说话人 1 00:01
Before we get into some real system design problems, let's first understand how to actually approach system design interview problems. And generally, what you can expect in a real interview, setting system design interviews are about taking real world problems and breaking them down and designing the core functionality of these systems. Since we're dealing with technical problems related to computers and the internet, we can expect that every design is gonna have some commonality, things like servers. We're gonna need to have some application running on some server. We can expect that if we need to scale these up, we're gonna have multiple servers. Now how do we route traffic to multiple servers? We have our friend, the load balancer. What if we need to store some data? We could have some type of database. And all these servers will be reading data from that database and writing data to that database as well.

Now, this sort of very, very high level design is a good start, but it's pretty hand wavy. We haven't talked exactly what we're gonna do. I this sort of design will apply to every single design problem. And your interviewer wants to know more than what a server and database and a load balancer are. Right? This is a good start, but it's not quite enough. We do generally need to dig down a bit more into the details. What is the schema of the data that we using sequence? Or no sequel? How are we handling the scale? Do we even need to handle any scale with the database? Maybe just a regular sequel, single instance, db is enough, it depends on the problem. Of course it's pretty common to be given a really really open ended problem for system design, something like design twitter or youtube or whatsapp.

It's impossible for us to cover this in typically a 45 minute interview. If twitter were really that simple, anyone could build it in 45 minutes, or at least as design it and then implement it relatively quickly. It's not that simple. It takes hundreds and thousands of engineers to actually do this. Typically, we are task with designing a small piece of twitter, something like the twitter feed or maybe twitter messaging, or maybe just the like functionality. When you click that heart like button on a tweet, maybe we're just designing this single feature because I bet it took dozens or hundreds of design document pages to design even just a simple feature like that.

So our job in the system design interview is to break it down. We need to dig into the

problem statement like design twitter. We need to clarify with the interviewer, what functionality are we focusing on? Are we just designing a particular feature or maybe the high level design just for the general site itself?

But in that case, we probably would not be digging too far into the details. But what i'm getting at is that it's all about scoping out the problem with your interviewer. This is not something you can do yourself, because your interviewer is ultimately the one guy guiding. They're sort of like your client and you're the one designing something that meets their requirements. That's a good way to think about it. And when it comes to scope ing and really understanding the requirements of the problem, we can typically categorize the requirements in two buckets. One is functional requirements. These are the actual features that we're implementing. Like, how do we want the design to actually function? How do we want the application to function? What do we want to happen? When we like a tweet? Are we allowed to like a tweet multiple times? Probably not a single user should not be able to like a tweet multiple times. Every time somebody likes a tweet, we should store their user id in a database so that we kind of remember that, right? This is a very high level scenario, but this gives you the general idea.

Also we want to know which features we're talking about the like feature here. Maybe we don't even care about the like feature. Maybe we just care about the news feed feature where we see a bunch of tweet and we can scroll through them. And every single user is going to have their own unique use feed that we need to generate. Maybe that's the functionality that we're trying to build, but it depends. You can see that this gets very, very open ended.

And that's why we need to clarify this with our interviewer. Because first of all, they already have something in mind. They probably have a small piece that they want to focus on as well. But at the same time, they're looking to see if you clarify, in general, because that's a very, very important thing on the real job. Imagine if you designed a very, very large feature or application, and it turned out to be be the wrong. One. It didn't meet the actual requirements that we wanted to implement. It was the wrong feature that we did. That's a very expensive mistake. That's something we try to avoid when we are actually designing systems. And one thing your interviewer is looking for is to see how careful you are, or if you just make assumptions without actually clarifying what we're trying to do, that's a mistake to

avoid. You don't want to just jump straight into the design and start solving the problem. It turns out you solve the wrong problem. Non functional requirements are pretty much everything that is not about the functionality of the application, namely, things around scale ability.

Any youtube tutorial can teach you how to build a twitter clone. But first of all, it won't have all of the features and functionality that real twitter does. And second of all, it definitely will not be able to handle the scale of real twitter. Maybe a tutorial will teach you how to store twitter posts in a regular sql database, and you just have a single instance database. This definitely is not going to scale to hundreds of millions of users, which is the traffic that twitter actually handles.

So we need to understand what type of scale are we handling. Possibly. We're gonna have 100 million daily active users. Maybe that's what your interviewer tells you. And so you need to design around this requirement in mind. And typically these non functional requirements are the difficult part of system design interviews, handling a large amount of scale. Is one problem. When it comes to scalability. We could have other things that fall into it, things like throughput, which is sort of like 1 hundred million daily active users that kind of falls into throughput. We may have hundreds of thousand thousands of requests hitting the database per second. Maybe we also wanna know what the storage capacity is, because if we have hundreds of millions of users, first of all, we're gonna have a lot of record requests going to our database that's throughput, but we're also gonna need to store a massive amount of data.

A single instance database theoretically can work, but even just a single query on that, much data is gonna go very, very slow. It could take seconds. It could take 10 seconds. It's gonna be really, really slow. So first, we need to understand how do we even store that data itself. And another set of non functional requirements comes from the perfect performance. When we talk about performance, latency is a big one. We want the performance of our application to typically be faster rather than slower. We want queries to our database to be as fast as possible. We would have to partition this database for the read, write latency to be lower. And then that will be directly impacting users as well, because they'll be the ones having to wait for these requests and queries to actually complete.

So latency is important, but maybe we only care about a subset of latency. Maybe we want the reads to be faster. We want reads to be less than 1 second. So every read should be less than a second, but maybe we have more tolerance for slower rights. Maybe rights can be slower. Maybe they can be 2 seconds or 3 seconds or something like that. Also, when it comes to performance, availability is pretty important for twitter. So imagine if somebody opens up twitter and they try to get their news feed, but they get an error message. Their internet is working completely fine, but maybe something went wrong with the day database, or maybe something went wrong with one of our servers crashed.

And then the user did not get the correct response. This could sort of fall into scalability as well if it's a scaling related issue. But in general, we think of availability as the number of requests is the denominator. And the numerator is the number of non error responses that we get the total number of non error. Error responsive divided by the number of total requests gives us generally the availability. And we want this to be as high as possible. Maybe we want 99 % or maybe 99 . 9 %. This is the percentage of requests that will not return an error. We want it to be as high as possible, but it's impossible to have 100 % availability. So you'll have to meet your interviewer somewhere in the middle. What's the availability that we're looking for? Because maybe we can actually design a system that will have a hundred nines of availability like nine. But but as we learned earlier, five nines of availability is something like 20 or 30 minutes of downtime per year.

So getting that even lower isn't a huge issue. Like if twitter goes down for 20 minutes per year, that's not the end of the world. I think we can even add another nine and get down to somewhere like 5 minutes. That's really not the end of the world. Maybe we can spend a lot more resources and get this down even lower. Maybe we can cut this to 1 minute or 10 seconds, but the improvement from that might not be worth the cost. So these are tradeoffs that we're gonna have to make. We definitely don't want to over engineer our solution. We can't have a perfect design. Our goal is to meet as many requirements as possible. And when we can't meet every requirement, we have to make intelligent tradeoffs, we have to sacrifice something that's not super important to gain the core functionality that we're actually looking for.

Now, when it comes to actually quantifying a lot of these metrics and requirements that we're trying to meet in an interview setting, you don't have time to actually get the exact value that you're looking for, like the exact throughput and to calculate a lot of these metrics.

So what we do to get around that and to have a shortcut is to do something called back of the envelope. Calculations are called that, because you do them on hand, like on the back of an envelope, or maybe on a napkin or something, but you don't need a calculator to do these. You can do them in your head at at a high level. Let me give you a quick example.

And then we'll talk about some numbers that you should generally be familiar with. Let's say we know that twitter has 100 million daily active users, and let's say among these 100 million people, each of them reads about, let's say, 100 tweet per day. But maybe for writing a tweet, on average, only 1/10 of these people make a tweet per day. So for each person, we have . 1 tweet. So how many tweet total do we expect to be written? Probably ten million writes per day, so 10 million write tweet. Now if each person is reading 100 million tweet, we'll have about 10 billion reads. I'm gonna change this zero to ab sorry, if it's not super readable. But we have 10 billion reads per day and 10 million writes.

So a quick calculation for throughput, for reads and writes, how many reads do we expect per second? While we take 10 billion and divide it by the number of seconds in a day? How do we get the number of seconds in a day? You can probably memorize it, or you can kind of do the quick calculation. I'll go through that really quickly. We know there's 60 seconds in a minute. We know that there's 60 minutes in an hour. We know that there's 24 hours in a day. So you can actually multiply this out. If you're decent enough at math. This is 3,600, 60 × 60, that because 6 × 6 is 36. We can add these two zeros together and then multiplying that by 24. While multiplying this by ten would be 36,000. Multiplying it by 20 would be 72,000, and then 3,600 × 4 is gonna be something like 14,000.

So this is gonna end up being something like 86,000. But I went through that exercise just to kind of show you that you can roughly approximate this to be 100,000. And that would be

perfectly good enough. I if you're off by a factor of two, like if you had this doubled, and this was 160,000, or if you were off by a factor of two in the other direction, where you approximated this to be 40 or 50,000, you would be fine. It's not a big deal to be off by that much.

So in this case, our denominator, the number of seconds in a day, is gonna be 100 k 100,000. So at this point, you can probably figure out how to do this in your head, or at the very least write these two numbers out, and then just kind of cross out the zeros. 100,000 has five zeros, and 10 billion has ten zeros. So we would end up with a number with five zeros, which is 100,000. So we expect 100,000 reads per second, which is definitely a large number. Now, if we were actually doing this with 10 million, we would basically just remove three of the zeros, because we know 10 million has three less zeros than 10 billion. So in terms of rights, we need to be able to handle a hundred rights per second. Now, when it comes to these high level designs and these applications that we're trying to design like twitter, for example, there's a lot of auxiliary services.

Our application may need to handle things like authentication and things like Payments. And there's a lot of kind of core functionality that a lot of applications have, unless the whole problem that you're trying to solve is a handle authentication or payments. You can typically ignore these. Like if we were designing the Twitter feed, we wouldn't have to talk about authentication. We would have to talk about the concept of users and user I DS to uniquely identify people, but not with how they're actually logging in and logging out or how they would maybe pay for services and things like that. But if you're unsure, you can always clarify this with your interviewer.

Now really quickly, I wanted to go over some really basic numbers that you probably already know things like 1 second is just a second. A second can get into a smaller unit like a millisecond. A millisecond is just a second divided by 1,000. And then we get to another smaller unit microseconds. This is the unit. It's actually a Greek letter, looks like au but a microsecond is 11 million of a second, 1 second divided by 1 million. And then we have nanoseconds. One nsa nanosecond is 1 second divided by 1 billion. So that's typically as small as we get to. And then we have bigger numbers. 1 second can get to 1 minute. We know that just has 60 seconds, and we can get to 1 hour, which has 60 minutes. And then

we can get to a day which has 24 hours. When it comes to storage. We're talking about bytes, and then we can get up to kilobytes, which kilo means a thousand a thousand bytes. We get to a megabyte, a million bytes. We get two gigabytes, a billion bytes. This is what we typically use for storage, at least persistent storage.

And then we get to a terabyte, which just adds to three more zeros. Instead of billion, we have a trillion, and then we get to a petabyte. Usually, this is as big as you need to go. This is 15, zero, ten to the power of 5th teen, a patabyte of data, quite a large amount of data. You typically won't go past this unit because you can have 1,000 patabytes or even 10,000 or 100,000. You don't really need to know the unit above that. So that's the good part. You don't need to memorize any units beyond this.

Now, lastly, I wanted to show you this site that I did not create. This is actually based off of some numbers that Jeff Dean talked about. Jeff Dean works at Google. He's actually responsible for a lot of solved problems in the distributed system space. And these are some latency numbers he talked about a decade ago, but let's take this roller and move it all the way to the right, because this is updated as of twenty, twenty, 20.

So these are some common latency numbers. You don't have to memorize them by heart, but you should generally have a high level idea of how these work. Like. We know l one cache is pretty quick. It's a nanosecond, but we know that main memory is slower than CPU caches and things like that.

So referencing memory is about 100 nanoseconds. Again, you don't need to memorize that exact number, but you should know that memory is a lot slower than cash and we have disk. So SSD reads are a lot slower than memory looks like about 100 times slower.

And we know that sending data over a network is even slower than reading from disk. You

can see 500 Micro seconds compared to 16 Micro seconds for an SSD and that's actually in the same data center, sending data over the internet all around the world is even slower. Here you can see it's 150 millisecond for a round trip from California to the Netherlands. And you can read this and get a decent idea. This is not created by me. It's created by Colin Scott. You can see the URL above feel Free to check it out, but again, don't need to memorize all this. Just have a decent conceptual idea of what's faster and what's slower. And generally by how much. So we talked about a lot of things. The only thing left you need to understand before getting into the system design practice problems is have a general high level overview of distributed systems and system design. I covered that in my system design for beginners course. So if you haven't checked that out, I highly recommend it. Or maybe if you already are familiar with distributed systems, you can skip that. But anyways, let's get started now.