Capacity Estimation and Constraints

Let's assume we have 500M total users, with 1M daily active users. 2M h d 23 h d

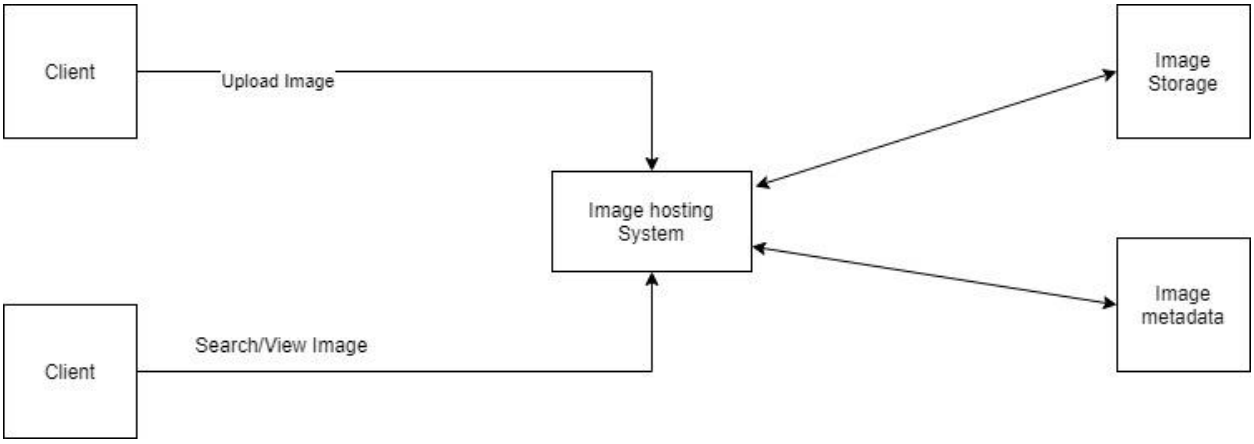
2M new photos every day, 23 new photos every second. Average photo file size => 200KB

Total space required for 1 day of photos

2M * 200KB => 400 GB Total space required for 10 years:

400GB * 365 (days a year) * 10 (years) ~= 1425TB

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 also some database servers to store metadata information about the photos.

SYSTEM API

Upload Photo (User_id , Photo_id)

User_id → Id of logged in user on Instagram

Photo_id → Photo to be uploaded

Search Photo(search title)

Search title → title of photo/video to be searched

Follow User(User_id ,username)

User_id → Id of user to be followed.

Username → name of user to be followed. If User_id is not available/follower does not have User_id, then username should be used for searching Followee.

DATABASE DESIGN

Nature of Data

- Require storage for billions of records
- Average Object Size(2-10 Mb depending upon the resolution)
- Service is read heavy and write heavy

User

Property	Type
User id	Varchar(100)
Username	Varchar(100)
Password	Varchar(100)
Email	Varchar(100)
DOB	Datetime
Last login	Datetime
Created date	Datetime

Photo

Photo Id	Int(primary key)
Photo Path	Varchar(256)
Photo description	Varchar(100)
Created date	Datetime

User Followers

Id	Int(primary key)
Follower Id	Int (Foreign key to User Table)
Followee ID	Int (Foreign Key to User Table)

User Photos

Unique id	Int(primary key)
Photo ID	Varchar(40) (Foreign key to photos table)
User ID	Int (Foreign Key to users table)

Although we can store mapping between **User** and **Photo** in **Photo** table, it is a bad DB design and violates DB Normalization principles. So, we have created new table **User Photos** which stores one to many mapping between **User** and **Photo**.

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 'Photo_id' and the 'value' would be an object containing Creation Timestamp, etc.

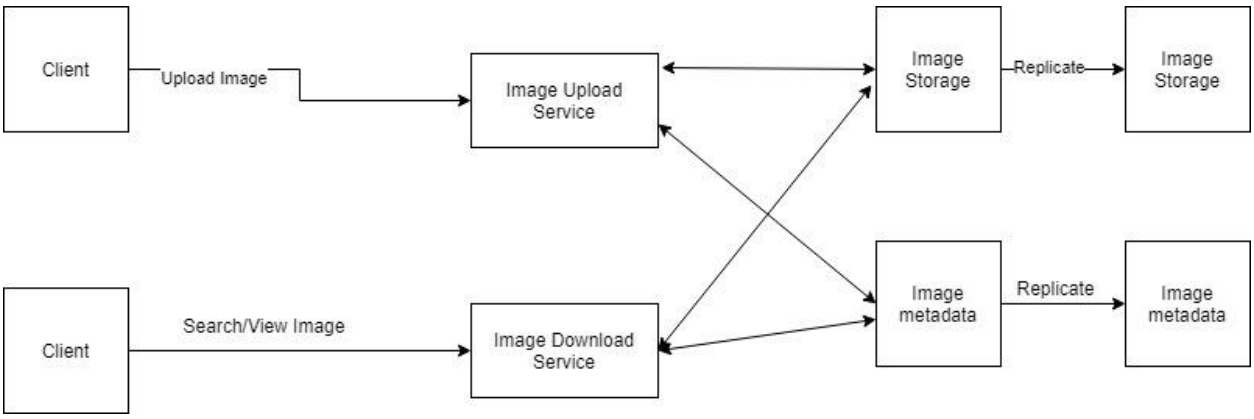
We need to store relationships between users and photos, to know who owns which photo. We also need to store the list of people a user follows. For both of these tables, we can use a wide-column datastore like Cassandra. For the 'User Photo' table, the 'key' would be 'User_id' and the 'value' would be the list of 'PhotoIDs' the user owns, stored in different columns. We will have a similar scheme for the 'User Follow' table.

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

Basic System Design and Algorithm

Our system is both read and right heavy. So we can have separate read and write servers for uploading and downloading/searching photos. It allows to scale system for million of users and improves the performance of system

Upload Photo



Download Image

Data Sharding

Let us discuss different schemes for metadata Sharding:

a. Partitioning based on User_id Let us assume we shard based on the 'User_id' 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 us assume for better performance and scalability we keep 10 shards.

So, we will find the shard number by User_id % 10 and then store the data there. To uniquely identify any photo in our system, we can append shard number with each Photo_id.

How can we generate Photo_id? Each DB shard can have its own auto-increment sequence for PhotoIDs and since we will append ShardID with each Photo_id, it will make it 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 they upload.
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 photos of a user on one shard can cause issues like unavailability of all the user's data if that shard is down or higher latency if it is serving high load etc.

b. Partitioning based on Photo_id If we can generate unique PhotoIDs first and then find a shard number through "Photo_id % 10", the above problems will have been solved. We would not need to append ShardID with Photo_id in this case as Photo_id 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 Photo_id because we need to know Photo_id 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 Photo_id 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 Photo_id of the new photo.

How can we plan growth of our system? We can have many 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 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 only must update the config file to announce the change.

News feed generation

We need to fetch latest, popular, and most relevant photos of user followers to create and show news feed for any user.

Our application server will first get a list of people the user follows and then fetch 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.) and return them to the user. A possible problem with this approach would be higher latency as we must 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 'User Newsfeed' 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 User News Feed 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

We have following approaches to send News Feed contents to the user

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 must 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.

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 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 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.

Cache and Load Balancing

Our service would need a massive-scale photo delivery system to serve the 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 Memcached 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.

Intelligent Cache

80-20 rule can help us build intelligent cache. It means 20% of read volume for photos is generating 80% of daily traffic which means certain photos are most popular and liked by majority people. So, we can cache these 20% of daily read of volume of data and photos to build intelligent cache.