadata servers to contain 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 the desired rows.

Least Recently Used (LRU) would be a reasonable cache eviction policy for our system — 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 — meaning 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.