Designing Twitter

Difficulty Level: Medium

Let’s design a Twitter-like social networking service.

Users of this service will be able to post tweets, follow other people, and favourite tweets.

1. What is Twitter?

Twitter is an online social networking service where users post and read short 140-character messages called “tweets”.

Registered users can post and read tweets, but those who are not registered can only read them.

Users access Twitter through a web browser, SMS, or the mobile app.

2. Requirements and Goals of the System

We will be designing a simpler version of Twitter with the following requirements.

Functional Requirements

1. Users should be able to post new tweets
2. A user should be able to follow other users
3. Users should be able to mark tweets as favourites
4. The service should be able to create and display a user’s timeline consisting of top tweets from all the people that the user follows
5. Tweets can contain photos and videos

Non-Functional Requirements

1. Our service needs to be highly available
2. Acceptable latency of the system is 200ms for timeline generation

3. Consistency can take a hit (in the interest of availability) — if a user doesn't see a tweet for a while, it should be fine

Extended Requirements

1. Searching for tweets
2. Replying to a tweet
3. Trending topics — current hot topics/searches
4. Tagging other users
5. Tweet notification
6. Who to follow? Suggestions?
7. Moments

3. Capacity Estimation and Constraints

Let's assume that we have one billion total users with 200 million daily active users (DAU).

Also assume that we have 100 million new tweets every day, and on average each user follows 200 people.

How many favourites per day?

If on average, each user favourites five tweets per day, we will have:

$$200M \text{ DAU} \cdot 5 \text{ favourites} = 1B \text{ favourites}$$

How many total tweet-views will our system generate?

Let's assume on average that a user visits their timeline two times a day, and visits five other people's pages.

On each page, if a user sees 20 tweets, then our system will generate 28B/day total tweet-views.

$$200M \text{ DAU} \cdot ((2+5) \cdot 20 \text{ tweets}) = 28B \text{ tweets/day}$$

Storage Estimates: Let's say that each tweet has 140 characters, and we need two bytes to store a character without compression.

Let’s also assume that we need 30 bytes to store metadata with each tweet (like ID, timestamp, user ID etc.).

Then, the total storage required is:

$$100M * (280 + 30) \text{ bytes} = 30 \text{ GB/day}$$

Over 5 years:

$$30 \text{ GB/day} \cdot 365 \text{ days/year} \cdot 5 \text{ years} \approx 55 \text{ TB}$$

We also need to consider storage needs for user data such as follows, favourites etc.

Some tweets will also have media, in addition to text.

Let’s assume that on average every 5th tweet has a photo, and every 10th has a video.

Let’s also assume that on average a photo is 200KB and a video is 2MB.

This leads us to 24TB of new media every day:

$$(\tfrac{100M}{5} \text{ photos} \cdot 200 \text{ KB/photo}) + (\tfrac{100M}{10} \text{ videos} \cdot 2 \text{ MB/video}) \approx 24 \text{ TB/day}$$

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

$$\frac{24 \text{ TB/day}}{86400 \text{ sec/day}} = 290 \text{ MB/sec}$$

Recall that we have 28B tweet views per day.

We must show the photo of very tweet (if it has a photo), but let’s assume that users watch every 3rd video that they see in their timeline.

So, the total egress will be:

$$\text{Text} : \frac{(28B \cdot 280 \text{ bytes})}{86400 \text{ seconds}} = 93 \text{ MB/s}$$

$$\text{Photos} : \frac{(28B/5 \cdot 280 \text{ bytes})}{86400 \text{ seconds}} = 13 \text{ GB/s}$$

$$\text{Videos} : \frac{(28B/10/3 \cdot 280 \text{ bytes})}{86400 \text{ seconds}} = 22 \text{ GB/s}$$

In total:

$$93 \text{ MB/s} + 13 \text{ GB/s} + 22 \text{ GB/s} \approx 35 \text{ GB/s}$$

4. System APIs

We can have SOAP or REST APIs to expose the functionality of our service.

The following could be the definition of the API for posting a new tweet:

```
tweet(api_dev_key, tweet_data, tweet_location, user_location, media_ids)
```

Parameters:

- `api_dev_key` (string) — the API developer key of a registered account; this will be used to, among other things, throttle users based on their allocated quota
- `tweet_data` (string) — the text of the tweet, typically up to 140 characters
- `tweet_location` (string) — optional location (longitude, latitude) that this Tweet refers to
- `user_location` (string) — optional location (longitude, latitude) of the user adding the Tweet
- `media_ids` (number[]) — optional list of media_ids to be associated with the tweet; all the media (photos, videos) need to be uploaded separately

Returns: (string)

A successful post will return the URL to access that tweet. Otherwise, an appropriate HTTP error is returned.

5. High-Level System Design

We need a system that can efficiently store all the new tweets.

Comparing reads and writes:

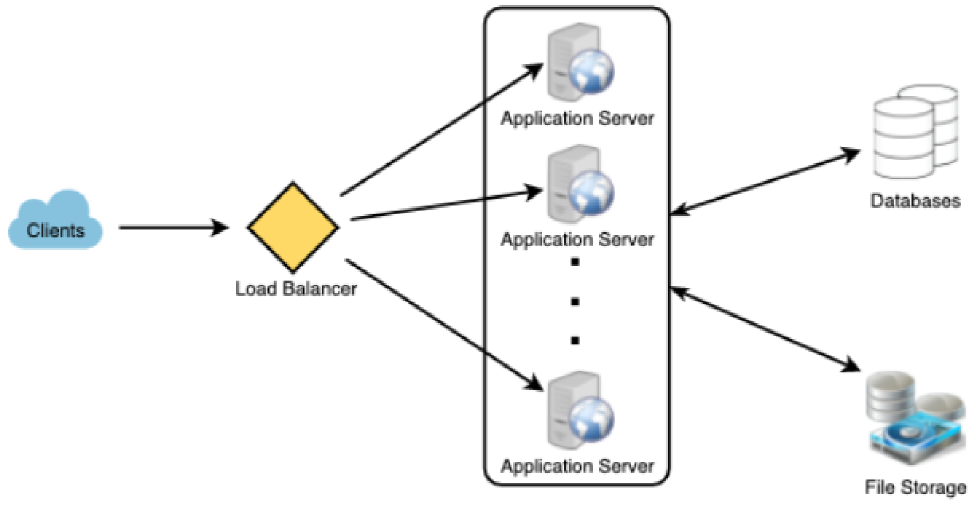
- Reads — $28B/86400 \text{ sec} = 325K \text{ tweets/sec}$
- Writes — $100M/86400 \text{ sec} = 1150 \text{ tweets/sec}$

It is clear then that this will be a read-heavy system.

At a high-level, we need multiple application servers to serve all of these requests, with load balancers in front of them for traffic distributions.

On the backend, we need an efficient database that can store all the new tweets and can support a huge number of reads.

We also need some file storage to store photos and videos.



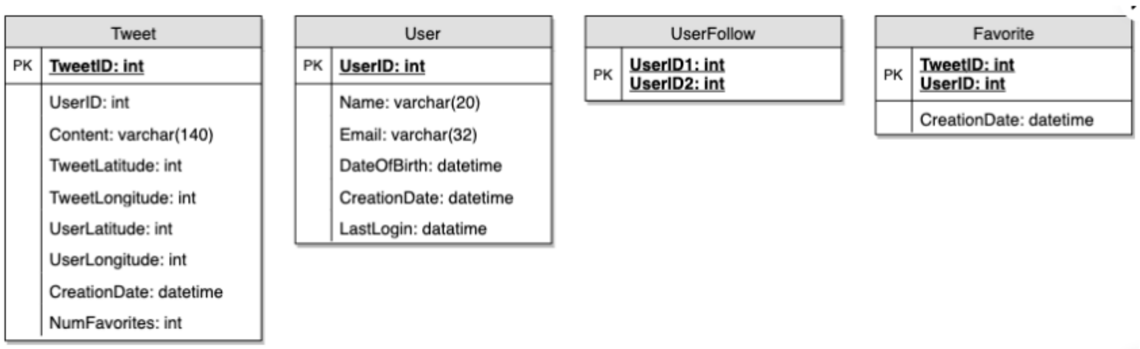
The traffic on our system may also be distributed unevenly throughout the day.

At peak times, we could expect a few thousand write requests per second, and around 1 million read requests per second.

We should keep this peak hour traffic in mind while designing the architecture of our system.

6. Database Schema

We need to store data about users, their tweets, their favourite tweets, and people that they follow.



For choosing between SQL and NoSQL databases to store the above schema, please see 'Database Schema' under *Designing Instagram*.

7. Data Sharding

Since we have a huge number of new tweets every day, and our read load is extremely high, we need to distribute out data onto multiple machines so that we can read/write it efficiently.

We have many options to shard our data. Let's go through them one-by-one.

Sharding based on UserID

We can try storing all of 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 of the user's tweets, favourites, follows etc.

While querying for the tweets, follows or favourites 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, we either have to 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 have to query all servers, and each server will return a set of tweets.

A centralised server will aggregate these results and return them to the user.

Let’s look into the timeline generation example. Here are the steps that our system has to perform to generate a user’s timeline:

1. Our application server will find all of the people that the user follows
2. The application server will send the query to all the 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. The application server will merge all of the results, sort them by recency and return the top tweets

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

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

Sharding based on Tweet Creation Time

Storing tweets based on their creation time will give us the advantage of fetching all of the top tweets quickly, and we only have to 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 sit idle.

Similarly, while reading, the server holding the latest data will have a very high load as compared to the servers holding old data.

Sharding by TweetID and Tweet Creation Time

If we don't store tweet creation time separately, and instead reflect this in the TweetID, we can get the benefits of both of these sharding approaches.

In this way, we make it quite quick to find the latest tweets.

For this, we must make each TweetID universally unique in our system, and have the TweetID contain a timestamp.

We can use epoch time for this.

Let's say that our TweetID will have two parts:

- The first part will be representing epoch seconds
- 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.

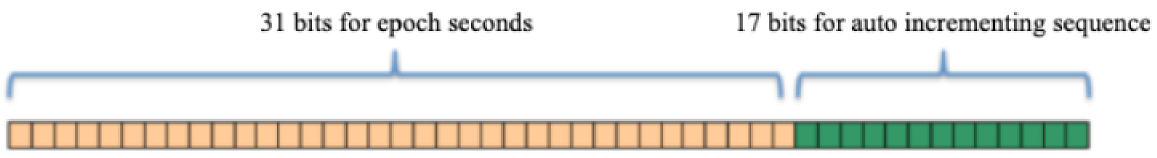
We can figure out the shard number from this TweetID, and then store the tweet on that shard.

What would then be the size of our TweetID?

Suppose that our epoch time starts today, and we want to store tweets for the next 50 years, or $50 \cdot 365 \cdot 86400 \approx 1.6B$ sec.

To do this, we would need $\lceil \log_2 1.6B \rceil = 31 \text{ bits}$

Since on average we are expecting 1150 new tweets per second, we can allocate at least $\lceil \log_2 1150 \rceil = 11 \text{ bits}$ to store an auto-incrementing sequence. We'll allocate an additional 6 bits to get a full-byte number of $31 + 11 + 6 = 48 \text{ bits} = 6 \text{ bytes}$


31 bits for epoch seconds
17 bits for auto incrementing sequence

So, in total, 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 auto-incrementing keys for us: one generating even keys, and the other generating odd keys.

If we assume that our current epoch seconds are “1483228800”, our TweetID will look something like this:

- 1483228800 000001
- 1483228800 000002
- 1483228800 000003
- …
- 1483228800 000101
- …

If we make our TweetID 64-bits (8 bytes) long, then we can easily store tweets for the next 100 years and even store them at the millisecond granularity.

In this approach, we will still have to query all the servers for timeline generation, but our reads and writes will be substantially quickey:

- We don’t have a secondary index on creation time, so this will reduce our write latency
- While reading, we don’t need to separately filter on creation time, as our primary key has the epoch time included in it

8. Cache

We can introduce a cache for database servers to cache hot tweets and users.

We can use an off-the-shelf solution like Memcached that can store the whole tweet objects.

Then, application servers, before going to the database, can quickly check for a cache hit on the desired tweets.

Based on clients' usage patterns we can determine how many cache servers we need.

Which cache eviction policy would best fit our needs?

When the cache is full and we want to replace a tweet with a newer/hotter tweet, Least Recently Used (LRU) seems to be a reasonable policy for our system.

How can we have a more intelligent cache?

If we go with the 80-20 rule, where 20% of tweets generate 80% of read traffic, meaning that certain tweets are so popular that a majority of people read from them, we can try to cache 20% of the daily read volume from each shard.

What if we cache the latest data?

Our service can benefit from this approach.

Let's say that 80% of our users see tweets from the past 3 days only. Then, we can try to cache all of the tweets from all the users from the past 3 days.

Suppose that we have dedicated cache servers that cache all of the tweets from all the users from the past 3 days.

As estimated above, we are getting 100 million new tweets or 30GB of new day every day (not including photos and videos).

If we want to store all of the tweets from the last 3 days, we will then need less than 100GB of cache memory.

This data can easily fit into one server, but we should replicate it onto multiple servers and distribute the read traffic using a load balancer to reduce the load on individual cache servers.

Then, whenever we are generating a user's timeline, we can ask the cache servers if they have all the recent tweets for that user.

If yes, then we simply return all the data from the cache.

If we do not have enough tweets in the cache, we have to query the backend server to fetch that data.

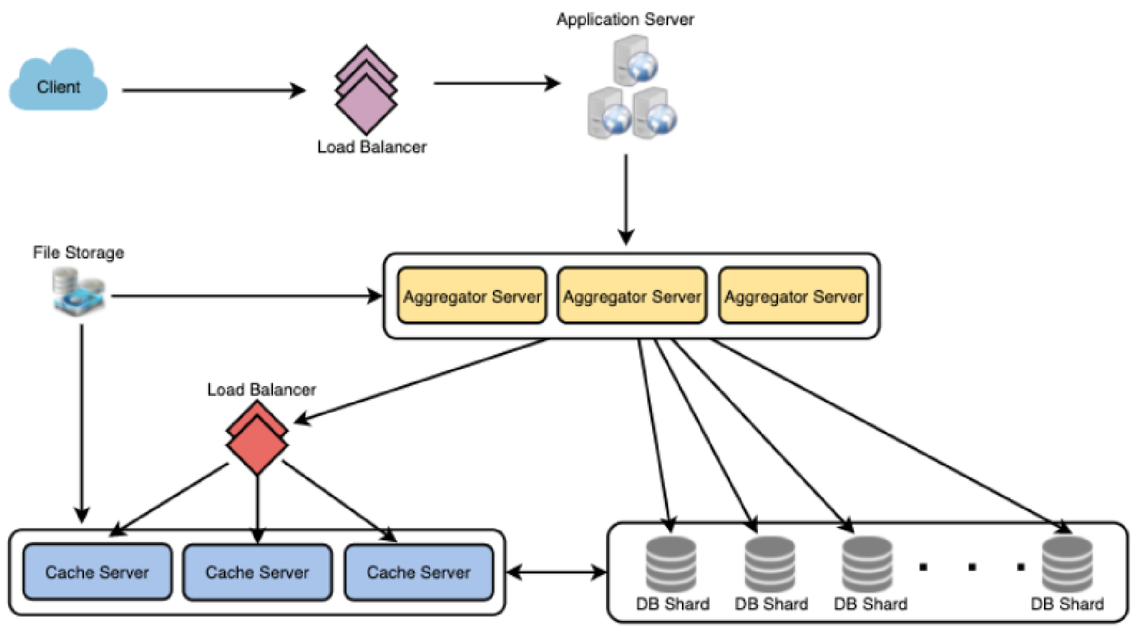
On a similar design, we can also try caching photos and videos from the last 3 days.

However, this seems expensive in terms of cache memory. Perhaps opting for the last day works well. Or we could only cache photos, and not videos.

Our cache would be like a hash table where the key would be OwnerID and the 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.



9. Timeline Generation

For a detailed discussion about timeline generation, refer to *Designing Facebook's Newsfeed.*

10. Replication and Fault Tolerance

Since our system is read-heavy, we can have multiple secondary database servers for each DB partition.

Secondary servers will be sued for read traffic only.

All writes will first go to the primary server, and then be replicated to secondary servers.

This scheme will also give us fault tolerance, since whenever the primary server goes down, we can failover to a secondary server.

11. Load Balancing

We can add a load balancing layer at 3 places in our system:

1. Between clients and the application servers
2. Between application servers and the database replication servers
3. Between aggregation servers and cache servers

Initially, a simple round robin approach can be adopted, which will distribute incoming requests equally among servers.

This LB is simple to implement and does not introduce any overhead.

Another benefit of this approach is that if a server is dead, the LB will take it out of the rotation and will stop sending any traffic to it.

A problem with round robin LB is that it won't take server load into consideration.

If a server is overloaded or slow, the LB will not stop sending new requests to that server.

To handle this, a more intelligent LB solution can be placed that periodically queries backend servers about their load, and adjusts traffic to that server based on the server load.

12. Monitoring

Having the ability to monitor our systems is crucial.

We should consistently collect data to get an instant insight into how our system is doing.

We can collect the following metrics to gather an understanding of the performance of our service:

1. New tweets per day, per second; what is the daily peak?
2. Timeline delivery stats: how many tweets per day, per second our service is delivering
3. Average latency that is seen by the user to refresh their timeline

By monitoring these metrics, we will get information on whether we need more replication, load balancing, or caching.

13. Extended Requirements

How do we serve feeds?

Get all the latest tweets from the people someone follows, and merge/sort them by time.

Use pagination to fetch/show tweets.

Only fetch the top N tweets from all of the people that someone follows. This N will depend on the client’s viewport, since on a mobile device we show fewer tweets compared to a web client.

We can also cache the next top tweets to speed things up.

Alternatively, we could pre-generate a user’s feed to improve efficiency. For details of this, see *Designing Instagram.*

Retweets

With each tweet object in the database, we can store the ID of the original tweet and not store any contents on any retweet objects.

Trending Topics

We can cache the most frequently occurring hashtags or search queries in the last N seconds, and keep updating them every M seconds.

We can rank trending topics based on the frequency of tweets, search queries, retweets or likes.

We can give more weight to topics that are shown to more people.

Who to follow? How to give suggestions?

This feature will improve user engagement.

We can suggest friends of people that someone follows.

We can go two or three levels down to find famous people for suggestions.

We can give preference to people with more followers.

As only a few suggestions can be made at any time, we can use machine learning to shuffle and reprioritise.

ML signals could include people with recently increased follow-ship, common followers, common location, common interests etc.

Moments

Get top news for different websites for the past 1 or 2 hours, figure out related tweets, prioritise them, categorise them (news, finance, entertainment) using ML — supervised learning or clustering.

Search

Search involves indexing, ranking, and retrieval of tweets.

A solution is discussed in *Designing Twitter Search*.