Design Twitter

https://github.com/filipegoncalves/interview-questions/blob/master/systems_design/Twitter.md
Design a simplified version of Twitter where users can post tweets, favorite tweets, and follow other people.

Defining use cases for the system is always useful. From the question, we can immediately spot at least 3 use cases:

- Post a tweet
- Favorite a tweet
- Follow a user

What other use cases / features can we think of? Well, the 3 use cases shown above are writing data into the system. Apparently we never read. At the very least, we have a 4th use case:

- Visit a user's profile (and read his/her tweets)

We could also think of analytics - for example, are we interested in allowing users to see other users' usage statistics, like total number of tweets posted, last seen time and date, ...? We assume that this is not required for now.

Step 2: Requirements and Constraints

Now it is time to do some maths, so that we have some concrete numbers to think about during the design phase. Useful metrics that more or less apply to every design problem include: number of users, storage needs (over 10 years and assuming a 70% capacity model), number of requests/second, how many of these are reads, how many of these are writes, and finally, we consider the read/write ratio.

What is the expected user base size? We assume that our service will have to accommodate 500 million users.

To estimate our total storage needs, we need to consider the different entities in the system. In particular, we need to store:

- Tweets and their identification (let's say an alphanumeric ID)
- Favorite relationships (for example, user X favorited tweet Y)
- Follower relationships (user X followed user Y)
- Misc user metadata (for each user, we store some basic information: name, date of birth, e-mail, location, ...)

We address each of these in turn.

It is expected that our service will handle 100 million new tweets / day. That's 360 billion tweets over 10 years. Each tweet is at most 140 characters long; assuming a character is 1 byte, each tweet takes at most 140 bytes, so we need about 51 TB to store every tweet. We will also have to identify tweets somehow. Assuming we use alphanumeric case-sensitive IDs with symbols in the ranges [a-z], [A-Z] and [0-9], we will need IDs of log_62(360 billion) symbols, which is roughly 6. So, each ID needs 6 bytes; with 360 billion tweets this means we need 2.52 TB to store IDs.

So, to store every tweet along with its ID, we need 54 TB.

What about favorites? Favoriting a tweet requires storing a user ID and a tweet ID. With 500 million users, a 5-byte ID will suffice. Assuming each tweet is favorited 10 times, each tweet incurs an additional (5+6)*10 bytes to store favorite relations. With 360 billion tweets, that's 39.6 TB.

If each user follows 200 users, then each user needs 1000 bytes (5 bytes * 200) to keep track of followers. With 500 million users, this represents 500 GB.

Adding all of these together yields an upper bound of 98 TB for the whole system. 98/7 = 14, so under a 70% capacity model, our system should be able to store 140 TB.

Let's estimate read and write traffic. We assume that there are 350 million requests/day. As mentioned before, there are 100 million new tweets/day, so this leaves us with 250 million read requests/day and 100 million write requests/day. This is a rough approximation, since we are ignoring *favorite* and *follow* requests. This boils down to roughly 4000 requests/sec., out of which 2600 are reads and the remaining 1400 are writes.

Step 3: Design

On a high-level view, like any other system, we break this into two main components: frontend and backend. The frontend component is the public face of the service - it processes clients requests, does some processing if necessary, forwards the requests to the backend, and when the reply is ready, sends it back to the client.

Conceptually, the backend is not very complex. It implements a data store that structures the data efficiently and makes it available to the frontend. We have 2 options for the datastore:

- Use a NoSQL model. We can use a NoSQL database or roll our own custom datastore implementation based on the specific needs of the system. Because our data fits the relational model, this may be harder to think about. However, NoSQL solutions are very popular nowadays and were designed with scale in mind, so in the long term it might be a good choice.
- Use a relational model. Database engines like MySQL seem like the perfect match: we could have a table for users, storing user ID, e-mail, name, date of birth, etc, for each user; a table for favorited tweets, storing pairs of (user ID, tweet ID); a table for follower relationships, storing pairs of user IDs (followee and follower), and a table for tweets

storing tweets IDs and their content. Indeed, many big companies like Twitter and Facebook run MySQL, albeit highly customized.

The advantages of using MySQL here is that it is a mature, tested technology that offers strong guarantees (ACID properties). It is easy to visualize and build the data model for our system because the concepts of tweet, followee / follower and favorites easily map to a relational database model. It also gives us the power to implement new, custom features easily. For example, if all of a sudden we decide that we want to allow users to see how many favorites or followers another user has, it's a matter of adding a button in the user interface and have the application layer handle a new request type that simply queries the database accordingly. In other words, we get the full power of SQL, essentially giving us free reign to look at and analyze the data as we please.

On the other hand, MySQL may have a hard time dealing with such a high volume of traffic. Queries are slow, and disk seeks are frequent. Queries are executed in the context of a transaction, which adds overhead and degrades overall throughput. When we think about scale, it is hard to shard. Common solutions in the MySQL world include master/slave replication or master/master replication, but there is always a master server dealing with writes, which represents a single point of failure and a global bottleneck.

Both models have advantages, but because our data model clearly fits a relational model, we will choose a SQL database engine at this time. The database layout has been described before:

- A single table stores userID <-> metadata associations. Metadata includes things like e-mail, name, birthday, location, etc.
- Another table stores tweets. It associates a tweet ID with the tweet's contents
- Favorites are stored in a table of (userID, tweetID) pairs
- Follower / followee relationships are stored in a table of (followerID, followeeID) pairs

Because we are not worried about scale yet, our database design is fully normalized.

Step 4: Identify Key Bottlenecks and Scale

We have a lot of reads and writes per second - 2,600 reads/sec. and 1,400 writes/sec. - but these numbers are not terribly high. It may be a little too much to expect that a single machine can handle all of these requests, so we should probably deploy a couple of application layer servers behind a load balancer to evenly distribute incoming requests. If each server can handle 400 requests/sec., 10 servers would be enough (although we might want to plan for future growth here; we can equally apply the 70% capacity rule).

But the real problem is the datastore. 140 TB is a lot of data. We can shard our database across a set of servers. If each server can store 2 TB, we would need about 70 servers, assuming an even distribution. The traditional master/slave replication scheme may not be enough here, because a single master is probably not able to serve 1,400 writes/sec. We can perform a hash-based or range-based distribution of keys at the application layer level, and have a bunch of master/slave MySQL clusters.

Finally, given the high read/write ratio, the system could greatly benefit from caching. We can add a cache layer between the application layer and the backend layer