**Designing Instagram**

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

Similar Services: Flickr, Picasa

## Requirements and Goals of the System

Plan for Minimum Viable Product in this discussion

Functional Requirements (also known as **FR** in **architect world**)

* **Upload/Download** of image
* **Search** of image
* Friends, **Follower & Followed** relationship social graph
* **Newsfeed** User page

Non-functional Requirements **(NFR)**

* HA (**High Availability of 5 9’s) – 99.999% (Customer have negligible unplanned downtime)**
* The **acceptable latency** of the system is **200ms** for News Feed generation.
* **Eventual consistency** (but there must be some timeline followed)
* **Reliability** – Opposite to vulnerability. Any system of the world is full proof, so this is the one area which is open for regular checks and upgrade according to the identified loopholes in system time to time.

**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.

* [Scale of system] Practically, users can upload as many photos as they like; therefore, efficient management of storage should be a crucial factor in designing this system
* [Latency] Low latency is expected while viewing photos.
* [ Reliability] 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 (Back Envelop calculation)

* Let’s assume we have 500M total users, with 1M daily active users.
* Every active user average upload 02 photo daily, it means average 2M photo upload per day
* Also assume that average size of each photo/video is roughly 200KB

So, storage required for a day will be

2M \* 200 KB = (2\*10^6) \* (200) KB

= 400 \* 10^6 KB

= 400 \* 10^3 MB

= 400 GB

Now on same rate if I must calculate the system storage requirement for 10 years

= 10 \* 365 \* 400 GB

= 4000 \* 400 GB (Approximately)

**= 1600 TB (Approximately)**

## 5. High Level System Design

At very high-level planning, Minimum viable architecture are like below?

Diagram

Description automatically generated

At a high-level, we need to support two scenarios, one to upload photos/video and the other to view/search photos/video.

Our service would need some [object storage](https://en.wikipedia.org/wiki/Object_storage) servers to store photos/videosand some database servers to store metadata information about the photos/videos.

## 6. Database Schema

Before starting any further level of breakup in system architecture, let’s design few DB schema for understanding the data flow requirement across the components.

Table

Description automatically generatedTable

Description automatically generated with low confidenceGraphical user interface, text, application

Description automatically generated

It will work well until you won’t require cases like below

Suppose you want to know which photo is owned by which user with their information.

Now, to achieve the above-mentioned case, you must use join in-case you are using SQL DB and if you are using the Key Value pair than you must create one more mapping where you will have mapping of Photo vs User.

We can store photos in a distributed file storage like [HDFS](https://en.wikipedia.org/wiki/Apache_Hadoop) or [S3](https://en.wikipedia.org/wiki/Amazon_S3).

For Meta data of photo can be stored in SQL or NoSQL DB, but why to choose one over other?

Let’s discuss:

We want NFR in our system like,

* **High availability over consistency**
* **Low latency**
* **High Reliability**

High Availability means your system should be up and running all the time => Through Replication of System=>Replication may hit on consistency but in your NFR it is ok to take hit on consistency

Low latency => Access more and more data from in-memory DB not from hard disk memory=>wire time should be decreased by providing CDN and closest to user location

So, major NFR, scale, low latency, and reliability, usually these all 03 possible if we use NO SQL DB instead of SQL DB.

What are the major No SQL DB decisions considered for above table requirement?

# For storing the Photo & User table, the best type is Key-value pair kind of NoSQL DB (Dynamo DB / Redis etc…

# For Storing the User Photo Table and UserFollow Table we have to use wide column DB (Casandra)

## You can also do Data Size Estimation for these table’s also in addition to above estimation.

Roughly Photo table have one row of 300 bytes roughly

Assume that 2M photo uploaded daily = 2M\*300 bytes = 2\*300\*10^6 bytes = 600 MB

So, for 10 Years, 10 \* 400 \* 600MB = 24\*10^5 MB = 2.4 TB

Similarly, we can calculate roughly the User Table and UserFollow

Until now we covered that how we can roughly estimate the storage requirement of your system so that you will make your strategy for handling this requirement

## 7. Component Design: Architectural tactic for **high availability** and **low latency** system

Photo Storage

As per the system requirement, our system might have more read requirement than write. I mean as per architect we must think about the more read biased system than write. Also, **to improve performance of this component, we must plan how to improve reading as fast as possible than write**.

**Another Architectural tactic here is, divide the read and write responsibility to different server** so that reading of content don’t get stalled while writing content to the server.

Also, we must consider that **connection from one server** is having max **limit**, so before reaching to that threshold value you must transfer request to the new server. This decision will be done at load balancer with some sticky meta data from client.

You must do this as per system requirement.

The above decisions are very basic, and we will go in more such decisions going forward.

## 8. Component Design: Architectural tactic for handling of **reliability**

As per NFR of the system, reliability is key attribute required in this system.

So, to make sure that if user has uploaded some file and process is successful than in any case this file not to be lost.

And so, you must support the reliability by introducing redundancy meaning multiple copy of service running at a time.

Now here multiple copy is organized in such a way that if one server is running than another server is standby and on any failover situation, we will just swap the active server and standby.

Diagram

Description automatically generated

## 9. Component Design: Data Sharding

So, for Image storage data replication is better choice for getting handled more reads

Now, in case of Image meta data, server replication might not be good choice, because in that case may be after reaching a certain limit blocking for reading to often will put server at unavailable and it will hamper our service availability or may introduce some latency to access the server.

For handling this situation, your replication read could not sufficient specially for relational database.

So, in that situation, what you will do….?

And for those who are architecting experience, can easily understand that another tactic available in your experience bucket is shard such big table and put them on different server so that your different server is distributed for read load to their relevant shards only. I hope, I am making sense here in my decision.

Now let’s understand what are the major strategy of data sharding possible and which one is the best suitable for this problem

If I see the system (Image meta data server) here, there are two major key parts of this service,

User

Image Meta Data

Approach of sharding is also based on these data

**(Approach 01) – Partitioning based on userID**

This approach is quite intuitive, and it is not helpful in many scenarios

|  |  |
| --- | --- |
| Pros | Cons |
| Easy to implement | Issue in case of Hot User |
| Intuitive | Making non uniform distribution and server usage |
|  | What if you are holding pictures of one use in more than one shard |
|  | Storing all images in one shard will have possibility to impact user experience in-case shard is not available, etc… |

**(Approach 02) – Partitioning based on PhotoID**

So, this approach is based one unique PhotoID and hashing index of shard by PhotoID % NShard, this will give you some shard number which still help you to reach the shard. What are the pros and cons of this approach?

Before moving on pros and cons, let’s discuss the

How can we generate PhotoIDs?

One intuitive way of generating the PhotoID is to use some dedicated DB service to generate unique PhotoID for all the Photo’s whose meta data is about to store in these shards.

One problem I can see immediately is the SPF (Single point of Failure).

Yes, this is kind of problem easily your audience or interviewer will catch here and obviously they are curious about, how you will solve this problem in your architectural decision.

So, solving this problem maybe you can run two parallel DB server who will generate PhotoID and would be back up to each other in special circumstances.

|  |  |
| --- | --- |
| Pros | Cons |
| Easy to implement | Does this solve your hot user problem? |
|  | Also, what if your few shards are down? |
|  | Also, how you deal with outburst data situation, where your one DB server for creating the Unique PhotoID is overloaded with work? |
|  | How well your shard is distributed? |

So, until now we understood that neither UserID nor PhotoID based shard consideration is enough for solving this shard problem, so you must consider another approach.

For now, I am assuming that you will go eventually with some hashing-based approach and in further refinement you will use consistent hashing approach. I will discuss this in detail with another article.

## 10. Component Design: 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.

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

In the final step, the server will submit all these photos to our **ranking algorithm**, which will determine the top 100 photos (based on time they uploaded or modified, likeness, etc.) and return them to the user.

If you look at the few lines mentioned above, it will easily observe here that if you will do all these steps every time than its sill increases your latency in service news’s feed for your user.

Now, latency point of view, it might not be acceptable for user and so you must think to optimise this latency in your **design decision**.

So, handle this situation you might have to cache this information before request from user and put this information ready for your user in advance, and they will fetch whenever they need.

This approach is called **Pre-Computation and ready with information.**

This is very handy and important for better user experience.

Before moving ahead, let’s discuss another key point here that, how you will serve this information to user. I mean, **what are the different approaches for sending News Feed contents to the users?**

>> Pull:

* Client will pull the news feed from server
* This has been done on regular interval

Pros:

* Client will pull as per their requirement. Also, Server cycle is preserved

Cons:

* Client can only see the update when they pull the data
* This approach will not give feed from their friend until they request from server. So, something does not update real time seems here
* Possibility of getting Empty response from server on client request, meaning you might be wasted many servers computing without anything meaningful

>> Push:

Server can push the data whenever anything happens meaningful. So, client don’t require to request, and they are getting updated as soon as server received data for client

Pros:

* No response is empty this case, meaning every computing power of server is used for purpose.
* Client will receive all update as soon as it happened

Cons:

* Clients must maintain a [Long Poll](https://en.wikipedia.org/wiki/Push_technology#Long_polling) request with the server for receiving the updates.
* User who is following many, will get lot’s computing power and bandwidth to receive all the push from server. It might freeze client other operations, and so it may be hampered user experience

>> Hybrid:

Another approach will be hybrid approach where we will use both approach for user who falls in category A or category B accordingly.

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. 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 updates to pull data regularly.

## 11. News Feed Generation 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.

## 12. Cache and Load balancing