# The Twitter Problem

In this example we will look at a concrete system design interview question and will try to go from a statement to a sound system design, which satisfies the initial conditions put by the interviewer. We will focus on the specifics of the system design interview, so that you know how to give your best in this particular environment. This e-book will mention technical notions and techniques but we will not go into very deep technical details about them. It will be your task to become more familiar with the different technical ideas used throughout the solution. You can also take a look at the system design learning section from HiredInTech if you haven't done that yet.

Are you ready? Let's begin with a very short statement that your interviewer gave you at the start of the interview.

## Statement

After the introductory few words your interview starts and the interviewer gives you the following task:

"Design a simplified version of Twitter where people can post tweets, follow other people and favorite* tweets."

* More about favoriting tweets on Twitter: https://support.twitter.com/articles/20169874-favoriting-a-tweet

## Clarifying Questions

The way it is given, this problem is very unclear. At first, it may seem that you don't need more than this one sentence. After all, it describes what the system should be doing. But think about it - being the architect and developer you need to know much more in order to make the proper decisions. Let's start thinking about them and get some answers from our imaginary interviewer.

First of all, how many users do we expect this system to handle? Our interviewer says:

"well… to make things interesting, let's aim for **10 million users** generating around **100 million requests per day**"

That's great but let's continue. Since we have the notion of following someone, how connected will these users be? After all, they will form a graph with the users being the nodes and the edges will represent who follows whom. Our imaginary interviewer has an answer for this too:

"we expect that each user will be **following 200 other users on average**, but expect some extraordinary users with tens of thousands of followers"

Now, our graph is starting to look like something more concrete and we could start thinking of a design, which would accommodate its size.

We have another dimension in the application: tweets. Producing new tweets and favoriting them should be the most common write operations in the application. But how many requests will that generate? Our interviewer says:

"Hm, that's hard to say… we expect that there will be a **maximum of 10 million tweets per day** and each tweet will probably be favorited twice on average but again, expect some big outliers."

Let's make a few very simple calculations with the information we just received. We will have around 10 million users. Average number of followed other users is 200. This means that the network of users will have about 200 * 10 million edges. This makes **2 billion edges**. If the average number of tweets per day is 10 million the number of favorites will then be **20 million**.

So far so good, we've got some numbers instead of nothing. In real life we can never have the exact numbers but it is always nice to have some predictions so that we know what kind of system we want to design. Imagine that the interviewer had in mind some very skewed version of Twitter in which there were only 30,000 users who produced thousands of tweets a day each and were all very strongly connected. This would probably throw us in a different direction for some of the design decisions. Each such problem could have many different interesting versions and it is amusing to think about each of them separately. In this case we will focus on the description given above.

Ok, so what else prevents us from getting our hands dirty and starting work on some high level design? In addition to the expected size of different things, for a system like that it will be important to know what availability is expected and what response times are tolerable. Naturally, our interviewer wants to have a system, which loads pretty quickly. None of the operations described above should take more than a few hundred milliseconds. The system should be online all of the time. It is not good to design into this application any downtimes.

That took a while to describe. There are many other questions that one could ask and we will get to some more in the next sections just to illustrate that sometimes questions pop up when you start thinking about the solution. However, interview time is very limited and you need to find the balance. This session where you ask questions and get answers should probably not last more than just a few minutes assuming your interview lasts 40-45 minutes and especially if this is not the only problem you get in it. Keep that in mind and practice will let you find your sweet spot.

Remember that not all system design questions will contain one or two sentences. In some cases the interviewer may give you a detailed specification, which includes more numbers, information about the UI and other important requirements. Even in such cases it is part of your task to extract what is useful for you and figure out if something important is missing.

To summarize, here are some numbers we now know:

- 10 million users
- 10 million tweets per day
- 20 tweet favorites per day
- 100 million HTTP requests to the site
- 2 billion “follow” relations
- Some users and tweets could generate an extraordinary amount of traffic

Now, we continue to our high-level system design!

## High-level Design

It’s a good idea to start from the top and define the main parts of our application. Then, for the high-level design, we can discuss their subparts.

As it is often the case, we can divide our architecture in two logical parts: 1) the logic, which will handle all incoming requests to the application and 2) the data storage that we will use to store all the data that needs to be persisted. If you learn enough about popular software systems, especially the ones that are accessed through a web browser you will notice that such a breakdown is quite popular and natural.

Now we begin with the first part. What should it take care of? The answer will come straight from the problem statement and the clarifying comments that the interviewer has made. At this point it has become obvious that our application will need to handle requests for:

- posting new tweets
- following a user
- favoriting a tweet
- displaying data about users and tweets

The first three operations require things to be written somewhere in our database, while the last is more about reading data and returning it back to the user. The last operation will also be the most common one as can be inferred from the numbers discussed in the previous section.

Let’s make it more concrete how the user will interact with our application and design something that will support such interactions. We can describe to the interviewer how we imagine this to work. For example, there will be a profile page for each user, which will show us their latest tweets and will allow for older tweets to be shown. Each such page will have a button for following the user. There will also be a button at the top of the page, which will allow logged in users to open a dialog box and write a message in it. After they click a button the message will be stored and will appear on their profile page. Once we have confirmed this very rough description of the UI we can begin with the design of the back-end supporting everything.

This kind of brief description of how you imagine the application to look could be useful because you make sure that you and the interviewer are on the same page. A simple drawing is also very helpful. You can look at one example in the Appendix section.

## Handling user requests

We know that the expected daily load is 100 million requests. This means that on average the app will receive around 1150 requests per second. Of course, this traffic is expected to be distributed unevenly throughout the day. Therefore our architecture should allow the app to handle at least a few thousand requests per second at times. Here comes the natural question about what is needed to handle this kind of load. As you may have guessed the answer depends on several things.

One aspect is the complexity of the requests that the application receives. For example, one request could require just one simple query to a database. It could also need a few heavier queries to be run and some CPU-intensive computations to be performed by the application.

Another aspect could be the technologies used to implement the application. Some solutions are better at concurrency and use less memory than others. In a situation like this one you should have some common knowledge about what kind of load can be handled by a single machine for sure and what is load that definitely needs more computing power. To build that it would help to spend some time reading about different real-life examples. Some people also compare the throughput they have achieved in different setups using different web frameworks, hosting services and so on. Any information like that could help you build better instincts about this matter. Finally, your personal work experience is always the best way to learn about these things.

When the expected load seems nontrivially high you can always consider scaling up or out. Scaling up would be an approach in which you decide to get a beefier and more expensive server, which is capable of handling the expected load. This approach has a natural limit for its scalability because after a given point the hardware of one machine just isn't capable of handling all the requests. In some cases doing that makes sense.

Scaling out would involve designing your architecture in a way that spreads the computations over a number of machines and distributes the load across them. This approach is better at scaling if your load grows significantly. However, it involves some other complications.

One more advantage of using more than one server for your application is the resilience that you add to your whole system. If you only have one machine and it goes down your whole application is down. However, if there are 10 servers handling requests and one or two go down the others will be able to handle at least part of the load if not all of it.

In our particular problem we would definitely suggest using a load balancer, which handles initial traffic and sends requests to a set of servers running one or more instances of the application.

One argument is the resilience that we gain as mentioned above. Another one is that our application doesn't seem to have any special requirements in terms of very high memory or CPU usage. All requests should be serviceable with code that runs on regular commodity machines. Using many such machines in parallel should give us the flexibility to scale out a lot.

How many servers we should have is something that can probably be determined experimentally with time. Also, if we set things up properly it should be fairly easy to add new servers if that is needed.

Under these links you can read more about load balancing and check out examples of load balancing solutions:

- Wikipedia's explanation
- Amazon's service for traffic load balancing
- nginx - HTTP load balancer
- HAProxy - TCP/HTTP load balancer

We'll say more about load balancing when we start analysing the bottlenecks in our architecture. For the moment, at this high level, we'll just mention that this is a needed component. Behind the load balancer we will be running a set of servers that are running our application and are capable of handling the different requests that arrive.

The application logic itself will most likely be implemented using some web framework, which in general allows us to write apps handling HTTP requests, talking to a database and rendering the appropriate HTML pages requested by the user. The details of which technology will be used are most likely not going to be important for this type of interview question. Nevertheless, it's strongly advised that you get a good understanding of the different types of modern web frameworks and how they work. This book does not cover such material but some things to look at are Node.js, Ruby on Rails, Angular.js, Ember.js. React.js, etc. This does not mean that you need to get down to learning these in detail but rather to take a look at the existing ecosystem of technologies, how they interact with each other and what are the pros and cons.

Now we have a load balancer and a set of application servers running behind it. The load balancer routes requests to the servers using some predefined logic and the application servers are able to understand the requests and return the proper data back to the user's browser. There is one more major component for our high-level architecture to be complete - the storage.

## Storing the data

We need to store data about our users and their tweets to make the application complete. Let's quickly look at what needs to be stored. First of all, users have profiles with some data fields attached to them. We'll need to store that. Each user has a set of tweets that they have produced over time. Moreover, users can follow other users. We need to store these relationships in our database. Finally, users can mark tweets as favorite. This is another kind of

relationship between users and tweets. The first one was recording users as authors of tweets. The second one will record users who favorited a tweet.

Obviously, there are some relations between our data objects - users and tweets. Let's assess the approximate size of the data to be stored. We said that we expect around 10 million users. For each user we'll have some profile data to store but overall that kind of data shouldn't be an issue for us. Tweets will be generated at an average speed of 10 million per day. This makes about 115 per second. Also, for a single year there will be 3.65 billion tweets. So, let's aim for a solution that can store efficiently at least 10 billion tweets for now and the incoming traffic mentioned above. We didn't really ask how big a tweet can be. Maybe it's safe to quickly ask that now and let's assume our interviewer told us it's the same as it is for the real Twitter - 140 characters. This means that for tweets we want to be able to store at least 140 * 10 bln = 1.4 trillion characters or **around 2.6 terabytes** if we assume 2 bytes per character and no compression of the data.

Finally, there are the connections between the users and the favorites of tweets. As we mentioned above the connections should be around 2 billion and each connection will most likely just contain two IDs of users where the first follows the second. So very likely it would be enough to store two 4-byte integer fields, making 8 * 2 bln = 16 bln bytes or **16 gigabytes**.

The favorites are expected to grow at a rate of 20 mln per day. So, for a year there will be 7.3 bln such actions. Let's say we want to be able to store at least 20 bln such objects. They can probably just point to one user and one tweet through their IDs. The IDs for the tweets will probably need to be 8 byte fields while for the users we could use 4 bytes only. This is because our tweets will grow a lot in number. So, our very rough calculation gives us 12 * 20 bln = 240 bln bytes or **240 gigabytes**.

After this quick analysis it is obvious that the tweets will take up the majority of our storage's space. In general, you don't need to make very detailed calculations especially if you don't have much time. However, it is important to build a rough idea about the size of the data that you will need to handle. If you don't figure that out any design decision at the higher or lower level may be inappropriate.

Now back to the storage. Our data has a number of relations between its main objects. If we decide to use a relational database for storing users, tweets and the connections between them this could allow us to model these relations easily. We know that the expected size of the data to store is around 2.6 - 2.7 terabytes. Real-life examples show that famous companies like Twitter and Facebook manage to use relational databases for handling much bigger loads than that. Of course, in most cases a lot of tuning and modifications were required. Below are some useful links related to how big companies use relational databases for large amounts of data:

- How Twitter used to store 250 million tweets a day some years ago
- How Facebook made MySql scale
- WebScaleSQL - an effort from a few big companies to make MySql more scalable
- Twitter's branch of MySql on GitHub
- Facebook's branch of MySql on GitHub

Let's say we decide to use a relational database like MySql or Postgres for our design.

The data that will be stored and the rate of the queries it will receive are not going to be absurdly high but they are not going to be trivial either. In order to handle the incoming read requests we may need to use a caching solution, which stands in front of the database server. One such popular tool is memcached. It could save us a lot of reads directly from the database.

In an application like ours it is likely that a given tweet or a user's profile becomes highly popular and causes many requests to be sent to our database server. The cache solution will alleviate such situations by storing the popular bits of data in memory and allowing for very quick access to them without the need to hit the database.

It is possible that at this moment the interviewer stops you and throws a question about what you were just talking about. For example, you just mentioned using a caching solution and imagine that the interviewer asks you:

"Sounds good, but could you tell me more about why reading from the cache would be better than just reading from the database?"

It is perfectly fine and expected to receive such questions from your interviewer throughout your discussion. You need to be prepared with enough knowledge about the methods and technologies that you use, so that you can justify their usefulness and talk about how they work. It is also very important to be able to explain why one solution is better than its alternatives and this way motivate your design choices.

To answer the question asked, we could say that a database stores data on disk and it is much slower to read from disk than from memory. A solution like memcached stores data in memory, which provides way faster access. We would need to clarify further that databases usually have their own caching mechanisms but with memcached we have better control over what gets cached and how. For example, we could store more complex pieces of data like the results from popular queries.

It is vital to have a good understanding of the differences in read/write access to different types of storage mechanisms. For example, you need to know how hard drive speeds compare to RAM to CPU cache. And it is worth mentioning that nowadays people are using a lot of SSDs, which provide better parameters than the older spinning ones.

Going further, in order to make it possible to answer read queries fast we will definitely need to add the appropriate indexes. This will also be vital for executing quick queries joining tables. Considering the size of the data we may also think about partitioning the data in some way. This can improve the read and write speeds and also make administration tasks like backups faster.

If you intend to be interviewing at places where relational databases are used, make sure you get comfortable with the main aspects of using them. You should be familiar with the general organization of a database schema, the ways different real-life situations are represented through tables and how indexes come into play, so that the planned queries run fast enough with the expected data size. Throughout this example we will talk about some other interesting topics related to scaling a relational database.

At this point we have a pretty good high-level architecture, which is built considering the important dimensions and requirements of our application. Described like this it looks like something that could work. Most likely you will have drawn one or more simple diagrams for your interviewer to make things more understandable. Now may come the time when you need to start digging deeper into some of the moving parts of this design. Or the interviewer may decide to tighten some of the initial requirements to see how you can alter your system design to accommodate the changes. We will look at a few such issues in the next section, which focuses on various bottlenecks and scalability issues.

Finally, notice that we never dug too deep into any of the aspects of the high-level design. This is our goal - we want to draw the whole picture rather quickly within a few minutes and to make sure our interviewer agrees with the ideas that we presented. If they explicitly want you to focus on a specific aspect of the design, then go ahead and do what they ask for. Otherwise, our advice is to not get into the details of any specific part of the application until everything else is outlined at the higher level. Once we've built this bigger picture of the system design the interview could go into more details about specific parts of the system.

## Low-level issues

Let's assume that we've shaped the main parts of our Twitter-like application. In a real-life interview situation this would have been more like a discussion with the interviewer where you talk and they interrupt you with questions and comments. It is ok to have to clarify things that did not become apparent to the interviewer. It is also normal to not get everything right from the first time. Be prepared to accept suggestions from the interviewer or to have your design challenged even if it is sound.

All of the above discussion related to the so-called high level design may seem like it would take a lot of time to describe and discuss. This may be true if done inefficiently. Remember that at the interview you are usually given a very limited amount of time. If you want to be able to fit within that time you will need to practice solving system design problems, talking about your solutions, computing things like expected load, storage needed, required network throughput

and so on. With time you will get the hang of it and will become better at having this special kind of discussion that the interview requires.

It is very likely that your interviewer would be interested to hear more details about a particular part of your designed system. Let's look at a couple such aspects.

## Database schema

If you've picked to use a relational database one possible topic for a more detailed discussion would be the schema that you intend to use. So, your interviewer asks:

"If you're going to use a relational database for storing all the data could you draft the tables and the relations between them?"

This is something that is very common and you should be prepared for such questions. Let's look at what we could draw and explain in this case.

We have two main entities: users and tweets. There could be two tables for them. For the users we would create a table like that with some column names suggested in brackets:

**Table `users`**

- ID (`id`)
- username (`username`)
- full name (`first_name` & `last_name`)
- password related fields like hash and salt (`password_hash` & `password_salt`)
- date of creation and last update (`created_at` & `updated_at`)
- description (`description`)
- and maybe some other fields...

Tweets should be slightly simpler:

**Table `tweets`**

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

Perhaps one can think of other values but this should be enough in our case.

These two entities have several types of relations between them:

1. users create tweets
2. users can follow users
3. users favorite tweets

The first relation is addressed by sticking the user ID to each tweet. This is possible because each tweet is created by exactly one user. It's a bit more complicated when it comes to following users and favoriting tweets. The relationship there is many-to-many.

For following user we can have a table like that:

**Table `connections`**

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

Let's also add a table, which represents favorites. It could have the following fields:

**Table `favorites`**

- ID of user that favorited (`user_id`)
- ID of favorited tweet (`tweet_id`)
- date of creation (`created_at`)

Now that we have this rough idea about the database tables we will need, our interviewer could ask us to think about what else is needed to serve the load of expected queries. We already discussed with some numbers the expected sizes of the data. There is also a pretty good idea about the types of pages that the application will need to serve. Knowing this we could think about the queries that will be sent to our database and to try to optimize things so that these queries are as fast as possible.

Starting with the basics there will be queries for retrieving the details of a given user. Our users' table above has both `id` and `username` fields. We will want to enforce uniqueness on both because IDs are designed to be unique and will serve as a primary key on the table and usernames are also meant to be different for all registered users. Let's assume that the queries to our data will be filtering users by their username. If that's the case we will definitely want to build an index over this field to optimize the times for such queries.

The next popular query will fetch tweets for a given user. The query needed for doing that will filter tweets using `user_id`, which every tweet has. It makes a lot of sense to build an index over this field in the `tweets` table, so that such queries are performed quickly.

We will probably not want to fetch all tweets of a user at once. For example, if a given user has accumulated several thousand tweets over time, on their profile page we will start by showing

the most recent 20 or something like that. This means that we could use a query, which not only filters by `user_id` but also orders by creation date (`created_at`) and limits the result. Based on that we may think about expanding our index to include the `user_id` column but to also include the `created_at` column. When we have an index over more than one column the order of the columns matters. If our index looks like that: `<user_id, created_at>`, making a query filtering by just `user_id` will take advantage of the index even though we are not filtering by the second column. So, such an index will allow us to filter either by just `user_id`, or by both columns. This will allow us to fetch all tweets authored by a given user or to isolate just the tweets created in a given time frame. Both will be useful queries for our application.

For each user we will want to show the users that they follow and the users that follow them. For this we will need the table `connections`. To get the users followed by someone we can simply filter the `connections` table by the column `follower_id`. To get the users following someone we can filter by `followee_id`. All this means that it will make a lot of sense to build two indexes in this table - one on `follower_id` and another one on `followee_id`. Voila, we are ready to show the connections that each user has in our application. Like for tweets you can figure out how to fetch the connections in a paginated manner.

What about favorited tweets by a user? We will definitely want to see something like that. For this we will need to use the table `favorites`. We will need to join favorites with tweets and to filter by `user_id` for the user whose favorites we want to fetch. The columns used for joining will be the `tweet_id` in `favorites` and `id` in `tweets`.

The above means that it makes sense to add two indexes - one on the `user_id` column and one on the `tweet_id` column.

Having discussed these base use cases, our interviewer suggests that you think about one more possible situation:

> "Do you think you could support with our database design the ability to display a page for a given user with their latest tweets that were favorited at least once?"

Let's think about that. We will need to use the table `favorites` but instead of filtering by the `user_id` column we will have to get the user_id from the `tweets` table. This means that we will again join the two tables - `favorites` and `tweets` - and this time will filter by the `user_id` column in `tweets`. It seems like we have the needed indexes in place already. One is in `favorites` over `tweet_id`, which will help us join this table with `tweets`. In table `tweets` the `id` field is a primary key so it has an index on it. And we also have an index on `user_id` in `tweets`, so this will help us filter by user.

One could think of other such query patterns and see what needs to be done in the database schema to address them. For example, for your own pleasure, you can think about supporting a

page, which shows for a given user a list of recent tweets that this user either authored or favorited, ordered by date of creation.

Here is another very good candidate for a homework exercise: for a given user A build a page showing the recent tweets authored by users that A is following. What would the query be, which indexes will it use and does it need more indexes to be added?

It is also worth mentioning that after creating the indexes our write queries will become slightly slower but the benefits that we get for our read operations are so significant with the amounts of data we have that we have no other choice.

In general, it is a useful skill to be able to design relational database schemas, to optimize them and to have a discussion about all that. We could cover much more on this topic but it is better to use specialized books and online resources for that. Be prepared to defend your database schema designs during your interviews. As mentioned already, if circumstances allow, it is always helpful to draw something on paper or a whiteboard. We've added a very simple diagram of our database schema in the Appendix section.

That was all great but with the expected size of our data we may have a separate discussion with our interviewer about partitioning the database in some way as a further step. We will touch on this a bit later in the text.

## Building a RESTful API

We have a simple plan for what our schema will be like. Another thing that our interviewer could be interested in is how our front-end would "talk" to the back-end system. Probably the most popular answer nowadays would be by exposing a RESTful API on the back-end side, which has a few endpoints returning JSON objects as responses. Many web applications are built this way nowadays and it is a good idea to take a look at what RESTful APIs are about if you don't feel confident about this area.

Let's see what we can draft for our particular task. The API endpoints will likely be built around the data entities that we have and the needs of the user-facing part of the application.

We will definitely need to fetch the profile details for a given user. So we could define an endpoint that looks like that:

```
GET /api/users/<username>
```

It will return a JSON object containing the data fields of a given user if such was found and will return status code 404 if no user matched the supplied **username** value.

To get the tweets of a given user ordered by date, we can expose an endpoint like that:

```
GET /api/users/<username>/tweets
```

This could be modified by supplying some query parameters telling the back-end that we want paginated tweets. For example, the default behavior will return the most recent 20 tweets only. But we could decide to load the next 20, and the next 20 and so on. A query fetching a subsequent "page" with queries could look like that:

```
GET /api/users/<username>/tweets?page=4
```

This tells the back-end to fetch the 4th page with 20 tweets, instead of the default behavior - 1st page with 20 tweets.

Let's continue! Our front-end will also be asking about the users following a given user and followed by that user. Here are two possible endpoints for that:

```
GET /api/users/<username>/followers
GET /api/users/<username>/followees
```

So far we defined a few GET requests. Let's look at creating new data. For example we will need an endpoint for posting a new tweet:

```
POST /api/users/<username>/tweets
```

Or how about following a given user:

```
POST /api/users/<username>/followers
```

If we look at the tweets, we may need API endpoints that cover them, too. For example it will be useful to be able to see a list of all users that favorited a tweet:

```
GET /api/users/<username>/tweets/<tweet_id>/favorites
```

And favoriting a twee can be done throught:

```
POST /api/users/<username>/tweets/<tweet_id>/favorites
```

As you can see, there will be a number of such endpoints that will be needed to make our application tick. Our endpoints are revolving around the main entities that we have. We highly recommend that you read more about building good RESTful APIs and perhaps think about more scenarios in which you will need to define additional endpoints.

Of course, we will need some sort of authentication to be put in place, so that we make sure that not everyone can query our exposed API.

Now, let's continue with some possible scenarios that the interview could follow. Imagine that you have outlined your ideas and the interviewer seems happy with what you have offered. They may want to test your ability to spot bottlenecks and to handle the need to scale your system. Probably one could think of many complications to add to the problem statement and to lead the discussion in various directions. We will cover a few things that seem quite normal to consider and likely to happen in a typical interview.

## Increased number of read requests

We have our implementation of the system and it handles everything perfectly. But what would happen if suddenly we got lucky and people started visiting our application 5 times more often generating 5 times more read requests caused by viewing posts and user profiles. What would be the first place that will most likely become a bottleneck?

One very natural answer is our database. It could become overwhelmed with all the read requests coming to it. One typical way to handle more read requests would be to use replication. This way we could increase the number of database instances that hold a copy of the data and allow the application to read this data. Of course, this will help if the write requests are not increased dramatically. An alternative approach could be to shard our database and spread the data across different machines. This will help us if we need to write more data than before and the database cannot handle it. Such an approach does not come for free and it involves other complications that we would need to take care of. Consider getting familiar with these popular techniques for scaling a relational database.

If we manage to stabilize our database, another point where we could expect problems is the web application itself. If we've been running it on a limited set of machines, which cannot handle all the load anymore this could lead to slow response times. One good thing about our high-level design is that it allows us to just add more machines running the application. If the database is ready to handle more incoming requests from additional application instances we can scale horizontally like that. We will just add more machines running our application and instruct our load balancer to send requests to these machines, too. Of course, this sounds simpler that it is in practice but that's the general idea.

One of the reasons why using the services of a company like Amazon or Heroku could be beneficial is that they make it easy to add new machines to your environment. You just need to have the money to pay for them. Having mentioned this, it is also useful to become familiar with some of these products and services that are in the market nowadays. For example, Amazon has a very big stack of services that can work together to provide a reliable and scalable environment for an application like ours. It is good to have an idea about what's out there if you need to deploy a web-facing application.

Finally, if all else is scaled and capable of handling the increased loads we may need to improve our load balancing solution. We mentioned in the high level design that it is a very good idea to use a load balancer, which would direct requests to different instances of our application. This way our load balancer could become a single point of failure and a bottleneck if the number of requests is really high. In such cases we could start thinking about doing additional load balancing using DNS and directing requests for our domain to different machines, which are acting as load balancers themselves.

## Scaling the database

We just touched on that aspect but let's talk a bit more about it. Let's say our application needs to be able to store even more data and the read/write requests per second are increased significantly. If we are using a relational database with the proper indexes running on a single machine it could very quickly become unable to handle the loads that our application experiences. One approach mentioned above is to add an in-memory cache solution in front of the database with the goal to not send repeated read requests to the database itself. We could use a key-value store like memcached to handle that.

This will also be really useful for handling situations in which a tweet becomes viral and people start accessing it or the same thing happens to a given user's profile.

But this could be insufficient if our data grows too quickly. In such cases we may have to start thinking about partitioning this data and storing it on separate servers to increase availability and spread the load. Sharding our data could be a good and necessary approach in such cases. It must be planned and used with care because it will cause additional complications. But sometimes it is just required. It is highly recommended that you read more about this topic and all the implication it brings along. This will help you justify your decision to shard and to have a discussion about the possible downsides of doing it.

Below are links to resources from Instagram and Flickr with some insights about how they managed to shard their data living in Postgres and MySql databases:

- Sharding and IDs at Instagram
- Sharding Postgres at Instagram (video & slides)
- Generating unique primary keys at Flickr when sharding

## Unexpected traffic

Let's look at one more possible issue. We already touched on this but it doesn't hurt to go through it one more time. In the beginning the interviewer warned us that there will be outliers in the data. This means that some users will have many more followers than the average. Also, some tweets will attract a lot of attention during a short period of time. In most cases such outliers will generate peaks in the number of requests that our application receives.

As we mentioned earlier this could increase the load on our database. In such a situation using a caching solution could help a lot. The idea is that it will be able to answer the popular and repeating requests coming from the application and these requests will never touch the database, which will be busy replying to other queries.

We could also experience unusual peaks in the requests hitting our application servers. If they are not enough to respond quickly enough to all requests this could cause timeout to occur for

some of the users. In such situations solutions that offer auto-scaling of the available computing nodes could save the day. Some companies offering such services are Amazon and Heroku and they were already mentioned in this example. You can take the time to investigate what is out there on the market, so that you can have a discussion about possible ways to handle peaks in traffic.

## Summary

We started from a seemingly simple problem statement and went through a lot of things. And still we haven't covered all the possible angles that one could use to look at the given task. But to do this we would need weeks or months rather than 30-40 minutes during an interview.

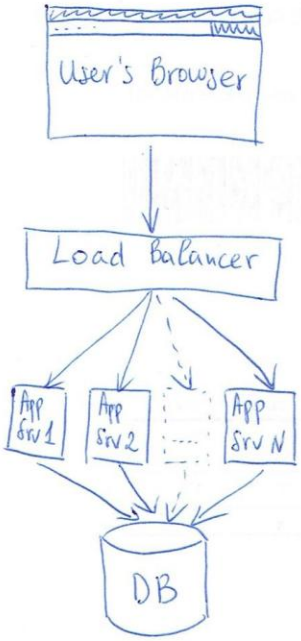
The important goal was to illustrate a few key points that we keep talking about at HiredInTech:

- Given a system design problem we need to make sure we have enough context and exact requirements
- In order to stay focused on the problem we need to describe a high-level design, which covers all aspects of the system that we need to build. We don't dig deep into any particular aspect at this point because we don't have the time for that, unless the interviewer explicitly asks us to
- If asked we may need to analyze the potential bottlenecks in our design and be ready to scale it efficiently, so that it is capable of handling more of anything
- Last but not least, practicing solving such problems in an interview-like environment is the key to becoming good at this. Remember that the interview is a special kind of format, which in most cases gives you a very limited time slot and you need to make the best out of it.

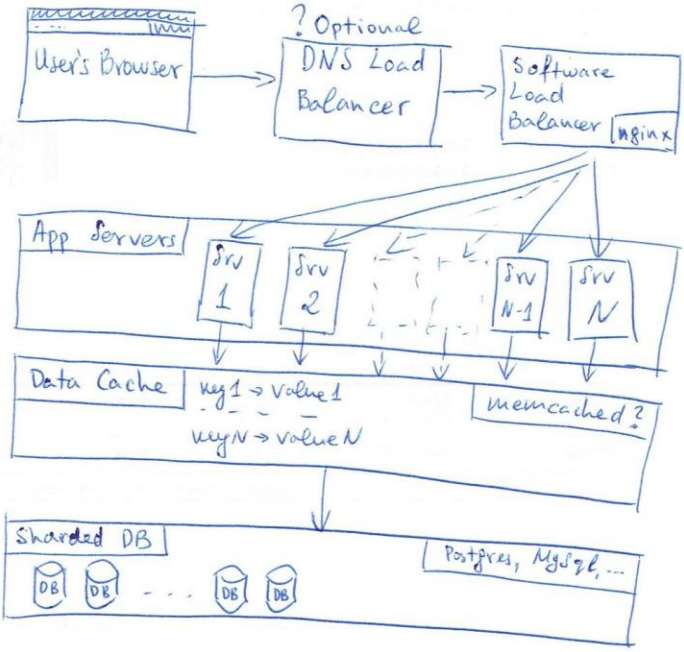
## Appendix

Below you can find a scanned copy of how one interview candidate's drawing could look like for this particular problem. There is no one recommended way to do it - this is just one possible illustration. We think it may be helpful to have a simple visual representation of how some of the discussed above could be drawn within a few minutes on a plain sheet of paper. Feel free and even encouraged to come up with your own diagrams for this problem.

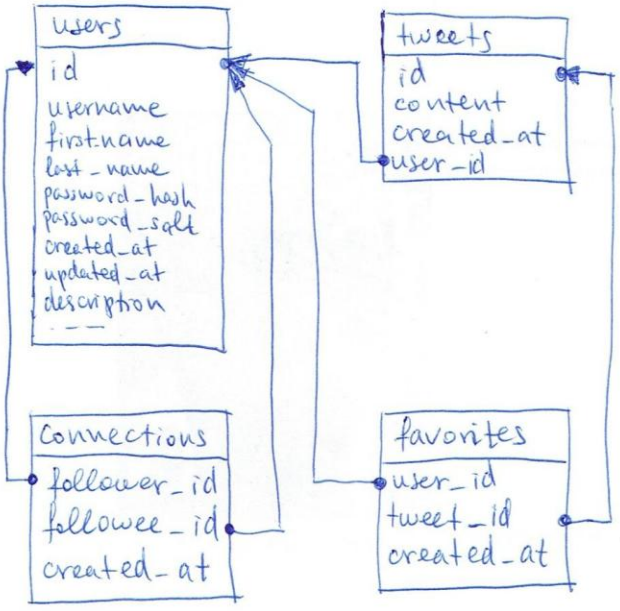
## High-level Design




## Design with optimizations allowing it to scale




## Database Schema




Indexes:

- users: <username>
- tweets: <user_id, created_at>
- connections:
  - <follower_id>
  - <followee_id>
- favorites:
  - <user_id>
  - <tweet_id>