9. System Design and Scalability

Handling the Questions

1. **Communicate**. Stay engaged with the interviewer. Ask questions. Be open about issues of your system.
2. **Go broad first**. Don't dive straight into the algorithm part or get excessively focused on one part.
3. **Use the whiteboard**. Easier to picture what you're saying
4. **Acknowledge interviewer concerns**. Don't brush them off; validate them and make changes accordingly.
5. **Be careful about assumptions**. An incorrect assumption can dramatically change the problem.
6. **State your assumptions**.
7. **Estimate when necessary**
8. **Take the wheel**. Ask questions. Be open about tradeoffs. Continue to go deeper and make improvements.

Design: Step by Step

Example: How to design a system like TinyURL.

Step 1: Scope the problem

Make a list of major features or use cases. For this example, it might be:

- Shortening a URL to a TinyURL
- Analytics for a URL.
- Retrieving the URL associated with a TinyURL.
- User accounts and link management.

Step 2: Make Reasonable Assumptions

For example, your system needs to handle a max of 1 million new URLs per day. This helps you calculate how much data your system might need to store.

Step 3: Draw the Major Components

You might need something like a frontend server that pull data from the backend's data store. You might have another set of servers that crawl the internet for some data, and another set that process analytics.

Walk through your system from end-to-end to provide a flow. A user enters a new URL. Then what?

Step 4: Identify the Key Issues

Once you have a basic design in mind, focus on the key issues. What will be the bottlenecks or major challenges in the system?

For this example, some URLs may be infrequently accessed, others can suddenly peak. You don't necessarily want to constantly hit the database.

Your interviewer might provide some guidance here. Use it.

Step 5: Redesign for the Key Issues

Adjust the design for it. Be open about any limitations in your design.

Algorithms that Scale: Step-by-Step

In some cases, you're not being asked to design an entire system, only a single feature or algorithm, but you have to do it in a scalable way. Try these following approach:

Step 1: Ask Questions

Ask questions to make sure you understand the question. There might be details left out.

Step 2: Make Believe

Set assumptions that might make your problem simpler. Then, use the assumptions to solve the problem (simply)

Step 3: Get Real.

Now, what if your assumption doesn't hold? Try to use the basic algorithm and improve it to solve this problem

Step 4: Solve problems

Finally, think about how to solve issues you identified in Step 2. Usually you can keep modifying the approach you outlined in Step 1. An iterative approach is typically useful.

Key Concepts

Horizontal vs Vertical Scaling

- Vertical Scaling: increasing the resources for a specific node. E.g. adding memory to a server to improve its ability to handle load changes
- Horizontal Scaling: Increase the number of nodes. E.g. add additional servers to decrease load on any one server.

Load Balancer

Typically some frontend parts of a scalable website will be put behind a load balancer. To do so, you have to build out a network of cloned servers that all have essentially the same code and access to the same data.

Database Denormalization and NoSQL

Joins in a relational database such as SQL can get very slow as the system grows bigger. You would generally avoid them.

Denormalization means adding redundant information into a database to speed up reads. (Having extra information in a table to prevent the need to make joins)

Alternatively, can go with a NoSQL database. A NoSQL database doesn't support joins and might structure data in a different way. It is designed to scale better.

Database Partitioning (Sharding)

Sharding means splitting the data across multiple machines while ensuring you have a way of figuring out which data is on which machine.

- Vertical Partitioning: Partitioning by feature. E.g. if you were building a social network, you might have one partition for tables relating to profiles, another one for messages, and soo on. Drawback: If one of these tables gets very large, might need to repartition that database (possibly using a different partitioning scheme).
- Key-Based (or Hash-Based) Partitioning: This uses some part of the data (e.g. ID) to partition it. Allocate N servers and put the data on `mod(key, N)`. Drawback: The number of servers you have is effectively fixed (N). Adding additional servers means reallocating all the data, which is very expensive task.
- Directory-Based Partitioning: You maintain a lookup table for where the data can be found. This makes it relatively easy to add additional servers, but it comes with 2 major drawbacks:
    1. The lookup table can be a single point of failure
    2. Constantly accessing the lookup table affects performance

Many architectures actually end up using multiple partitioning schemes.

Caching

An in-memory cache can delivery very rapid results. It's a simple key-value pairing and typically sits between your application layer and your data store.

When an application requests a piece of information, it first tries the cache. If the cache does not contain the key, it will then look up the data in the data store.

When you cache, you might cache a query and its results directly. Alternatively, you can cache the specific object.

Asynchronous Processing and Queues

Slow operations should ideally be done asynchronously. In some cases, we can do this in advance. E.g. we might have a queue of jobs to be done that update some part of the website. If we were running a forum, one of these jobs might be to re-render a page that lists the most popular posts. That list might end up being slightly out of date, but that's better than a user stuck waiting on the website to load simply because someone added a new post and invalidated the cached version of this page.

Networking Metrics

- Bandwidth: Maximum amount of data that can be transferred in a unit of time.
- Throughput: Actual amount of data that is transferred in a unit of time.
- Latency: How long it takes data to go from one end to the other

Unlike throughput where at least you can speed things up through data compression, there is often little you can do about latency.

MapReduce

A MapReduce program is typically used to process large amounts of data. A MapReduce program requires you to write a `Map` step and a `Reduce` step. The rest is handled by the system.

- Map takes in some data and emits a <key, value> pair
- Reduce takes a key and a set of associated values and "reduces" them in some way, emitting a new <key, value> pair. This result might be fed back into the Reduce program for more reducing.

MapReduce allows us to do a lot of processing in parallel, which makes processing huge amounts of data more scalable.

Considerations

- Failures: Plan for many or all of these failures.
- Availability and Reliability: Availability is a % of time the system is operational. Reliability is a function of the probability that the system is operational for a certain unit of time.
- Read-heavy vs. Write-heavy. If it's write-heavy, consider queuing up the writes. If it's read-heavy, you might want to cache.
- Security issues.