System Design : Instagram

(Photo Sharing Social Network)

Interview Daemon Jan 2 · 9 min read

Source: *interviewdaemon.com (Link)*

Instagram is the most popular photo-oriented social network on the planet today.

Introduction

This system will allow users to share photos and videos with other users. Additionally, users can follow other users based on follow request and they can see other user's photos and videos. In this system, you can search users and see their profile if their account is public. Otherwise you need to send a follow request.

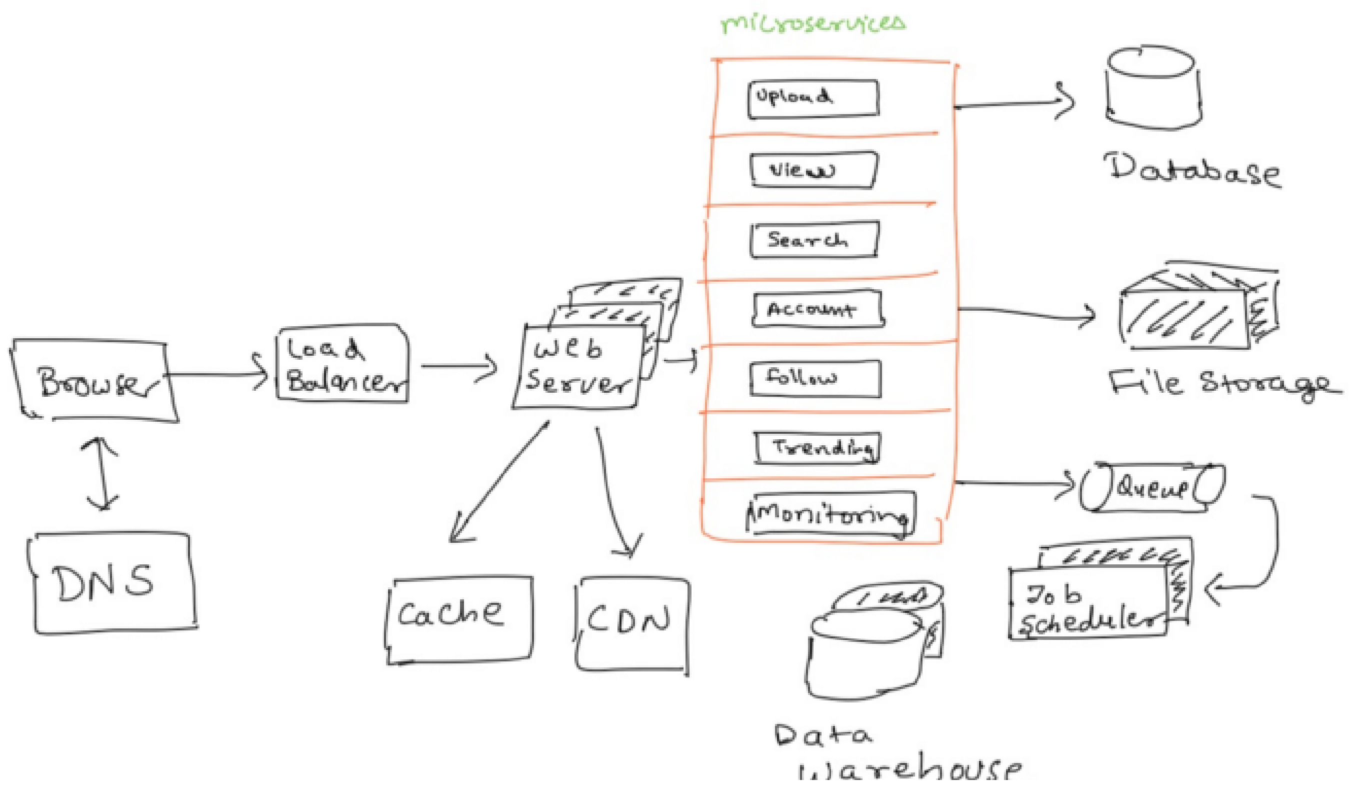
Requirements

- Users must be able to **create/delete an account**.
- Users must be able to see other users' photos and videos in their **timeline**.
- Users must be able to **upload/delete/view/search photos and videos**.
- Users must be able to **search users**.

- System must be able to support **public and private account**.
- Users must be able to send a **follow request** to other users.
- Users must be able **like other users photos and videos**.
- System should be **highly available**
- System should be **highly reliable**
- System should be **durable**
- System should be **resilient**
- Service will be both **write-heavy and read-heavy**.
- Service will be stay **consistent** (Eventual Consistency is fine) and reliable which means there should not be any data loss.
- Service will be durable which means all piece of system should exists until they are delete manually.

High Level System Design

At a high-level, we need to support multiple micro-services in the current system.



API Design

We can use REST or SOAP to serve our APIs. Basically, there will be three important API’s of photo and video sharing service system.

1. postMedia

```
postMedia(api_dev_key, media_type, media_data, title, description,
tags[], media_details)
```

PostMedia will responsible for uploading photo or video. api_dev_key is the API developer key of a registered account. We can eliminate hacker attacks with api_dev_key. This API returns HTTP response. (202 accepted if success)

2. getMedia

```
getMedia(api_dev_key, media_type, search_query, user_location, page,
maximum_video_count = 20)
```

Return JSON containing information about the list of photos and videos. Each media resource will have a title, creation date, like count, total view count, owner and other meta informations.

3. deleteMedia

```
deleteMedia(api_dev_key, ID, type)
```

Check if user has permission to delete media. It will return HTTP response based on your response.

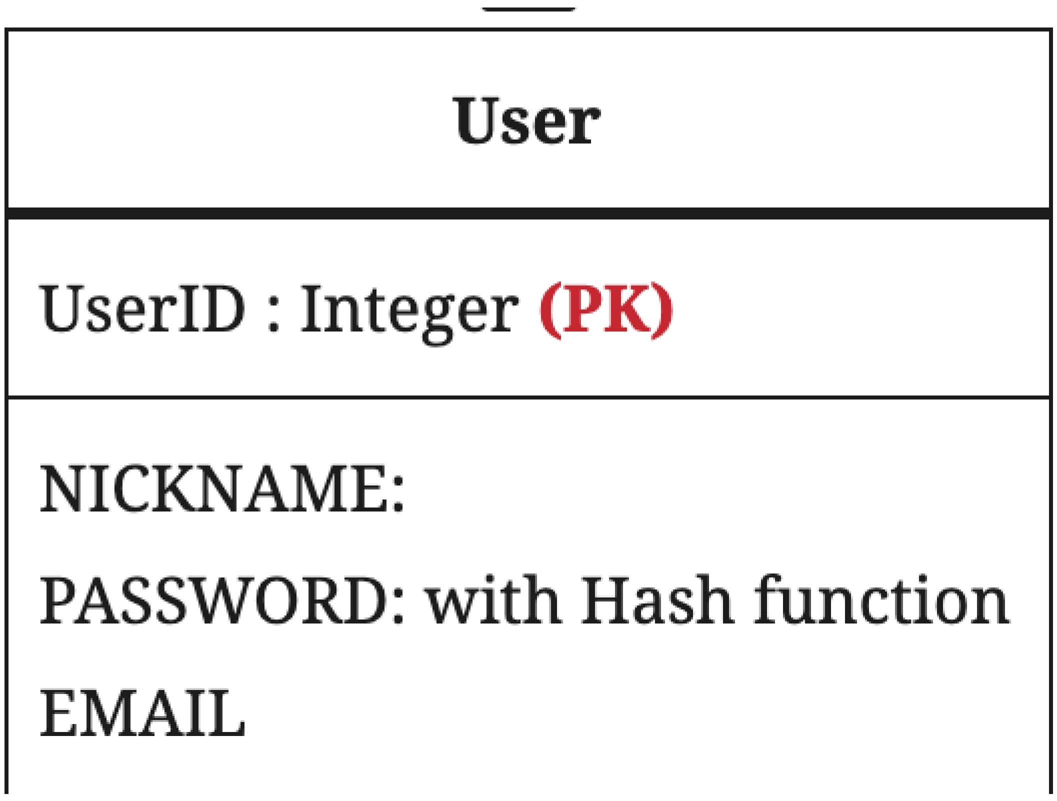
- 200 (OK),
- 202 (Accepted) if the action has been queued, or

- 204 (No Content)

Database

Instagram uses PostgreSQL (Relational Database) as primary database. It stores information such as user data, photo, tags, meta-tags etc. As the platform grew bigger they moved to more microservice model keeping the database as SQL but started to shard(partition) the database across multiple servers. (Database Sharding) (Sharding in Instagram)

- In-memory database is used to store the activity feed, sessions & other app's real-time data. (Instagram uses Redis)
- Distributed memory caching system is used for caching throughout the service. (Instagram uses Memcache)
- SQL Connection pooler is used when connecting to backend to get perfomance boost. (Instagram uses Pgbouncer)
- We can store photos in a distributed file storage like HDFS or S3.
- We can use Cassandra, column-based data storage (Non-Relational Database), to save follow-up of users.



BIRTHDATE

REGISTERDATE

LASTLOGINDATE

PROFILE_VISIBILITY

Post

ID: Integer **(PK)**

USERID: INT (FK with UserId)

MEDIA_TYPE_ID: INT (FK with)

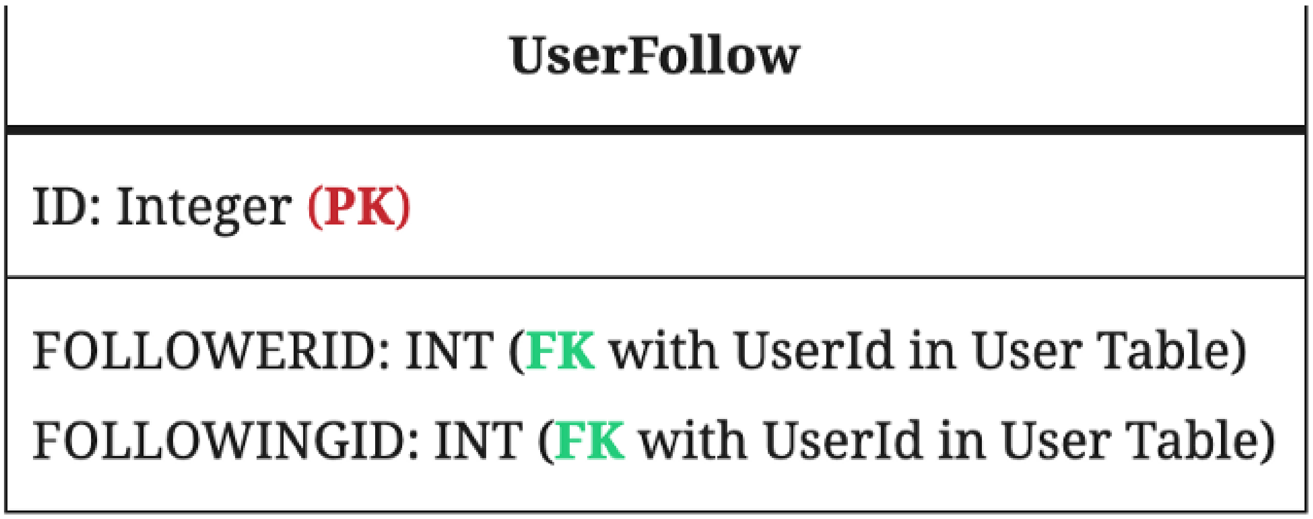
PATH

DESCRIPTION

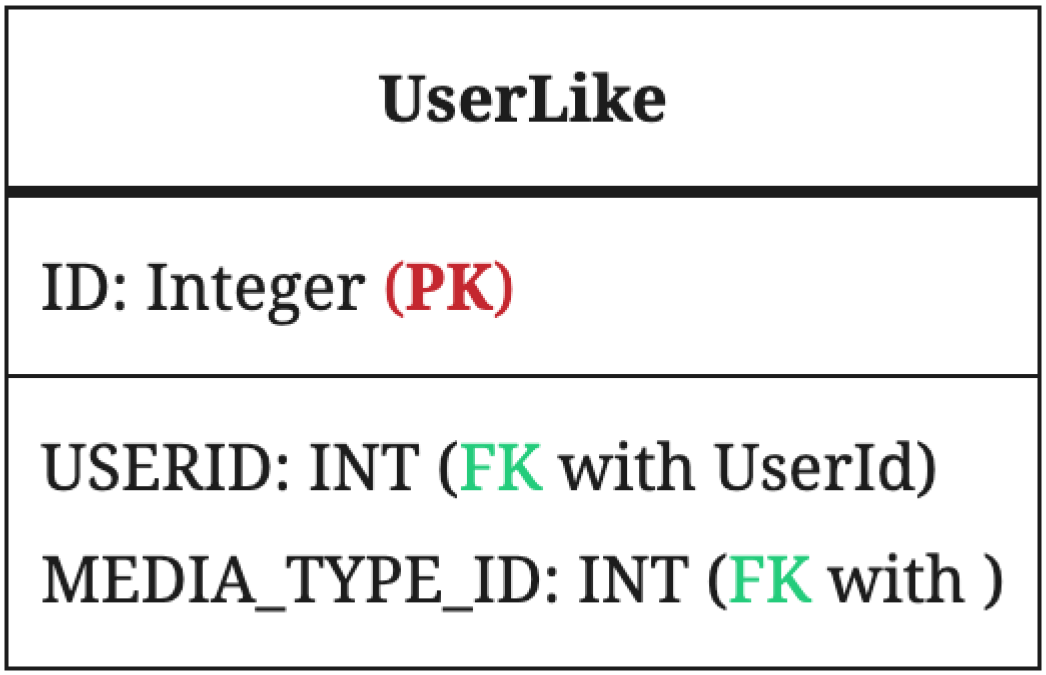
VISIBILITY

ADDEDDATE

VIEWS_COUNT



UserFollow
ID: Integer **(PK)**
FOLLOWERID: INT (**FK** with UserId in User Table) FOLLOWINGID: INT (**FK** with UserId in User Table)



UserLike
ID: Integer **(PK)**
USERID: INT (FK with UserId) MEDIA_TYPE_ID: INT (FK with)

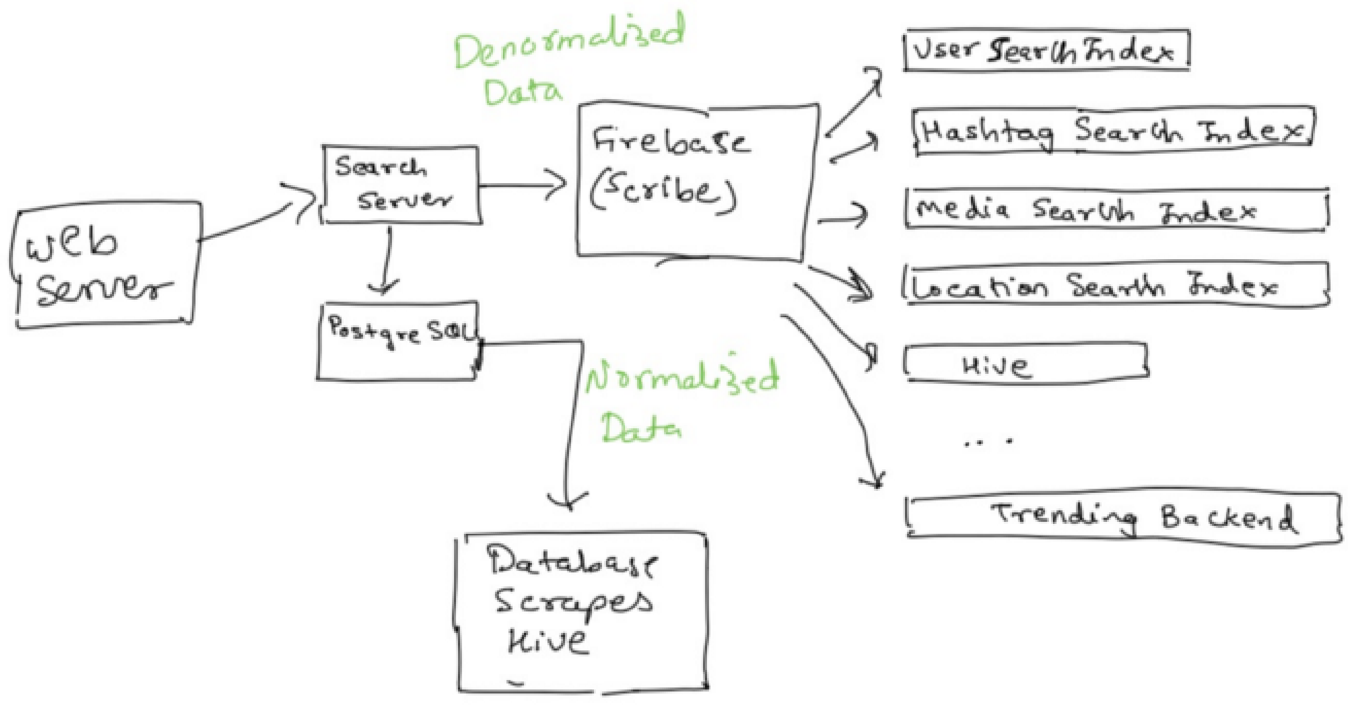
Component Design

1. Upload Service

Uploading images and videos can consume all time and connections as upload is a slow process. We can keep upload server seperate which uploads the metadata to PostgreSQL (Relational Database) and images to file storage like S3.

2. Search Service

Instagram initially used Elasticsearch (Distributed search) for its search feature but later migrated to *Unicorn* (a social graph aware search engine built by Facebook in-house). Unicorn powers search at Facebook & has scaled to *indexes* containing trillions of documents. It allows the application to save locations, users, hashtags etc & the relationship between these entities.



Speaking of the Insta's search infrastructure it has denormalized data stores (Non-Relational Database like Cassandra) for users, locations, hashtags, media etc.

The search infrastructure has a system called *Slipstream* which breaks the user uploaded data, streams it through a *Firehose* (Load Data Relational Data into Data Stores) & adds it to the search indexes.

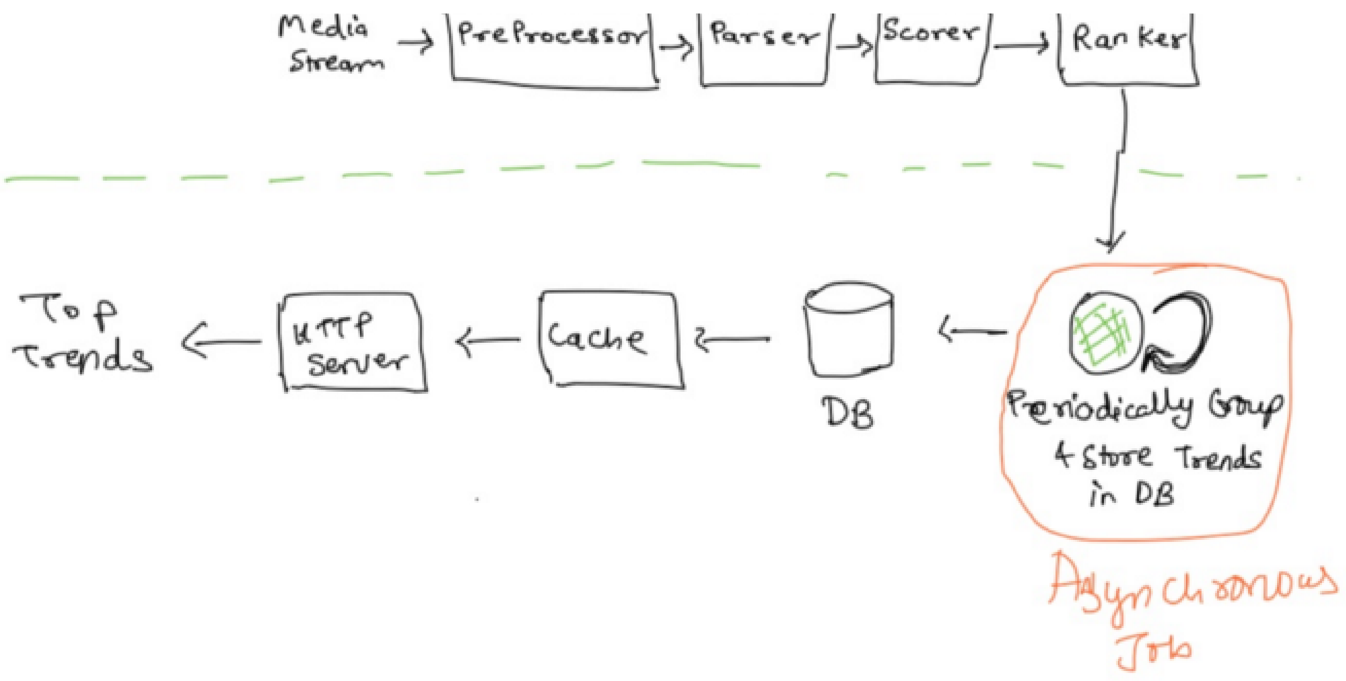
The data stored by these search indexes is more search-oriented as opposed to the regular persistence of uploaded data to *PostgreSQL DB*.

3. Top Trending HashTag (Ranking and News Feed)

The role of the service is to consume a **stream of event logs** and produce the **ranked list of trending content** I.e. hashtags and places.

Pre-generating the News Feed

We can have dedicated servers that are continuously generating users' News Feeds and storing them in a 'UserNewsFeed' table.



- **Pre-processor Node**

The *pre-processor* node attaches the necessary data needed to apply filters on the original media that has metadata attached with it.

- **Parser Node**

The *parser* node extracts all the hashtags attached with an image and applies filters to it.

- **Scorer Node**

Scorer node keeps track of the counters for each hashtag based on time. All the counter data is kept in the cache, also persisted for durability.

- **Ranker Node**

The role of the *ranker* node is to compute the trending scores of hashtags. The trends are served from a read-through cache that is *Memcache* & the database is *Postgres*.

No we have the top trends in database and local cache. We can use one of the following approach to get top trends to the user.

Background Processing:

Asynchronous tasks such as sending notifications to the users & other system background processes are done using task queue and message broker.

- We can use an asynchronous task queue based on distributed message communication, focused on real-time operations. It supports scheduling too. (Instagram uses Celery)
- A message broker use to push the content in queue and pull it later (Instagram uses RabbitMQ)

What are the different approaches for sending News Feed and TopTrends) contents to the users?

- **Pull:** Clients can pull the News-Feed contents from the server at a regular interval or manually whenever they need it. (Problems: New data might not be shown until clients issue a pull request, Pull requests will result in an empty response if there is no new data)
- **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. (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)
- **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. 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.

4. Account Service

This service is solely responsible for creation and deletion of user details. This can call postgresql and update the User table.Can also user the “**PROFILE_VISIBILITY**” boolean to make user profile public/private based on user preferences.

5. Follow Network Service

This service is responsible to keep track of followers a user has and profiles which a user follows. **UserFollow** table is used which keeps track of people a user follows. We can also keep this in Cassandra or any key value stores.

6. Monitoring

With so many instances powering the service, monitoring plays a key role in ensuring the health & availability of the service.

Network & infrastructure monitoring tool used by Instagram to track metrics across the service & get notified of any anomalies. (Instagram uses Munin)

A network daemon is used to track statistics like counters and timers. Counters at Instagram are used to track events like user signups, number of likes etc. Timers are used to time the generation of feeds & other events that are performed by users on the app. These statistics are almost real-time & enable the developers to evaluate the system & code changes immediately. (Instagram uses StatsD)

We need to watch the running processes & take snapshot of any process taking longer than the decided time by the middleware and the file is written to the disk. (Instagram uses Dogslow)

Monitors on external website's, ensuring expected performance & availability. (Instagram uses Pingdom)

PagerDuty can be used for notifications & incident response.

Replication and Backup

Since we want a high available and consistent system. Replication and back-up are two important concepts to provide pillars we mentioned before. Replication is a very important concept to handle a failure of services or servers. Replication can be applied database servers, web servers, application servers, media storages and etc.. Actually we can replicate all parts of the system.

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.

Data Sharding (Data Partitioning)

Instagram does generate its own unique id and also shards the

Unique ID Generation

1. Generated IDs should be sortable by time (so a list of photo IDs, for example, could be sorted without fetching more information about the photos)
2. IDs should ideally be 64 bits (for smaller indexes, and better storage in systems like Redis)

Solutions

1. Generate IDs in web application (**Cons:** UUID generated in web application completely random and have no natural sort)
2. Database Auto Increment (Cons: Can eventually become a write bottleneck, For distributed storage can no longer guarantee that they are sorted over time)
3. Key Generation Server that generates new Keys (Create unique key with respect to each table. **Feasible Approach**)

Database Partitioning

1. Partition based on UserID

Here we can keep all photos of a user on the same shard. 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.

Problems:

- How would we handle hot users? Several people follow such hot users, and a lot of other people see any photo they upload.
- Some users will have a lot of photos compared to others, thus making a non-uniform distribution of storage.

- 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?
- 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.

2 Partition 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.

Cache and Load Balancing

We can introduce a cache for metadata servers to cache hot database rows. Least Recently Used (LRU) can be a reasonable cache eviction policy for our system. (Instagram uses Memcache)

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.

References:

- *HighScalability*,
- 8bitmen.com,
- under-the-hood-instagram,
- GeeksForGeeks,
- what powers instagram

System Design Interview System Design Interview Daemon