

Part 2: Introduction

If you read Part 1, then you should have a mental model of what a system design interview looks like. You also learned common pitfalls to avoid and discovered high-level strategies to prepare you for your interview. Like Part 1, Part 2 is still introductory and grounded in theory, providing you with a 30k-foot overview of the whole case. Part 2 is more targeted, however, and we'll begin by teaching you 15 fundamental system design concepts that will help you succeed in your interview.

About these 15 fundamental concepts

The first three concepts aren't purely technical—they're based on tacit knowledge. We learned these three concepts through collectively experiencing thousands of hours of system design interviews, but you'll be able to glean these insights with only a small amount of your time. The final 12 concepts are purely technical.

One note about the 12 technical topics

We've identified two categories of technical concepts: Topics you'll want to know quite well, and topics that are worth knowing only a little bit about. For the first version of our guide, we are going to skip the latter topics, but we plan to include this information in future iterations of the guide.

Remember

There are endless things you could learn about system design, but Part 2 of this guide focuses on the 12 technical concepts that will give you the best bang for your buck in a system design interview. By the end of Part 2, you'll understand these 12 topics well enough in theory to begin putting them into practice (which we'll cover in Parts 3 and 4).

Three core concepts for system design interviews

a. There's no right way to design a system

By now you've heard (or read) that "there's no right way to design a system," and you might think it's true. But how do you know for sure?

Watch this video of two experts designing the same system side by side. By the time you're done, you'll have a practical example that proves "there's no right way to design a system". Pay attention, and you'll notice how effective it is when you guide the interview toward your strengths and when you're open about gaps in your understanding. The video is split into two parts.

This is one of the most important lessons in the entire guide.

Here is Part 1

<https://www.youtube.com/watch?v=ZiOpPkiFemE>

Here is Part 2

https://www.youtube.com/watch?v=PU_sgwZvm6s

b. General rules of thumb

We have "rules of thumb" scattered throughout this guide. In those cases, they apply directly to the material. But in this instance, these rules of thumb don't fit anywhere, because they apply to, well, everything.

Interviewer behavior

As an interviewer, it's hard to tell the difference between a bad candidate and a good candidate who is stuck.

If the interviewer interrupts you, it's probably because you're going off track.

If your interviewer interrupts you to suggest that you explore another avenue, then most likely you're designing the system in contradiction to what the interviewer

expects. In this case, let the interviewer explain what they expect, and then you should ask clarifying questions to ensure you understand the new direction before moving on.

It's fine if the interviewer asks you questions, but it's a bad sign if the interviewer starts telling you how to do things. This is a negative signal because the interviewer feels that you need help to move forward, and this will lower your score.

Prior experience affects both sides

In a system design interview, you may encounter two different situations:

1. The interviewer has read your resume and wants to see you demonstrate your experience in building something you're familiar with. This should be easy because you can apply your knowledge from your current/previous position.
2. The interviewer has read your resume and decides to purposely challenge you by asking you to design something you have not worked on. In this case, don't worry - just remember that "there is no right way to design a system." Use your best judgment and industry knowledge to come up with something reasonable. Also, be honest about gaps in your knowledge and don't be afraid to ask questions. Demonstrate that you are curious and willing to learn.

When the interviewer decides to challenge you with something new, it may be a topic that is based on their own particular expertise or skill set.

Tip

If you know a little about your interviewer's background, you should have a hint about what to expect, which can allow you to prepare a little ahead of time.

Time management

It's more important to cover everything broadly than it is to explain every small thing in detail.

By the end of the interview, the interviewer is inherently asking themselves “Could this person get an MVP off the ground?” If the answer is “no”, then you’ve drastically reduced your chances of passing the interview.

Approaching the problem

Whatever decision you make, explain **why**. In a system design interview, **why** is more important than **what**. For anything you say, be prepared to explain **why**.

Your interviewer cares less about whether your design is good in itself, and more about whether you are able to talk about the trade-offs (positives and negatives) of your decisions.

Keep it simple. The first rule of distributed systems is that you should avoid them if you don’t need them! Always consider maintenance costs. People don’t build distributed systems for fun. If all of Google could run on just one machine, you can bet they would do it.

In other words, if there is a simple way to do things and a complex way to do things, aim for the simple path. Not because the simple way is more likely to be correct, but because you have to make more assumptions for more complicated explanations to be true.

Accept that there are some things that you will not know, and be ready to admit this to your interviewer. In the third core concept (below), we will teach you exactly how to say this without losing points in the interview.

A likely pitfall

Once you reach a certain level of competency, you're more likely to need to hone your communication skills than improve the way you design the system. This guide will help mid-level / senior candidates surpass that level of competency, which means that after finishing this guide, you’ll want to refine your communication skills to continue improving. Mock interviews with different types of interviewers are the best solution we’ve found to refining your communication skills, or working with a

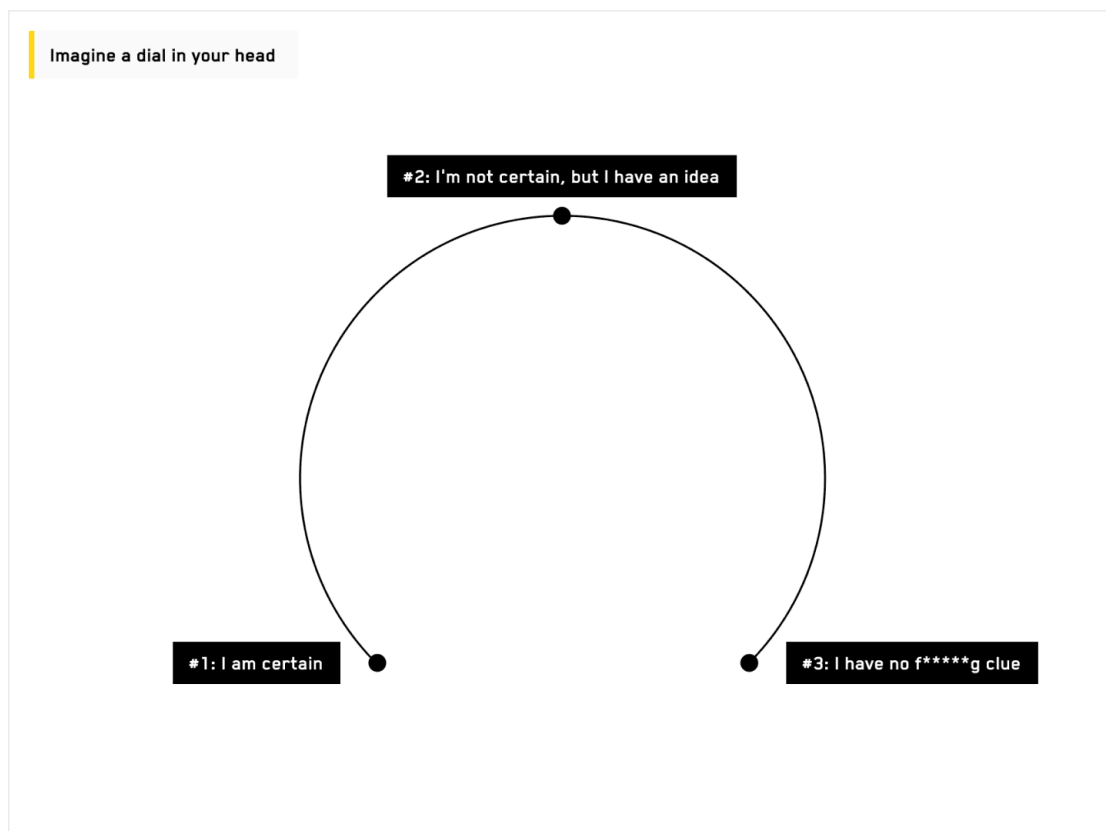
dedicated coach who can get to know you (and your areas of expertise and improvement) very well.

c. Exactly what words to say in specific scenarios

What to say when you don't know what to do

Weak interview candidates are scared to ever utter the phrase “I don’t know.” Stronger interview candidates say “I don’t know” more often and then strengthen this with a buffer—the words you put around your uncertainty. A naked “I don’t know” is a dead end, so your interviewer has nowhere to go. Adding a buffer gives your interviewer several paths they can take to move the conversation forward.

Before we tell you exactly what words to say, let’s unpack a concept. There are different levels of “not knowing.” Sometimes you have no clue. Sometimes you have a clue. And other times, you’re certain.



Here is an example of all three levels on the dial

Let's say the interviewer asks three different candidates about load balancing. Imagine that each candidate has a different number on the "dial" in their head when it comes to load balancers. We'll show you exactly how each candidate should respond.

Candidate #1

Knows a shit ton about load balancers, load balancing algorithms, and everything in between. Has worked with them first hand for a while and knows them very well.

Tip

Most mid-level to senior engineers aren't this kind of candidate

Rock on with your bad self, Candidate #1 -> Go ahead and strut your stuff here. You've got the knowledge, so all you need to do is share it.

Candidate #2

Knows nothing about what load balancing algorithms are but knows what a load balancer is and what it's supposed to do.

Tip

The best thing to say here is:

"I don't know, I'm definitely going to look that up right after this interview, but if I had to give my best guess I'd say... [x] and here is why [explanation/thought process]"

You can also say things such as:

- "I'm going to throw some things out there, but don't hold me to them..."
- "I don't have experience with that yet but I've been reading [Thing you read] I would approach it like [Idea]."

"I don't have hands-on experience with this, but I have read about it and here's what I know."

- "This reminds me of this thing I built one time..."

- (This is best said with some charisma) "This is super interesting, tell me more about that."

Tip

If you pay attention, the interviewer will probably give you indications if you're going down a fruitful path.

Candidate #3

Has been writing shrink-wrap software their whole life and has never heard of a load balancer. The candidate should tell the interviewer exactly that. And then the interviewer will either:

- Tell the candidate what a load balancer is and ask them how they'd approach that problem.
- Or they'll move on to something else.

How to push back against your interviewer in a helpful way

Example of a candidate pushing back:

Interviewer: "I wouldn't use a cache here in my opinion"

Interviewee: "Sure we can take out the cache, though the reason I think a cache might be useful here is [insert technical reasoning here]. Do you think there might be a better way we can approach this?"

Choose your words carefully

1. Acknowledge and affirm your interviewer. Start with “sure,” “OK,” or “yes,” so they are open to what you’re saying. This results in them being less likely to push back against your pushback.

Tip

The best language to cause your counterpart to let their guard down is “I’m sorry”. If you want to push back, and you don’t want the interviewer to challenge it, start your push back with “Ok. I’m sorry, but if we take out the cache won’t that result in [insert technical reasoning here]...”

2. Remember that you are colleagues on the same team. The best way to demonstrate this is to use collaborative language. This is the easiest way to build empathy and score micro points to get them on your side.

Tip

The best language to demonstrate collaboration is “We” or “Let’s.” For example: “We could take out the cache. However, if we did that one drawback would be [insert technical reasoning here]...”

Handwave stuff for the sake of time

No one can design a real-world system in 40-60 minutes. You can only design partial non-usable systems in that amount of time. As a result, you won’t be able to cover everything in depth.

Remember

It's more important to cover everything broadly than it is to explain every small thing in detail.

Anecdote

It is common for interviewers to ask a candidate to “design Gmail.” There are so many different dimensions to the product we know as “Gmail,” so no candidate can actually design Gmail in the amount of time they have in an interview. Whenever people tell you to ‘design gmail’ that is to scare you.”

Tip

Handwaving stuff is a smart time-management choice. It’s also a tactic to avoid getting derailed.

When we say “handwave stuff,” this means that you can say, “I’m going to skip going into [detailed thing] for now, but if we want, we can come back to it later.” If you dive deeply into the details of every single thing, you’ll fall down too many rabbit holes to be productive. But if you notice and address this so your interviewer understands, by saying something like, “Hey, here is a rabbit hole we could go down, but let’s skip it for now,” you’re killing a few birds with one stone. You’re demonstrating your knowledge by calling it out, and simultaneously you’re keeping the interview on track, because in an open-ended situation like this, it’s just as important to know where you’re not going as it is to know where you are going.

Here’s an example of a smart way to handwave stuff (taken from one of our mock interviews)

<https://www.loom.com/share/01a74e894a574652a63f210c40d14c67?t=1>

Be proactive when you know there’s a particularly tricky part coming up, and you know you want to take it on

If you feel some part of the question could/would become problematic, fight the instinct to avoid it and instead pinpoint it right away.

You can say something like this:

"The challenge we would face very soon would be with [multiple workers updating their offsets while grabbing the next task concurrently?]. Let me finish the [API specs / DB schema / etc] part and then begin attacking that challenge. I know that would be a hard part, and might kindly ask you to navigate it if I begin approaching the problem from the wrong end."

The two types of interviewers

Let's place interviewers into two types: warm and cold.

Warm interviewers enjoy collaboration. They like when you check in with them. You'll recognize this type of person because you can feel them engaging with you during the interview.

Cold interviewers would rather you get the thing done on your own. Maybe they'd rather not be in the interview, or maybe it just seems that way. You'll identify this type of person in an interview when they visibly withdraw after you start engaging with them too much. Engagement happens in two ways: how many questions you ask them (a question is a request for engagement), and how much thinking you do out loud.

Outlaw idea

Cold interviewers are more likely to let you screw yourself into getting rejected. However, it'd be hard to find an interviewer who'd admit they would let a candidate harm their own chances. Every interviewer wants to think they're an above-average interviewer who helps the candidates stay on track. If you watch the tape, however,

there are clear stylistic differences, and interviewers usually fall into one of these two groups.

Tip

A lot of candidates fail because they expect engagement from the interviewer, and they don't know how to adjust when they get a cold interviewer. When candidates don't get what they expected, they often become more cautious, which causes them to make progress more slowly and get a worse score for that lower level of productivity as a result. This is a good reason to do mocks with different people so you can get comfortable with different styles.

How to treat the two types of interviewers

There is a balance between asking questions and making statements. Cold interviewers get turned off when you constantly check in with them by asking questions. They'd prefer you to proceed without involving them so much. Questions necessitate a response. But well-worded statements only necessitate a response if certain conditions are met... which is less work for our cold colleagues!

With a cold interviewer, say things like

"Correct me if I'm wrong, but I think I can do X because of Y."

"Stop me if I'm going off track, but I think the next thing to do is X because of Y."

Anecdote

From a cold interviewer: "If you go down the wrong road, I will just let you go there."

With a cold interviewer

Do less thinking out loud. You get more of a pass for silences with cold interviewers. And it's better to not open your mouth until you're reasonably sure about what you're going to say, as opposed to having a completely open dialogue where you're not worried about getting it right the first time.

With a warm interviewer

The usual status quo advice applies. Check in with them. Ask them questions. Above all, treat them like colleagues you're working with on the job. You're in a room and you're trying to figure this out together.

With a cold interviewer, say things like

"I think I can do X because of Y. What do you think?"

"I think we can proceed with either X, Y, or Z. Personally, I think X because N. What do you think?"

Anecdote

From a warm interviewer: "It's better to ask first than to run with something you aren't sure about because it'll waste time and you'll go down the wrong path."

Warning

One caveat: The more questions you ask, the more junior (or incompetent) you can seem. However, not all questions are created equally. Juniors (or weak candidates) tend to ask dumb clarifying questions where they don't know why they're asking the question, except that they think they're "supposed to." This makes it clear they're

talking above their level, which exposes them as being more junior than they are trying to appear. The lesson: don't ask a question or do something simply because you think you're supposed to.

Example:

A seasoned interview candidate knows to check in with their interviewer. A noob interviewer candidate knows that as well. But checking in after every single decision you make is a good way to be outed as a noob. A better idea, to be perceived as a higher level candidate, is to check in with your interviewer at major milestones in the interview.

Do

Check in at major milestones (e.g., once you're done taking requirements, or after you're done with the high-level design)

Don't

Check in after every single requirement

Check in after every single time you make a decision

12 fundamental (technical) system design concepts

1. APIs
2. Databases (SQL vs NoSQL)
3. Scaling
4. CAP theorem
5. Web authentication and basic security
6. Load balancers
7. Caching
8. Message queues
9. Indexing
10. Failovers
11. Replication
12. Consistent hashing

a. APIs

What is an API?

Imagine it's the year 3000. Mars has been colonized and has vibrant urban cities. You took your vitamins, exercised regularly, and thanks to some advances in medicine, you are in the prime of your life. You take a commercial spaceship to Mars and arrive only to discover that not a single person on the planet speaks English. People come from all over the galaxy, and there isn't a common language. Instead, people rely on hand gestures to communicate, barter, and trade.

You see a futuristic hot-dog stand and your mouth waters at the thought of grabbing a midday snack, but quickly you experience culture shock because you are having a

hard time conveying what sort of bun and hot dog you want. You stand to the side and observe how others are making their purchases.

What you notice is that others first hold up interesting and weird hand signals to indicate what bun and hot dog they want. Then the person at the stand uses their fingers to indicate the number of minutes it will take to prepare, along with the cost of the hot dog. Finally, a nod is given and money is exchanged.

Congratulations, you now know how to order hot dogs in the future. You also have a good understanding about what APIs are for. In the absence of a common language, cooperation between two people, or computers, needs to have a structured and mutually agreed upon way of exchanging critical information. In this case, there were two parties: the futuristic hot-dog man and you. The formal definition of an API (Application Programming Interface) is the set of rules or contracts through which two or more software entities communicate.

API architectural styles

Back to our futuristic Mars adventure. It goes without saying that buying a hot dog and renting a fancy floating Mars condo won't be exactly the same. From intuition we know that a hot dog is a much smaller transaction, while renting an apartment requires a lot more information to be exchanged. Hot dog exchanges usually happen right on the sidewalk, while renting an apartment is best done sitting down at a table. These differences in communication patterns exist because of differing amounts and types of required information and the differing amount of time both parties need to stay in touch.

Likewise, when we design the communication mechanism of our software, we will need to make choices that best suit our use case. Here we will explore the common architectural styles, review their strengths, and explain when they are more suitable.

1. REST or Representational State Transfer (hot dog style)

This is the style that our hot dog example used. When we want to buy a hot dog we make a GET request with our hot dog type (German sausage, of course) in mind. We then got back information about how much it costs and how long it will take. Once we made a decision, we sent out a post request with our money and got back the hot dog.

Under this style, the APIs must be modeled based on the resources in the system. A single URI to access that resource is used, and various actions based on the HTTP verbs (GET/PUT/POST/PATCH/DELETE) can be performed on the resources wherever possible.

Strengths: This approach creates structured ways of getting and modifying information from your database. It is the most universally used and works for most circumstances. This method also tends to have tooling that supports generation of documentation that can make it easier for developers to understand, especially for external services accessing the API through network calls.

Weaknesses: It requires you to write the requests for each type of entity in your database, in contrast to GraphQL where no separate inquiries are required to grab all the data the caller needs. It also isn't as space efficient as RPC.

2. RPC or the Remote Procedure Call (family style)

RPC is like communication in a family or with close friends. When you are with family and you notice your favorite snacks in the fridge, you can usually skip a lot of communication and make assumptions that you can eat some without asking. Since you have close and frequent communication, you make certain processes more efficient.

RPC allows the execution of a procedure or command in a remote machine. In other words, you can write code that executes on another computer internally in the same

way you write code that runs on the current machine. In this approach, the API is more thought of as an action or command. And it is easier to add these functions to extend the functionality.

RPC - `/placeAnOrder (OrderDetails order)`

REST - `POST /order/orderNumber={} [Order body]`

Strengths: It is more space efficient than REST, and it makes development easier since the code you write that requires communication to other computers does not require much special syntax.

Weaknesses: It can be only used for internal communication. There are complications that can occur, such as timing issues, when you are communicating between machines, and RPC could make this distinction less clear, leading developers to miss corner cases that cause faults in the system.

3. GraphQL (Amazon Go Style)

GraphQL can be thought of as those Amazon Go stores where you can walk in, grab what you need, and walk out. There are cameras that track what you took, and you are automatically charged for the items you left with. Items in Amazon Go stores are placed in a way that they can be easily discovered, allowing customers to decide what they need. Likewise, in GraphQL you structure the data in graph relationships and then leave it for those using your service to define what they need.

This modeling technique enables building a perfect request to fetch all the data that is needed by the client without making multiple calls. A very important step in this is building the right schema, which encompasses all the queries you can make using the APIs and the data types that can be returned. It is more widely used in mobile applications and other systems where graph-like data fits in.

Strengths: GraphQL works particularly well for customer-facing web and mobile applications, because once you set up the system, frontend developers can craft

their own requests to get and modify information without requiring backend work to build more routes.

Weaknesses: There is initially some upfront development work required to set up this communication system, both on the frontend and backend. It is also less friendly for external users when compared with REST APIs, where documentation can be generated automatically. In addition, GraphQL is not suitable for use cases where certain data needs to be aggregated on the backend.

b. Databases (SQL vs NoSQL)

How is this topic used in system design interviews?

Since you'll be tasked with designing a performant, scalable system, it's important to decide how your system should store its data.

Remember

There is no strictly wrong answer for which database to use, as long as you can justify yourself and demonstrate an understanding of the alternative options.

Here's an analogy: Imagine that I tell you to deliver a set amount of packages across town and ask you whether you want to use a car or a motorcycle. Let's assume that motorcycles can arrive at any destination faster because they can bypass traffic jams by sneaking in between cars. Before you can answer me properly, there are a few things to consider:

How comfortable are you with using motorcycle vs. a car?

How many packages are there?

What is the average length of time it takes from the starting point and drop-off location while driving a car? What about a motorcycle?

How many packages can I carry in the car at once? How many can I carry on the motorcycle at once?

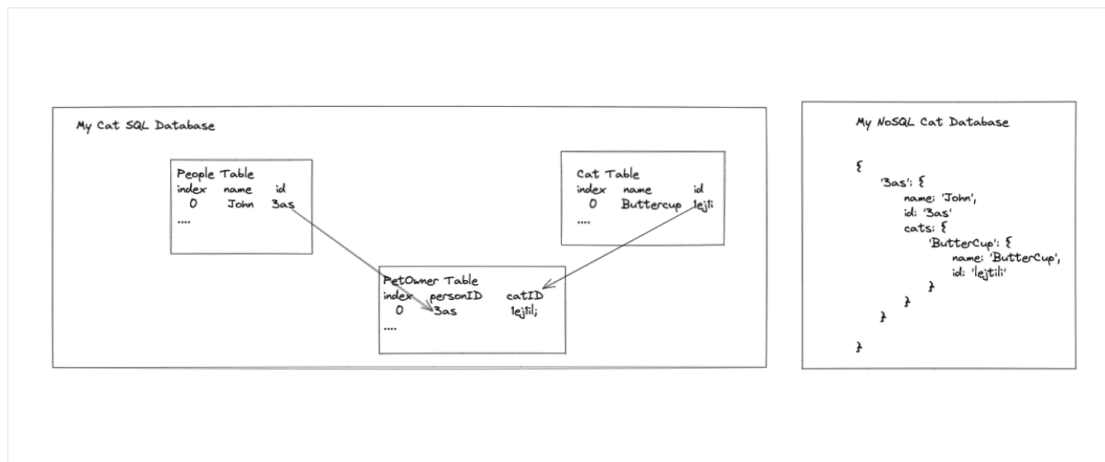


The questions we considered above are similar to the type of thinking we use when deciding which type of database to use. Both SQL and NoSQL databases have their merits depending on the requirements. This section will explore how to make a decision and justify your choice.

Here is an illustration of the same cat database in both types of databases. Take a look below, and we'll go into more details in later sections:

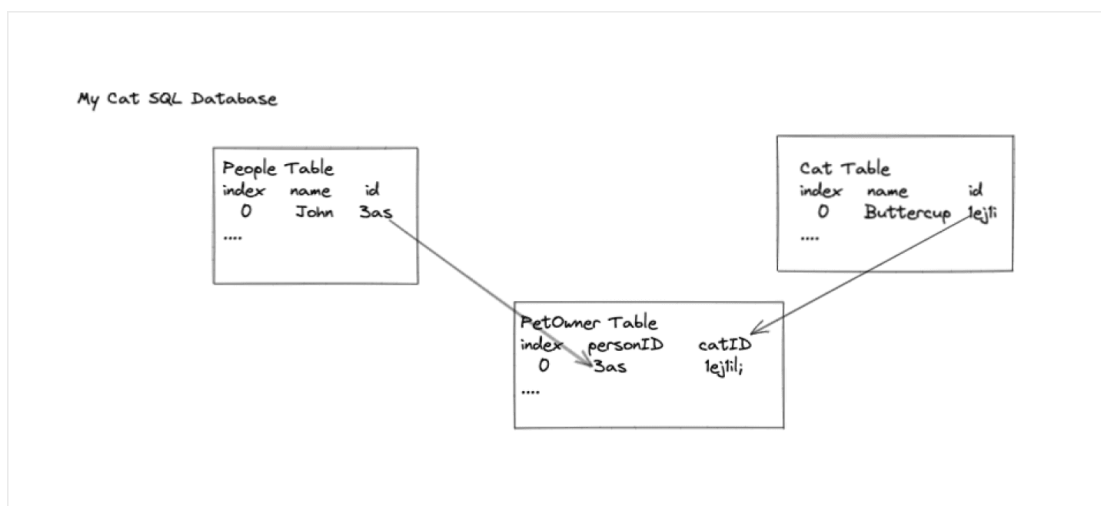
An SQL database is a relational database that is composed of tables where each row reflects a data entity and each column defines specific information about the field.

For example, in the cat database below, you can see that there are three tables: People Table, PetOwner Table, and Cat Table. Data within an SQL table is stored in these strict relationships.



Notice that the SQL database is composed of separate tables holding bits of information that connect while the NoSQL database is a nested key-value store.

SQL Database



Advantages of SQL Databases

SQL offers more powerful querying out of the box

In an SQL database, you can **craft a specific query using the language SQL** that searches for the data that you want without having to write custom code. This is an advantage over time because the compiler that transforms your SQL query into machine code can be optimized over time independent of business logic.

SQL has stronger ACID guarantees out of the box

SQL is preferred if it is more important that customers using your service always see up-to-date information, even if sometimes there is a slight delay. This is known as having strong consistency. A good use case for this is if you decide to build a service to manage payments. In such a service it is important to always show the right values and for the system to reliably handle situations of failure to avoid issues like overcharging a customer. Compare this with a service like Signal or Whatsapp, in which it is completely acceptable for a message to be out of sync for a couple of seconds.

SQL ensures ACID out of the box which makes it a good choice for cases where high consistency is needed:

- **Atomicity**: the transaction (changes) completes in its entirety without failures or the transaction is reverted. There is no case where only half the writes are changed leading to inconsistency.
- **Consistency**: By default, SQL has strong consistency. Put another way, you are guaranteed to be internally consistent. This is done by locking certain parts of the database when it is being written, forcing other requests to wait. This is in contrast with NoSQL databases that by default have eventual consistency. This means that when you query the database the data might be stale for a couple of seconds before it becomes consistent.
- **Isolation**: All read and write requests within a transaction are not impacted by other transactions. This ensures that the changes being made in one transaction are not available in other transactions.
- **Durability**: In case of failure for a given transaction the data is not lost and can be recovered.

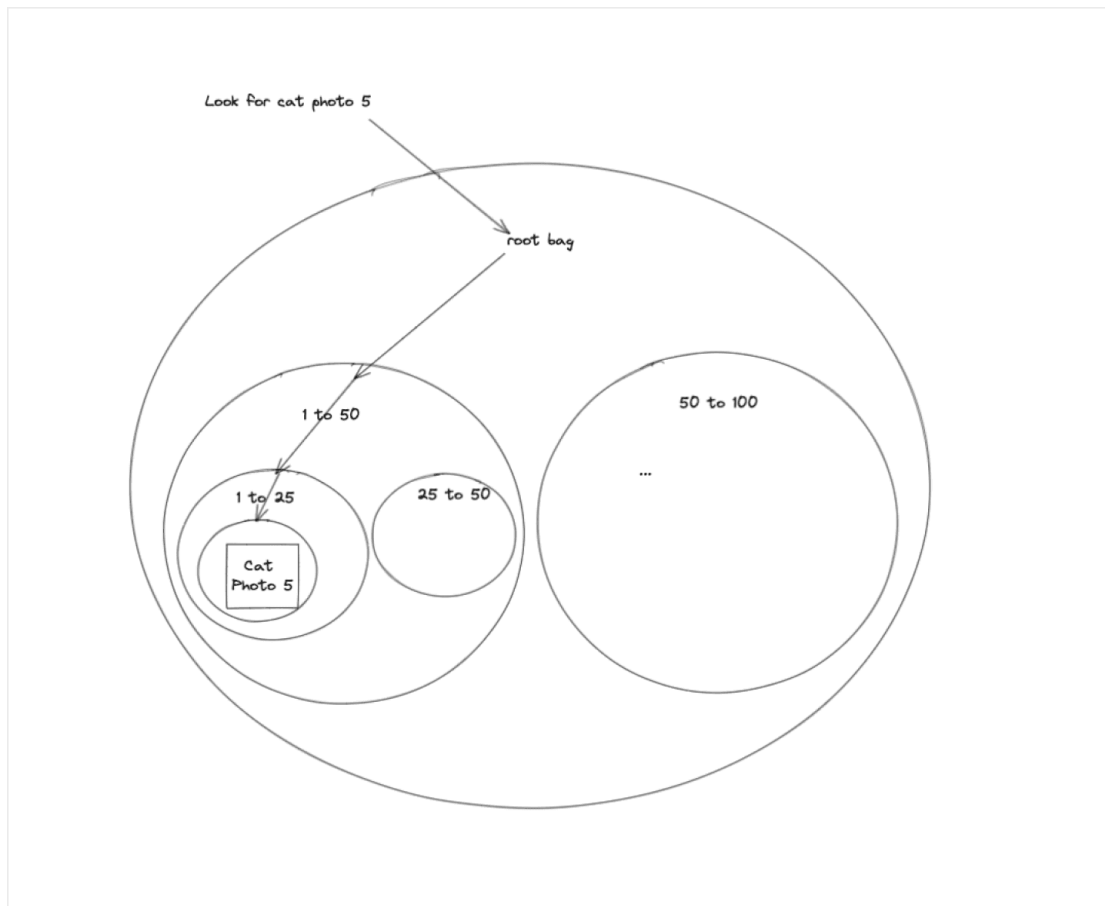
Disadvantages of SQL Databases

B-Trees, used in SQL DBs, are slower to write into (compared to what NoSQLs use under the hood.)

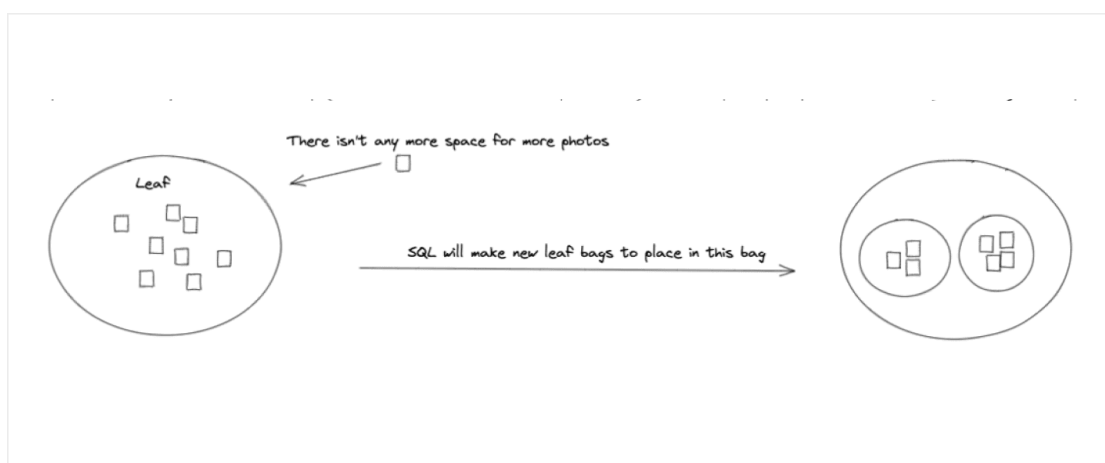
Put simply, SQL databases are slower for writing than NoSQL databases due to the way data is stored. In contrast to this, NoSQL databases typically have slower reads because of their implementation, which we will explore later. Below is a summary of how that works:

SQL Databases are stored in a B-Tree structure that divides the available space into fixed-size blocks called pages. As a mental model, imagine that you have 100 cat photos labeled 1 to 100, and you want to be able to find a given photo in the fewest steps. One approach is to have layers of bags with ranges on them so you can narrow down your search.

You can read more about B-Trees a bit later in the guide, or you [can check out a wikipedia link about them now.](#)



If you need to change something in cat photo 5, you will look in the root bag for the next layer of bags until you reach a bag that only has photos (leaf). You will then change the photo and return it. Each bag can only hold a certain amount of items; thus, when you reach that limit, then the leaf bag will split into multiple layers.



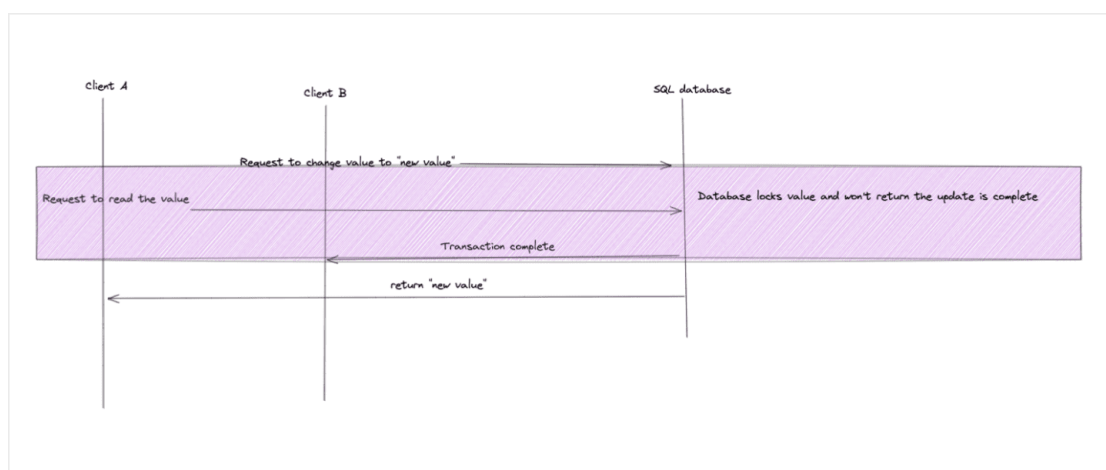
In SQL databases these bags are called pages, and each page is traditionally 4 KB in size and maps onto a specific sector of hard drive space. Once a page becomes too

large, this page is repartitioned to point to new children pages and the values get sorted into them.

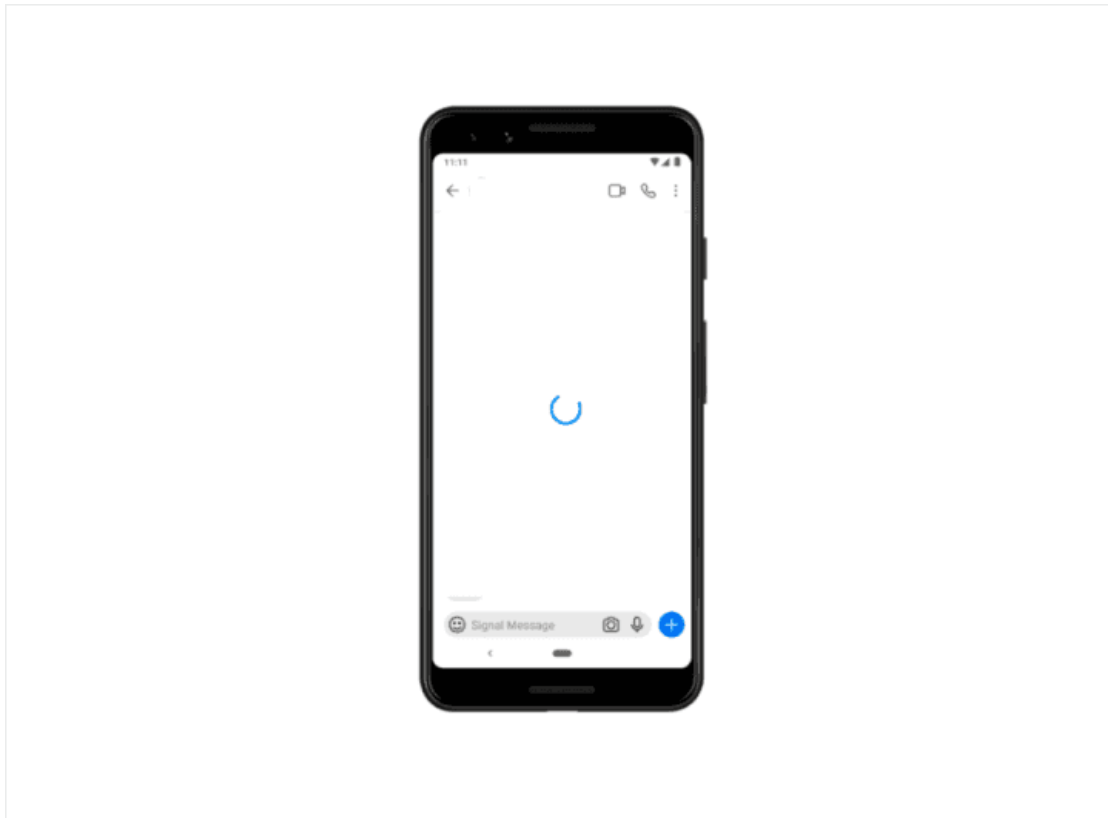
When a write operation occurs, it needs to overwrite existing values. This becomes an issue with SSDs since an SSD must erase and rewrite fairly large blocks of storage chip at a time. When new pages are created and data is moved, this causes the operation to perform poorly when compared to NoSQL databases that use a log structure that only appends existing data.

Strong consistency is expensive to reduce latency for

For customer experiences that do not require strong consistency, SQL databases will have much higher latency compared with NoSQL databases. This is because strong consistency requires that the database lock particular fields when it is being modified.



In the example above an update query locks the field so that any user requesting that data has to wait. In the example of a messaging app, this will degrade the experience for the user when it would have been perfectly acceptable to have the new message become eventually consistent.



SQL does not work well for mixed schema data

Since data in SQL has a fixed schema, each time a new value is to be introduced to a row a migration is needed to transform and migrate data across all nodes. This process is expensive and time-consuming. In contrast, NoSQL databases do not have to have a fixed schema, so different versions of data can exist without extra work.

NoSQL Database

NoSQL-style databases have existed since 1960, but the term "NoSQL" was introduced in the 21st century. Unfortunately, it is a catch-all term for many alternative technologies that aren't relational.

There are four types of NoSQL databases, each of which has advantages depending on their use case:

- **Key Value:** these are the most popular type of NoSQL database that stores data with unique keys and the system is opaque to the contents of the stored data. The architecture scales through the usage of sharding of data across nodes and by default is eventually consistent.
- **Document databases:** these databases are very similar to key value stores, except that they have the ability to perform aggregate searches across data and store in a variety of formats like JSON, XML, and YAML.
- **Columnar databases:** these databases also store information in tables but allow you to have denormalized data. The indexing is based on columns rather than rows. This type of database is most efficient when your queries involve computing the same value types across multiple values.
- **Graph databases:** these databases help store complicated node and edge relationships. They also allow for easy graph transversal and modification without writing and maintaining your own code.

Remember

The common aspect of these databases is that they do not require rigid schemas like SQL models.

In this section, we will focus on the document store database because it is the most commonly used.

Advantages of NoSQL Database

Generally speaking, NoSQL is faster for writes but slower to query.

Most NoSQL databases operate with the underlying data structure of a Log-structured merge tree (LSMT), in contrast to SQL database's use of B-Trees. This difference makes it so on a theoretical level NoSQL databases are faster for writes and slower for reads. Below we will describe a simplified mental model of why this is the case.

Imagine we are an official scorekeeper for a NASCAR race. Our job is to stand at a given position and record down the time each car passes that point. There are a lot of cars and speed is important here. The faster way to note down the time is to just record the driver's shorthand in the next blank line.

Drivers	Lap 1	Lap 2	Lap 3
T.F	2:31
G.K	2:29
T.P	2:40

Having a table like this is easier to read but is too slow since you will need to search for the driver then the lap.

G.K	2:29
T.F	2:31
T.P	2:40
...	

Have a log list like this where you just add to the bottom is much faster for recording times. Sure its a little harder to read but we can deal with that later.

As you can see, the log list on the right is much faster for writes since you don't do anything to maintain structure when you add to it. It is notably a bit harder to read if you are looking for a particular driver's time or for a particular lap, since it requires you to scan down the page for all entries for the driver and count in your head until you find the correct entry.

As a heuristic, the left side can be thought of as an SQL database that is more structured and is slower for writes but faster for reads. The right side can be thought of as a NoSQL database that is faster for writes but slower for reads.

Readers should be aware that a NoSQL database is much more complex in its implementation and is more efficient when it comes to reads than looking through a long list. We strongly encourage readers to read more into LSMT data structure and how compaction and SS tables improve efficiency. (Links for further reading: [LSMT data structure](#) + [SStables](#))

Managed NoSQL services like DynamoDB or Azure MongoDB include sharding and scaling out of the box

Managed services like DynamoDB and MongoDB come with sharding and scaling out of the box. Although this is also possible with SQL databases, it requires a bit more planning for the development team, whereas scaling NoSQL databases often just involves increasing paid capacity.

Disadvantages of NoSQL Database

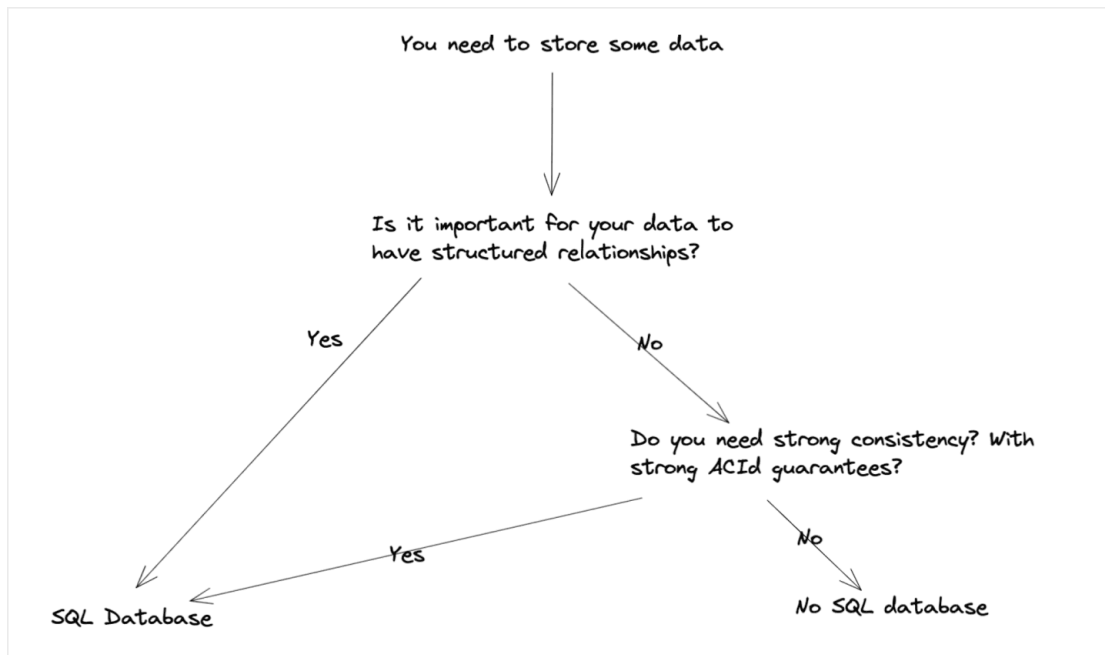
NoSQL databases are more limited in the types of efficient queries that can be done. In addition, they are less suitable for circumstances where strong consistency is required, a fact we covered in the SQL section above.

Conclusion

As managed cloud services advance, the lines between databases become more blurred. For example, Amazon RDS SQL Databases provides some of the box options for sharding and scaling, while AWS Dynamo DB provides an option to enable strong consistency, though it will introduce some latency during read and writes.

The type of database you should use depends on what you are storing, how often you are writing, and what sorts of retrievals you need to make. Though there are certainly creative solutions to accomplish the same end, choosing the right database to simplify the design and improve its robustness is the overall goal.

Here is a simple map to guide your decision making.



c. Scaling

What is scaling?

For system design interviews all problems you are asked to solve involve scaling to many users in unique circumstances. Thus, understanding the core principles of scaling and the various approaches will give you a strong foundation throughout the interview.

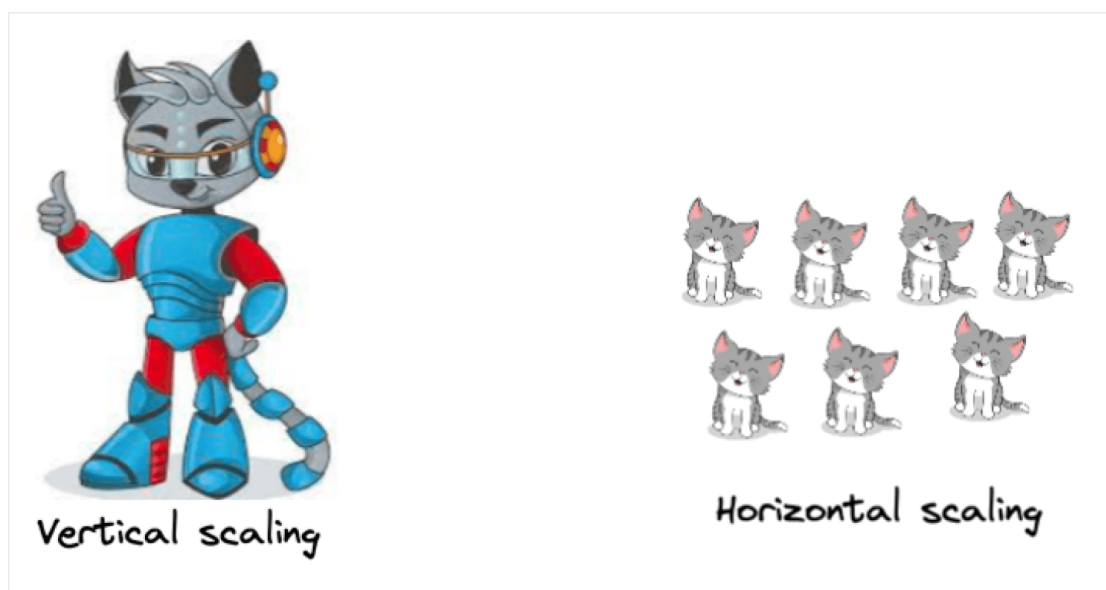
Imagine that you know for a fact that you have the cutest cat in your neighborhood. Many come by to visit your cat and admire how fluffy and cute he looks. You want more people to see your cat, but you are limited in the number of people who can come over to your place at a given time (condo capacity limit). You also realize that your cat will become unhappy if he has to entertain too many people (cat attention limit).

Scaling is considering how you can solve these problems so that more people can see your cat. The idea is to find a way to scale your cat's reach as he gains more fans over time. The important thing to understand is that there are different factors to scale (condo capacity limit and cat attention limit) as well as different approaches (vertical

and horizontal scaling). We will go into more detail about each type in the sections below.

The table below shows how you might solve scaling your cat, depending on whether your focus is on horizontal or vertical scaling.

	Vertical scaling	Horizontal scaling
Cat attention scaling	give your cat cybernetic upgrades to make him into a super cat robot	clone your cat
Condo capacity scaling	renovate your condo so it can accommodate more guests at once	buy more condos



In real-life applications, the process of scaling is very similar to our scenario. We build products that customers want, and then as more and more customers use our products, we scale up our infrastructure to make sure it continues to work.

Vertical and Horizontal Scaling

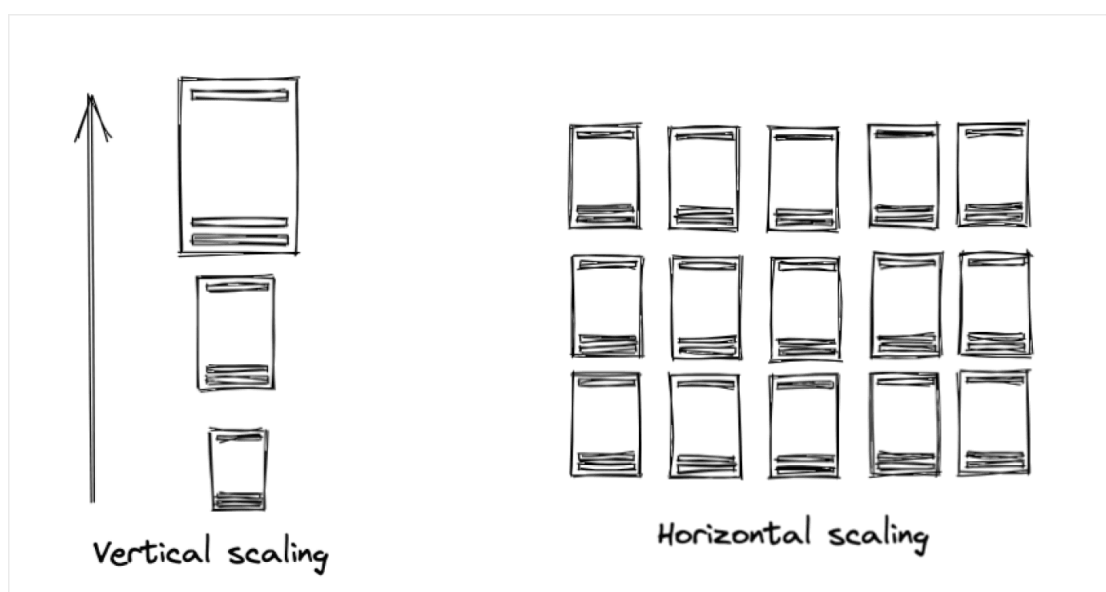
There are two major ways to scale. If you scale vertically, you make the current computer (that is serving your customers) more powerful. If you scale horizontally, you buy or rent more computers and distribute the load among the computers.

Imagine that you innovate your cat business from in-person viewing to virtually showing photos of your cat to customers. You start off holding the photos on your computer, which acts as the server whenever customers visit your website. This works fine at first, but soon you find that you are running out of space on your computer. What's worse, customers are complaining that loading the images takes too long.

To solve this problem you have two options:

- Vertical scaling: Upgrade your computer's hard drive, CPU, and RAM.
- Horizontal scaling: Buy more computers.

It is common to view these types of scaling as opposing ideas, but in reality, a combination of both types of scaling occurs in many software architectures. In the next section, we will explore the pros and cons of the types of scaling as well as their use cases.



Vertical Scaling

Vertical scaling alone is the easiest way to scale initially because it does not require you to change your software architecture. It also has the benefit of reducing latency since the communication between different parts of your software architecture is done locally on one computer rather than through a network.

However, there are a few disadvantages to relying on vertical scaling alone:

- Due to physical hardware limitations, upgrading a computer's hardware past a certain level becomes very expensive or even impossible beyond a certain point. Meaning: no matter how much money you pay, you'll never find a single server beefy enough to run a major web service on its own.
- There is a risk of the single system failing to create a single point of failure.

Horizontal Scaling

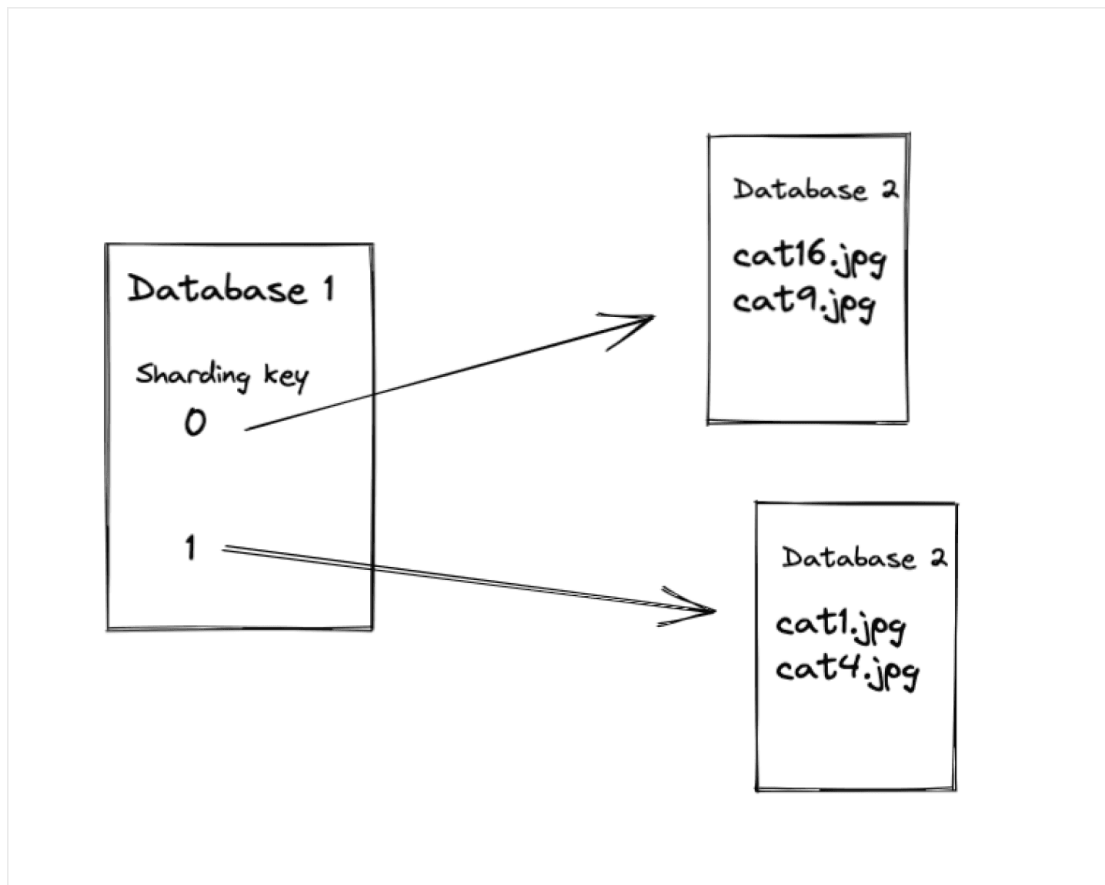
Horizontal scaling relies on building systems that communicate between multiple computers to store and process information. The two major forms of horizontal scaling are database scaling and compute scaling. We can also use horizontal scaling with caching.

Database Scaling

Imagine you need to store billions of cat photos, which will make it impossible to store it all on one computer. The way to overcome this problem is to store your images in a bunch of computers and to keep track of which computer has which images so you can get a specific one back.

One thing you can do here to improve performance is that you can add database replicas, but eventually you get to the point where you have to shard into separate databases. This technique is known as “database sharding” and involves sharding a larger database into smaller databases. Fun fact: For a special trick to remember what sharding is, think about how when Voldemort split his soul into seven horcruxes, he was basically sharding his soul.

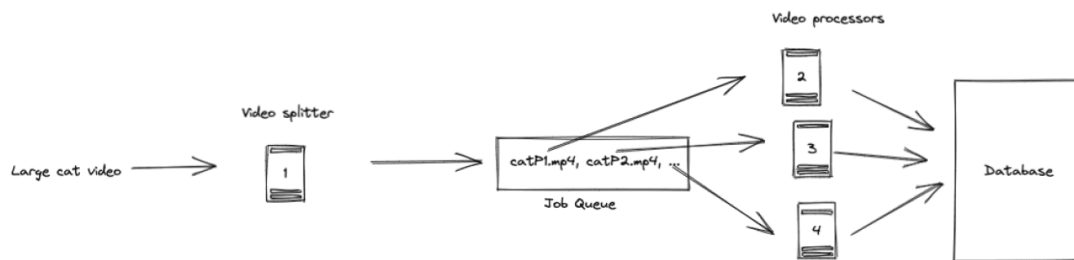
With database sharding, each time you need to store something you would have a hash function that determines in a consistent manner which database to store the item. When you need to retrieve the data again, you will use the same hash function to determine which database to look for your item.



As the data you need to handle becomes larger, you can also add additional layers of sharding to make certain queries more efficient.

Compute scaling

Imagine now that you need to process large cat videos, and you notice that it is too slow to be done on a single computer. You can use the concept of compute scaling by dividing the video into pieces and designating each piece as a job in a queue so that multiple computers can work together in parallel.



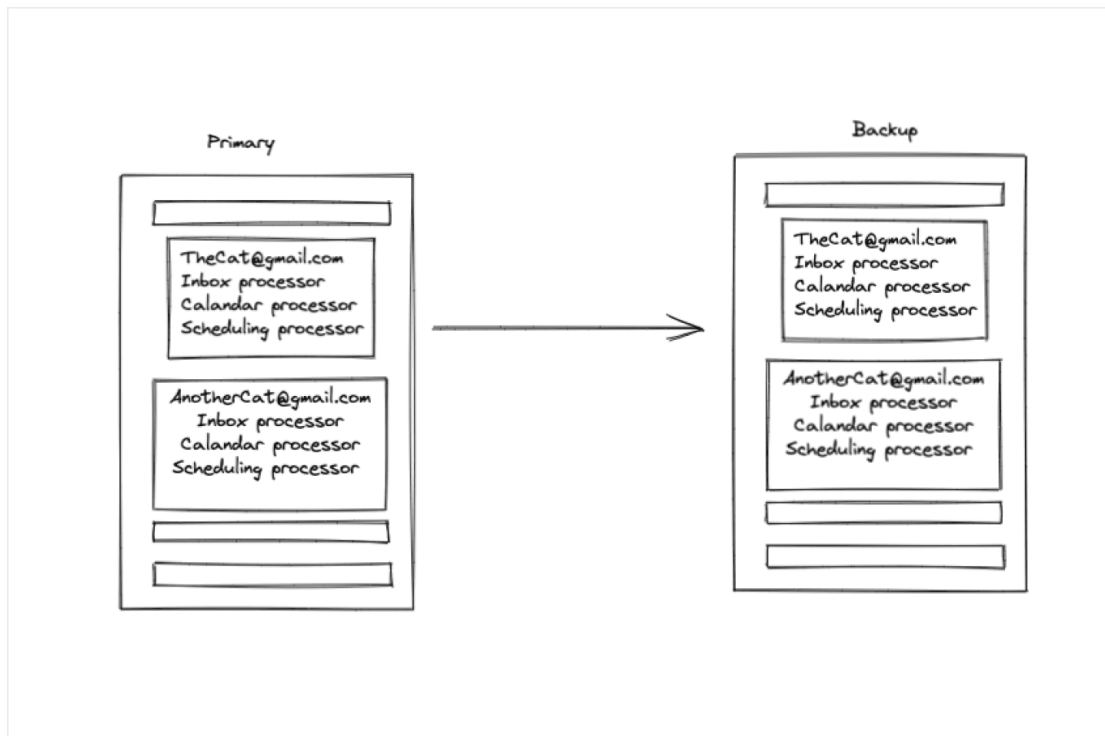
How horizontal and vertical scaling are used together

Remember

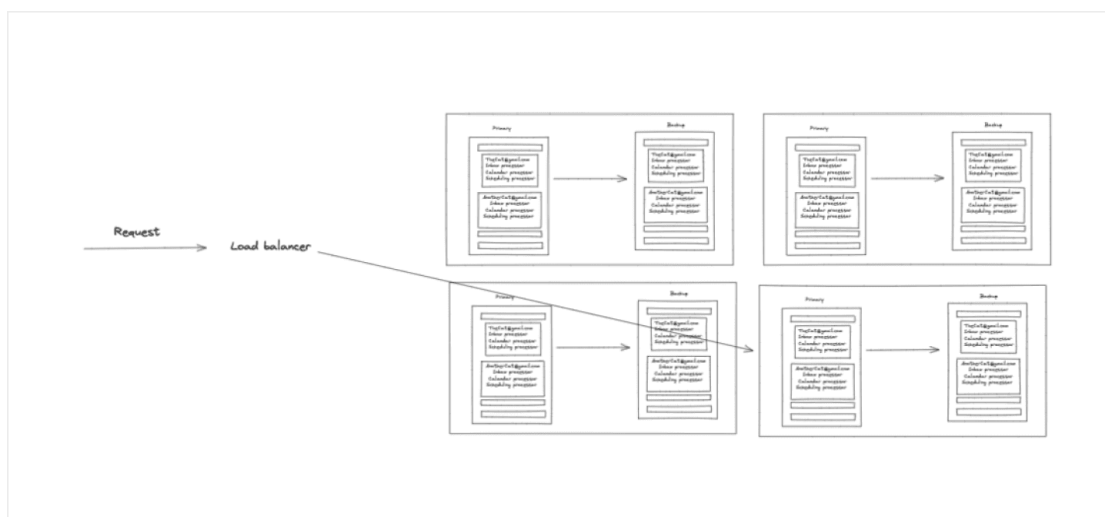
While it might seem like horizontal scaling is generally the right approach, you still need to consider both types of scaling.

As mentioned earlier, keeping certain processes and services together within a single machine can reduce the latency between calls. It is important to consider which services and processes would be more efficient within the same computer and adjust the compute and storage of each node accordingly.

A real-world example is email architecture like Gmail, Yahoo Mail, and Outlook. There are many processes that run in parallel for a given user, all of which are computationally inexpensive. In addition, systems must be very careful about guaranteeing the customer's privacy per each country's laws. Rather than distributing the processing across different nodes, a single node contains a set of users based on geographic proximity. To address problems of reliability and a single point of failure, backup nodes follow and duplicate the primary node in case it goes down.



In this case, vertical scaling is used when deciding the computer and storage of a given node. We need to ensure that it can handle the jobs for a given set of users. This architecture is also making use of horizontal scaling, where millions of primary and backup nodes are spun up to handle all the users using the email platform. When a request is made, a load-balancing service will decide which primary node is responsible for handling the given user.



Another note on consideration of scaling during interviews

A lot of system design interviews will involve how to scale to a large number of users. Do not assume, however, that a particular scaling strategy is required. Instead, have an understanding of the number of users and the amount of data that is required to inform your decision-making. Make a choice, and then explain your rationale.

d. CAP Theorem

What is the CAP theorem?

Consistency, Availability, Partition tolerance. The CAP theorem is one of the most fundamental theorems for a distributed system design.

An analogy to help understand the CAP theorem

Picture a bank in your neighborhood (let's say in New York). Imagine you visit this bank to deposit and withdraw your money. If this bank has only this one New York branch, then you depend on a single bank to access your money.



Now, let's say the bank expands internationally and opens a branch in London. This will help the bank to scale to more customers and avoid a single point of failure.

You decide to vacation in London and go to the bank to withdraw some money. You visit the London branch, but they don't allow you to withdraw, as they have no records of you depositing any money. You are frustrated, and you send bad feedback to the bank manager.

The bank officials meet to discuss this issue. It happened because the London branch didn't have access to the transactions registered by the New York branch and vice versa. In other words, they didn't have consistent data. So the banks come up with a strategy. To ensure consistent data, the branches will have to phone each other (similar to a computer network) before marking any transaction complete. Only when all the branches note the transaction information and send their acknowledgement can a transaction be marked complete.

What happens if the London branch has a holiday but New York is business as usual? None of the transactions will go through, as New York branches will keep ringing the London branch but no one in London answers the phone. To avoid this roadblock, and to make the system available even when one or more branches are unavailable, the New York branch decides they won't wait for the acknowledgments because they will send an email to the London branch instead. The London branch must record all the incoming information in their email (async communication) about the transactions before processing any new transaction on the next day when it reopens.

What happens if the telephone company has an outage? If the network has a fault, and your transaction information cannot be conveyed to other branches, the bank has a choice to make:

1. Either don't allow the transaction until the communication is back. This means that the New York branch will be unavailable to deposit money because data consistency cannot be ensured.

2. Or have inconsistent data between different branches. This means that if the customer tries to withdraw the money in London they deposited in New York, they will not be able to do it.

This is what the CAP theorem is all about: Trade Offs

It revolves around the following terms:

- **C**onsistency means that every node in a network will have access to the same data.
- **A**vailability means that the system is always available to the users.
- **P**artition tolerance means that in case of a fault in the network or communication, the system will still work.

CAP theorem says that a distributed system needs to make a choice between consistency, availability, and partition fault tolerance. It cannot achieve all three, so systems should be designed to achieve two out of these three.

In most systems, fault tolerance is necessary. So the choice is to be made between availability and consistency.

For some systems like financial systems, consistency is very important. For others like TikTok, where it is OK if some users get access to certain videos later than the rest, we try to aim for availability over consistency. Even in these cases, we want our system to eventually have the same view of the data. And that is called eventual consistency, where systems become consistent eventually, if not immediately.

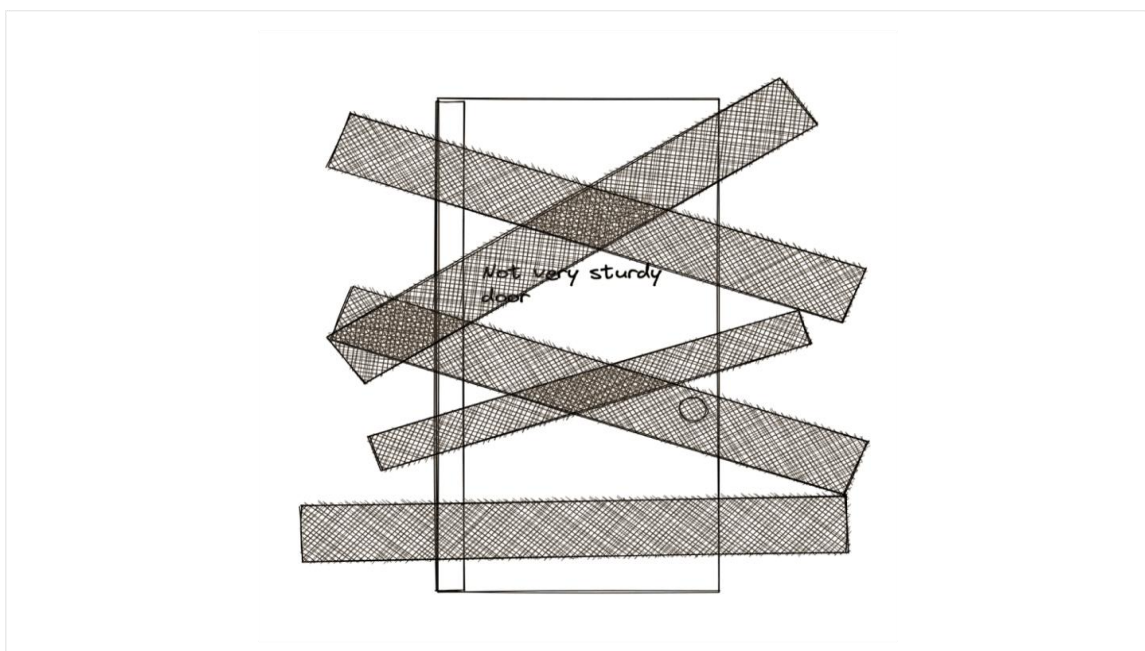
e. Web authentication and basic security

You and your close friends are in the zombie apocalypse. You're running for your lives when you happen upon an abandoned grocery store. You rush inside, closing the door behind you.

Your group is safe for now, though not for long because the entrance door is thin and made of wood.



There is enough food to survive for a while, so you nail down the door with planks of wood.



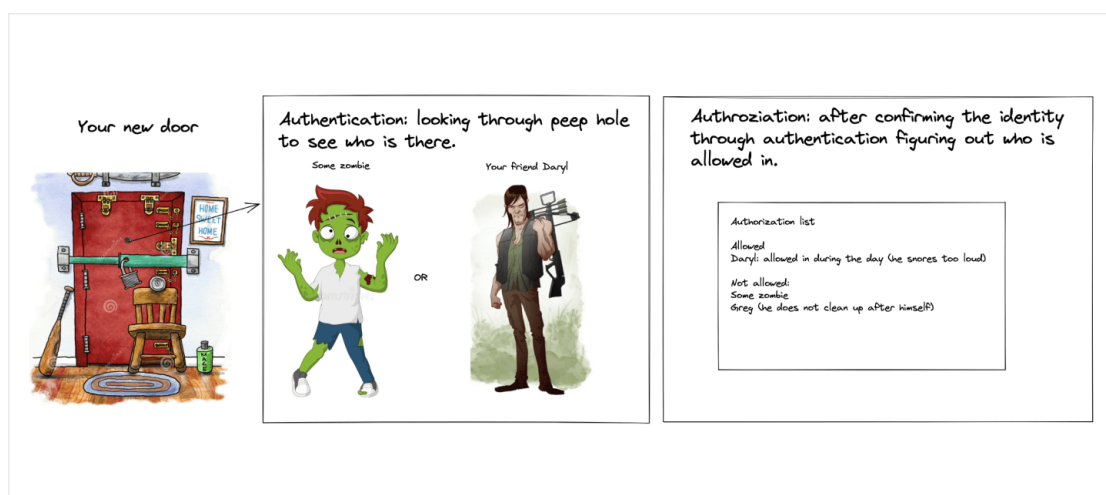
With enough pieces of word securing the door, you've essentially turned the door into a wall. One day you'll run out of food, so you know you'll eventually need to go back outside to find more resources. Plus, if there was an emergency in the grocery store (maybe a fire, maybe a monster) you'll need to get out the door then too. **The question of security is all about the trade off between total safety (a wall) vs total convenience (a hole in the wall).** The same can be said about the question of security within system design. We want to have the right amount of security that minimizes risks while still providing functionality to users. This section explores the intricacies of such tradeoffs.

Note

This section is intentionally longer than the other 11 technical topics. Mid-level to senior candidates can score brownie points with their interviewer by going into some security stuff. As a result, we made this topic more in-depth to help you get more of those extra points.

Authentication, aka "figuring out who you're talking to"

The plan is to have a very strong door with a tiny peephole so that your group can determine whether or not the door should be opened.



In software architecture, authentication refers to verifying the identity of our service's users. It should not be confused with authorization, which is the related but separate concept of determining which users have permission to take which actions. For the moment, we're just concerned with validating identities.

Storing Passwords

The “peephole in the door” approach works for a while, but the zombies are smarter than you anticipated. Recently, there were a couple of close calls where a zombie masqueraded as a survivor and folks guarding the door almost bought it. So, the group decides to create some secret passwords. That way you can rely on two factors to authenticate who’s at the door: looking through the peephole and asking for the password to authenticate. You also make sure that nobody writes the password down on a piece of a paper (in case zombies find it later).

Likewise, software often relies on passwords as a means of authentication. The most typical, classic way to do authentication is with a username and password combination. The username may be a chosen name, email address, phone number, member ID, or any other unique identifier. The password should be a text string known only to the user. To authenticate themselves, the user provides their username and password. As long as only the actual user knows the password, this verifies that they are who they say they are. So how do we implement this?

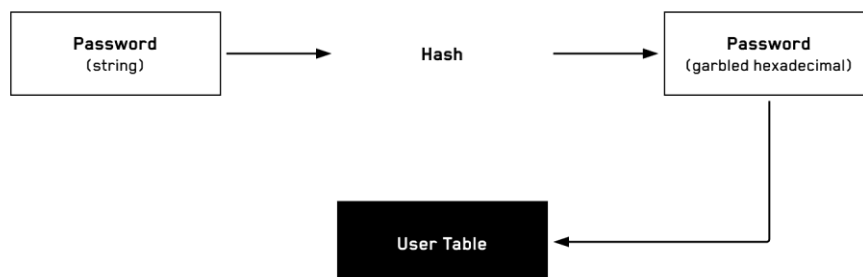
The most simple approach would be to store the password text in a data store alongside the username, and then you could compare it directly to the user's password submission. DO NOT DO THIS. The critical flaw of this approach is that if your database is ever compromised, the attacker will have access to all of your users' passwords. Because many users reuse the same passwords across multiple services, this could cause problems for your users even beyond your own service.

Hashing

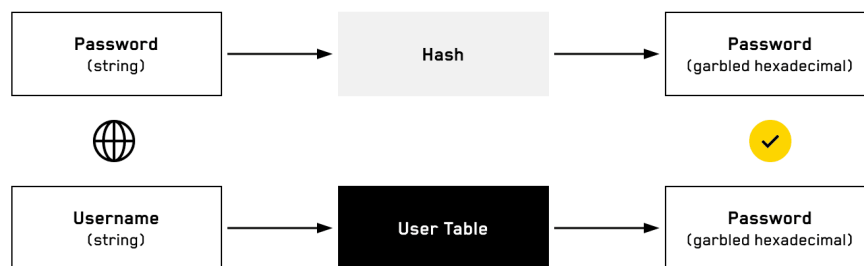
As your mini survival colony grows, you start to have more and more passwords. You are not only guarding against zombies now, but also against rival communities that want to get inside and steal precious resources. So, you decide to use a cipher wheel to hide the contents of the letter that way even if the letter is stolen the secrets within will still be safe.

Just as your group wants to avoid having their secrets stolen, databases want to keep sensitive user data safe even if a hacker gains access. The first step we can take toward secure password storage is the use of a cryptographic hash. **A hash function** is a function that accepts an input and maps it to a smaller value. **Cryptographic hash functions** in particular are designed to make it as mathematically difficult as possible to derive the input value from the output value. If you use a modern, secure hash function, it would be nearly impossible to reverse the hash for all intents and purposes.

We can use this phenomenon to our advantage. When your user first signs up and provides their password, don't store the password itself. Instead, run the password through a secure hash function and store the output - a value that we will refer to as the password's hash.



Now if we want to authenticate a user, we can apply the same hash function to the password they provide. If the hash matches the one stored in your database, then the password was correct and we can sign the user in.



Salting

Another issue you encounter is that sometimes the passwords your group sets are easily guessable. One way to solve this is to salt the password by adding in some random words that aren't obvious. For example, instead of groupPasswordRainToMainArea you could salt it by adding "spider" to the password. No one likes spiders so it makes it way more secure.



Password is easy to guess if rival groups know
you guys like rainbows, cats, and dogs

```
groupPasswordRainbowToMainArea  
groupPasswordCatsToStorageArea  
groupPasswordDogsToBackgardenArea
```

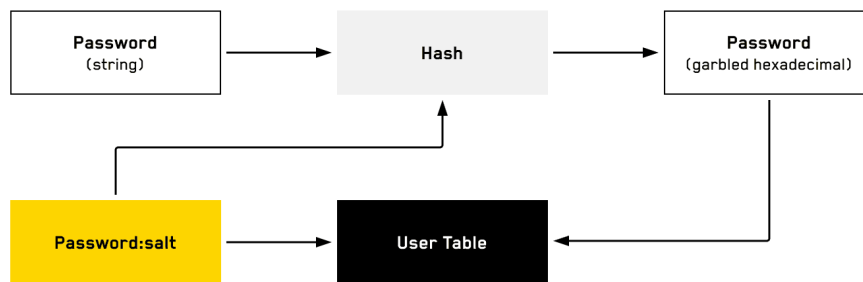


Password way more secure if you add spider
no one in your group likes spiders

```
SPIDERgroupPasswordRainbowToMainArea  
SPIDERgroupPasswordCatsToStorageArea  
SPIDERgroupPasswordDogsToBackgardenArea
```

In software security, salting our passwords makes them much harder for an attacker to uncover, but there's another vulnerability we need to address: rainbow tables. A rainbow table is an enormous lookup table containing millions of common passwords and their variations, along with their hashes for a common hash algorithm. An attacker with a rainbow table who gets their hands on a credential database could rapidly search the database for users whose hashes are in their rainbow table and then quickly compromise those passwords.

Unfortunately, users frequently use common passwords (that is, after all, what makes them common), so we should defend against this attack. We can do that with a technique called "salting." The basic idea behind salting is that rather than hashing the password itself, we hash the password concatenated to a random value, which we can store in plain text along with the hash. Then we apply the same salt when hashing user-provided passwords for login attempts.



The salt is not a secret. Its purpose is simply to render rainbow tables unusable. Say we use ten random characters as our salt, and user A's password salt is "a8h2rmd1tb". User A's password is "password123". If an attacker obtains the credential database, they will certainly have the hash of "password123" in their rainbow table. But they almost definitely will not have the hash of "password123:a8h2rmd1tb" in their rainbow table, which means they won't be able to derive user A's password from the table.

The attacker still has the salt, and it doesn't stop them from using brute force attacks in which they simply try random combinations of words and characters until they find the correct password. But the salt forces them to do this work for every user rather than simply looking them up in a table. Using a slower hash function will further frustrate brute force attacks by increasing the amount of computer time the attacker needs in order to compromise each password.

Real World Password Hashing

In actual practice, avoid writing your own code to salt, hash, and store passwords. The principles are simple, but the implementation details are tricky, and a subtle mistake could severely compromise your password security. Instead, look for

modern, vetted, secure libraries that can do hash generation and password comparison for you.

Web Sign-in

Now that we've taken care of storing our password hashes and salts, we need to actually sign our users in. The classic way to do this in a web UI is with a form submission, but nowadays we could use an async request as well. Sending the user's username and password submission, of course, is a sensitive moment in the authentication flow. Login handlers should always use HTTPS to ensure that credentials don't pass in plain text over the network where they could be intercepted. Of course it's a good practice to encrypt all web traffic with HTTPS, but password submissions in particular must be protected.

Another pitfall to avoid with password submissions is logging code. It's all too easy while setting up a web project to write automatic logging code that captures the raw contents of all requests. If this type of automatic logging runs on login endpoints, you'll write all of your users' passwords in plain text to your logs, completely defeating any progress you've made toward secure password storage. A good rule of thumb is to avoid automatically logging POST bodies and GET parameters. Instead, keep automatic logging to things like URL paths and response codes, and log parameters manually when necessary after verifying that they don't include sensitive credentials.

After your user has submitted their password to log in, assuming they pass verification, you need to give them a way to prove their authentication on subsequent requests. We don't want the user to have to submit their password with every single page request, of course.

Session Tokens

A classic, simple way to track authentication is to generate a token the user can submit with subsequent requests to track that they are, in fact, signed in. This token

should be randomly generated—be sure to use a secure random number generator—and long enough to be infeasible to brute force. This token will represent this particular sign-in session for this user. Now you can check for this session token in a cookie on subsequent requests (more on cookies later) to verify that you're still dealing with the same logged-in user.

Crucially, **the session token is equivalent to a password** and should be treated accordingly. Don't store the token in your database in plain text, but rather salt and hash it the same way you would a password. Session tokens should also come with an expiration date, as short as feasible. The expiration of a session token doesn't mean that the user has to log in again—you can rotate them silently in the background between one request and the next. It will limit the amount of time the user can remain logged in without visiting your site, so use that as your guiding principle when selecting a token expiration time. The shorter the expiration time, the less opportunity an attacker who steals a user's session token will have to exploit it.

This is, of course, another fantastic reason to encrypt all of your traffic with HTTPS. If you secure only the login endpoint, the user's subsequent unencrypted requests will be vulnerable to interception and each of them will include their session token.

JSON Web Tokens

Rather than plain session tokens, you may also opt to use JSON Web Tokens, or JWTs. While a session token is an opaque string that means nothing without access to the session database, a JWT explicitly encodes the user's access. It may list the logged in user's username or ID, as well as explicit capabilities or permissions that have been authorized for that specific login. The JWT standard defines a canonical way to encode this data along with expiration and other metadata, although you can use the same concept for your own custom payloads if needed.

JavaScript

```
{  
  "sub": "user123",  
  "email": "user@domain.com",  
  "iat": 1665385660  
}
```

Of course a JSON payload listing the user's ID and permissions would be comically easy to falsify, so we can't use the JSON alone to verify a logged-in user. To render the JWT valid, the server must do one of two things.

1. Sign the payload

Attaching a signature from a private key held only by your service verifies the token's legitimacy. This approach has two primary advantages. The first is that the token is fully transparent. You can use it on the client side to tell who's logged in and optionally what permissions they have. The other advantage, and perhaps the strongest selling point for JWT as a whole, is that you can use the token with other services. As long as you publish the corresponding public key to the private key used to sign tokens, not only your own services but services from entirely different providers can validate the token, allowing your user to authenticate elsewhere on the web.

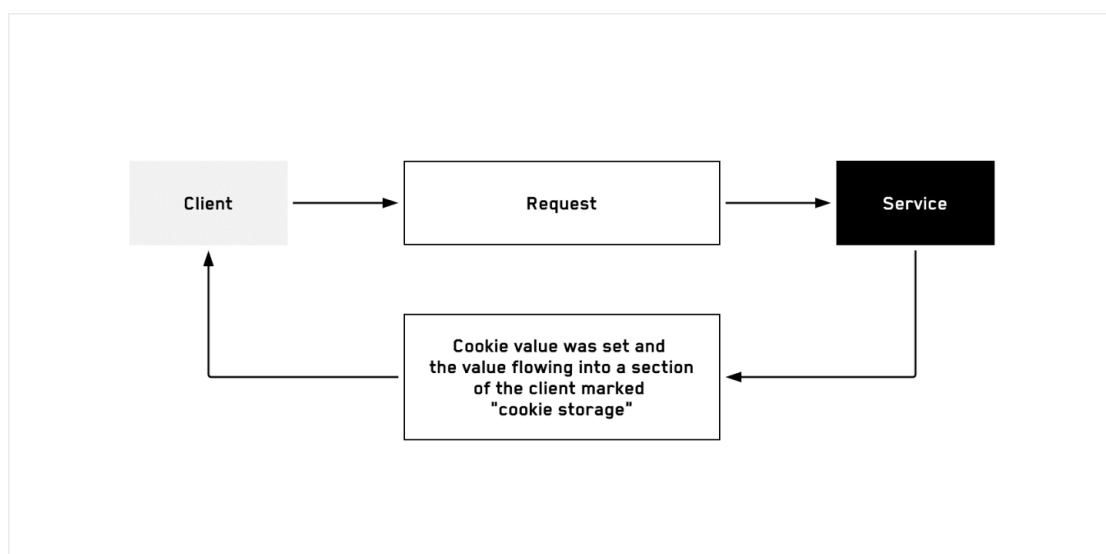
2. Encrypt the payload

Alternatively, you may encrypt the payload so that only your service can read it. This is a less common approach to JWT, and it turns the token into something closer to a simple session token—neither the client nor third-party services can derive any information from it. This approach may still be useful, however. If you distribute the decryption key to different services within your project, they will be able to validate encrypted tokens without needing access to a centralized session database. In more niche use cases, you can also use this approach to "store" small amounts of data for the duration of the session without writing them to a database or making them visible to the user. Just insert the data you need into the encrypted token, and you'll have access every time the user presents their token to verify the session.

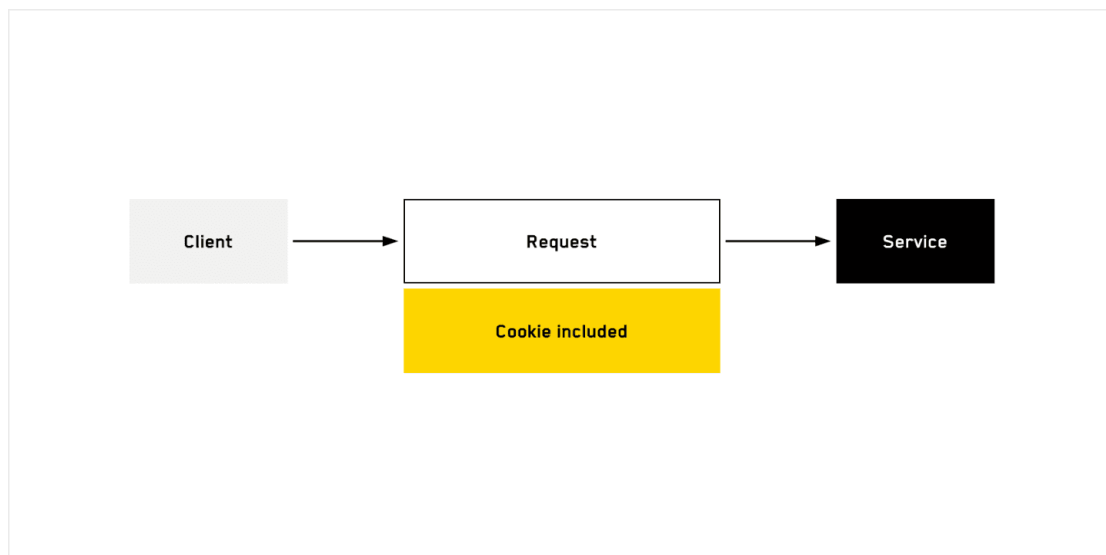
Cookies

Whether you use session tokens or JWTs, you'll need a mechanism to store the token on the client side and send it back to the server with each request. The standard way to do this in web applications is to use browser cookies.

A cookie is simply a text string that the browser associates with a given key and sends back to the server whenever it makes a request. The server may include, in any response, one or more "Set-Cookie" headers. The header indicates the key and value to store, along with some metadata regarding expiration and security.



Subsequently, the browser automatically includes those cookie values in every request made to the same domain (potentially limited by path, if one was specified while setting the cookie). By storing a session token or JWT in a cookie, we can ensure that all subsequent requests will include it and allow the server to validate the current user session.



Since our auth tokens—whether session token or JWT—are equivalent to passwords, we need to be very careful about who can access these cookies and how. Thankfully, browsers have made great strides in recent years toward keeping cookies safe.

Setting the "Secure" flag on a cookie tells the browser only to include it in HTTPS requests, keeping it out of unencrypted traffic that could be intercepted. The "HttpOnly" flag tells the browser not to allow access to the cookie through JavaScript. The cookie will still be sent in request headers, but snooping frontend code won't have direct access to it. The "SameSite" flag allows you to specify the degree of caution the browser should take when sending requests that originate from a different site, to help prevent cross-site request forgery (CSRF) attacks.

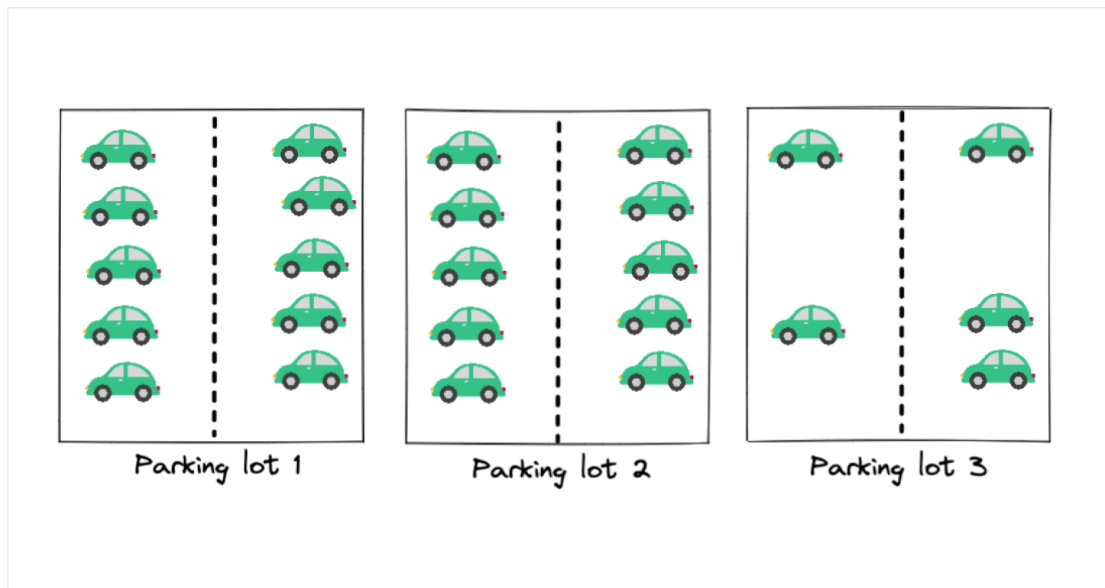
Summary of web authentication and basic security

We can view this entire process as a lifecycle for a user's account, which would go something like this.

1. The user signs up. At this point, we need to salt and hash their password and store those values **(but not the password itself!)**.
2. The user logs in with their username and password. We verify the password by hashing it with the stored salt and checking to see if it matches the stored hash (ideally using a secure library to make the comparison). We then send some kind of identifying token, either a simple session token or a JWT or similar token, back to the client in a cookie set header.
3. On subsequent requests, the browser sends the cookie back to the server, where we can verify the session token or check the signature on/decrypt a JWT.
4. Periodically, the session token or JWT should be expired and a new one generated and sent down to the client with a cookie set header.
5. Eventually, the user's session may expire from inactivity. In this case, we go back to step 2.

f. Load balancers

Imagine that you're making good money because you own three parking lots near a busy shopping district. If you could maximize the amount you make daily, you might even retire early and forget about all this system design nonsense. Recently though, you discovered that you aren't making as much as usual through parking fees, so you decide to investigate. Here is what you see from today's security footage:



Well, that explains it! People seem to favor going to parking lots 1 and 2, so when it fills up they end up leaving thinking there isn't any parking available here. As a result, you work to design a system that would direct people to parking lots in a way that ensures each lot is being used in a balanced manner. Ta-da! You now understand why we need load balancers.

Load balancers are an important component of any distributed system that helps distribute traffic across the machines. As distributed systems are designed to scale, the basic requirement is to add or remove machines in case of increased load or in case of failures. Load balancers also help manage this.

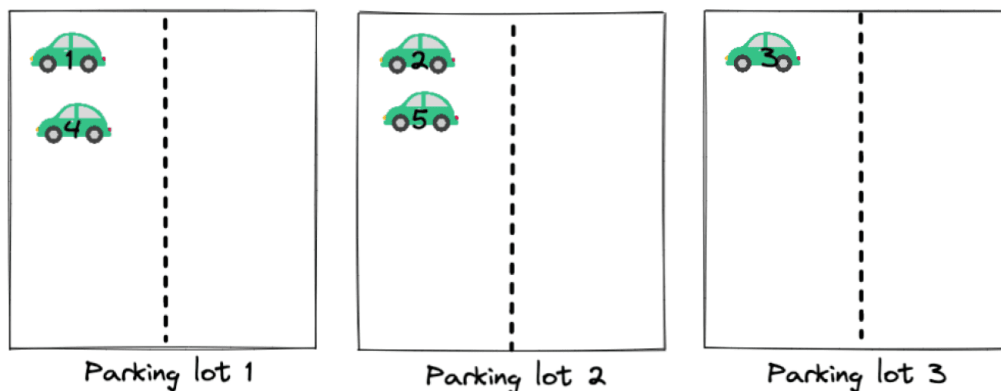
An analogy to help you understand load balancers

You could stare at cameras and then let people know through an LED sign which lot has available space, but that seems like a lot of work. Maybe you could place sensors at each lot to detect if a car is parked there? Well, you're not a hardware person, and all those sensors are going to cost money that you'd rather save for your early retirement. Are there smarter ways to accomplish this without a bunch of words or extra sensors? Absolutely. Let's take a look at the options.

Round robin

In this technique, if there are “N” machines, then the load balancer will send requests to each of them one by one. Although this should mostly work, in the case where a machine is loaded (not able to process requests in a timely fashion), the load balancer keeps bombarding the machine.

For our parking lot example, let’s assume there are 3 parking lots. The person at the entrance decides to send each of the next vehicles in the next lane. The first vehicle goes into the first lot, the second vehicle into the second lot, and the third vehicle into the third parking lot. But then the fourth vehicle starts from the beginning and enters the first lot.

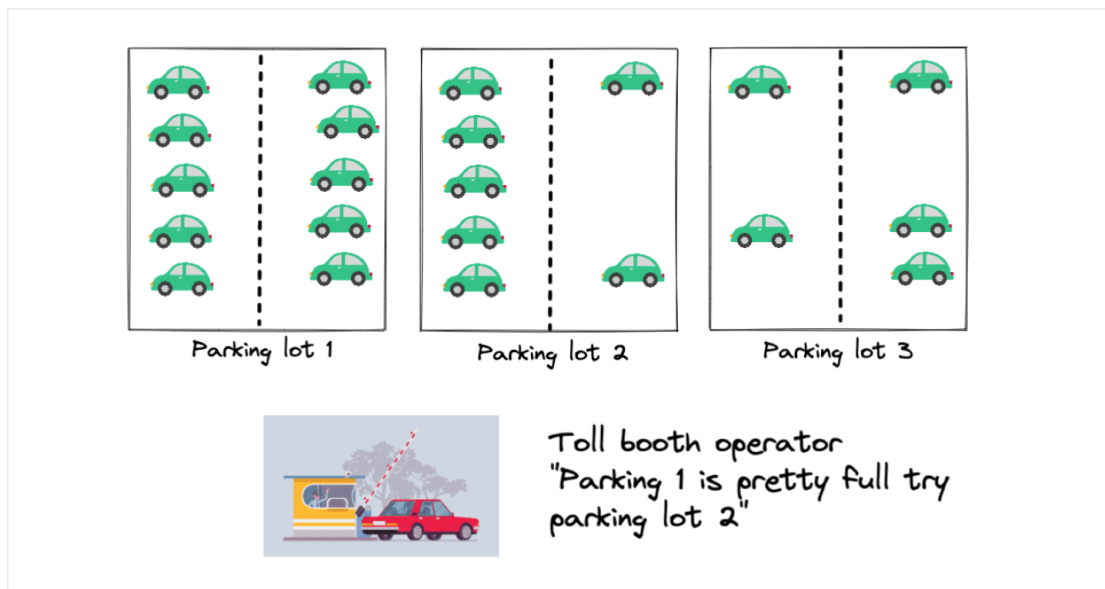


Least connections / response time

We send the requests to the machine with the least connections or the minimum response time. This option is useful for incoming requests that have varying connection times and a set of servers that are relatively similar in terms of available compute and other resources.

For our example, let’s now assume that there is a faulty parking ticket machine in one of the lots, so to make up for the broken machine, someone is manually (and

slowly) handing out tickets. The vehicles in that lot are more full, so the person in charge decides to send fewer vehicles in that lane to avoid overloading that lot.



Hashing

Hashing is based on an arbitrarily chosen key. The key for hashing can be a request id for a given user or an IP address. The hash function returns a numeric result for each key. Using the hashing result, this is how we can obtain the server id to which the request should be sent:

$$\text{machine_id} = H(\text{request_id}) \% n$$

where $H()$ is the hash function, "%" is the modulo operator, and "n" is the number of machines present in the system.

Example (values are arbitrarily chosen to demonstrate the concept):

$$\text{request_id} = 292341$$

$$n = 160$$

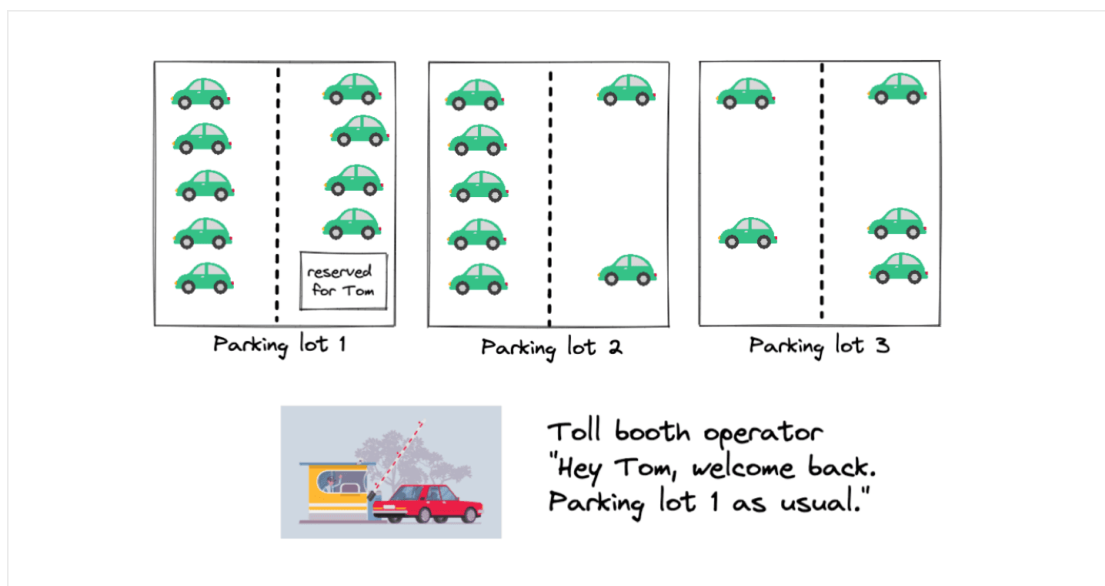
$$\text{machine_id} = H(292341) \% 160 = 4276751699036 \% 160 = 112.$$

In this example, the resulting machine id is 112, and the request will be sent to the corresponding machine/server.

With hashing, the addition and subtraction of machines leads to the same request being sent to multiple machines.

If there are too many vehicles and lanes, we might need to keep track of the previous vehicles and their parking lots for the round-robin method. To scale, we would need multiple load balancers (so we'd have to hire multiple people to work at the entrance). To have a fair distribution of vehicles in the lanes, the parking lot attendants come up with a strategy. First, they will locate the first numerical digit of any incoming car's license plate. And then they will send the vehicle to the lane that corresponds to that numerical digit. Let's say that the vehicle number is ABC 456, so the vehicle will be sent to the 4th parking lot. (This assumes that there are 10 parking lots, and the digit 0 will go into the 10th parking lot.)

In this scenario, we could also ensure that returning customers get sent to the same parking lots that they are used to, which could be a nice benefit when a particular parking lot is closer to the customer's favorite store. In system design, sometimes it is important for a client to connect to the same server; for example, when there needs to be a WebSocket established for two-way communication.



g. Caching

What is caching?

Let's use a small example to understand. If you had to write a function that calculates the number of lines in this row, how would you do that?



A row of 11 vertical bars, each representing a line, arranged horizontally.

You write a function that calculates and loops through it, and it counts the number of lines. In this case, the function would count 11 lines.



A row of 11 vertical bars, each representing a line, arranged horizontally. To the right of the bars is a right-pointing arrow, followed by a yellow circle containing the number 11.

Let's say someone asks, "How many lines are in this row?" Well, since the function already counted the lines earlier, someone would be able to see "11 lines" and not have to have the function count up the number of lines again.

In essence, that is exactly what caching is: storing an expensive computation so you don't have to compute it again.

Why use caching?

To reduce latency of an expensive network computation or network calls or database queries or asset fetching.

Generally, caching is used at the expense of more storage (or more memory). Generally we store it in memory (RAM), but it can be stored on disk too.

What concepts related to caching come up often in interviews?

Rule of thumb

Generally, caching is used for read-heavy systems. Popular read-heavy systems are Twitter and YouTube. Are most of their millions of users tweeting and posting videos? No, their most common users are reading tweets and watching videos.

Rule of thumb

The 80/20 rule: You want to store 80% of read requests in 20% of storage (or memory). Generally, these are the most popular requests.

For example, if there is a tweet from a very popular person on twitter you would cache it everywhere (on a device, on a CDN, on the application) so that whenever that tweet is finished you can fetch it very quickly and serve it up to your users.

A CDN (content delivery network) is something that's commonly used to deliver cached data.

Taking a closer look at the example of Youtube

If there is a video from a very popular channel, you'd want to cache that data everywhere so users can get to it quickly. We can use the 80/20 rule here: 80% of the queries should be served by 20% of the popular data which is stored.

Caches store data in local regions. For example, let's say there is a Britney Spears video which is more popular in America. You should cache the Britney Spears video for your users in America, so they can access that video quickly.

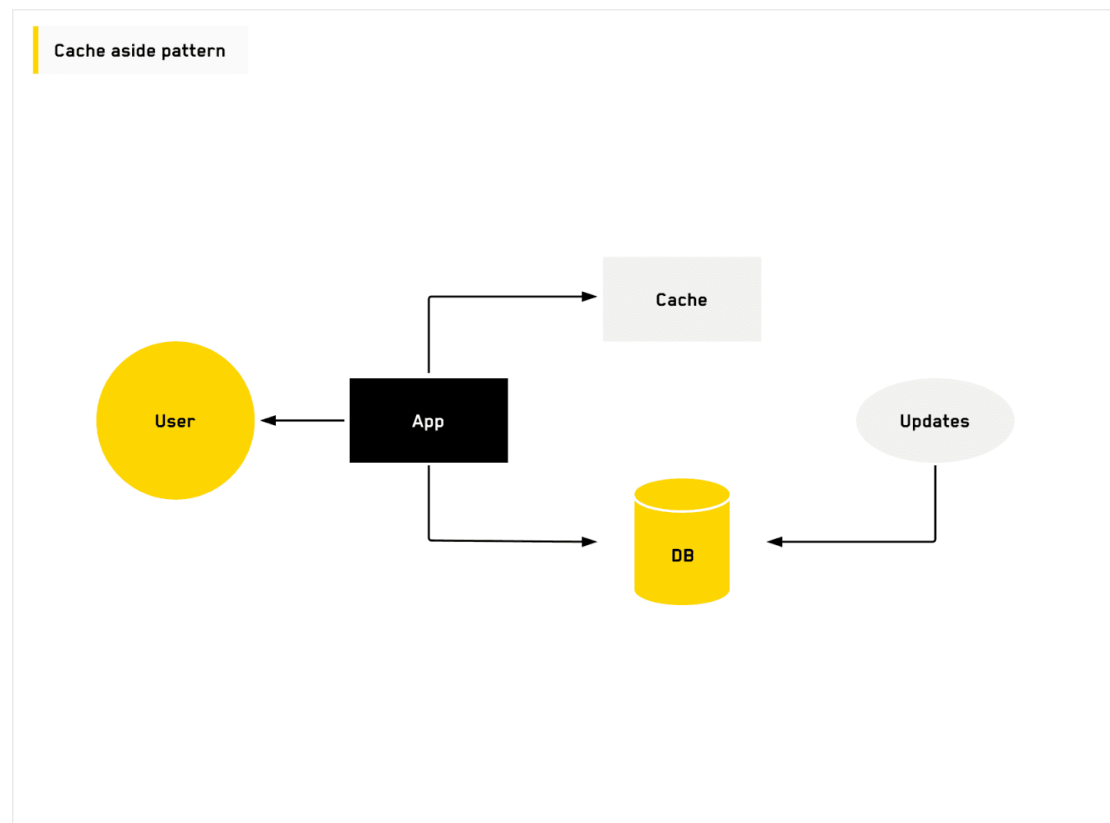
Let's say there's also a Korean drama that's more popular with users in Korea. You'll want to cache this Korean drama for the users in Korea, so they can watch it quickly whenever they want.

Popular Caching Patterns

1. Cache aside pattern
2. Write-through pattern and write-back

Cache-aside pattern

This is the most popular cache pattern. In this pattern, we have an application which will try to fetch data from the cache, and if the data is not found (also known as a "cache miss") it will fetch data from the database. Or it will do an expensive computation. And then it will put that data back to the cache before returning the query back to the user.



In this pattern we only cache the data that we need, which is advantageous. One disadvantage of this pattern is that the data can become stale if there are lots of updates to the database. This disadvantage can be mitigated by having a “Time To Live” (or any other expiry pattern)—this is a concept we can skip for now since we’ll return to it later.

Another disadvantage to this pattern: If there are a lot of cache misses in our application, then the application has to do a lot more work than in the regular flow of just fetching data solely from the database. In this case, the application will first go to the cache, then there will be a cache miss, then it will go back to the database, and then write that data back to the cache before going back to the user.

If there are a lot of cache misses, then this cache is causing more problems than it’s worth.

Remember

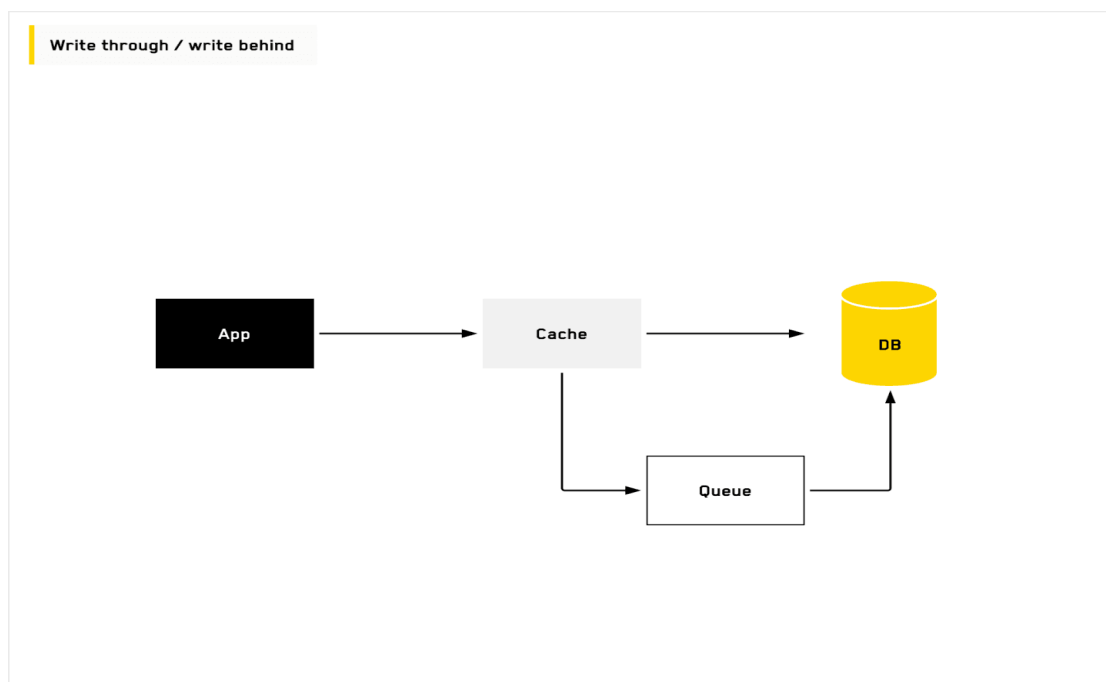
We use caches to reduce latency, but in this case (of many cache misses) we're making latency worse because of all the extra work the application has to do.

Write-through and Write-back patterns

In these patterns, the application directly writes the data to the cache. And then the cache synchronously (or asynchronously) writes the data to the database. When we write it synchronously it's called "write-through," and when we write it asynchronously it's called "write-back" (or "write-behind"). In the asynchronous example, we put the data in a queue, which writes the data back to the database and improves the latency of writes on your application.

Rule of thumb

If you're experiencing slow writes, a quick fix is async writes to the database.



In both of these patterns, there is an obvious disadvantage. Here we are writing all the data to the cache, which might not even be read. Hence, we are overloading the cache (or cache memory) with expensive calls that might not even be required. For example, there are some accounts on Twitter that are not followed by many people. If we put every tweet in the cache from these unpopular accounts, it will take up expensive memory. Also, if the database goes down before the data is written to the database, this causes inconsistency.

Using caching on the client side

As we discussed earlier, caching can happen anywhere—it can be for database queries, expensive computations, or to save network costs.

Caching is also very popular on the client side. Let's use the example of a Netflix browser or mobile app. Some recommended movies will remain recommended over a period of time. For these movies, it makes sense to cache the thumbnails of these movies in the device or browser. That way when the user visits Netflix again, these thumbnails will be quickly served from the cache (or local memory) instead of fetching it from the network.

Cache invalidation

One of the problems we have observed with caching is that data can become stale if there are lots of updates to the database. Therefore, it is important to expire or invalidate data from the cache, so your data doesn't get stale! **It's a good practice to include a brief point about cache invalidation during system design interviews.**

There are many policies to invalidate data from the cache. The most important is called "Least Recently Used" (or LRU). Here's an example to demonstrate LRU.

Let's say you have some memory that can only store 4 items, and you want to cache these items in the memory. The first item you have is "A" and you cache it at timestamp "1", the next item is "B" and you cache it at timestamp "2", the next item is "C" and you cache it at timestamp "3".

LRU

A	B	C	
1	2	3	

If the next item is “B” again, you’ll have to update the timestamp for “B”: it’s now “4” instead of “2”.

LRU

A	B	C	
1	4	3	

If the next item is “D”, then the timestamp for “D” would be “5”.

LRU

A	B	C	D
1	4	3	5

If the next item is “E” (with a timestamp of 6), then we have to make some room in our memory! Who do we get rid of? Which is the least recently used item? “A” right? Because it was the lowest timestamp of “1”. So we replace “A” with “E”, and with “E” comes the new timestamp of 6.

LRU

E	B	C	D
6	4	3	5

What if the 7th item is C? We don't need to replace "C" with "C", we simply need to update the timestamp for "C" to "7" because the latest value when "C" was accessed was 7.

LRU

E	B	C	D
6	4	7	5

What if the 8th item is "F"? Who do we get rid of now? That's right, "B"! Because unfortunately for "B" they are the least recently used; their timestamp of 2 is currently the oldest one on the board. So we replace "B" with "F" and with "F" comes the timestamp "8". And this pattern would continue as we bring new items into this memory store.

LRU

E	F	C	D
6	8	7	5

Remember

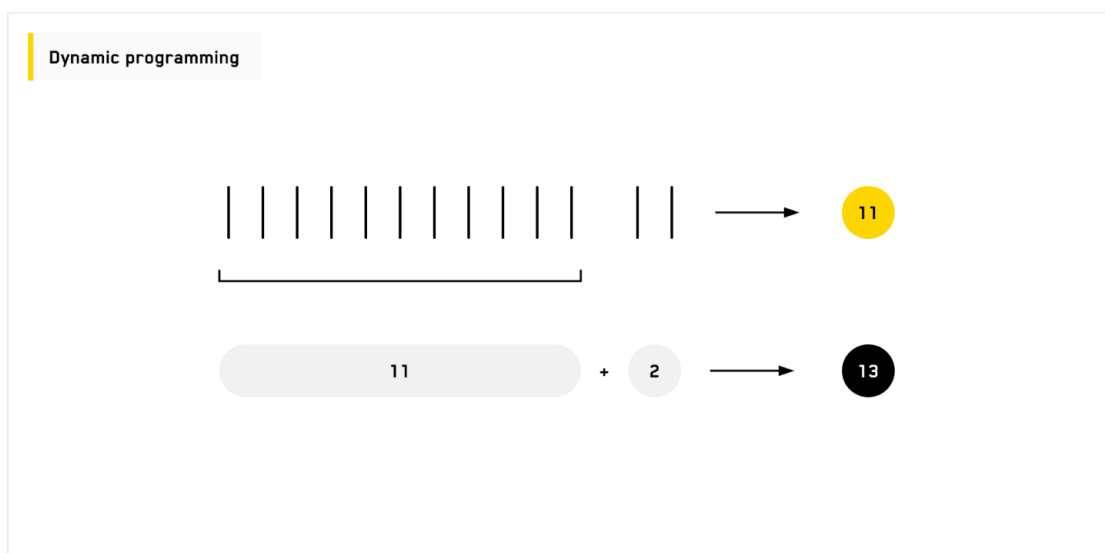
For most systems 20% of the data accounts for 80% of the reads. So using LRU will result in fewer cache misses. Because of the 80/20 rule, we want to give special treatment to the most popular data! That's why we use LRU. As a result, we can throw stuff in the cache (and not miss), which reduces latency for 80% of your requests.

Bonus

Think back to the earlier example in which you're asked, "How many lines are here?" And you'd say "11" because this data is cached.

But what if we added 2 more lines? Now, your data is stale.

How can you count the new number of lines without recounting all of the lines? How can you use the value you already have (11) and then add 2 to update the value in your database or cache to 13? This is dynamic programming.



h. Message queues

It's sometime in the far future. You wake up to find that you have been frozen and perfectly preserved at your current age. You climb out to find yourself in an abandoned city with remnants of 21st century life.

You start calling out and asking if anyone is there. Silence. All you hear are the sounds of insects chirping and birds in the distance. You start to panic. Loneliness washes over you. You want to go back to your friends, family, netflix and chilling, and most of all you want to finish reading that awesome system design guide from interviewing.io.

Suddenly, off in the distance you see movement. As it comes closer, you realize they are people. They are dressed kind of like the Flintstones cartoons. At first, you're not sure how you feel about them. Then you realize they speak some offshoot of English so you decide it's safe to live among them while you try to figure out a way back home. They respect you immensely and you quickly become regarded as a leader of the tribe.

Other than the giant step back in technology, you notice other less obvious things. In particular, there's one quirk that causes a lot of social conflict. These future cavemen seem to have forgotten the concept of lining up and waiting their turn for things. Whenever there is food or water, or anything desirable, there is chaos which usually crescendos in a heated argument or a fight: and the resources don't get distributed efficiently. It takes a person who has authority and respect from the tribe (you) to calm things down again, so everyone can get what they need.

This makes you realize how important queues are for a community to be efficient. It shouldn't take a leader to watch over things everytime resource needs to be distributed. Instead, the leader of a tribe should introduce the concept of queues. That way people can just line up and be served when it is their turn. You decide that if this whole "queue" thing goes well you might make future amendments to it: such as food and water going to children and elderly people first.

Congratulations you are well on your way to rebuilding human civilization. Bonus, your brain just downloaded a cheatsheet about message queues and how they make distributed systems more efficient. These systems are made so you can add a task and then move on knowing the task will be handled afterwards, similar to how in a well-functioning community you can add a basket of food and people will line up to be served in an orderly fashion.

What are asynchronous systems and message queues?

An asynchronous system is one where the client sends the request and does not wait for the message to be delivered. For the client, sending the message is like any other function or an API call, but in contrast to the APIs, the clients do not expect a response. Such functions or API calls are also known as “fire and forget” calls. The client’s message is not immediately sent over to the processing system but instead sent to an intermediary. This intermediary is the message queue or the broker.

Advantages

1. A queue stores messages that need to be stored in a database. This is used if there's the possibility that traffic will spike, causing the CPU of the database to go up like crazy and kill the server. That would cause the database to be down and probably lose data. Instead... throw it in the queue!
2. If a message has to be processed by some very expensive code, you may also hold them in a queue while previous messages are being processed so you don't overload (and potentially kill) servers.
3. Queues can deliver messages to multiple systems, instead of the client having to send them to all the required systems.
4. Queues decouple the client from the server by eliminating the need to know the server address.

A few things to understand about message queues

- In an asynchronous system, there are producers and consumers/subscribers.
- Message is the data that flows through an asynchronous system. There is usually no particular data model enforced.
- Producers are the systems that produce the messages.
- Consumers are the systems that process those messages.
- Message Queue or topic is the link between the producers and the consumers.
- Producers publish the messages to a named queue, and the queue delivers these messages to the consumer of the queue.

Remember

There can be **different message queues for different needs** and consumers subscribe to them based on their requirements. Alternatively, there can be a single message queue delivering all messages to all the consumers and filtering out the messages that they can process. Or a single message queue can be smart enough to selectively deliver messages to the appropriate subscribers.

Based on the different implementations of message queues, there can be different combinations of the following properties:

1. Guaranteed delivery.
2. No duplicate messages are delivered.
3. Ensure that the order of messages is maintained.
4. At least once delivery with idempotent consumers.

Popular production-grade products

Remember, you don't need to say the brand names in interviews, unless you have enough experience to talk about alternative products as well.

Kafka

RabbitMQ (based on AMQP protocol)

i. Indexing

You are Doreamon, a magical robotic Japanese cat that has a futuristic gizmo to solve any problem you run into.

Even though you're magic, one thing that frustrates you is how much time you spend looking for the gadget you need. Since your pouch is extremely large, you wish you had a more organized way to store things.

This is where the concept of indexing can save the day! Whether you are a magical robotic cat or you are designing a complex software architecture, understanding indexing will help you find items faster and more efficiently.

For example, have you ever experienced a slow website or a customer portal that takes too long to fetch data? A pervasive issue with such systems is the lack of proper indices. As a result, the database queries take much longer than expected to find the information you are looking for.

What is indexing?

Indexing is a mechanism by which the underlying data is mapped for faster retrieval. For a system to process an instruction involving data access, these are the certain steps involved:

1. Fetch the block of data from the hard disk (secondary/permanent storage) to the primary memory (e.g. RAM).
2. Look for the desired data in the RAM.
3. Repeat 1 and 2 until desired data is fetched.

Remember

Indexing makes looking for this desired data faster.

How does indexing achieve faster data access?

Consider a book from which you want to read a specific chapter. To do so, you would look for the desired chapter in the index. And based on the page mentioned in the index, you can quickly reference the desired chapter. Essentially, this is what a database index does.

In the hard disk, there might be GBs or TBs (or higher orders) of data stored. If we try to look at each record in the disk to search for a specific record (which is like going through each page of the book to look for the desired chapter) this would take a long time. Instead, if there was a way to keep an index of records on the disk, the process would be much faster. This index for records is more like a map or a key-value pair, where the key identifies a record (or a set of records) and the value is the location of the record on the disk.

When the index maintains a key-value mapping for each of the records in the database, it is called a dense index. On the other hand, when the index maintains a mapping for a subset of the records, then it is called a sparse index.

An example to take a closer look at indexing:

- Assume that a hard disk has 100 data records, each of size 128B.
- Size of each block in the disk = 512B
- This means each block can hold 4 records.

If our RAM loads one block at a time from the disk, then to search for the desired data in all the records, in the worst case, we would need to fetch 25 blocks ($100/4$).

Consider that now we maintain a map (a set of key-value pairs) of the record ID (key) to the block address (value). And let's say that each key-value pair takes 2B. This map would be 200B.

This map is nothing but an index. Now, if we search a record in the hard disk, the index would first be loaded in the RAM, and then based on the key, we will retrieve the block address.

Once we have the block address, the specific block from the hard disk would be loaded instead of loading all the blocks. And from the block, we can fetch the desired record.

This means that by maintaining an index, we could reduce the I/O calls to the disk substantially, from 25 calls before the index to 2 calls (one for the index and the other for the specific block).

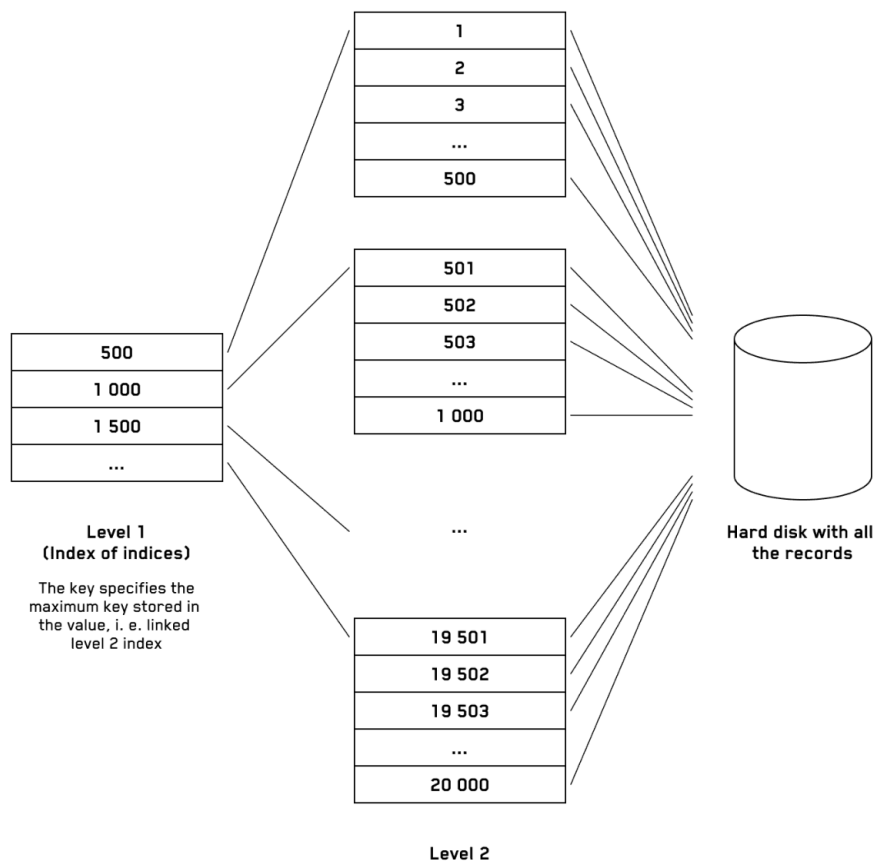
Multilevel index

As the size of the database grows, the size of the index will also grow. A large index requires multiple I/O calls to read the index itself.

In the above example, if the size of the database grows from 100 to 10,000, the size of the index would be 20,000B ($2 \times 10,000$). To load the index in the RAM, we will require ~ 40 ($20k/\text{size of the block}$) I/O calls. This is still better than the scenario without the index, in which 2500 ($10000/4$) I/O calls would be required in the worst case. However, we can further improve this with the help of multilevel indices.

Multilevel indices maintain an index of indices. This means that to fetch a record, first an index is referenced to get the address for the block storing the correct index, as shown in the following diagram.

Multilevel index



These multiple levels of indices reduce the I/O calls from 40 to 2 (excluding the call to load the record):

1. One call to load the level 1 index.
2. The second call is to load the level 2 index as determined from the level 1 index.

The levels of indices can keep on increasing as the data grows to keep the indexing efficient.

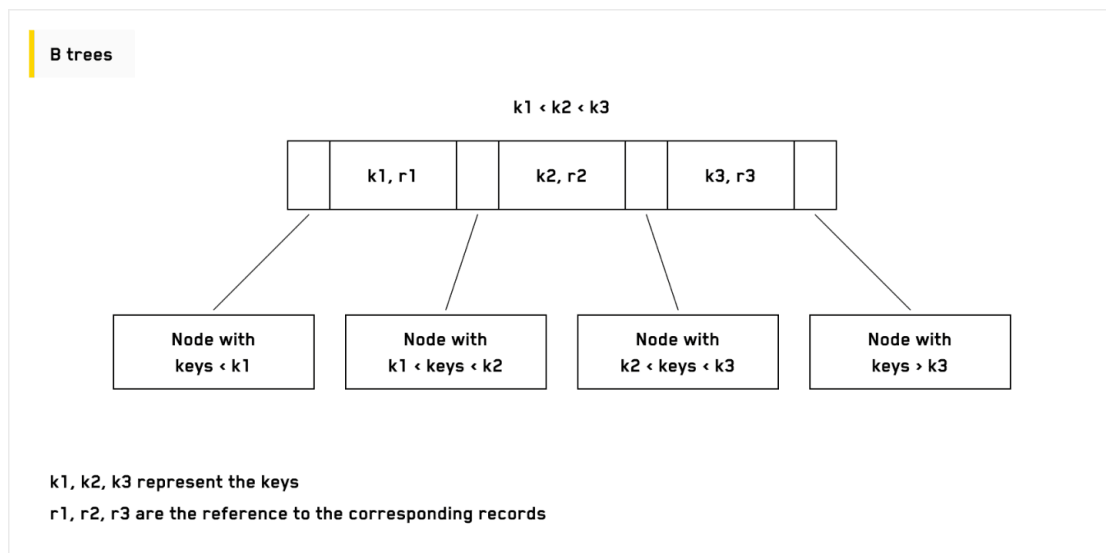
B-trees

B-trees are self-adjusting trees that can achieve multilevel indexing. They are a generalized form of Binary Search Trees. The data is stored in sorted order in the B-trees. B-tree achieves the efficient utilization of space in nodes, along with keeping the height of the tree small.

For a B-tree of order n , following are the properties of the B-tree:

1. Each node can have at most $n-1$ keys.
2. Each node can have a maximum of n children.
3. All the keys of the left subtrees are smaller than the keys of any subtree on the right,
4. Each internal node (non-leaf node) except the root node must have at least $\lceil n/2 \rceil$ children, where x represents the ceiling function.
5. All leaves are at the same depth.

Following is an example of a B-tree of order 4.



B+ trees

- B+ trees are an extension of B-trees.
- The major differences in the data structure are:

1. Only the leaf nodes store the record or reference to the record.
2. All the leaf nodes are connected to form a linked list. This enables sequential access along with direct access.

j. Failover

Good news! You were able to retire early because you made so much money from your parking lot business (see the previous section on “load balancers” for that story). You decide to charter a plane to your small vacation island. On the plane are your closest friends and family, along with your cat, and the pilot you hired to fly the plane. Everything was fine until a giant spider came crawling onto the engine controls. The pilot has a deep phobia of spiders, so he flees the cockpit and locks himself in the plane's bathroom.

Now you are in a situation where no one is piloting your plane! You need to quickly figure out which one of your friends and family has the best chance of taking over the job. This situation is an example of a failover plan. The leader—in this case the pilot—has failed, and you must find a new leader to take over.

When a leader node fails:

1. One of the follower nodes needs to be promoted to the leader.
2. Client node(s) must be reconfigured to send the write request to the new leader.
3. Other followers need to be reconfigured to consume data from the new leader.

For failover to be triggered, a prerequisite is that the leader's failures should be tracked. We can track it by sending some health-status pings to the nodes from time to time, and use the response time to determine the failure.

Replication

In our situation it would have been helpful to have a backup pilot (one who is not afraid of spiders) ready to take over. In system design we want to have servers ready to take over at a moment's notice when a leader fails. Take a look at the replication session (below) to learn we can handle failover with replication strategies.

Complications of Failover

Failover can lead to some tricky issues:

1. Failover leads to lost updates in the case of asynchronous replication when the leader goes down.
2. How to detect if the leader has gone down? Deciding the threshold for when to mark the leader as unavailable can be challenging. Sometimes it can happen that the traffic on the system is high, and that's why it is taking longer to respond. In such scenarios, if we bring down the leader, the system would be even more stressed.
3. The problem of having more than one leader.

It is important to ensure that once a failover is triggered, the old leader is aware of their changed role whenever it comes back up. It should pick up its new role as a follower, should stop accepting the write requests, and be configured to listen to the new leader.

k. Replication

Imagine that it is 480 BC, and you are the Spartan general Leonidas. You have been tasked with defending against an army of Persians who vastly outnumber your forces. You decide to make your stand at Thermopylae, where you hope the narrow passage and careful maneuvering will give your soldiers a chance against overwhelming odds.

You stand at a high point above the battlefield where you are able to see the battle from a bird's-eye view. You have messengers who will help you pass orders and give you key information that you need to make the correct decisions.

But what happens if a messenger gets injured or killed? Your orders will never arrive, and you'll surely lose the battle. To prevent this from happening, you create a system where a messenger will tell other messengers and replicate critical information so that there isn't a single point of failure.

Congratulations! You saved Sparta! You also understand at a high level why replication is important. Whether it's commanding thousands of soldiers or scaling software services to hundreds of millions, understanding the foundations of replication will make you ever more effective.

In system design, replication refers to making multiple copies of the same data on different machines.

Why replication?

Replication is done to achieve one or more of the following goals:

1. To avoid a single point of failure and increase availability when machines go down.
2. To better serve the global users by organizing copies by distinct geographical locations in order to serve users from copies that are close by.
3. To increase throughput. With more machines, more requests can be served.

Some key terms to understand for replication


Replica: Copy of data

Leader: Machine that handles write requests to the data store.

Followers: Machines that are replicas of the leader node, and cater to read requests.

Synchronous vs asynchronous replication

When a write request to a replica is marked as acknowledged, only then is it called synchronous replication. This means that the leader waits for an acknowledgment from all of the followers. In our example, Leonidas makes sure that his orders are understood by waiting until there is confirmation that all the messengers have heard the news.



Leonidas: messenger alpha squad are you here?

Messenger leader: yes general awaiting orders.


Leonidas: tell falcon cohort and lentil cohort to retreat to secondary defensive position. I'll wait for you to confirm that every messenger in your group understands the orders and knows where they are to go.

Messenger leader: Yes general. I will proceed immediately.


Leonidas literally freezes and does not nothing until messenger leader returns.

Messenger leader: Everyone go the orders and are on their way.

Leonidas satisfied moves to other pressing matters.




When the leader doesn't wait for the acknowledgment from the followers before marking the client's write requests as successful, it is called asynchronous replication. In our example, it would be as if Leonidas barks orders out, assumes they are heard, and moves on.



Leonidas: messenger alpha squad tell falcon cohort and lentil cohort to retreat to secondary defensive position.

Leonidas satisfied moves to other pressing matters.



Synchronous replication ensures that the information is replicated before moving on. This can be nice when it is vital that nothing is missed. The downside is that it slows down the stream of information being passed.

Remember

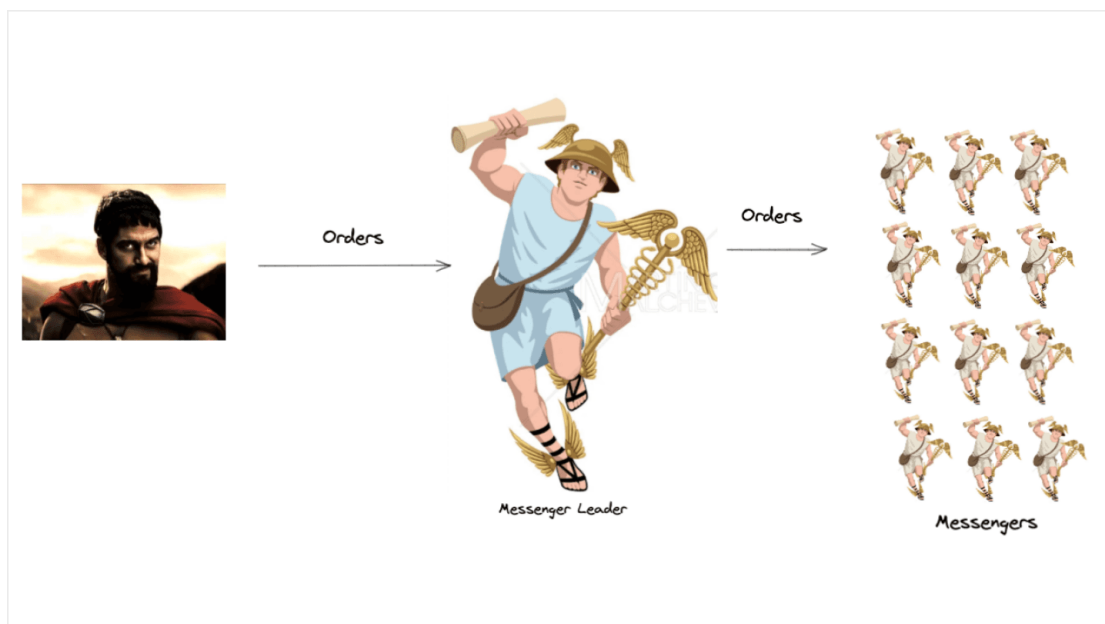
Sync replication ensures guaranteed delivery to all the followers, while async replication is less time-consuming for the client.

Sometimes in the database a semi-synchronous approach is taken, where only one follower is synchronously updated and the rest are asynchronous. When the former crashes, one of the latter is made a synchronous follower. This ensures that the up-to-date copy is at least available in two nodes and the client is also not kept waiting for long.

Most common types of Replication Systems

Single leader

This would be like if there was a single leader for all messengers who takes orders from Leonidas and then passes this information to all messengers.

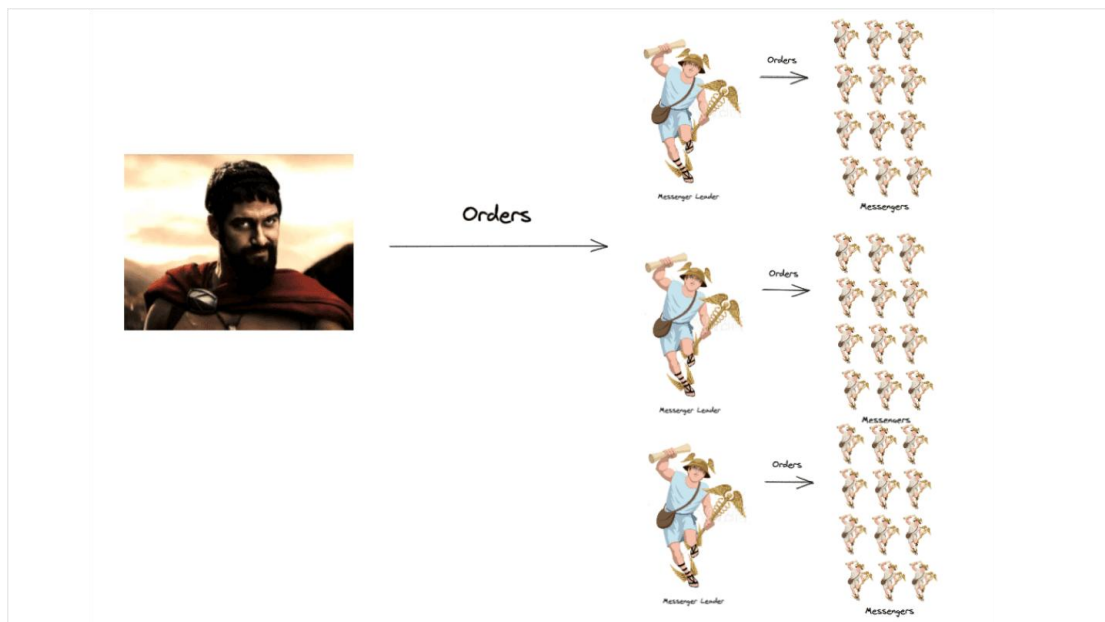


In system design, a single machine acts as a leader, and all write requests (or updates to the data store) go through that machine. All the other machines are used to cater to the read requests. This was previously known as “master-slave” replication, but it’s currently known as “primary-standby” or “active-passive” replication.

The leader also needs to pass down the information about all the writes to the follower nodes to keep them up to date. In case the leader goes down, one of the follower nodes (mostly with the most up-to-date data) is promoted to be the leader. This is called failover.

Multi leader

This would be like if there were multiple leaders who were responsible for their own messenger group who take orders from Leonidas and pass this information to all messengers.



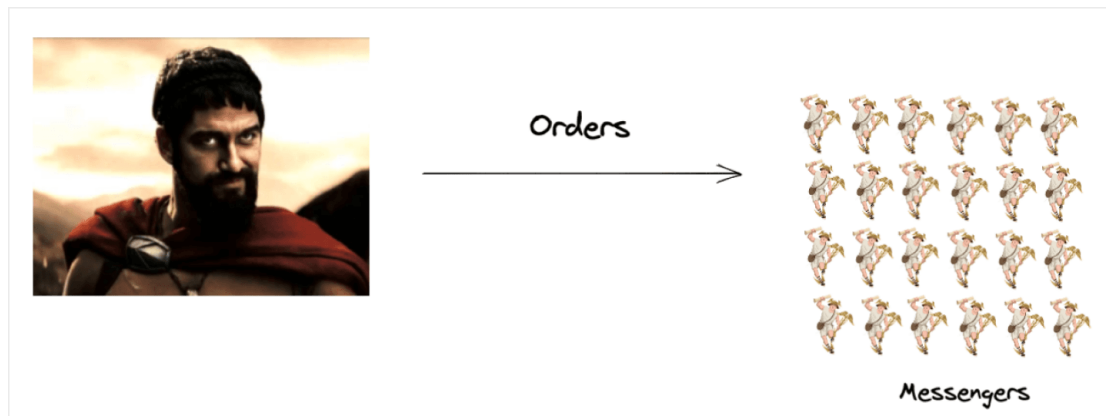
In system design, this means that more than one machine can take the write requests. This makes the system more reliable in case a leader goes down. This also means that every machine (including leaders) needs to catch up with the writes that happen over other machines.

Conflict resolution for concurrent writes:

1. Keeping the update with the largest client timestamp.
2. Sticky routing—writes from same client/index go to the same leader.
3. Keeping and returning all the updates.

Leaderless replication

In our analogy, leaderless replication would be like having messengers without group leaders who then go back and forth from the battlefield to Leonidas with information. Just before they deliver orders, they compare notes to see which messenger has the latest order.



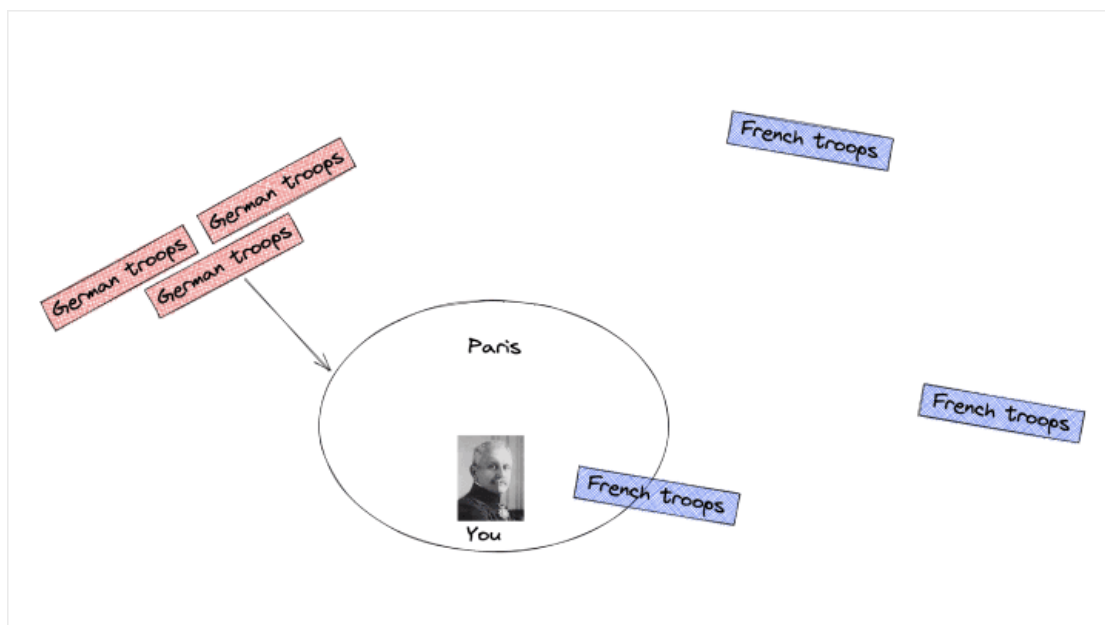
In such a system, all machines can cater to write and read requests. In some cases, the client directly writes to all the machines, and requests are read from all the machines based on quorum. Quorum refers to the minimum number of acknowledgements (for writes) and consistent data values (for reads) for the action to be valid. In other cases, the client request reaches the coordinator that broadcasts the request to all the nodes.

I. Consistent hashing

It is Sept. 6, 1914. You are General Manoury. You're in charge of the French army. You must defend against an oncoming German invasion led by Kaiser Wilhelm.



A messenger runs into the room and tells you some bleak news. A large German force is marching on Paris and is only a dozen miles away. You're outnumbered: you have some French troops in the city but most are farther away than the quickly approaching German soldiers. "Sacrebleu!" You exclaim in frustration. You must find a way to quickly get your forces back to the city and take up defensive positions before the invading Germans arrive.



You walk outside for a smoke break (like we said, you're French) and you happen to see a French taxi. Aha! You make an announcement asking all taxi drivers in the city to report to you immediately. You explain that they will be given assignments to go to various locations to pick up French troops and bring them back to the city. Of course, you give the drivers a choice whether or not they want to help, but you

emphasize: it is for France! As taxi drivers start to show up, you give them instructions on where to pick up French soldiers and where to drop them off afterwards.

While eating a croissant, you ask yourself the question of how to best organize your taxis. You know that there are 1000 taxis and 2000 troops to move. So, then the most efficient way to coordinate is for each taxi to pick up two soldiers and bring them back to the city. The issue is: you don't know how many total taxis are going to volunteer or when they will show up. You also don't know how many trips each driver is willing to make before they call it a day. So, you'll need some sort of system which is able to distribute the work evenly among all the taxis you know are available, but also accounts for new drivers signing up to help and existing drivers deciding they're done for the day.

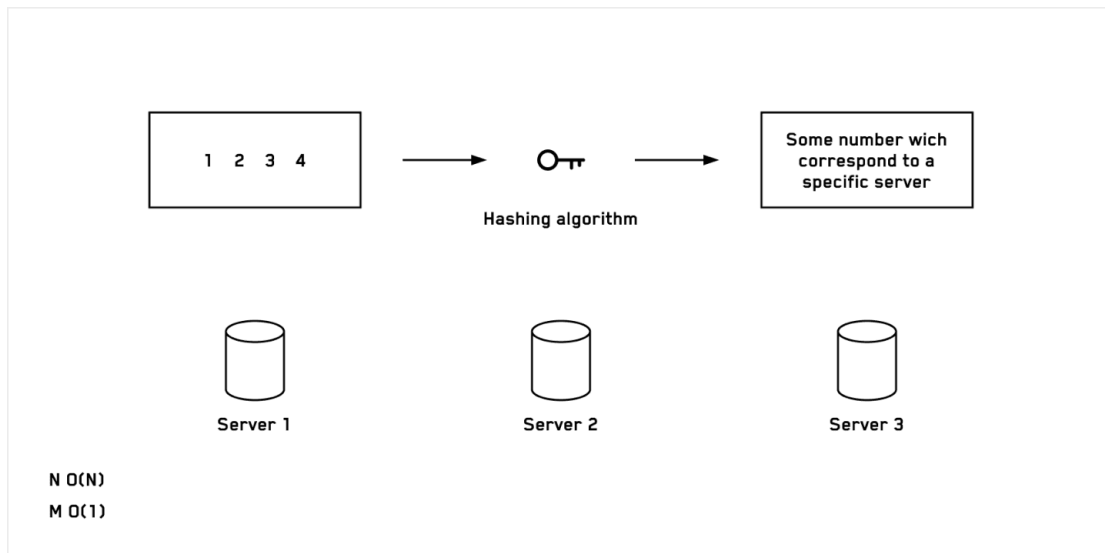
The situation above happened once during WWI, but a similar need for dynamic coordination happens everyday in system design interviews and real life software architecture. Servers are added and removed for a variety of unpredictable reasons, and we need a way to make sure we are evenly balancing the work given to each. One way of tackling this problem is a technique called “consistent hashing”.

What is consistent hashing?

Consistent hashing is a way to effectively distribute the keys in any distributed storage system—cache, database, or otherwise—to a large number of nodes or servers while allowing us to add or remove nodes without incurring a large performance hit.

What does an example look like?

Consider an example where we have a cache key we want to read or lookup—let's say it's 1234—and we have three cache servers that we'll number 0, 1, and 2. We could take the cache key, feed it to a hashing algorithm, and modulo the number it spits out by 3. Let's say the result is 1, so we'll assign this datum to server #1.

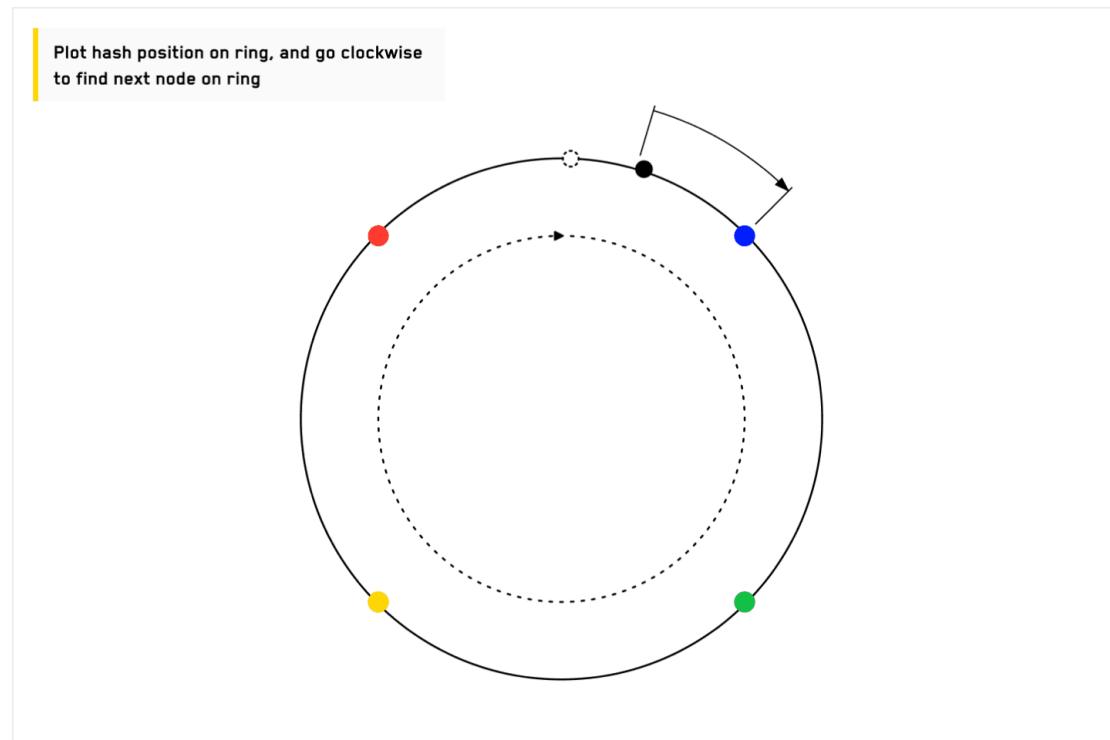


This only really works when we have a static number of servers. Because as soon as one of these gets added or removed, we have to change the number we're modulo-ing by. And now we're getting results for our node assignments that are no longer consistent with the results we got when there were only three nodes. This is why this solution is called "consistent hashing."

But in this naive approach we now have to take all the keys—let's say there are N keys and M nodes in our system—and relocate them to new nodes. That makes this an $O(N)$ operation, which is very expensive. This is particularly bad if your node just went down due to excessive traffic, because now you're trying to reallocate all of those keys on top of dealing with the elevated traffic. That's bad news.

A consistent hashing approach is more effective for these kinds of systems. Instead of having a fixed number of servers and having a modulo operations, we're going to map our hash results onto a ring. We can have whatever range we want here, but for now let's pretend that it goes from 0 to 100. Next, we'll assign each node to a point on this ring.

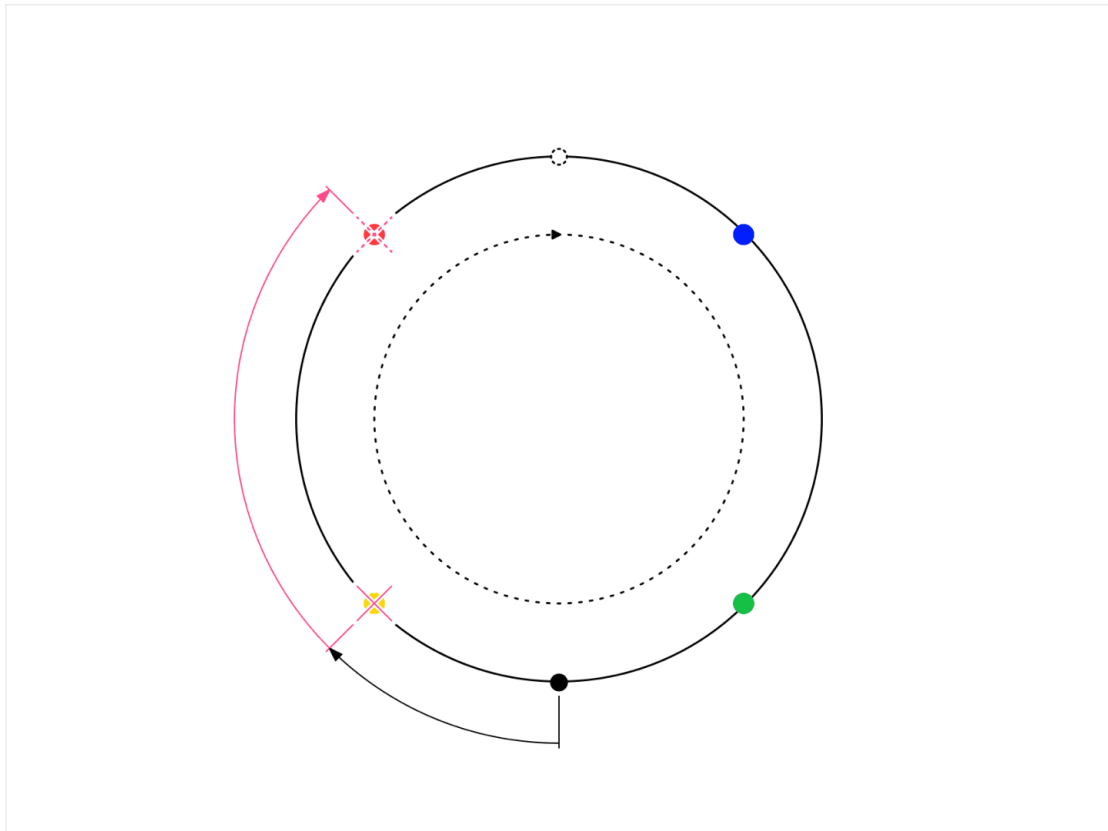
When we write or read to or from this cache, we plot the hash position on the ring and then go clockwise until we find the next node on the ring. That becomes the node that owns this key.



So now, if one of these goes away, we only have to reallocate the keys that were assigned to it specifically. They go to the next node on the ring.

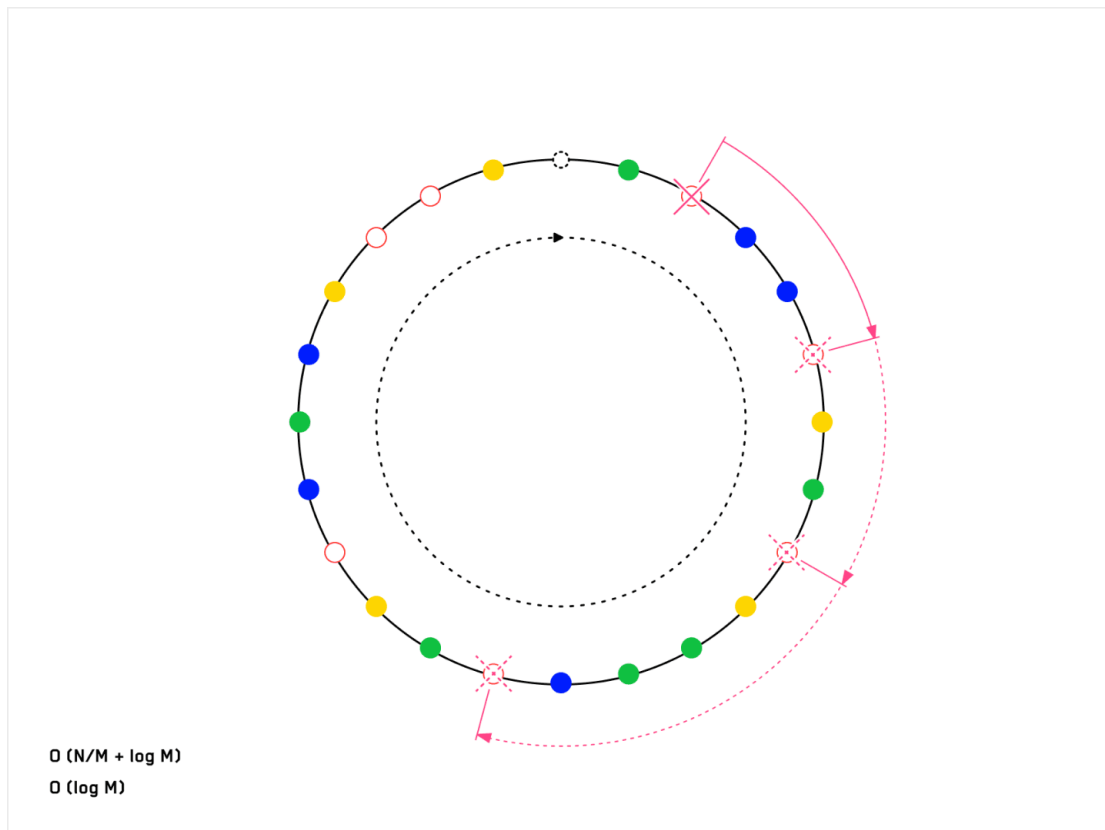
In this set up, we're only reallocating N/M keys. We also have to add a $\log M$ component to account for a binary search to find the next node on the ring, which gives us an overall $O(N/M + \log M)$ operation. Because M will be a much smaller number than N , this still gives us a huge performance improvement.

We still have a problem, though. If, say, this purple node goes down, we're going to reallocate all of its keys to the red node. So if purple went down due to high traffic, red is going to inherit all of that high traffic on top of the keys it already had.



So now it's likely that red will go down, and all those keys will move to blue, and so on until the entire system just collapses in a cascading failure.

To avoid this, instead of putting a single point on the ring for each node, we put a bunch of points on the ring for each node—we call these virtual nodes. As long as these virtual nodes are randomly distributed, when one node gets knocked offline or we add a new node, the reallocated keys should be more or less evenly distributed among the entire system.



Remember

In our original naive hashing model, it costs us $O(N)$ every time we need to add or remove a node, but inserting or removing a key is $O(1)$. In our consistent hashing model, we can change the number of nodes with only $O(N/M + \log M)$ runtime; however, inserting or looking up a cache key will now take $O(\log M)$ instead of $O(1)$ because we have to do a binary search to find the next node on the hash ring.

Part 2 Outro

Great work! You made it through the theory Achievement unlocked now you are ready for Parts 3 and 4, which are all about getting super practical. In the latter half of this guide we will integrate all the theories we've discussed. We'll also build some simple systems from scratch, show you how to get unstuck, and teach you a 3-step framework that will help you crush any system design interview.