Requirements:

Functional Requirements:

1. Upload/ Download/View Photos and videos
2. Add comments
3. Search based on photo titles
4. News Feed from top 10
5. Follow other users

Non-Functional Requirements:

1. Reliable – should not lose any pictures
2. Highly available – should not be down
3. Consistency of news feed can take a hit

This will be a read heavy system.

Number of users:

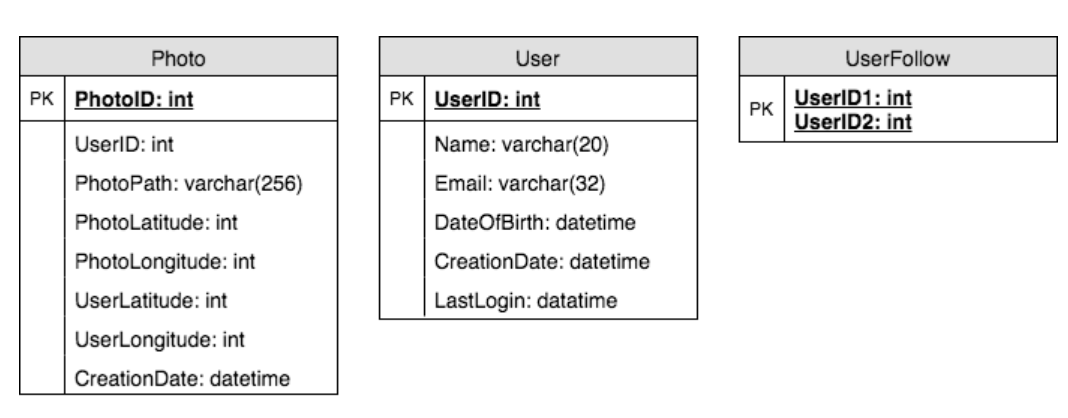
1. 1M daily active users
2. 2M photos each day – 2M/ 24 \* 60 \* 60= 23 photos/sec
3. Each photo takes 500kb space
4. 2M \* 500Kb = 100GB per day
5. 100GB \* 30 \* 12 \* 10 = 3600TB

API Routes and Methods:

1. Create New User
   1. Username
   2. Password
   3. Email
   4. Phone
2. Upload Picture
   1. UserId
   2. Picture Stream
   3. Picture Title
3. Follow User
   1. UserId
   2. UserId To Follow

We need object storage to store this data and the reference for this picture can be stored in a simple relation database

Define database models:



As relation tables are not scalable, we can store the metadata in NoSQL database where we could enjoy the benefits of it

We can maintain the relationship tables like UserID (PK) – PhotoID and UserID to follow.

System Design:

Writes can be slow, but reads should be faster

1. Writes cannot consume all the connections on an app server
2. So, we can create a different service for writes, that has only responsibility to write to the object database
3. Reads can happen from a different app server

Reliability:

1. We can have a master slave architecture for our database, to help with reliability
2. Master can take all the writes and propagate them to the slave node

Availability:

1. Like data we can have multiple app servers which can be load balanced
2. If one dies or have high latency, then the redundant server can take the additional calls

Data Sharding:

1. If we have all the data in single database storage, it will hit the performance
2. We can shard by the user
   1. This will create uneven distribution of photos
   2. If we have hot users like celebrities, they need more shards and space than a regular user
3. Shard by photo id
   1. Generate a unique phot id based using the key generation service
   2. Modulo of this photo id by number of shards gives us the right shard to store the metadata
4. We can store these logical partitions on a single database, or we can separate based on the number of users

News Feed:

1. We can have a different service that can pick the recently uploaded pictures of the followers of a user and store them in a database for that user
2. As the latency would be high if we start generating the news feed every time user requests
3. Pre generating would be a good idea
4. Once we pre generate, we can use
   1. Pull mechanism to send updates only when required
   2. Push using web sockets when the user logins
   3. Hybrid, pull for users with more followers and push with a smaller number of followers
   4. We can push data incrementally
5. As we require latest photos to generate this, we can add the data time to the unique PhotoID which would give us easy way to get these pictures

Block Diagram:

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.

We need to have geographically distributed caches to push the data to the users quickly