Capacity Estimation and Constraints

Storage Estimates Let us say each tweet has 140 characters and we need two bytes to store a character without compression. Let us assume we need 30 bytes to store metadata with each tweet (like ID, timestamp)

Not all tweets will have media, let us assume that on average every fifth tweet has a photo and every tenth has a video. Let us also assume on average a photo is 200KB and a video is 2MB. This will lead us to have 24TB of new media every day.

(100M/5 photos * 200KB) + (100M/10 videos * 2MB) ~= 24TB/day

Bandwidth Estimates: Since total ingress is 24TB per day, this would translate into 290MB/sec.

Remember that we have 28B tweet views per day. We must show the photo of every tweet (if it has a photo) but let us assume that the users watch every 3rd video they see in their timeline. So, total egress will be:

(28B * 280 bytes) / 86400s of text => 93MB/s + (28B/5 * 200KB) / 86400s of photos => 13GB/S + (28B/10/3 * 2MB) / 86400s of Videos => 22GB/s

Total ~= 35GB/s

System API

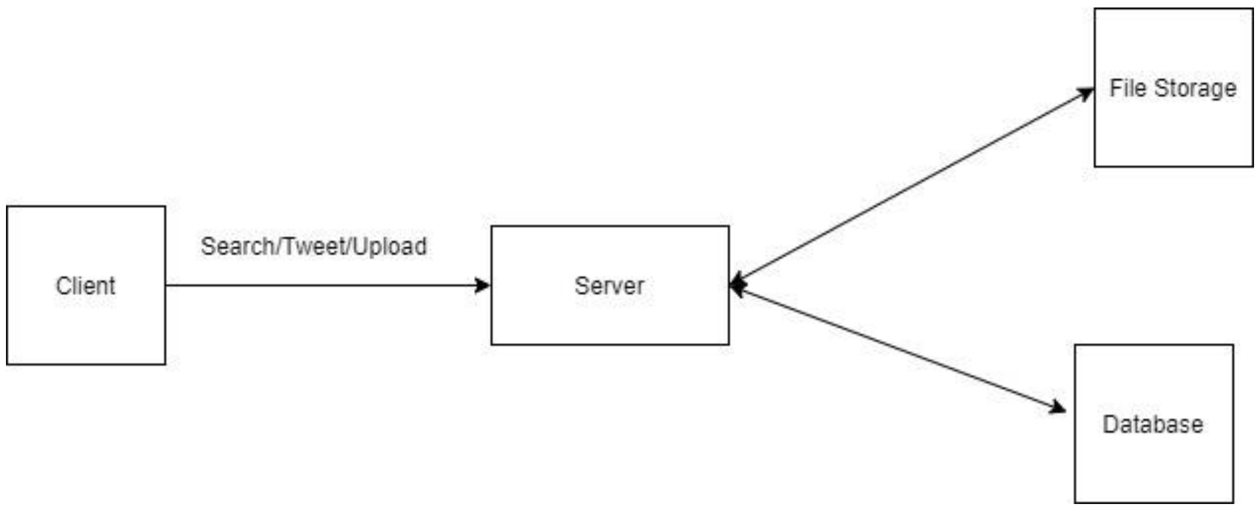
Post tweet (user id ,tweet id)

User feed (user id)

Follow Tweet(follower id ,Followee id)

Favoriting/Bookmark tweet(user id, tweet id)

High Level Design



Database Design

Design considerations

- System is read and write heavy
- Average Size of Tweet(140 characters and multimedia i.e. 2-5 MB per tweet with multimedia)
- System should be highly available

User

Column	Type
User_id	Integer
Username	String
Email	String
Password	String
Creation date	Timestamp
Last login	Timestamp
Longitude	Integer
Latitude	Integer

Photos

Column	Type
Photo id	Integer
Photo path	String
Created date	Timestamp

Videos

Column	Type

Video id	Integer
Video path	String
Created date	Timestamp

Tweets

Column	Type
Tweet id	Integer
Tweet	String
Created Date	Timestamp
Last update	Timestamp
Longitude	Integer
Latitude	Integer

User Tweets

Column	Type
User id	Integer
Tweet id	Integer

User Favorite Tweets

Column	Type
User id	Integer
Tweet id	Integer

User Followers

Column	Type
Follower id	Integer
Followee id	Integer
Following start time	timestamp

User Photos

Column	Type
User id	Integer
Tweet id	Integer
Photo id	Integer

User Videos

Column	Type
User id	Integer
Tweet id	Integer
Video id	Integer

Depending upon the schema, we can store user tweets in relation database such as MySQL.

For building user timeline of feeds generation can flood the relational database .So we can use NoSQL database or Memory store for fast writes.SO we propose to utilize both MySQL and NoSQL database for our system.

News feed generation

We need to fetch latest, popular, and most relevant tweets 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 followers from each user. In the final step, the server will submit all these tweets to our ranking algorithm which will determine the top 100 tweets (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/tweets/videos 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.

Data Partitioning and Replication

Sharding based on User_id: We can try storing all the data of a user on one server. While storing, we can pass the UserID to our hash function that will map the user to a database server where we will store all the user's tweets, favorites, follows, etc. While querying for tweets/follows/favorites of a user, we can ask our hash function where we can find the data of a user and then read it from there. This approach has a couple of issues:

1. What if a user becomes hot? There could be a lot of queries on the server holding the user. This high load will affect the performance of our service.
2. Over time some users can end up storing a lot of tweets or having a lot of follows compared to others. Maintaining a uniform distribution of growing user data is quite difficult.

To recover from these situations either we must repartition/redistribute our data or use consistent hashing.

Sharding based on TweetID: Our hash function will map each TweetID to a random server where we will store that Tweet. To search for tweets, we must query all servers, and each server will return a set of tweets. A centralized server will aggregate these results to return them to the user. Let us investigate timeline generation example; here are the number of steps our system must perform to generate a user's number of steps our system has to perform to generate a user's

1. Our application (app) server will find all the people the user follows.
2. App server will send the query to all database servers to find tweets from these people.
3. Each database server will find the tweets for each user, sort them by recency and return the top tweets.
4. App server will merge all the results and sort them again to return the top results to the user.

This approach solves the problem of hot users, but, in contrast to sharding by UserID, we must query all database partitions to find tweets of a user, which can result in higher latencies.

We can further improve our performance by introducing cache to store hot tweets in front of the database servers

Sharding based on Tweet creation time: Storing tweets based on creation time will give us the advantage of fetching all the top tweets quickly and we only must query a very small set of servers. The problem here is that the traffic load will not be distributed, e.g., while writing, all new tweets will be going to one server and the remaining servers will be sitting idle. Similarly, while reading, the server holding the latest data will have a very high load as compared to servers holding old data.

Sharding based on Tweet Id and Tweet creation timestamp

If we do not store tweet creation time separately and use TweetID to reflect that, we can get benefits of both the approaches. This way it will be quite quick to find the latest Tweets. For this, we must make each TweetID universally unique in our system and each TweetID should contain a timestamp too.

We can use epoch time for this. Let us say our TweetID will have two parts: the first part will be representing epoch seconds and the second part will be an auto-incrementing sequence. So, to make a new TweetID, we can take the current epoch time and append an auto-incrementing number to it. We can figure out the shard number from this TweetID and store it there.

We would need 31 bits to store this number. Since on average we are expecting 1150 new tweets per second, we can allocate 17 bits to store auto incremented sequence; this will make our TweetID 48 bits long. So, every second we can store (2^17 => 130K) new tweets. We can reset our auto incrementing sequence every second. For fault tolerance and better performance, we can have two database servers to generate autoincrementing keys for us, one generating even numbered keys and the other generating odd numbered keys.

Caching

We can introduce a cache for database servers to cache hot tweets and users. Also Application servers, before hitting database, can quickly check if the cache has desired tweets. Based on clients' usage patterns we can determine how many cache servers we need. Keeping above requirements ,we propose to utilize Memcache for our system.

Intelligent Cache
80-20 rule can help us build intelligent cache. It means 20% of tweets is generating 80% of daily traffic which means certain tweets are most popular and liked by majority people. So, we can cache these 20% of daily read of volume of data and tweets to build intelligent cache. Our service can benefit from this approach.

Our cache would be like a hash table where ‘key’ would be ‘OwnerID’ and ‘value’ would be a doubly linked list containing all the tweets from that user in the past three days. Since we want to retrieve the most recent data first, we can always insert new tweets at the head of the linked list, which means all the older tweets will be near the tail of the linked list. Therefore, we can remove tweets from the tail to make space for newer tweets.

Load Balancing

We can add a Load balancing layer at three places in our system:

1. Between Clients and Application servers
2. Between Application Servers and database servers
3. Between Application Servers and Cache servers

We can use simple round robin mechanism for load balancing ,it does not take the server load into consideration. So, we can use Weighted Round Robin approach for load balancing the weighted round robin load balancing algorithm allows site administrators to assign weights to each server based on criteria like traffic-handling capacity. Servers with higher weights receive a higher proportion of client requests.

Timeline Generation

Monitoring

We can collect following metrics/counters to get an understanding of the performance of our service:

1. New tweets per day/second. This should be collected on daily basis

2. Timeline delivery stats, how many tweets per day/second our service is delivering.

3. Average latency that is seen by the user to refresh timeline