Features:
This is the first part of any system design interview, coming up with the features which the system should support. As an interviewee, you should try to list down all the features you can think of which our system should support. Try to spend around 2 minutes for this section in the interview. You can use the notes section alongside to remember what you wrote.
Q: What are some of the Twitter features we should support?
A: Let's assume that we are looking at posting tweets, following people and favoriting tweets. A user should also be able to see a feed of tweets of his/her followers.
Q: Do we need to support replies to tweets / grouping tweets by conversations?
A: Let's assume we don't need to for this case.
Q: How about privacy controls around each tweet?
A: Not required. Let's assume for this case that all tweets are public.
Q: Do we need to support trending tweets? If so, do we need to support localization and personalization?
A: For this case, lets just assume we are not focussing on building the trending tweets feature.
Q: How about Direct messaging ?
A: No. Lets leave that out for this question. That could be another question by itself.
Q: How about mentions / tagging?
A: Let's assume we don’t need to support mentions/tagging.
Q: Do we need to support a notification system?
A: For the purpose of this question, no.
Estimations:
This is usually the second part of a design interview, coming up with the estimated numbers of how scalable our system should be. Important parameters to remember for this section is the number of queries per second and the data which the system will be required to handle.
Try to spend around 5 minutes for this section in the interview.

Lets estimate the volume of tweets. Assume that our system would be the second most popular tweeting service after Twitter.

Q: What is the number of users and traffic that we expect the system to handle?
A: Twitter does around 500 million tweets per day with 100 million daily active users. Lets assume similar numbers.
Q: How many followers does every user have?
A: The behavior should be similar to Twitter here. Each user has on average 200 followers, with certain hot users having a lot more followers. For example, users like Justin Bieber would have millions of followers.
Q: How many times is a tweet favorited?
A: Assuming the same behavior as Twitter, we can assume that each tweet is favorited twice. However, in this case as well, there will be outliers. There are certain tweets which might be favorited by millions of people.
Q: Assuming the network of users, how many user to follower edge would exist?
A: We had 100 million active users with 200 followers on average. This means 100M x 200 edges = 20B edges
Design Goals:

Latency - Is this problem very latency sensitive (Or in other words, Are requests with high latency and a failing request, equally bad?). For example, search typeahead suggestions are useless if they take more than a second.
Consistency - Does this problem require tight consistency? Or is it okay if things are eventually consistent?
Availability - Does this problem require 100% availability?
There could be more goals depending on the problem. It's possible that all parameters might be important, and some of them might conflict. In that case, you'd need to prioritize one over the other.
Q: Is Latency a very important metric for us?
A: Yes. A twitter like system needs to be fast, especially when you are competing with Twitter.
Q: How important is Consistency for us?
A: Not really. Assuming a lot of activity on this system, if I miss out on a tweet of a person I am following every now and then, its not the end of the world. Compare this to direct messaging where consistency is extremely important.
Q: How important is Availability for us?
A: Yes. If Twitter becomes unavailable, it becomes a news. As a product, it needs to be highly available.
Skeleton of the design:
The next step in most cases is to come up with the barebone design of your system, both in terms of API and the overall workflow of a read and write request. Workflow of read/write request here refers to specifying the important components and how they interact. Try to spend around 5 minutes for this section in the interview.
Important : Try to gather feedback from the interviewer here to indicate if you are headed in the right direction.

As discussed before, there are 4 major operations we need to support:

- Posting new tweets
- Following a user
- Favoriting a tweet
- Get the feed for a user

Q: What would the API look like for the client?
Q: What data would need with every Tweet we fetch?
A: We should have the content of the tweet, the person who posted the tweet, the timestamp when tweet was created and number of favorites for the tweet.
Q: Would we need all the user profiles of users who have favorited a tweet?
A: Given thats a lot of data to fetch, we can be more intelligent about it and just fetch top 2 people in the list. In this scheme, we would show every tweet as 200 favorites which on hover shows Favorited by X, Y and 198 others
Q: How many tweets should we fetch at a time?
A: At a time, only a certain number of tweets will be in the viewport (lets say 20). Lets call it a page of tweets. For a user, we would only want to fetch a page of tweets at a time.
Gotcha: Would the page size remain constant across different situations?

Probably not. The page size would be different across clients based on screen size and resolution. For example, a mobile's page size might be lower than that of a web browser's.
A: The first 3 operations end up doing a write to the database. The last operation does a read. Following is an example of how the API might look like :

- Posting new tweets : addTweet(userId, tweetContent, timestamp)
- Following a user : followUser(userId, toFollowUserId)
- Favorite a tweet : favoriteTweet(userId, tweetId)
- TweetResult getUserFeed(user, pageNumber, pageSize, lastUpdatedTimestamp)
  where TweetResult has the following fields :
  TweetResult {

  List(Tweets) tweets,
  boolean isDeltaUpdate

  }
  Tweet {

  userId,
  content,
  timestamp,
  numFavorites,
  sampleFavoriteNames

  }

There could be other APIs as well which would help us fetch the most recent tweets of a user, or fetch the followers for a tweet.
Q: How would a typical write query (addTweet) look like?
A: Components:

- Client (Mobile app / Browser, etc) which calls addTweet(userId, tweetContent, timestamp)
- Application server which interprets the API call and tries to append the tweet to user's tweet with the timestamp in the database layer.
- Database server which appends the tweet

Q: How would a typical read query (getUserFeed) look like?
A: Components:

- Client (Mobile app/Browser, etc) which calls getUserFeed
- Application server which interprets the API call and queries the database for the top user feed.
- Database server which looks up the followers' tweet to get the result.

Deep Dive:
Lets dig deeper into every component one by one. Discussion for this section will take majority of the interview time(20-30 minutes).

Lets dig deeper into every component one by one.

Application layer:

Think about all details/gotchas yourself before beginning.
Q: How would you take care of application layer fault tolerance?
Q: How do we handle the case where our application server dies?
A: The simplest thing that could be done here is to have multiple application server. They do not store any data (stateless) and all of them behave the exact same way when up. So, if one of them goes down, we still have other application servers who would keep the site running.
Q: How does our client know which application servers to talk to. How does it know which application servers have gone down and which ones are still working?
A: We introduce load balancers. Load balancers are a set of machines (an order of magnitude lower in number) which track the set of application servers which are active (not gone down). Client can send request to any of the load balancers who then forward the request to one of the working application servers randomly.
A: If we have only one application server machine, our whole service would become unavailable. Machines will fail and so will network. So, we need to plan for those events. Multiple application server machines along with load balancer is the way to go.

Database layer:

This is the heart of the question. In the skeleton design, we assumed that the database is a black box which can magically store or retrieve anything efficiently. Lets dig into how we will build that magic black box.

Q: What data do we need to store ?

A:

- For every tweet, we need to store content, timestamp and ownerID.
- For every user, we need to store some personal information (Name, age, birthdate, etc.)
- We need to store all u1->u2 follower relations.
- We need to store all user_ids against a tweet of users who have favorited the tweet.

Q: RDBMS or NoSQL?
Q: Are joins required?
A: NoSQL databases are inefficient for joins or handling relations. As such, NoSQL databases store everything in a denormalized fashion. In this case, we do have relations like

- user -> followers
- tweets -> favorited by users

SQL seems to win on this parameter on ease of use.
Q: How much data would we have to store?
A: If the size of the data is so small that it fits on a single machine's main memory, SQL is a clear winner. SQL on a single machine has next to zero maintenance overhead and has great performance with right index built. If your index can fit into RAM, its best to go with a SQL solution. Lets analyze our current case :

- Size of tweets :
  Number of tweets per day : 500 million
  Maximum size of a tweet : 140 chars + 1 byte for timestamp + 1 byte for userId = 142 bytes
  Provisioning for : 5 years = 365 * 5 days
  Space required : 142bytes * 500M * 365 * 5 = 129.5TB

- Size of user - follower relation :
  Assuming total of 1 Billion users and every user has 200 followers on average, we end up with 200B total connections. To store it, we would need 200B * 2 bytes (size of 2 userIDs) = 400G.

- Size of tweet to favorites relation :
  Average number of favorites per tweet : 2 (Ref. Estimations section)
  Total number of tweets daily : 500M
  Provisioning for : 5 years = 365 * 5 days
  Space required : (2 bytes + 1 byte for tweetId) * 500M * 365* 5 = 2.7TB
  So, total space required is close to 130TB. That'd definitely not fit on a single machine's hard disk.

Q: How important is technology maturity?
A: SQL DBs like MySQL have been around for a long time and have hence been iterated enough to be very stable. However, most NoSQL databases are not mature enough yet. Quoting an article from PInterest Engineering Blog :
We intentionally ran away from auto-scaling newer technology like MongoDB, Cassandra and Membase, because their maturity was simply not far enough along (and they were crashing in spectacular ways on us!).

So, a SQL solution will have a sharding overhead. Most NoSQL solutions however are built with the assumption that the data does not fit on a single machine and hence have sharding builtin. NoSQL wins on this parameter.
A: Things to consider :

- Are joins required?
- Size of the DB
- Technology Maturity

In practice, the score is equal for both RDBMS or NoSQL for this one. In theory, NoSQL would be a better fit.
We can choose either to proceed further. Lets go with a relational DB like MySQL for this one.
Q: What would the database schema look like?
A: Always be prepared for this question in cases where the schema might be a bit more elaborate. We have two main entities: users and tweets. There could be two tables for them. Table users would have personal information about the user. A sample table schema for that could look like the following :
Table **users**

- ID (id) - primary key
- username (username) - unique key
- First Name (first_name)
- Last Name (last_name)
- password related fields like hash and salt (password_hash & password_salt)
- Created at (created_at)
- Last updated at (updated_at)
- Date of Birth (dob)
- description (description)

Tweets should be slightly simpler:
Table **tweets**

- ID (id) - primary key
- content (content)
- date of creation (created_at)
- user ID of author (user_id)

Now, lets look at the relations that we need to model :

Follower relation (User A follows another user B)
Table **connections**

- ID of user that follows (follower_id)
- ID of user that is followed (followee_id)
- date of creation (created_at)

Favorite : A user can favorite a tweet.
Table **favories**

- ID of user that favorited (user_id)

- ID of favorited tweet (tweet_id)
- date of creation (created_at)

Now, based on the read API queries, we can look at the additional index we would need :

Get the feed of a user - This would require us to quickly lookup the userIds a user follows, get their top tweets and for each tweet get users who have favorited the tweet.
This means we need to following index :

- An index on follower_id in connections to quickly lookup the userIds a user follows
- An index on user_id, created_at in tweets to get the top tweets for a user (where user_id = x sort by created_at desc)
- An index on tweet_id in favorites

Q: Now the bigger question, How would we do sharding?
Question: Approach1: How can we shard on users?
A:
Detail : Whats stored in each table :

- users : part of the table with user_ids which belong to the shard
- tweets : part of the table with author_ids which belong to the shard (Or in other words, tweets by the users in the current shard)
- connections : All entries where follower_id belongs to the current shard
- favorites : All entries where tweet_id belongs to the tweets table in this shard

Pros:

- Equal load distribution
- Cheap writes : All of the write queries are simple and rely on just one shard (Assuming tweet favorite API encodes the tweet owner ID in the tweet ID when sending request).

Cons:

- While looking up the userIds a user follows is easy on the machine, getting the top tweet for each of those userIds would require querying different shards.
- Even when we need to favorite a tweet, finding the tweet would require us to query all the shards. We can work around it, however, by encoding owner_id with the tweet_id from the client.

Question: Approach2: Can we shard on recency(timestamp)?
A: Shard on recency(timestamp) -
Most recent tweets in the most recent shard. The idea is that most of the time we are only working with most recent tweets. Its rare to dig up tweets which are more than a few weeks old. New tweets are requested most frequently.

- users : All of users table resides in one shard separately.

- tweets : This table is sharded across shards by recency. When the most recent shard starts getting full, we create a new shard and assign the new incoming tweets to the newly created shard.
- connections : All of the table resides in the shard with users.
- favorites: Stored with the tweets in their shard

Pros:

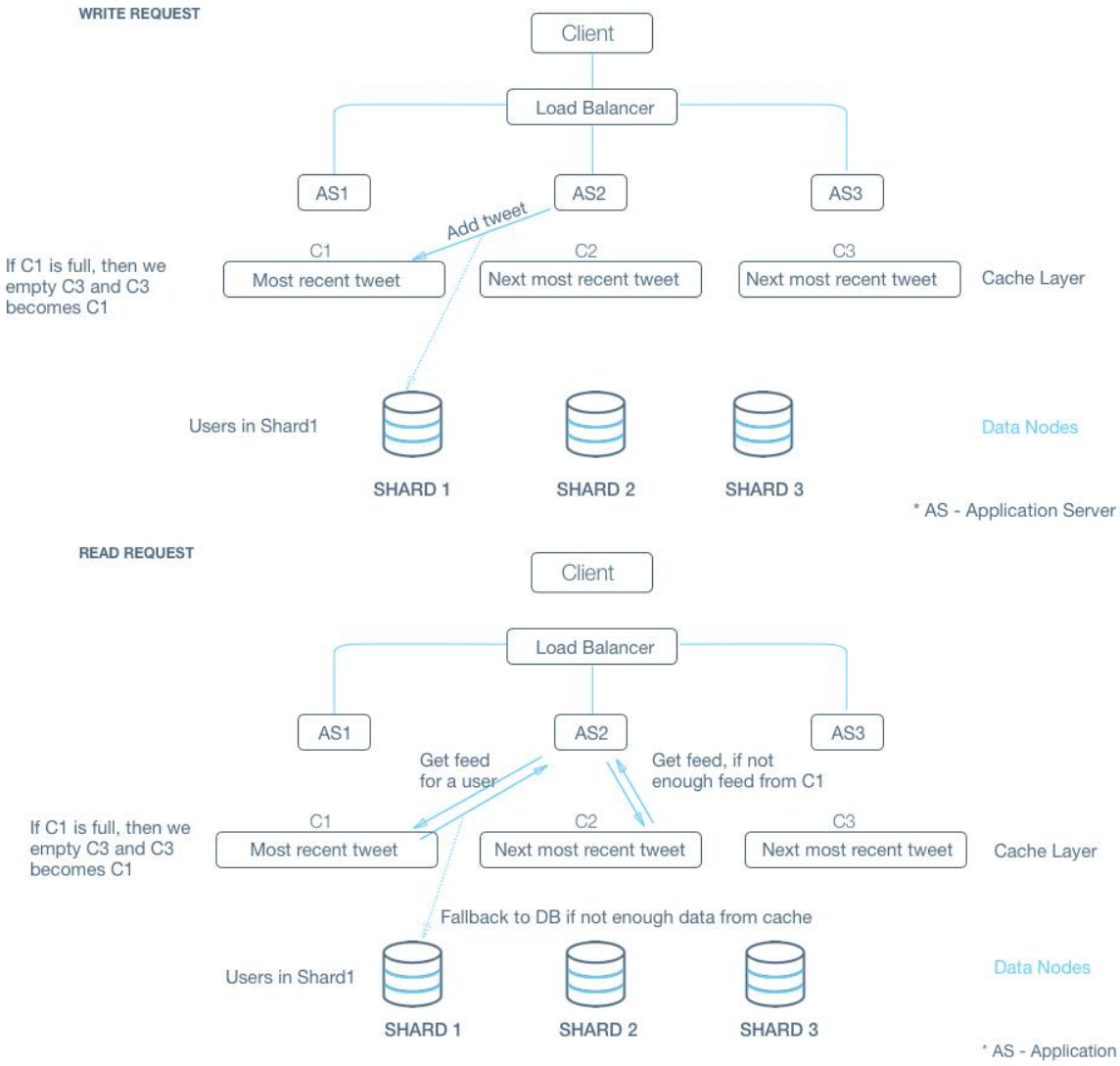
- Fetching the user feeds requires just querying 2 shards (users and tweets). More reliable and has low latency. Most of the queries would only interact with 2 shards.

Cons:

- Load imbalance : The most recent tweet shard will be handling almost all of the traffic while the other shards will remain idle.
- Huge maintenance overhead : Every time, you need to create a new shard, you'll need to allocate new machines, setup replication and make things switch almost instantly so that no downtime is induced. All in all, a nightmare for DBAs at that scale.

A: We have already established earlier that we would need to shard as data would not fit into a single machine. The read query we are optimizing :

Get the feed of a user - This would require us to quickly lookup the userIds a user follows, get their top tweets and for each tweet get users who have favorited the tweet.
We can model our data with two basic approaches, sharding based on recency of tweet or based on users, we will call these approaches Approach1 and Approach2 respectively. Our answer consists of a hybrid approach of Approach1 and Approach2, so we highly recommend you to go through hints which explains both of them in detail.
Both solutions have their downsides and don't seem ideal. What is a clean solution then? Lets go back to our design goals. We want low latency and high availability. Consistency is not a big deal for us (If I miss a tweet once in a while in the feed, its not the end of the world).
With that in mind, we can look for a hybrid model. We will definitely need to heavily cache. We can have a cache which simulates the recent shard in Approach 2 and a DB which stores stuff as in Approach 1. The idea being that most reads will be served by the cache itself and it has the collection of all recent tweets. In the rare case of not so recent tweet, we will go to the DB and in such cases, latency outliers are alright. Notice that the DB writes would be cheap as discussed in Approach 1.



Q: Do we need special handling for spiky cases (Users with unusual number of followers / Tweets with likelihood of getting unusually high number of favorites) ? Think about the case when Katy Perry (with more than 70M followers) tweets.
A: Lets look at both cases one by one.
Let's say Katy Perry tweets. Following is what happens :

- We write the tweet to the shard where Katy Perry belongs. Not a problem.
- We add it to the recent tweets cache. Again, not a problem.
- As all followers get their feed update by specifically requesting for an update, the resultant change is that a lot of followees will get an update when they request for it. This should manifest as an uptick in the upload bandwidth. In the worst case, assuming that 30% of the followers are online at a time, we would need around 3G of upload bandwidth which is a really small number for a datacenter.

Lets look at really popular tweets now. They'll have an unusually high rate of being favorited (The highest being 3M total favorites). This means a really high rate of write to the shard which can cause deadlocks. We can add some optimizations here if required in terms of batching the updates to favorites table in a queue before flushing them. Nitty Gritty : Would the queue be persistent? If not, what happens if the machine dies. That would cause data loss. If yes, where does the queue reside? How do you merge the query results?

Q: How would we handle a DB machine going down?

A: As stated in design goals, we need to make sure our system is more available at all times. We had sharded the database based on users. We can have a replica for each of them which follows the updates happening on the master database shard. When the master goes down, the slave can take over. Now there is a problem here. What if there were some updates which the slave had not caught up to yet. Do we lose that information? We can take a call either way. If we are particular about getting the data back, we know that we can get that information from the cache layer and resolve stuff on the DB layer.