**Design Scalable System like Instagram**

**Why?**

A system like Instagram is highly desired as a tool for social media. A high traffic website such as Instragram requires a large deal of organization to ensure user-end satisfaction is at an all time highest. We will focus solely on pictures.

**Requirements and Goals**

Functional

* Users can login/register
* Users can also logout
* Users have full range of upload and sharing of video and pictures. We will restrict downloads to only the specific user.
* Users may restrict other users from seeing their profile along with publicize or privatize their own content.

Non Functional

* The system should be highly available.
* Consistency is desired through availability at all times is regarded as higher.

Other Functional

* The system should monitor overall communication within the platform.
* The systems timeline creation must be effective for most recent to least recent (Decreasing).
* Being read heavy, the systems main purpose of presentation of pictures, videos is desired and uploading. Latency must be at an all time low!

**Estimations, Capacity, Constraints**

Due to the high volume of traffic expected for read operations with minimal write operations, we must prioritize and focus a larger deal to read. ~~We will suggest a 100:1 relation for read to write.~~

**Traffic of the system**

Suppose that we have 500 million users within our system with 10 million active daily users. Suppose then each active user a day uploads 2 pictures a day, therefore we will be obtaining 20 million pictures a day. Assuming that the size of a picture is 500KB, for a single day it would take

1 day \* 24 hours \* 60 minutes \* 60 seconds \* 500KB \* 20 million = ~10TB of space a day! Cray.

**Bandwidth estimates**

Uploads per second

20 million / (24 \* 3600) = 231 UPS (Uploads per second)

231 UPS \* 500KB (Storage per picture) = 115.7MB/s

Since we said the factor was 100 to 1 for read and write, the computed write above this would be used to calculate the proposed reads.

115.7MB \* 100 = 11.5GB/s

**~~Memory estimates~~**

Do we store this in cache?

Come back to later

**API Design**

We can either use REST or SOAP for the API

Functionalities

**uploadPicture(dev\_api\_key, picture, title, description, tags[])**

*Taking a dev\_api\_key which can also be of the registered account, we can eliminate attacks.*

**deletePicture(dev\_api\_key, pictureId)**

*Deletes the picture from the system/database.*

**downloadPictures(dev\_api\_key, searchQuery, userLocation, maxCount, page)**

*Allows dev to download multiple images based on queried results.*

*This will be a JSON formatted data return with the available keys mapping to the data sent.*

**Database design**

* We must be able to hold a large quantity of records, calculated to be 10TB a day.
* Each record/picture is 500KB but since holding pictures within a database is well… meaningless, storing them on the system will suffice. We will only need to know the picture information and where it’s pointed to within the system.

**User**

UserID : int(8)

UserName : nvarchar

UserRealName : nvarchar

UserSurname : nvarchar

Main : nvarchar

BirthDate : DateTime

RegisterDate : DateTime

LastLoginDate : DateTime

**Picture**

PictureID : int

UserID : int

PicturePath : nvarchar

PictureLatitude : int

PictureLongitude : int

CreationDate : DateTime

**UserFollow**

UserID1 : int

UserID2 : int

**Basic System Design and Algorithm**

We would need Client, Services, Web Server, Application Server, Picture Storage, Database, Caching, Redundancy, Load balancing, Replication, and Sharding.

There will be two separate services in the main system which are upload and download images. The storage of pictures will be important as well and keeping track of the records/information within the database. Any request to the web server will be redirected towards the application server.

We would also require a backup server, rule of 3 or more as any data replication on one server wouldn’t be lost on another and would prevent redundancy.

Designing the news feed to handle such requests as to refresh their newsfeed is important. We could pregenerate the users timeline but we might run in numerous problems doing so.

**Pulling**; When the user requests any new changes to the news feed, the data brought in might not be overall complete was the pre-generate hasn’t finished.

**Push;** We could implement a push that would send the data to the client as soon as its ready.

**Hybrid (Optimal):** Prefetch any data at all times, when the user issues a pull request we could post any of the new data generated. We could look further into the number of users a person follows. A pull would be easily built for a small user base and a push for large number of followers.

**Data Management**

We should partition the database based on the UserID and PictureID. By partitioning based on the UserID, we can divide the incoming requests based on that specific ID. This will highly reduce the number of requests required to fill a news feed. Conditioning with shard based on UserID. The problem with this is that the system may be nonuniformly distributed and it might not fit all in one shard/feed. By creating unique identifiers for nonuniformly based data, we can seamingly link them together.

**Caching**

Cache memory is important to get near flawless latency between the user and the respective content. A suggestion is to use a CDN (Content Delivery Network) for our cache to deliver from the file system to the user. To keep any meta data underlining the pictures, we can use Memcache.

**Load Balancing**

A load balancing layer would be important to our system to evenly distribute traffic among the servers available in our system. We would need a load balancer between our Client and Web Server, Web Server to Application Server, Application Server to Database, application Server to image storage device, and finally Application Server to Cache database.

A good implementation for the load balancer would be Round Robin.

Issues: Overwhelming traffic on a specific server isn’t neglected in the selection process, meaning that the Round Robin algorithm will place requests on that overloaded server no matter the condition.