



What Is a System Design Interview?

Learn about system design interviews (SDIs) and how to strategically approach them.

We'll cover the following



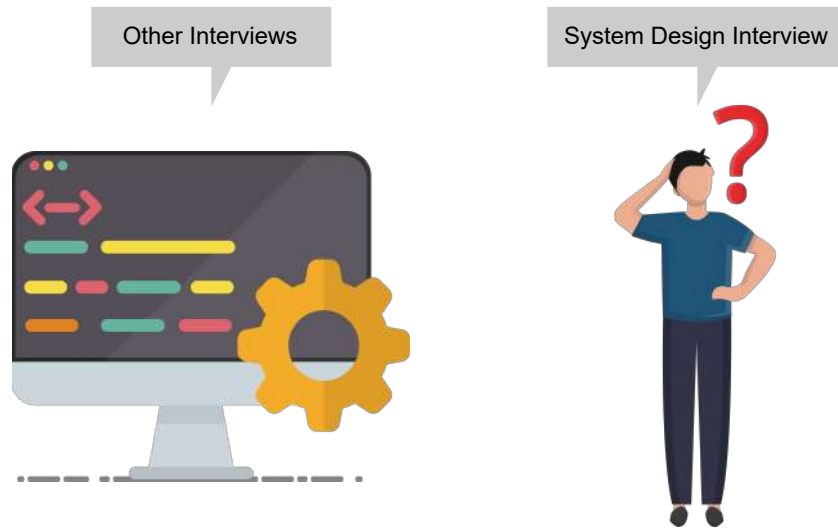
- How are SDIs different from other interviews?
- How do we tackle a design question?
 - Present the high-level design
- Possible questions for every SDI
 - The design evolution of Google
 - Design challenges
 - The responsibility of the designer
- Who gets a system design interview?
- Theory and practice

Our system design course is equally useful for people already working and those preparing for interviews. In this chapter, we highlight the different aspects of a system design interview (SDI) and some helpful tips for those who are preparing for an upcoming interview. We encourage learners to read this chapter even if they aren't preparing for an interview because some of the topics covered in this chapter can be applied broadly.

How are SDIs different from other interviews?

Just like with any other interview, we need to approach the systems design interviews strategically. SDIs are different from coding interviews. There's rarely any coding required in this interview.





Other interviews versus a systems design interview

An SDI takes place at a much higher level of abstraction. We figure out the requirements and map them on to the computational components and the high-level communication protocols that connect these subsystems.

The final answer doesn't matter. What matters is the process and the journey that a good applicant takes the interviewer through.

Note: As compared to coding problems in interviews, system design is more aligned with the tasks we'll complete on our jobs.

How do we tackle a design question?

Design questions are open ended, and they're intentionally vague to start



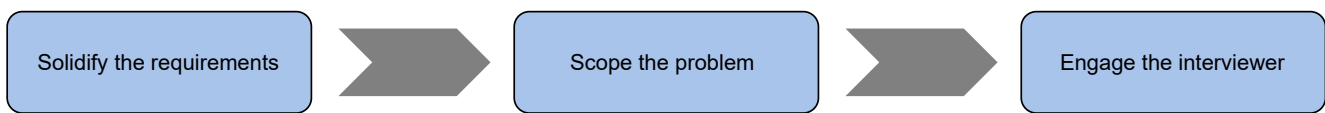
Interviewers often ask about a well-known problem,—for example, designing WhatsApp. Now, a real WhatsApp application has numerous features, and including all of them as requirements for our WhatsApp clone might not be a wise idea due to the following reasons:

- First, we'll have limited time during the interview.

- Second, working with some core functionalities of the system should be enough to exhibit our problem-solving skills.

We can tell the interviewer that there are many other things that a real WhatsApp does that we don't intend to include in our design. If the interviewer has any objections, we can change our plan of action accordingly.

Here are some best practices that we should follow during a system design interview:



Best practices for system design interview

- An applicant should ask the right questions to solidify the requirements.
- Applicants also need to scope the problem so that they're able to make a good attempt at solving it within the limited time frame of the interview. SDIs are usually about 35 to 40 minutes long.
- Communication with the interviewer is critical. It's not a good idea to silently work on the design. Instead, we should engage with the interviewer to ensure that they understand our thought process.

Present the high-level design

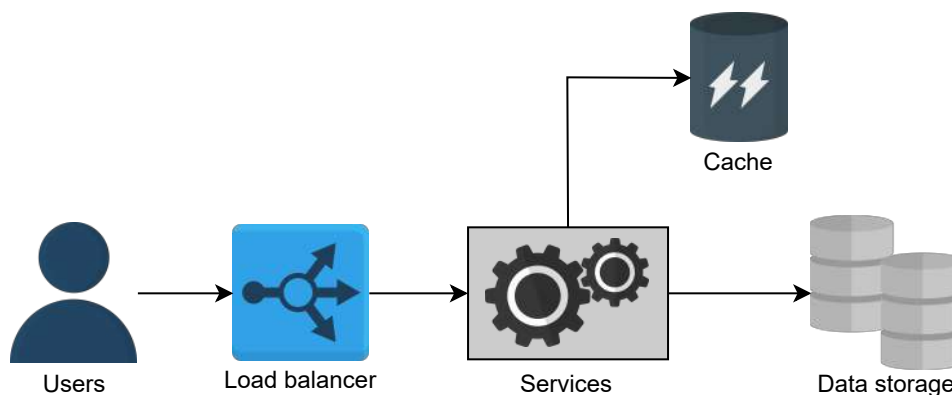
At a high level, components could be frontend, load balancers, caches, data processing, and so on. The system design tells us how these components fit together.

An architectural design often represents components as boxes. The arrows between these boxes represent who talks to whom and how the boxes or components fit together collectively.

?

Tt

☾



A sample design

We can draw a diagram like the one shown above for the given problem and present it to the interviewer.

Possible questions for every SDI

SDIs often include questions related to how a design might evolve over time as some aspect of the system increases by some order of magnitude—for example, the number of users, the number of queries per second, and so on. It's commonly believed in the systems community that when some aspect of the system increases by a factor of ten or more, the same design might not hold and might require change.

Designing and operating a bigger system requires careful thinking because designs often don't linearly scale with increasing demands on the system.

Another question in an SDI might be related to why we don't design a system that's already capable of handling more work than necessary or predicted.

💡 Show the answer

?

Tt

The design evolution of Google

☾

The design of the early version of Google Search may seem simplistic today, but it was quite sophisticated for its time. It also kept costs down, which was necessary for a startup like Google to stay afloat. The upshot is that whatever we do as designers have implications for the business and its customers. We need to meet or exceed customer needs by efficiently utilizing resources.

Design challenges

Things will change, and things will break over time because of the following:

- There's no single correct approach or solution to a design problem.
- A lot is predicated on the assumptions we make.

The responsibility of the designer

As designers, we need to provide fault tolerance at the design level because almost all modern systems use off-the-shelf components, and there are millions of such components. So, something will always be breaking down, and we need to hide this undesirable reality from our customers.

Who gets a system design interview?

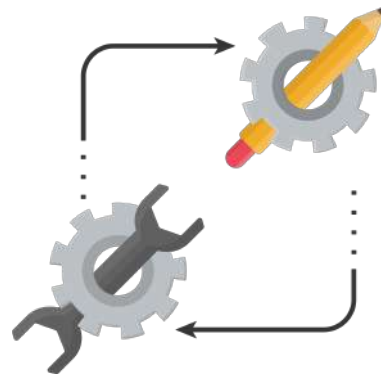
Traditionally, mid-to-senior level candidates with more than two years of experience get at least one system design interview. For more senior applicants, two or three system design interviews are common.

Recently, large companies have also put forth system design questions to some junior candidates. It's never too early to learn system design to grow or even expedite our careers.

Theory and practice



Most of the theory of system design comes from the domain of distributed systems. Getting a refresher on these concepts is highly recommended. Educative has an excellent [course on distributed systems](#) that we can use to refresh our knowledge of distributed systems concepts.



Distributed systems give us guideposts for mature software principles. These include the following:

- Robustness (the ability to maintain operations during a crisis)
- Scalability
- Availability
- Performance
- Extensibility
- Resiliency (the ability to return to normal operations over an acceptable period of time post disruption)

Such terminology also acts as a lingua franca between the interviewer and candidate.

As an example, we might say that we need to make a trade-off between availability and consistency when network components fail because the CAP theorem indicates that we can't have both under network partitions. Such common language helps with communication and shows that we're well versed in both theory and practice.

?

Tt



Remember: It's a candidate's job to exhibit their skills to the interviewer.

← Back

Next →





How to Prepare for Success

Get an overview of some interview preparation guidelines and materials.

We'll cover the following



- This course
- Technical blogs
- Ask why a system works
 - The right direction
- Mock interviews

Substantial preparation is necessary to increase our odds of getting the job we apply for.

Depending on a candidate's seniority and proficiency, different candidates need different times for interview preparation.

For an average candidate, three to four months might be required to prepare for a system design interview.



This course

This course helps readers learn or brush up on their system design skills. We've carefully curated some traditional and fresh design problems to cover the substantial depth and breadth of the system design.

The following activities might expedite the preparation and add variety and further depth to a candidate's knowledge.

Technical blogs

Many companies regularly publish the technical details of their significant work in the form of technical blogs.



Why are companies eager to share the technical details of their work?

💡 Show the reason

Note: There's a fine line between what can be held back by a company due to a competitive edge and what can be made public.

?

Tt

☾

We can study these blogs to get insight into what challenges or problems the company faced and what changes they made in the design to cope with these challenges.

Note: Staying informed about these innovations is important for professionals, and it's even more crucial for an applicant.

Some important technical blogs are [Engineering at Meta](#), [Meta Research](#), [AWS Architecture Blog](#), [Amazon Science Blog](#), [Netflix TechBlog](#), [Google Research](#), [Engineering at Quora](#), [Uber Engineering Blog](#), [Databricks Blog](#), [Pinterest Engineering](#), [BlackRock Engineering](#), [Lyft Engineering](#), and [Salesforce Engineering](#).

Note: We should always take non-peer-reviewed material with a grain of salt. Think about what blogs say critically and with technical acumen to decide if what they say makes sense or not. If it doesn't make sense, that could be an excellent opportunity to discuss the issue with peers.

Ask why a system works

By asking themselves the right questions, candidates can think through dense and ambiguous situations.



- Learn how popular applications work at a high level—for example, Instagram, Twitter, and so on.
- Start to understand and ask why some component was used instead of another—for example, Firebase versus SQL.
- Build serious side projects. Start with a simple product and then improve and refine it.
- Build a system from scratch, and get familiar with all the processes and details of its construction.

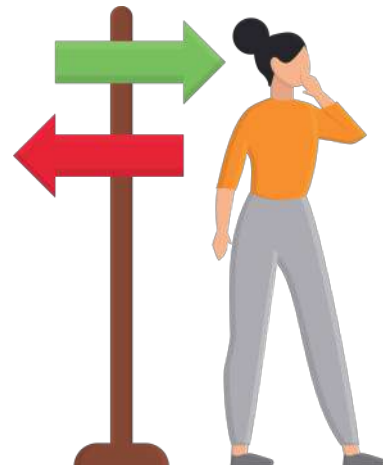


We can clone a popular application without tutorials.

The right direction

System design deals with components at a higher level, and we need to avoid going into the trenches.

We should focus less on mechanics and more on trade-offs.



For example, discussing whether to use Room library instead of raw SQLite isn't helpful because Room library is a mere wrapper around SQLite. A better discussion might be about using traditional databases like MySQL or using NoSQL stores like MongoDB. This second discussion will help us pinpoint trade-offs between the two.

?

T

C

We should start with high-level stuff because the low-level details will automatically come up.

Mock interviews

Mock interviews are a great way to prepare for system design interviews. They involve pairing up with a friend and allowing them to ask questions. If it's not possible to use a friend, another strategy is to record ourselves and play the role of both interviewer and interviewee. With this approach, we can critique ourselves or ask a knowledgeable friend for feedback.



Note: Since no mock interview can fully mimic an actual interview, it's a good idea to take real interviews in companies. Once we've been through an interview or two, we'll be better able to evaluate what went right and what didn't.

[< Back](#)[Next >](#)



How to Perform Well

Learn some helpful tips on how to perform during a system design interview.

We'll cover the following



- What to do during the interview
 - Strategize, then divide and conquer
 - Ask refining questions
 - Handle data
 - Discuss the components
 - Discuss trade-offs
- What not to do in an interview

What to do during the interview

We stress that a candidate should make an effort to avoid looking unoriginal. The interviewer has probably asked the same question to many candidates. Reproducing a run-of-the-mill design might not impress the interviewer.

At the same time, an interview can be a stressful situation. Having a plan to attack the problem might be a good strategy. Depending on the candidate, there can be multiple strategies to



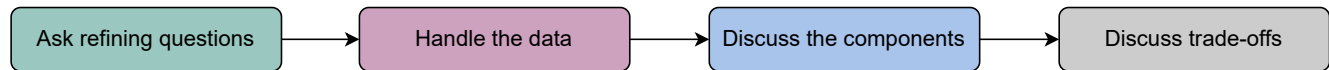
The dos of a system design interview



attack a design problem. We suggest the following technique.

Strategize, then divide and conquer

We recommend including the following activities somewhere in the interview:



Activities to include in the interview

Ask refining questions

We need to understand the design problem and its requirements. To do this, we can put on our product manager hat and prioritize the main features by asking the interviewer refining questions. The idea is to go on a journey with the interviewer about why our design is good. These interviews are designed to gauge if we're able to logically derive a system out of vague requirements.



We should ensure that we're solving the right problem. Often, it helps to divide the requirements into two groups:

- Requirements that the clients need directly—for example, the ability to send messages in near real-time to friends.
- Requirements that are needed indirectly—for example, messaging service performance shouldn't degrade with increasing user load.

?

Note: Professionals call these functional and nonfunctional requirements.

Tt



Handle data

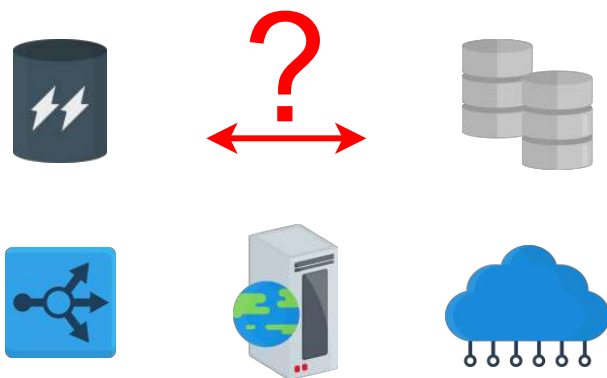
We need to identify and understand data and its characteristics in order to look for appropriate data storage systems and data processing components for the system design.



Some important questions to ask ourselves when searching for the right systems and components include the following:

- What's the size of the data right now?
- At what rate is the data expected to grow over time?
- How will the data be consumed by other subsystems or end users?
- Is the data read-heavy or write-heavy?
- Do we need strict consistency of data, or will eventual consistency work?
- What's the durability target of the data?
- What privacy and regulatory requirements do we require for storing or transmitting user data?

Discuss the components



Components

At some level, our job might be perceived as figuring out which components we'll use, where they'll be placed, and how they'll interact with each other.

An example could be the type of database—will a conventional

?

Tt

☾

database work, or should we use a

NoSQL database?

There might be cases where we have strong arguments to use NoSQL databases, but our interviewer may insist that we use a traditional database. What should we do in such a case?

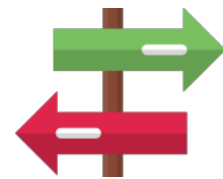
 Show the solution

Note: We often abstract away the details of the components as boxes and use arrows to show the interactions between them. It might help to define the user-facing APIs at a high level to further understand system data and interaction requirements.

Front-end components, load balancers, caches, databases, firewalls, and CDNs are just some examples of system components.

Discuss trade-offs

Remember that there's no one correct answer to a design problem. If we give the same problem to two different groups, they might come up with different designs.



These are some of the reasons why such diversity exists in design solutions:

- Different components have different pros and cons. We'll need to carefully weigh what works for us. ?
- Different choices have different costs in terms of money and technical complexity. We need to efficiently utilize our resources. Tt



- Every design has its weaknesses. As designers, we should be aware of all of them, and we should have a follow-up plan to tackle them.

We should point out weaknesses in our design to our interviewer and explain why we haven't tackled them yet. An example could be that our current design can't handle ten times more load, but we don't expect our system to reach that level anytime soon. We have a monitoring system to keep a very close eye on load growth over time so that a new design can be implemented in time. This is an example where we intentionally had a weakness to reduce system cost.

Something is always failing in a big system. We need to integrate fault tolerance and security into our design.

What not to do in an interview

Here are a few things that we should avoid doing in a system design interview:

- Don't write code in a system design interview.
- Don't start building without a plan.
- Don't work in silence.
- Don't describe numbers without reason. We have to frame it.
- If we don't know something, we don't paper over it, and we don't pretend to know it.



The don'ts of a system design interview

Note: If an interviewer asks us to design a system we haven't heard of, we should just be honest and tell them so. The interviewer will either explain it to us or they might change the question.

?

Tt



 **Back**

How to Prepare for Success

Next 

Why Are Abstractions Important?

 Mark as Completed





Availability

Learn about availability, how to measure it, and its importance.



- What is availability?
 - Measuring availability
 - Availability and service providers

What is availability?

Availability is the percentage of time that some service or infrastructure is accessible to clients and is operated upon under normal conditions. For example, if a service has 100% availability, it means that the said service functions and responds as intended (operates normally) all the time.

Measuring availability

Mathematically, availability, **A**, is a ratio. The higher the **A** value, the better.

We can also write this up as a formula:

$$A \text{ (in percent)} = \frac{(Total \ Time - Amount \ Of \ Time \ Service \ Was \ Down)}{Total \ Time} * 100$$

We measure availability as a number of nines. The following table shows how much downtime is permitted when we're using a given number of nines.



The Nines of Availability

Availability Percentages versus Service Downtime			
Availability %	Downtime per Year	Downtime per Month	Downtime per Week
90% (1 nine)	36.5 days	72 hours	16.8 hours
99% (2 nines)	3.65 days	7.20 hours	1.68 hours
99.5% (2 nines)	1.83 days	3.60 hours	50.4 minutes
99.9% (3 nines)	8.76 hours	43.8 minutes	10.1 minutes
99.99% (4 nines)	52.56 minutes	4.32 minutes	1.01 minutes
99.999% (5 nines)	5.26 minutes	25.9 seconds	6.05 seconds
99.9999% (6 nines)	31.5 seconds	2.59 seconds	0.605 seconds
99.99999% (7 nines)	3.15 seconds	0.259 seconds	0.0605 seconds



Availability and service providers

Each service provider may start measuring availability at different points in time. Some cloud providers start measuring it when they first offer the service, while some measure it for specific clients when they start using the service. Some providers might not reduce their reported availability numbers if their service was not down for all the clients. The planned downtimes are excluded. Downtime due to cyberattacks might not be incorporated into the calculation of availability. Therefore, we should carefully understand how a specific provider calculates their availability numbers.

← Back

The Spectrum of Failure Models

Next -

Reliability

☒ Mark as Completed



[← Back To Course Home](#)

Grokking Modern System Design Interview for Engineers & Managers

16% completed



Introduction



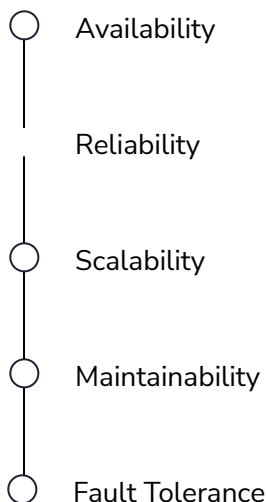
System Design Interviews



Abstractions



Non-functional System Characteristics



Reliability

Learn about reliability, how to measure it, and its importance.

We'll cover the following



- What is reliability?
- Reliability and availability

What is reliability?

Reliability, R , is the probability that the service will perform its functions for a specified time. R measures how the service performs under varying operating conditions.

We often use **mean time between failures (MTBF)** and **mean time to repair (MTTR)** as metrics to measure R .

$$MTBF = \frac{\text{Total Elapsed Time} - \text{Sum of Downtime}}{\text{Total Number of Failures}}$$

?

$$MTTR = \frac{\text{Total Maintenance Time}}{\text{Total Number of Repairs}}$$

T_r

(We strive for a higher MTBF value and a lower MTTR value.)

Reliability and availability

Reliability and availability are two important metrics to measure compliance of service to agreed-upon service level objectives (SLO).

The measurement of availability is driven by time loss, whereas the frequency and impact of failures drive the measure of reliability. Availability and reliability are essential because they enable the stakeholders to assess the health of the service.

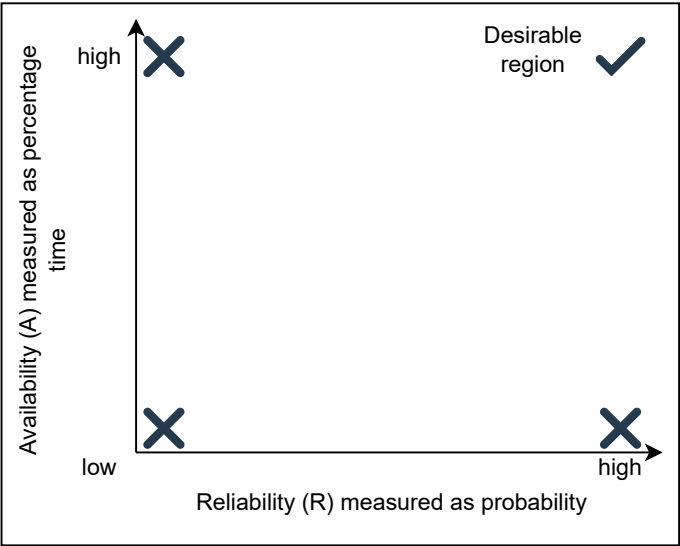
Reliability (**R**) and availability (**A**) are two distinct concepts, but they are related. Mathematically, **A** is a function of **R**. This means that the value of **R** can change independently, and the value of **A** depends on **R**. Therefore, it's possible to have situations where we have:

- low **A**, low **R**
- low **A**, high **R**
- high **A**, low **R**
- high **A**, high **R** (desirable)

?

Tt





Availability as a function of reliability

← Back

Availability

Next →

Scalability

☒ Mark as Completed





Scalability

Learn about scalability and its importance in system design.

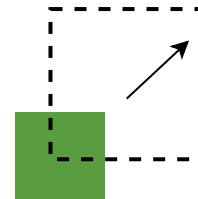
We'll cover the following



- What is scalability?
- Dimensions
- Different approaches of scalability
 - Vertical scalability—scaling up
 - Horizontal scalability—scaling out

What is scalability?

Scalability is the ability of a system to handle an increasing amount of workload without compromising performance. A search engine, for example, must accommodate increasing numbers of users, as well as the amount of data it indexes.



The workload can be of different types, including the following:

- **Request workload:** This is the number of requests served by the system.
- **Data/storage workload:** This is the amount of data stored by the system.



Dimensions

Here are the different dimensions of scalability:

- **Size scalability:** A system is scalable in size if we can simply add additional users and resources to it.
- **Administrative scalability:** This is the capacity for a growing number of organizations or users to share a single distributed system with ease.
- **Geographical scalability:** This relates to how easily the program can cater to other regions while maintaining acceptable performance constraints. In other words, the system can readily service a broad geographical region, as well as a smaller one.

Different approaches of scalability

Here are the different ways to implement scalability.

Vertical scalability—scaling up

Vertical scaling, also known as “**scaling up**,” refers to scaling by providing additional capabilities (for example, additional CPUs or RAM) to an existing device. Vertical scaling allows us to expand our present hardware or software capacity, but we can only grow it to the limitations of our server. The dollar cost of vertical scaling is usually high because we might need exotic components to scale up.

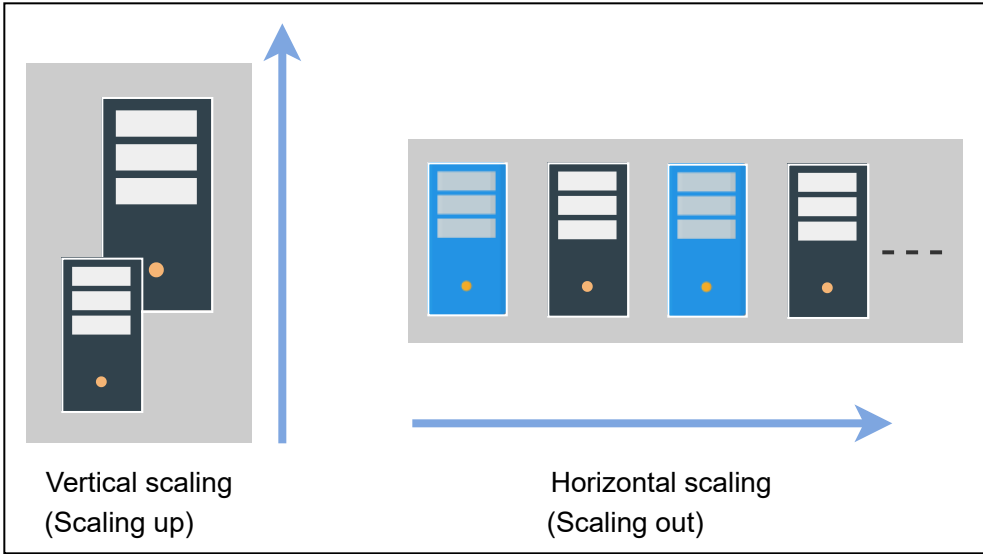
Horizontal scalability—scaling out

Horizontal scaling, also known as “**scaling out**,” refers to increasing the number of machines in the network. We use commodity nodes for this purpose because of their attractive dollar-cost benefits. The catch here is that we need to build a system such that many nodes could collectively work as if we had a single, huge server.

?

Tt





Vertical scaling versus horizontal scaling

← Back

Reliability

Next →

Maintainability

☒ Mark as Completed





Fault Tolerance

Learn about fault tolerance, how to measure it, and its importance.

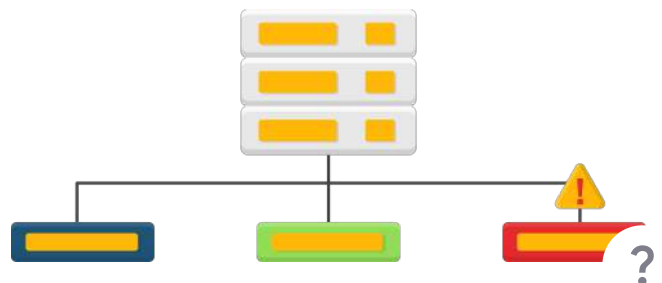
We'll cover the following ^

- What is fault tolerance?
 - Fault tolerance techniques
 - Replication
 - Checkpointing

What is fault tolerance?

Real-world, large-scale applications run hundreds of servers and databases to accommodate billions of users' requests and store significant data. These applications need a mechanism that helps with data safety and eschews the recalculation of computationally intensive tasks by avoiding a single point of failure.

Fault tolerance refers to a system's ability to execute persistently even if one or more of its components fail. Here, components can be software or hardware. Conceiving a system that is hundred percent fault-tolerant is practically very difficult.



Tt



Let's discuss some important features for which fault-tolerance becomes a necessity.

Availability focuses on receiving every client's request by being accessible 24/7.

Reliability is concerned with responding by taking specified action on every client's request.

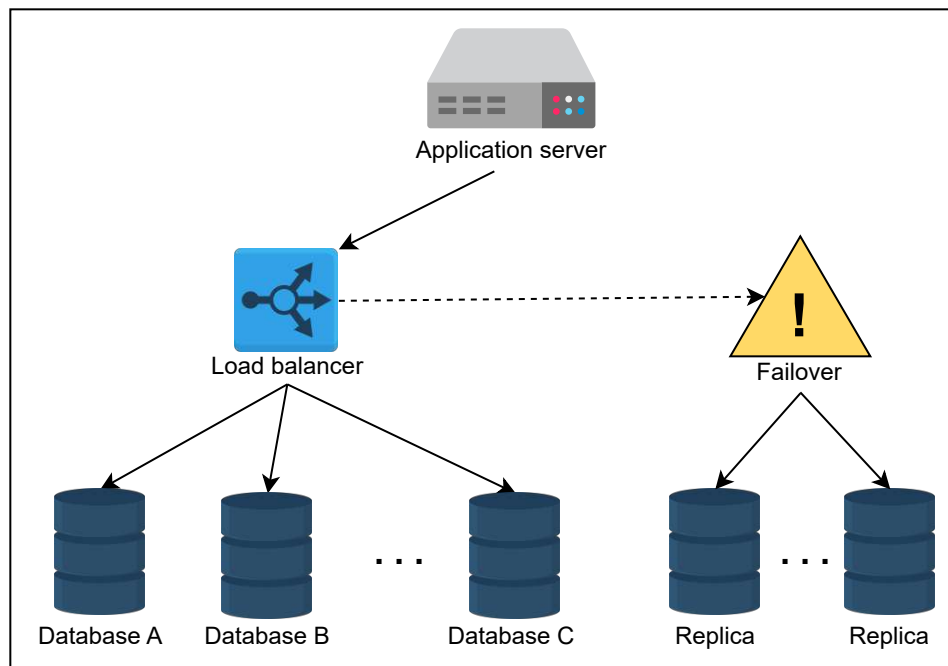
Fault tolerance techniques

Failure occurs at the hardware or software level, which eventually affects the data. Fault tolerance can be achieved by many approaches, considering the system structure. Let's discuss the techniques that are significant and suitable for most designs.

Replication

One of the most widely-used techniques is **replication-based fault tolerance**. With this technique, we can replicate both the services and data. We can swap out failed nodes with healthy ones and a failed data store with its replica. A large service can transparently make the switch without impacting the end customers.

We create multiple copies of our data in separate storage. All copies need to update regularly for consistency when any update occurs in the data. Updating data in replicas is a challenging job. When a system needs strong consistency, we can synchronously update data in replicas. However, this reduces the availability of the system. We can also asynchronously update data in replicas when we can tolerate eventual consistency, resulting in stale reads until all replicas converge. Thus, there is a trade-off between both consistency approaches. We compromise either on availability or on consistency under failures—a reality that is outlined in the [CAP theorem](#).



Replication-based fault tolerance

Checkpointing

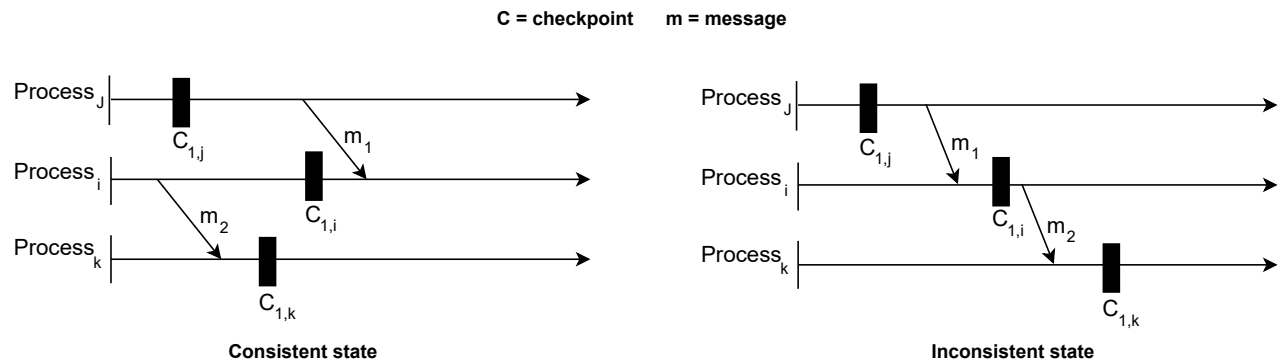
Checkpointing is a technique that saves the system's state in stable storage when the system state is consistent. Checkpointing is performed in many stages at different time intervals. The primary purpose is to save the computational state at a given point. When a failure occurs in the system, we can get the last computed data from the previous checkpoint and start working from there.

Checkpointing also comes with the same problem that we have in replication. When the system has to perform checkpointing, it makes sure that the system is in a consistent state, meaning that all processes are stopped except read processes that do not change the state of the system. This type of checkpointing is known as **synchronous checkpointing**. On the other hand, checkpointing in an inconsistent state leads to data inconsistency problems. Let's look at the illustration below to understand the difference between a consistent and an inconsistent state:

?

T





Checkpointing in a consistent and inconsistent state

Consistent state: The illustration above shows no communication among the processes when the system performs checkpointing. All the processes are sending or receiving messages before and after checkpointing. This state of the system is called a consistent state.

Inconsistent state: The illustration also displays that processes communicate through messages when the system performs checkpointing. This indicates an inconsistent state, because when we get a previously saved checkpoint, *Process i* will have a message (m_1) and *Process j* will have no record of message sending.

[< Back](#)
[Next >](#)

Maintainability

Put Back-of-the-envelope Numbers in Persp...

☒ Mark as Completed

?

Tt





Introduction to Domain Name System (DNS)

Learn how domain names get translated to IP addresses through DNS.

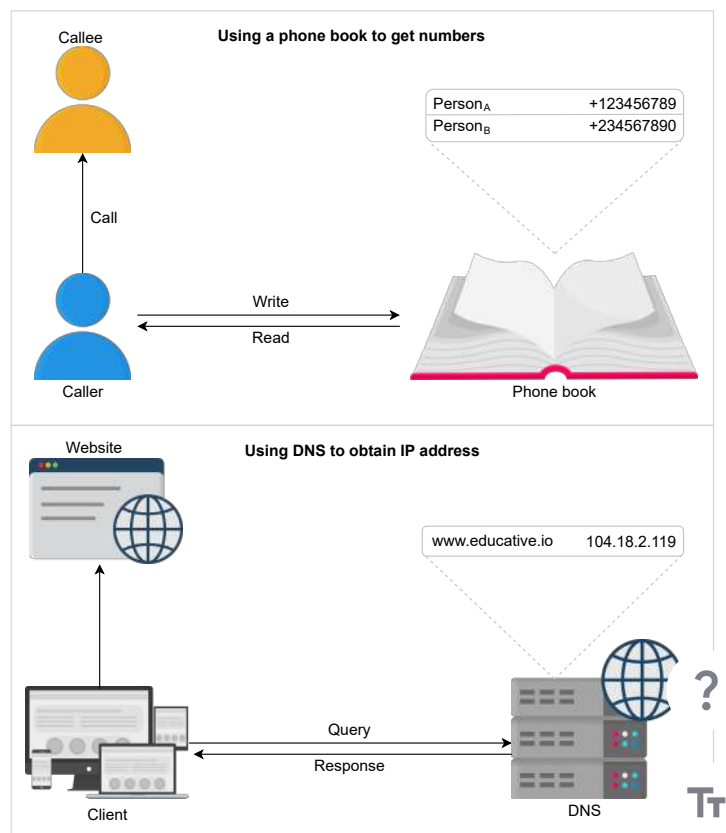
We'll cover the following ^

- The origins of DNS
- What is DNS?
- Important details

The origins of DNS

Let's consider the example of a mobile phone where a unique number is associated with each user. To make calls to friends, we can initially try to memorize some of the phone numbers. However, as the number of contacts grows, we'll have to use a phone book to keep track of all our contacts. This way, whenever we need to make a call, we'll refer to the phone book and dial the number we need.

Similarly, computers are uniquely identified by IP



Using a phone book analogy to understand domain name system (DNS)



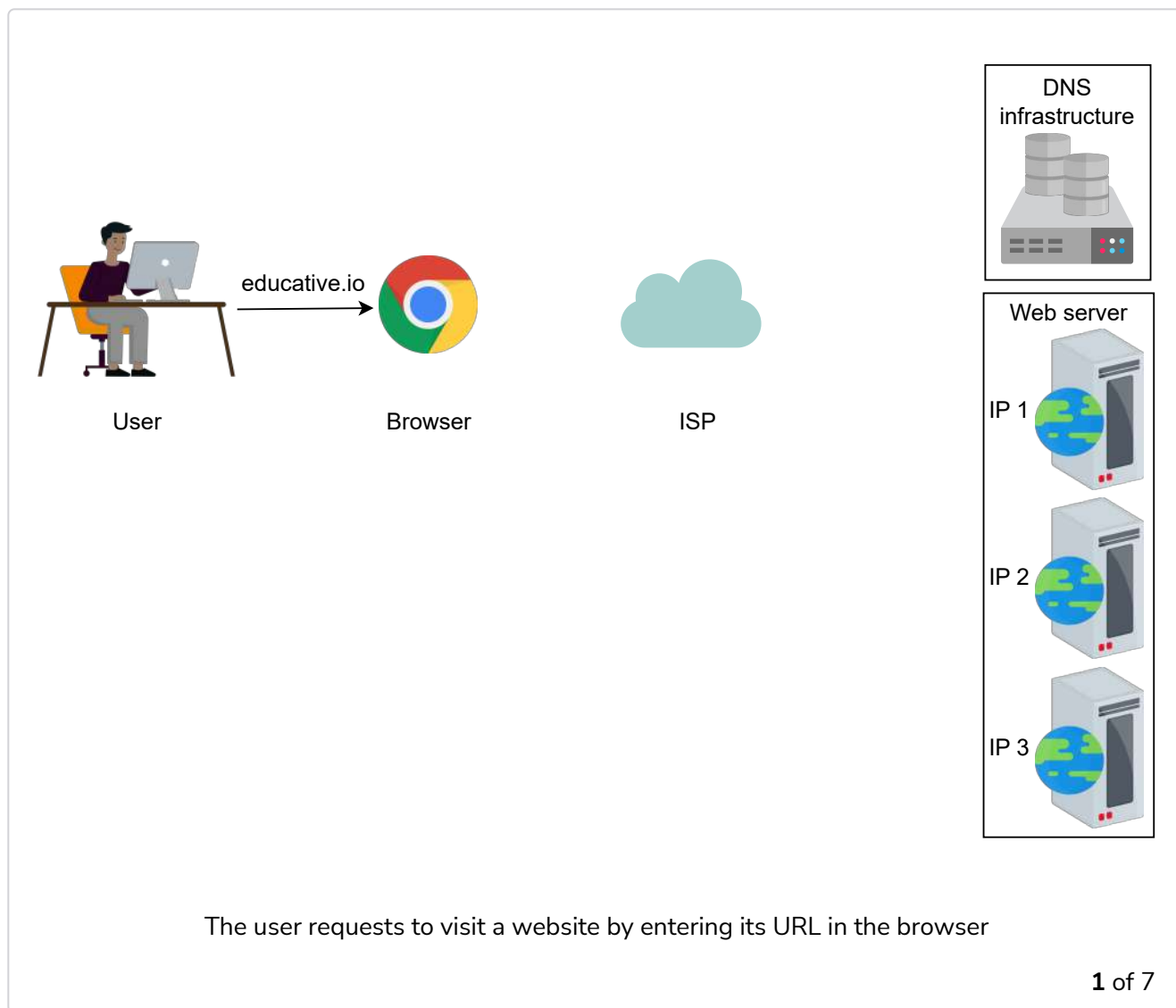
addresses—for example, **104.18.2.119** is an IP address. We use IP addresses to visit a website hosted on a machine. Since humans cannot easily remember IP addresses to visit domain names (an example domain name being **educative.io**), we need a phone book-like repository that can maintain all mappings of domain names to IP addresses. In this chapter, we'll see how DNS serves as the Internet's phone book.

What is DNS?

The **domain name system (DNS)** is the Internet's naming service that maps human-friendly domain names to machine-readable IP addresses. The service of DNS is transparent to users. When a user enters a domain name in the browser, the browser has to translate the domain name to IP address by asking the DNS infrastructure. Once the desired IP address is obtained, the user's request is forwarded to the destination web server.

The slides below show the high-level flow of the working of DNS:





The entire operation is performed very quickly. Therefore, the end user experiences minimum delay. We'll also see how browsers save some of the frequently used mappings for later use in the next lesson.

Important details

?

Let's highlight some of the important details about DNS, some of which we'll cover in the next lesson:

Tt



- **Name servers:** It's important to understand that the DNS isn't a single server. It's a complete infrastructure with numerous servers. DNS servers that respond to users' queries are called **name servers**.
- **Resource records:** The DNS database stores domain name to IP address mappings in the form of resource records (RR). The RR is the smallest unit of information that users request from the name servers. There are different types of RRs. The table below describes common RRs. The three important pieces of information are *type*, *name*, and *value*. The *name* and *value* change depending upon the *type* of the RR.

Common Types of Resource Records

Type	Description	Name	Value	Example
A	Provides the hostname to IP address mapping	Hostname	IP address	relay.iana.org
NS	Provides the hostname that is the authoritative DNS for a domain name	Domain name	Hostname	(NS, authoritative)
CNAME	Provides the mapping from alias to canonical hostname	Hostname	Canonical name	stale.example.com
MX	Provides the mapping of mail server from alias to canonical hostname	Hostname	Canonical name	mail.example.com

- **Caching:** DNS uses caching at different layers to reduce request latency for the user. Caching plays an important role in reducing the burden on DNS infrastructure because it has to cater to the queries of the entire Internet.
- **Hierarchy:** DNS name servers are in a hierarchical form. The hierarchical structure allows DNS to be highly scalable because of its

increasing size and query load. In the next lesson, we'll look at how a tree-like structure is used to manage the entire DNS database.

Let's explore more details of the above points in the next lesson to get more clarity.

[← Back](#)[Next →](#)



How the Domain Name System Works

Understand the detailed working of the domain name system.

We'll cover the following



- DNS hierarchy
 - Iterative versus recursive query resolution
- Caching
- DNS as a distributed system
 - Highly scalable
 - Reliable
 - Consistent
- Test it out
 - The nslookup output
 - The dig output

Through this lesson, we'll answer the following questions:

- How is the DNS hierarchy formed using various types of DNS name servers?
- How is caching performed at different levels of the Internet to reduce the querying burden over the DNS infrastructure?
- How does the distributed nature of the DNS infrastructure help its robustness?



Let's get started.

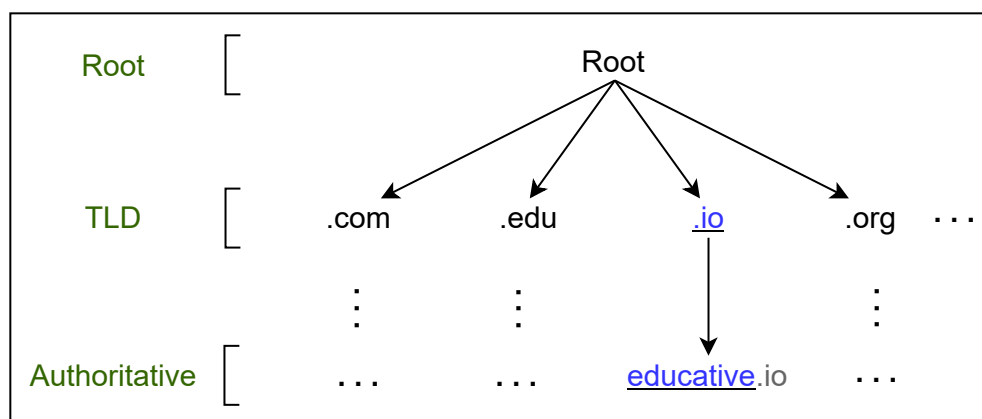


DNS hierarchy

As stated before, the DNS isn't a single server that accepts requests and responds to user queries. It's a complete infrastructure with name servers at different hierarchies.

There are mainly four types of servers in the DNS hierarchy:

1. **DNS resolver:** Resolvers initiate the querying sequence and forward requests to the other DNS name servers. Typically, DNS resolvers lie within the premise of the user's network. However, DNS resolvers can also cater to users' DNS queries through caching techniques, as we will see shortly. These servers can also be called local or default servers.
2. **Root-level name servers:** These servers receive requests from local servers. Root name servers maintain name servers based on top-level domain names, such as **.com**, **.edu**, **.us**, and so on. For instance, when a user requests the IP address of **educative.io**, root-level name servers will return a list of top-level domain (TLD) servers that hold the IP addresses of the **.io** domain.
3. **Top-level domain (TLD) name servers:** These servers hold the IP addresses of authoritative name servers. The querying party will get a list of IP addresses that belong to the authoritative servers of the organization.
4. **Authoritative name servers:** These are the organization's DNS name servers that provide the IP addresses of the web or application servers.



DNS hierarchy for resolution of domain/host names

Point to Ponder

Question

How are DNS names processed? For example, will [educative.io](#) be processed from left to right or right to left?

[Show Answer](#) ▼

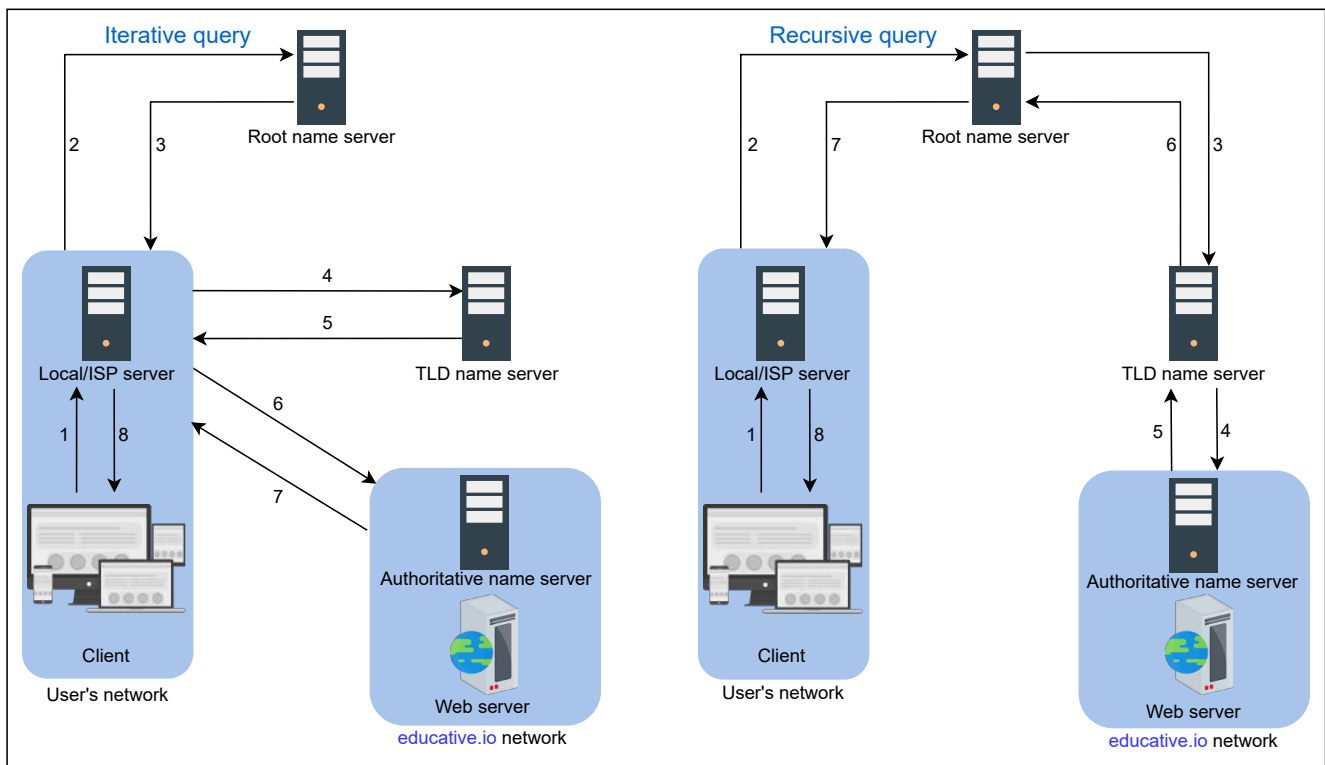
Iterative versus recursive query resolution

There are two ways to perform a DNS query:

1. **Iterative:** The local server requests the root, TLD, and the authoritative servers for the IP address.
2. **Recursive:** The end user requests the local server. The local server further requests the root DNS name servers. The root name servers forward the requests to other name servers.

In the following illustration (on the left), DNS query resolution is iterative from the perspective of the local/ISP server:





Iterative versus recursive query

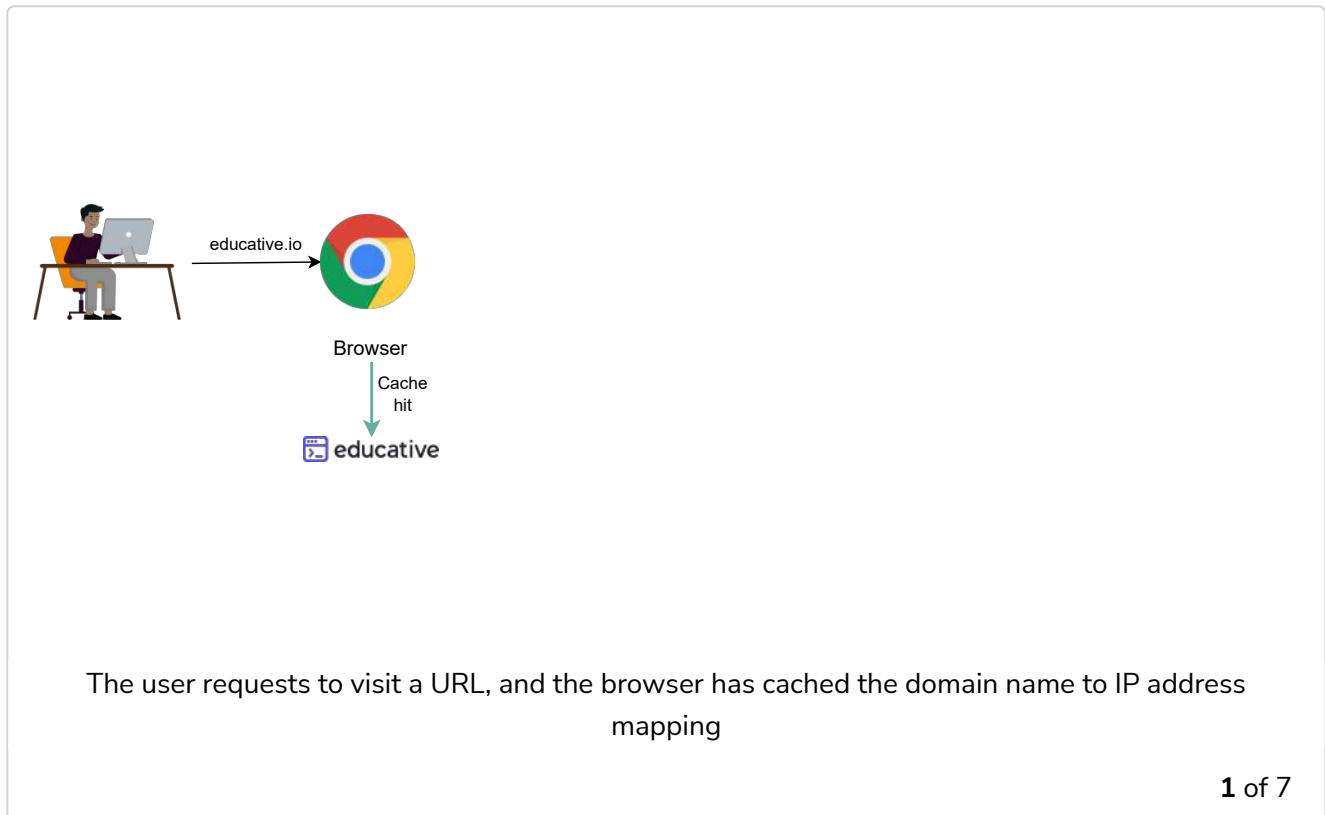
Note: Typically, an iterative query is preferred to reduce query load on DNS infrastructure.

💡 Fun Fact

Caching

Caching refers to the temporary storage of frequently requested **resource records**. A **record** is a data unit within the DNS database that shows a name-to-value binding. Caching reduces response time to the user and decreases network traffic. When we use caching at different hierarchies, it can reduce a lot of querying burden on the DNS infrastructure. Caching can be implemented in the browser, operating systems, local name server within the user's network, or the ISP's DNS resolvers.

The slideshow below demonstrates the power of caching in the DNS:



Note: Even if there is no cache available to resolve a user's query and it's imperative to visit the DNS infrastructure, caching can still be beneficial. The local server or ISP DNS resolver can cache the IP addresses of TLD servers or authoritative servers and avoid requesting the root-level server.

DNS as a distributed system

Although the DNS hierarchy facilitates the distributed Internet that we know today, it's a distributed system itself. The distributed nature of DNS has the following advantages:

- It avoids becoming a single point of failure (SPOF).

- It achieves low query latency so users can get responses from nearby servers.
- It gets a higher degree of flexibility during maintenance and updates or upgrades. For example, if one DNS server is down or overburdened, another DNS server can respond to user queries.

There are 13 logical root name servers (named letter **A** through **M**) with many instances spread throughout the globe. These servers are managed by 12 different organizations.



Let's now go over how DNS is scalable, reliable, and consistent.

Highly scalable

Due to its hierarchical nature, DNS is a highly scalable system. Roughly 1,000 replicated instances of 13 root-level servers are spread throughout the world strategically to handle user queries. The working labor is divided among TLD and root servers to handle a query and, finally, the authoritative servers that are managed by the organizations themselves to make the entire system work. As shown in the DNS hierarchy tree above, different services handle different portions of the tree enabling scalability and manageability of the system.

Reliable

Three main reasons make the DNS a reliable system:

1. **Caching:** The caching is done in the browser, the operating system, and the local name server, and the ISP DNS resolvers also maintain a rich cache of frequently visited services. Even if some DNS servers are temporarily down, cached records can be served to make DNS a reliable system. 
2. **Server replication:** DNS has replicated copies of each logical server spread systematically across the globe to entertain user requests at low 

latency. The redundant servers improve the reliability of the overall system.

3. **Protocol:** Although many clients use DNS over unreliable user datagram protocol (UDP), UDP has its advantages. UDP is much faster and, therefore, improves DNS performance. Furthermore, Internet service's reliability has improved since its inception, so UDP is usually favored over TCP. A DNS resolver can resend the UDP request if it didn't get a reply to a previous one. This request-response needs just one round trip, which provides a shorter delay as compared to TCP, which needs a three-way handshake before data exchange.

Point to Ponder

Question

What happens if a network is congested? Should DNS continue using UDP?

[Show Answer](#) ▼

Consistent

DNS uses various protocols to update and transfer information among replicated servers in a hierarchy. DNS compromises on strong consistency to achieve high performance because data is read frequently from DNS databases as compared to writing. However, DNS provides eventual consistency and updates records on replicated servers lazily. Typically, it can take from a few seconds up to three days to update records on the DNS

servers across the Internet. The time it takes to propagate information among different DNS clusters depends on the DNS infrastructure, the size of the update, and which part of the DNS tree is being updated.

Consistency can suffer because of caching too. Since authoritative servers are located within the organization, it may be possible that certain resource records are updated on the authoritative servers in case of server failures at the organization. Therefore, cached records at the default/local and ISP servers may be outdated. To mitigate this issue, each cached record comes with an expiration time called **time-to-live (TTL)**.

Point to Ponder

Question

To maintain high availability, should the TTL value be large or small?

Show Answer ▼

Test it out

Let's run a couple of commands. Click on the terminal to execute the following commands. Copy the following commands in the terminal to run them. Study the output of the commands:

1. `nslookup www.google.com`
2. `dig www.google.com`

Terminal 1






Loading...

The following slide deck highlights some important aspects of `nslookup` and `dig` output.

```
Non-authoritative answer:
Name:   www.google.com
Address: 74.125.201.99
Name:   www.google.com
Address: 74.125.201.147
Name:   www.google.com
Address: 74.125.201.105
Name:   www.google.com
Address: 74.125.201.103
Name:   www.google.com
Address: 74.125.201.104
Name:   www.google.com
Address: 74.125.201.106
Name:   www.google.com
Address: 2607:f8b0:4001:c07::67
Name:   www.google.com
Address: 2607:f8b0:4001:c07::63
Name:   www.google.com
Address: 2607:f8b0:4001:c07::93
Name:   www.google.com
Address: 2607:f8b0:4001:c07::68
```



Cached response



The output of nslookup www.google.com

1 of 2



Let's go through the meaning of the output:

The nslookup output

- The **Non-authoritative answer**, as the name suggests, is the answer provided by a server that is not the authoritative server of Google. It isn't in the list of authoritative nameservers that Google maintains. So, where does the answer come from? The answer is provided by second, third, and fourth-hand name servers configured to reply to our DNS query—for example, our university or office DNS resolver, our ISP nameserver, our ISP's ISP nameserver, and so on. In short, it can be considered as a cached version of Google's authoritative nameservers response. If we try multiple domain names, we'll realize that we receive a cached response most of the time.
- If we run the same command multiple times, we'll receive the same IP addresses list but in a different order each time. The reason for that is DNS is indirectly performing **load balancing**. It's an important term that we'll gain familiarity with in the coming lessons.

The dig output

- The **Query time: 4 msec** represents the time it takes to get a response from the DNS server. For various reasons, these numbers may be different in our case.
- The **300** value in the **ANSWER SECTION** represents the number of seconds the cache is maintained in the DNS resolver. This means that Google's ADNS keeps a TTL value of five minutes ($\frac{300 \text{ sec}}{60}$).

?

Tt

☾

Note: We invite you to test different services for their TTL and query times to strengthen your understanding. You may use the above terminal for this purpose.

Point to Ponder

Question

If we need DNS to tell us which IP to reach a website or service, how will we know the DNS resolver's IP address? (It seems like a chicken-and-egg problem!)

[Show Answer](#) ▼

[← Back](#)

[Next →](#)





Learn about the basics of load balancers and the services offered by them.

We'll cover the following



- What is load balancing?
- Placing load balancers
- Services offered by load balancers

What is load balancing?

Millions of requests could arrive per second in a typical data center. To serve these requests, thousands (or a hundred thousand) servers work together to share the load of incoming requests.

Note: Here, it's important that we consider how the incoming requests will be divided among all the available servers.

A **load balancer (LB)** is the answer to the question. The job of the load balancer is to fairly divide all clients' requests among the pool of available servers. Load balancers perform this job to avoid overloading or crashing servers.

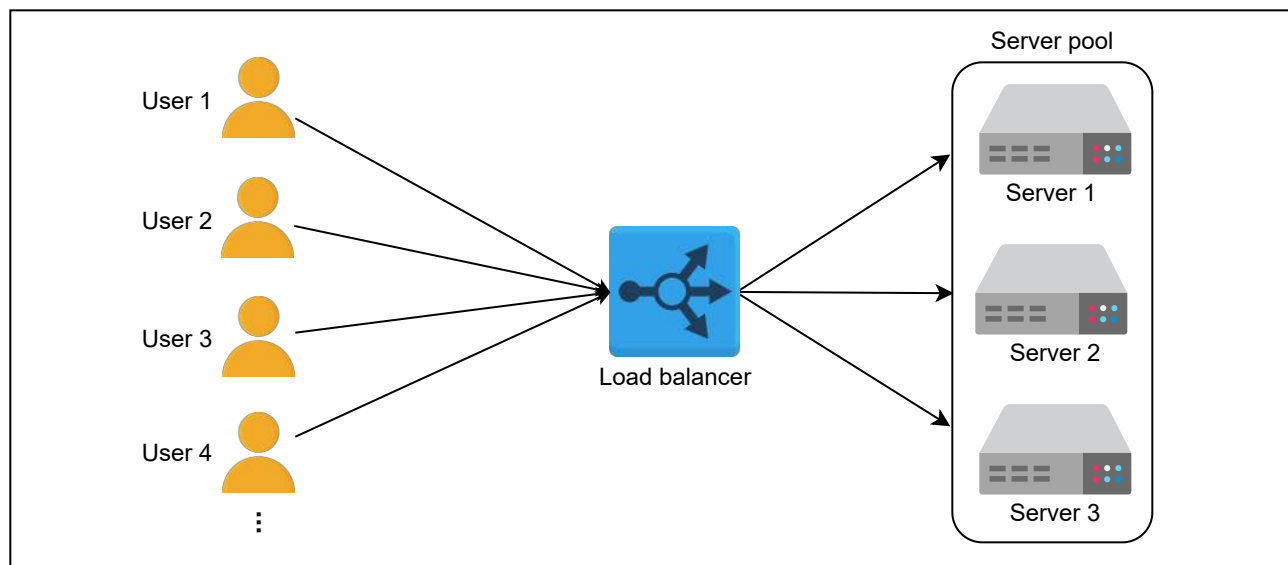
The load balancing layer is the first point of contact within a data center after the firewall. A load balancer may not be required if a service entertains



a few hundred or even a few thousand requests per second. However, for increasing client requests, load balancers provide the following capabilities:

- **Scalability:** By adding servers, the capacity of the application/service can be increased seamlessly. Load balancers make such upscaling or downscaling transparent to the end users.
- **Availability:** Even if some servers go down or suffer a fault, the system still remains available. One of the jobs of the load balancers is to hide faults and failures of servers.
- **Performance:** Load balancers can forward requests to servers with a lesser load so the user can get a quicker response time. This not only improves performance but also improves resource utilization.

Here's an abstract depiction of how load balancers work:



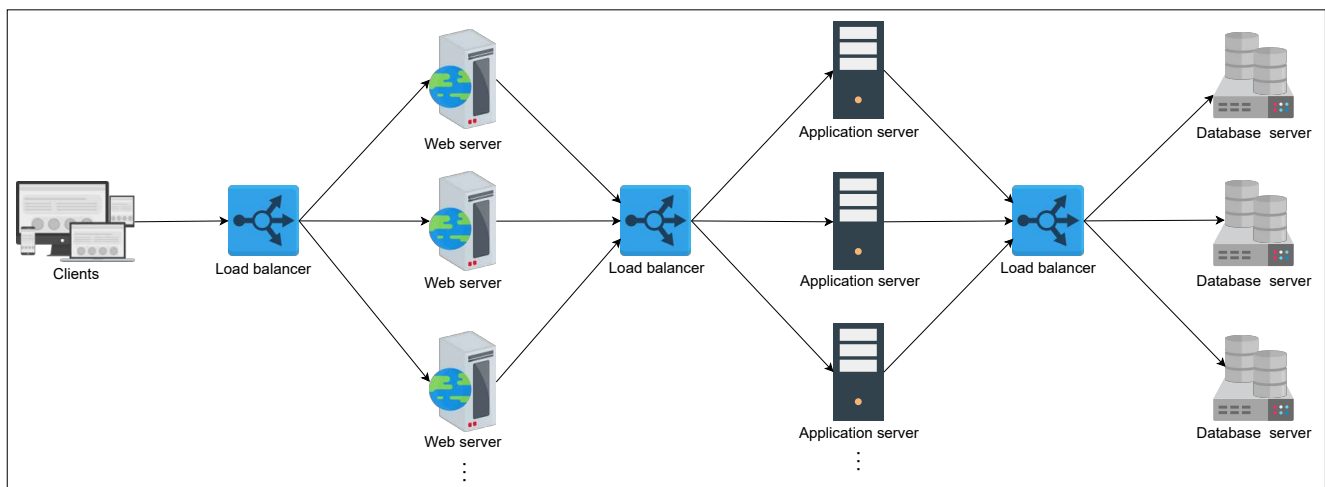
Simplified working of a load balancer

Placing load balancers

Generally, LBs sit between clients and servers. Requests go through to servers and back to clients via the load balancing layer. However, that isn't the only point where load balancers are used.

Let's consider the three well-known groups of servers. That is the web, the application, and the database servers. To divide the traffic load among the available servers, load balancers can be used between the server instances of these three services in the following way:

- Place LBs between end users of the application and web servers/application gateway.
- Place LBs between the web servers and application servers that run the business/application logic.
- Place LBs between the application servers and database servers.



Possible usage of load balancers in a three-tier architecture

In reality, load balancers can be potentially used between any two services with multiple instances within the design of a system.

Services offered by load balancers

LBs not only enable services to be scalable, available, and highly performant, they offer some key services like the following:

- **Health checking:** LBs use the heartbeat protocol to monitor the health and, therefore, reliability of end-servers. Another advantage of health checking is the improved user experience.

?

T

☾

- **TLS termination**: LBs reduce the burden on end-servers by handling TLS termination with the client.
- **Predictive analytics**: LBs can predict traffic patterns through analytics performed over traffic passing through them or using statistics of traffic obtained over time.
- **Reduced human intervention**: Because of LB automation, reduced system administration efforts are required in handling failures.
- **Service discovery**: An advantage of LBs is that the clients' requests are forwarded to appropriate hosting servers by inquiring about the service registry.
- **Security**: LBs may also improve security by mitigating attacks like denial-of-service (DoS) at different layers of the OSI model (layers 3, 4, and 7).

As a whole, load balancers provide flexibility, reliability, redundancy, and efficiency to the overall design of the system.

Food for thought

Question

What if load balancers fail? Are they not a single point of failure (SPOF)?

Show Answer ▼

?

Tt

In the coming lessons, we'll see how load balancers can be used in complex applications and which type of load balancer is appropriate for which use

case.

← Back

How the Domain Name System Works

Next →

Global and Local Load Balancing

☒ Mark as Completed





Global and Local Load Balancing

Understand how global and local load balancing is performed.

We'll cover the following



- Introduction
- Global server load balancing
 - Load balancing in DNS
 - The need for local load balancers
- What is local load balancing?

Introduction

From the previous lesson, it may seem like load balancing is performed only within the data center. However, load balancing is required at a global and a



- **Global server load balancing (GSLB):** GSLB involves the distribution of traffic load across multiple geographical regions.
- **Local load balancing:** This refers to load balancing achieved within a data center. This type of load balancing focuses on improving efficiency and better resource utilization of the hosting servers in a data center.

Let's understand each of the two techniques below.

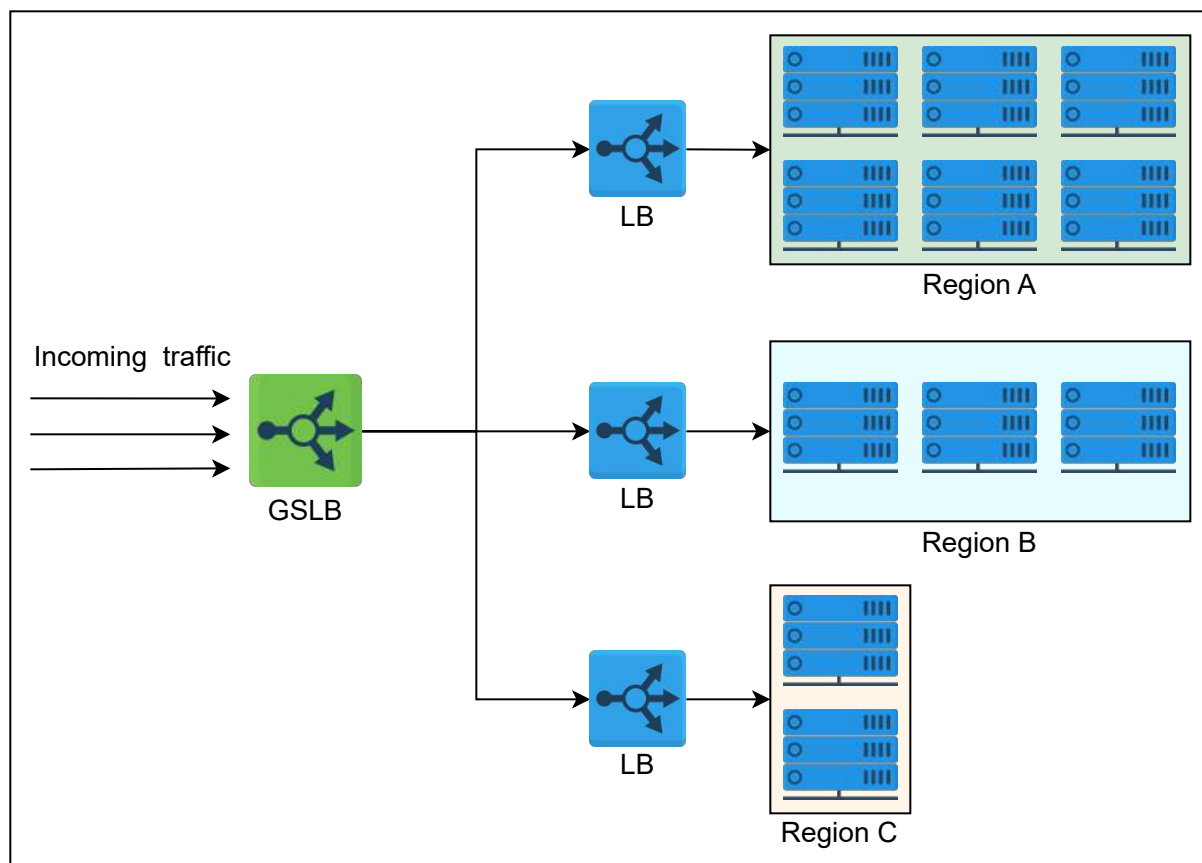
Global server load balancing



GSLB ensures that globally arriving traffic load is intelligently forwarded to a data center. For example, power or network failure in a data center requires that all the traffic be rerouted to another data center. GSLB takes forwarding decisions based on the users' geographic locations, the number of hosting servers in different locations, the health of data centers, and so on.

In the next lesson, we'll also learn how GSLB offers automatic zonal failover. GSLB service can be installed on-premises or obtained through Load Balancing as a Service (LBaaS).

The illustration below shows that the GSLB can forward requests to three different data centers. Each local load balancing layer within a data center will maintain a control plane connection with the GSLB providing information about the health of the LBs and the server farm. GSLB uses this information to drive traffic decisions and forward traffic load based on each region's configuration and monitoring information.

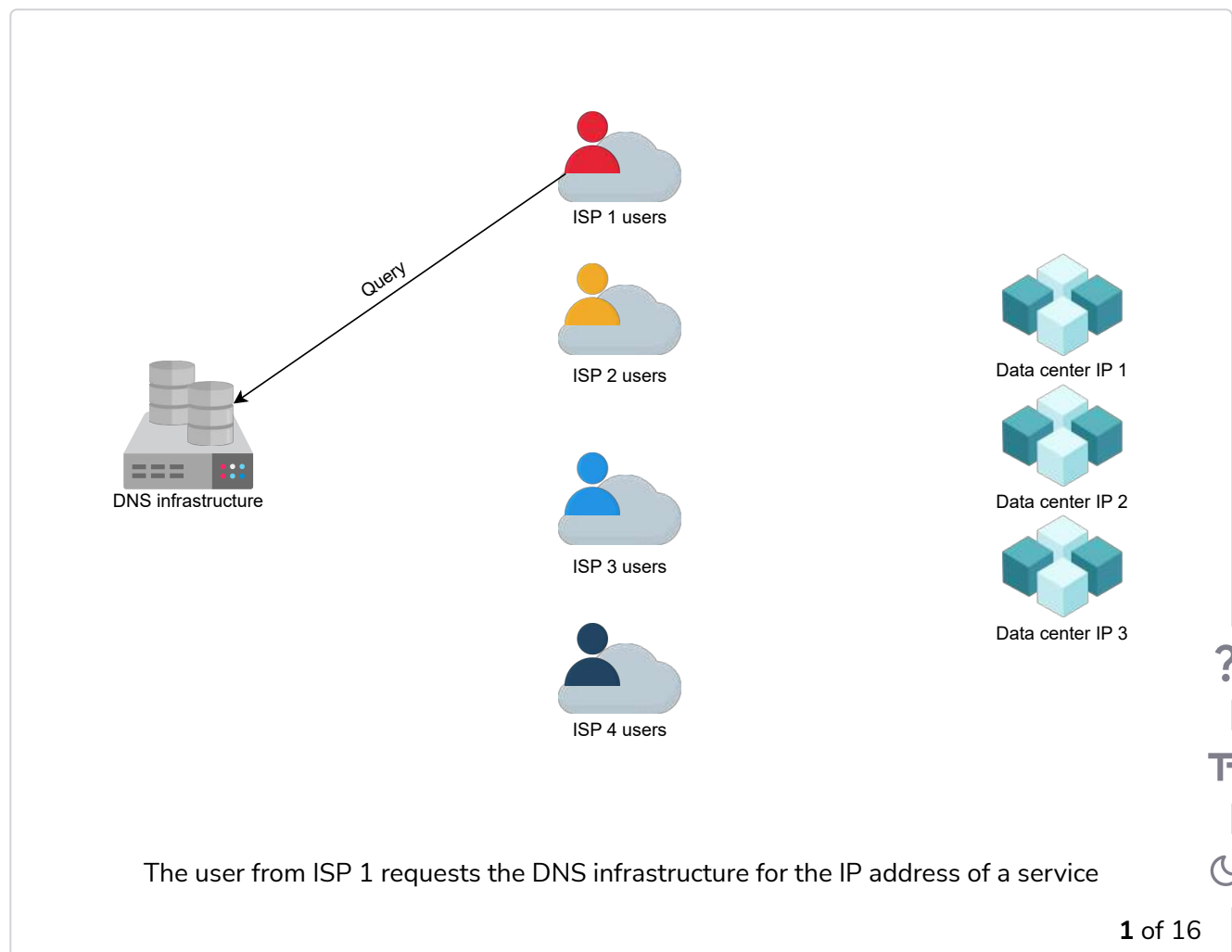


Usage of global load balancing to send user requests to different regions

Now, we'll discuss how the domain name system (DNS) can perform GSLB.

Load balancing in DNS

We understand that DNS can respond with multiple IP addresses for a DNS query. In the lesson on [DNS](#), we discussed that it's possible to do load balancing through DNS while looking at the output of `nslookup`. DNS uses a simple technique of reordering the list of IP addresses in response to each DNS query. Therefore, different users get a reordered IP address list. It results in users visiting a different server to entertain their requests. In this way, DNS distributes the load of requests on different data centers. This is performing GSLB. In particular, DNS uses round-robin to perform load balancing as shown below:





As shown above, round-robin in DNS forwards clients to data centers in a strict circular order. However, round-robin has the following limitations:

- Different ISPs have a different number of users. An ISP serving many customers will provide the same cached IP to its clients, resulting in uneven load distribution on end-servers.
- Because the round-robin load-balancing algorithm doesn't consider any end-server crashes, it keeps on distributing the IP address of the crashed servers until the TTL of the cached entries expires. Availability of the service, in that case, can take a hit due to DNS-level load balancing.

Despite its limitations, round-robin is still widely used by many DNS service providers. Furthermore, DNS uses short TTL for cached entries to do effective load balancing among different data centers.

Note: DNS isn't the only form of GSLB. Application delivery controllers (ADCs) and cloud-based load balancing ([discussed later](#)) are better ways to do GSLB.

💡 What are application delivery controllers (ADCs)?

The need for local load balancers

?

DNS plays a vital role in balancing the load, but it suffers from the following limitations:

T

☾

- The small size of the DNS packet (512 Bytes) isn't enough to include all possible IP addresses of the servers.
- There's limited control over the client's behavior. Clients may select arbitrarily from the received set of IP addresses. Some of the received IP addresses may belong to busy data centers.
- Clients can't determine the closest address to establish a connection with. Although DNS geolocation and anycasting-like solutions can be implemented, they aren't trivial solutions.
- In case of failures, recovery can be slow through DNS because of the caching mechanism, especially when TTL values are longer.

To solve some of the above problems, we need another layer of load balancing in the form of local LB. In the next lesson, we'll discuss different details about local load balancers.

What is local load balancing?

Local load balancers reside within a data center. They behave like a reverse proxy and make their best effort to divide incoming requests among the pool of available servers. Incoming clients' requests seamlessly connect to the LB that uses a virtual IP address (**VIP**↴).

Point to Ponder

Question

Can DNS be considered a global server load balancer (GSLB)?

Show Answer ▼



In the next lesson, we'll explore some advanced details of local load balancers.

[← Back](#)[Next →](#)[Introduction to Load Balancers](#)[Advanced Details of Load Balancers](#)☒ Mark as Completed



Advanced Details of Load Balancers

Understand load balancers and their usage within a system.

We'll cover the following



- Algorithms of load balancers
 - Static versus dynamic algorithms
 - Stateful versus stateless LBs
 - Stateful load balancing
 - Stateless load balancing
- Types of load balancers
- Load balancer deployment
 - Tier-0 and tier-1 LBs
 - Tier-2 LBs
 - Tier-3 LBs
 - Practical example
- Implementation of load balancers
 - Hardware load balancers
 - Software load balancers
 - Cloud load balancers
- Conclusion



This lesson will focus on some of the well-known algorithms used in the local load balancers. We'll also understand how load balancers are connected to form a hierarchy, sharing work across different tiers of LBs.



Algorithms of load balancers



well-known algorithms are given below:

- **Round-robin scheduling:** In this algorithm, each request is forwarded to a server in the pool in a repeating sequential manner.
- **Weighted round-robin:** If some servers have a higher capability of serving clients' requests, then it's preferred to use a weighted round-robin algorithm. In a weighted round-robin algorithm, each node is assigned a weight. LBs forward clients' requests according to the weight of the node. The higher the weight, the higher the number of assignments.
- **Least connections:** In certain cases, even if all the servers have the same capacity to serve clients, uneven load on certain servers is still a possibility. For example, some clients may have a request that requires longer to serve. Or some clients may have subsequent requests on the same connection. In that case, we can use algorithms like least connections where newer arriving requests are assigned to servers with fewer existing connections. LBs keep a state of the number and mapping of existing connections in such a scenario. We'll discuss more about state maintenance later in the lesson.
- **Least response time:** In performance-sensitive services, algorithms such as least response time are required. This algorithm ensures that the server with the least response time is requested to serve the clients.
- **IP hash:** Some applications provide a different level of service to users based on their IP addresses. In that case, hashing the IP address is performed to assign users' requests to servers. ?
- **URL hash:** It may be possible that some services within the application are provided by specific servers only. In that case, a client requesting service from a URL is assigned to a certain cluster or set of servers. The URL hashing algorithm is used in those scenarios. Tt ?

There are other algorithms also, like randomized or weighted least connections algorithms.

Static versus dynamic algorithms

Algorithms can be static or dynamic depending on the machine's state. Let's look at each of the categories individually:

Static algorithms don't consider the changing state of the servers.

Therefore, task assignment is carried out based on existing knowledge about the server's configuration. Naturally, these algorithms aren't complex, and they get implemented in a single router or commodity machine where all the requests arrive.

Dynamic algorithms are algorithms that consider the current or recent state of the servers. Dynamic algorithms maintain state by communicating with the server, which adds a communication overhead. State maintenance makes the design of the algorithm much more complicated.

Dynamic algorithms require different load balancing servers to communicate with each other to exchange information. Therefore, dynamic algorithms can be modular because no single entity will do the decision-making. Although this adds complexity to dynamic algorithms, it results in improved forwarding decisions. Finally, dynamic algorithms monitor the health of the servers and forward requests to active servers only.

Note: In practice, dynamic algorithms provide far better results because they maintain a state of serving hosts and are, therefore, worth the effort and complexity.

?

Tr



Stateful versus stateless LBs

While static and dynamic algorithms are required to consider the health of the hosting servers, a state is maintained to hold session information of different clients with hosting servers.

If the session information isn't kept at a lower layer (distributed cache or database), load balancers are used to keep the session information. Below, we describe two ways of handling session maintenance through LBs:

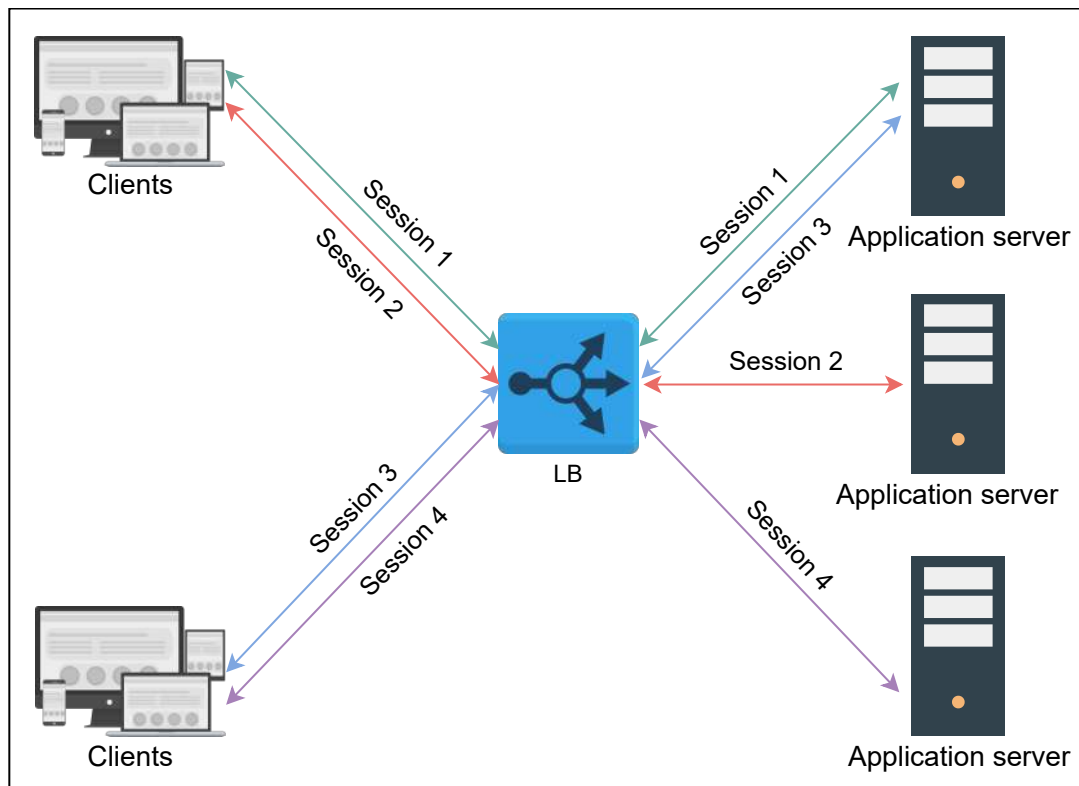
- Stateful
- Stateless

Stateful load balancing

As the name indicates, **stateful load balancing** involves maintaining a state of the sessions established between clients and hosting servers. The stateful LB incorporates state information in its algorithm to perform load balancing.

Essentially, the stateful LBs retain a data structure that maps incoming clients to hosting servers. Stateful LBs increase complexity and limit scalability because session information of all the clients is maintained across all the load balancers. That is, load balancers share their state information with each other to make forwarding decisions.





Stateful load balancing

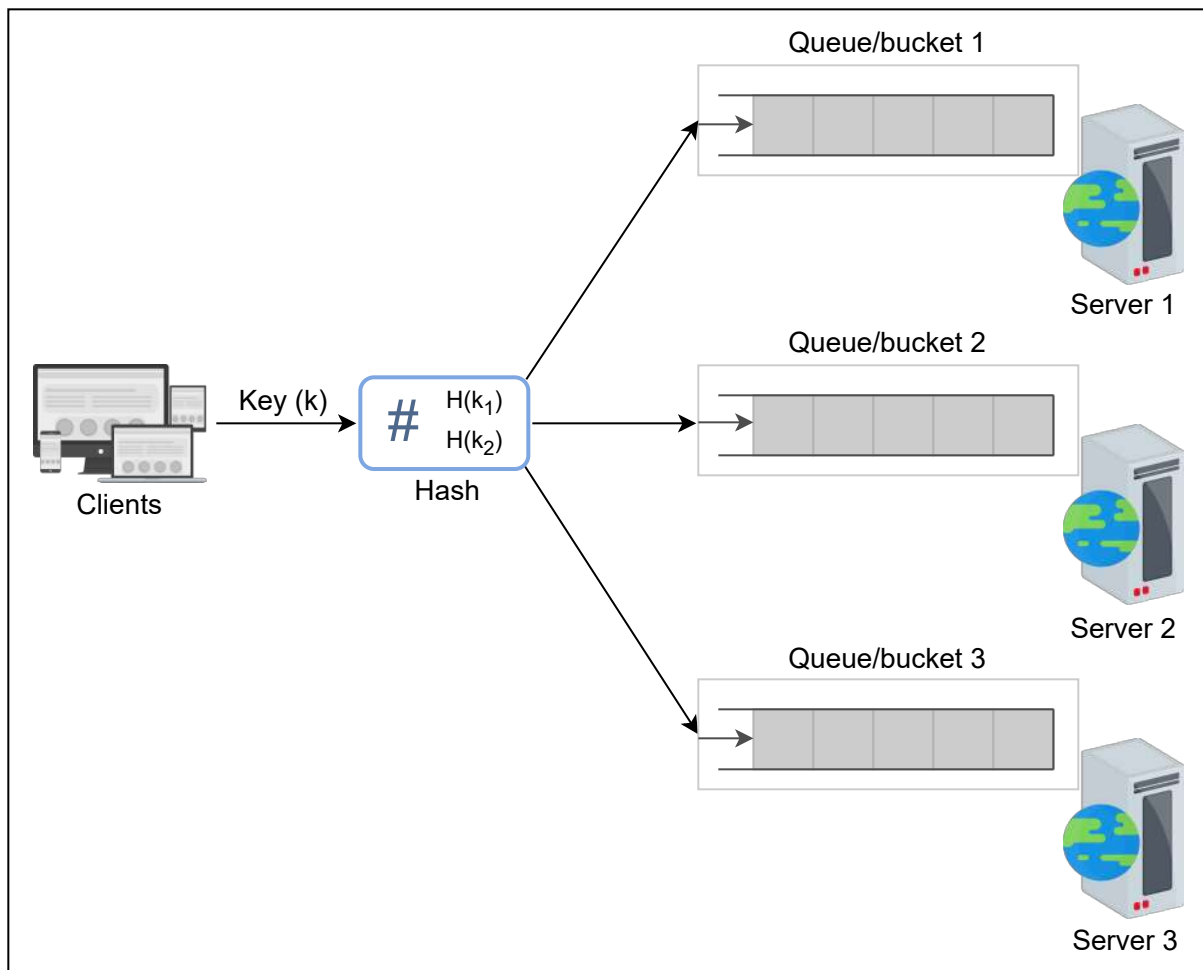
Stateless load balancing

Stateless load balancing maintains no state and is, therefore, faster and lightweight. Stateless LBs use consistent hashing to make forwarding decisions. However, if infrastructure changes (for example, a new application server joining), stateless LBs may not be as resilient as stateful LBs because consistent hashing alone isn't enough to route a request to the correct application server. Therefore, a local state may still be required along with consistent hashing.

?

Tt





Stateless load balancers using hash buckets to map requests to end servers

Therefore, a state maintained across different load balancers is considered as stateful load balancing. Whereas, a state maintained within a load balancer for internal use is assumed as stateless load balancing.

Types of load balancers

Depending on the requirements, load balancing can be performed at the network/transport and application layer of the open systems interconnection (OSI) layers.

- **Layer 4 load balancers:** Layer 4 refers to the load balancing performed on the basis of transport protocols like TCP and UDP. These types of LB maintain connection/session with the clients and ensure that the same (TCP/UDP) communication ends up being forwarded to the same back.

end server. Even though TLS termination is performed at layer 7 LBs, some layer 4 LBs also support it.

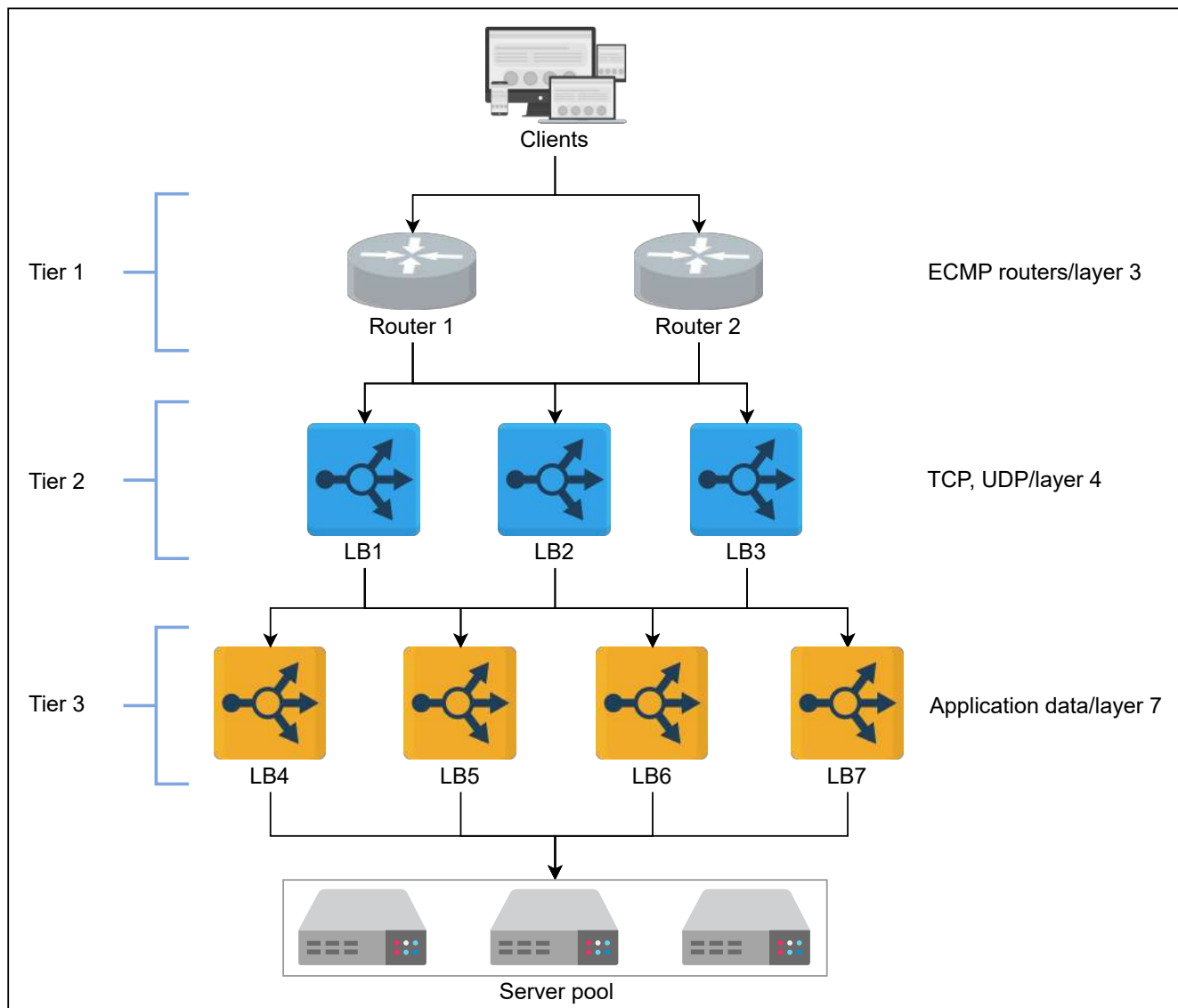
- **Layer 7 load balancers:** Layer 7 load balancers are based on the data of application layer protocols. It's possible to make application-aware forwarding decisions based on HTTP headers, URLs, cookies, and other application-specific data—for example, user ID. Apart from performing TLS termination, these LBs can take responsibilities like rate limiting users, HTTP routing, and header rewriting.

Note: Layer 7 load balancers are smart in terms of inspection. However layer 4 load balancers are faster in terms of processing.

Load balancer deployment

We discussed the trade-offs of load balancing performed at different OSI layers. In practice, however, a single layer LB isn't enough for a large data center. In fact, multiple layers of load balancers coordinate to take informed forwarding decisions. A traditional data center may have a three-tier LB as shown below:





Three-tier load balancer in a typical data center

Tier-0 and tier-1 LBs

If DNS can be considered as the tier-0 load balancer, equal cost multipath (ECMP) routers are the tier-1 LBs. From the name of ECMP, it's evident that this layer divides incoming traffic on the basis of IP or some other algorithm like round-robin or weighted round-robin. Tier-1 LBs will balance the load across different paths to higher tiers of load balancers.

?

ECMP routers play a vital role in the horizontal scalability of the higher-tier LBs.



Tier-2 LBs

The second tier of LBs include layer 4 load balancers. Tier-2 LBs make sure that for any connection, all incoming packets are forwarded to the same tier-3 LBs. To achieve this goal, a technique like consistent hashing can be utilized. But in case of any changes to the infrastructure, consistent hashing may not be enough. Therefore, we have to maintain a local or global state as we'll see in the coming sections of the lesson.

Tier-2 load balancers can be considered the glue between tier-1 and tier-3 LBs. Excluding tier-2 LBs could result in erroneous forwarding decisions in case of failures or dynamic scaling of LBs.

Tier-3 LBs

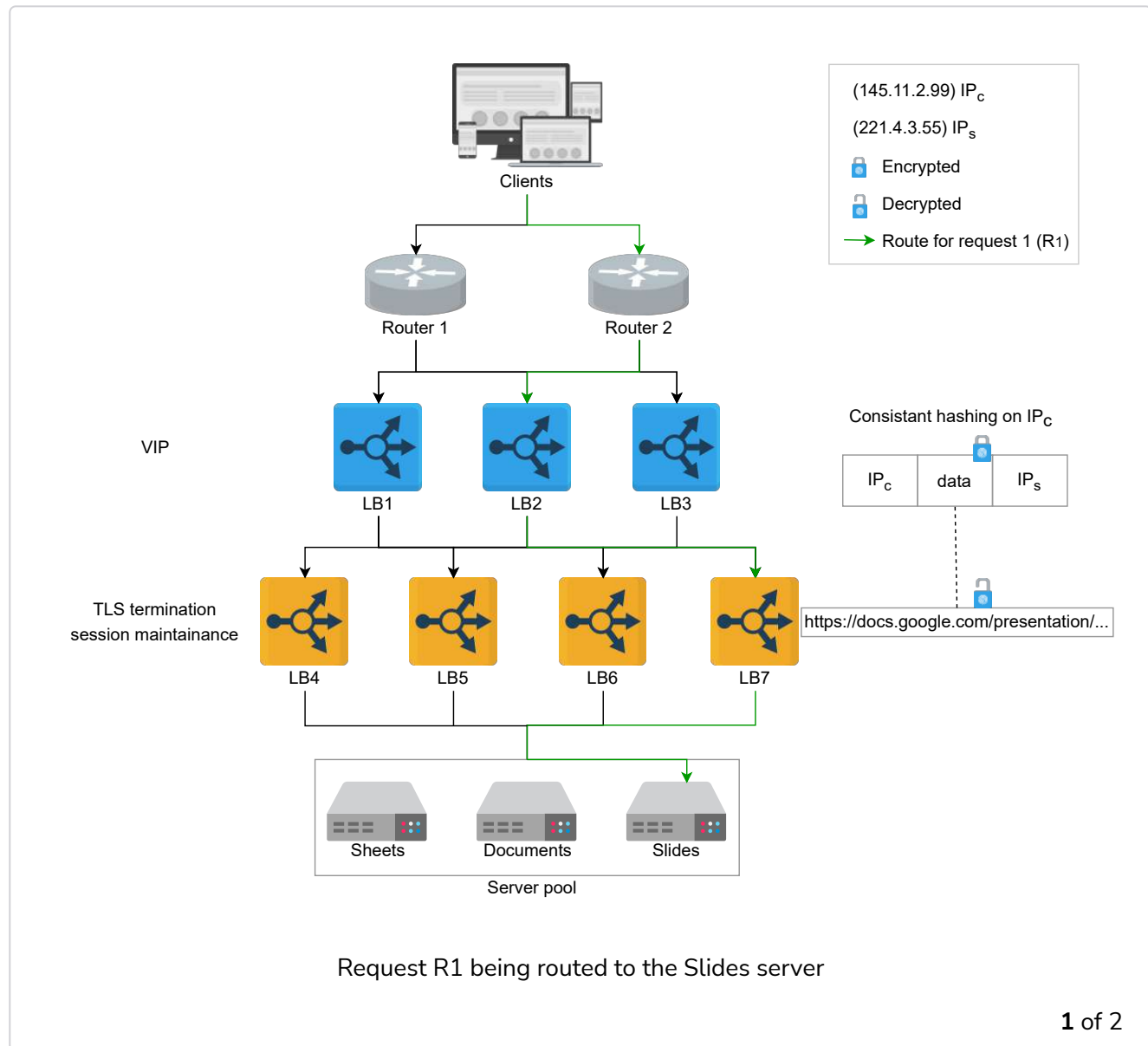
Layer 7 LBs provide services at tier 3. Since these LBs are in direct contact with the back-end servers, they perform health monitoring of servers at HTTP level. This tier enables scalability by evenly distributing requests among the set of healthy back-end servers and provides high availability by monitoring the health of servers directly. This tier also reduces the burden on end-servers by handling low-level details like TCP-congestion control protocols, the discovery of Path MTU (maximum transmission unit), the difference in application protocol between client and back-end servers, and so on. The idea is to leave the computation and data serving to the application servers and effectively utilize load balancing commodity machines for trivial tasks. In some cases, layer 7 LBs are at the same level as the service hosts.

To summarize, tier 1 balances the load among the load balancers themselves. Tier 2 enables a smooth transition from tier 1 to tier 3 in case of failures, whereas tier 3 does the actual load balancing between back-end servers. Each tier performs other tasks to reduce the burden on end-servers.



Practical example

Let's look at an example where requests from a client come in and get forwarded to different application servers based on the application data inside the client's network packets.



Let's look at the illustration above in the following steps:

1. R_1 indicates request 1 coming through one of the ECMP routers (tier-1 LBs).

2. ECMP routers forward R_1 to any of the three available tier-2 LBs using a round-robin algorithm. Tier-2 LBs take a hash of the source IP address (IP_c) and forward the packet to the next tier of LBs.
3. Tier-3, upon receiving the packet, offloads TLS and reads the HTTP(S) data. By observing the requested URL, it forwards the request to the server handling requests for `slides`.

R_2 takes the same path but a different end-server because the requested URL contains `document` instead of `slides`. Tier-3 LBs are preconfigured to forward requests to application servers based on the application data. For instance, a typical HAProxy server can have the following configuration in tier-3 LBs:

```
mode HTTP //define which mode the LB will work in
acl slidesApp path_end -i /presentation //define a category of applications
use_backend slidesServers if slidesApp // use a set of backend servers if
backend slidesServers // listing servers serving slidesApp
server slides1 192.168.12.1:80 //using slides1 server to serve slidesApp
```

HAProxy sample configuration for layer 7 load balancers

Quiz

Question 1

After a request reaches a back-end server, should the response be routed back through each tier of the load balancers?

Show Answer ▼



Implementation of load balancers

Different kinds of load balancers can be implemented depending on the number of incoming requests, organization, and application-specific requirements:

Hardware load balancers

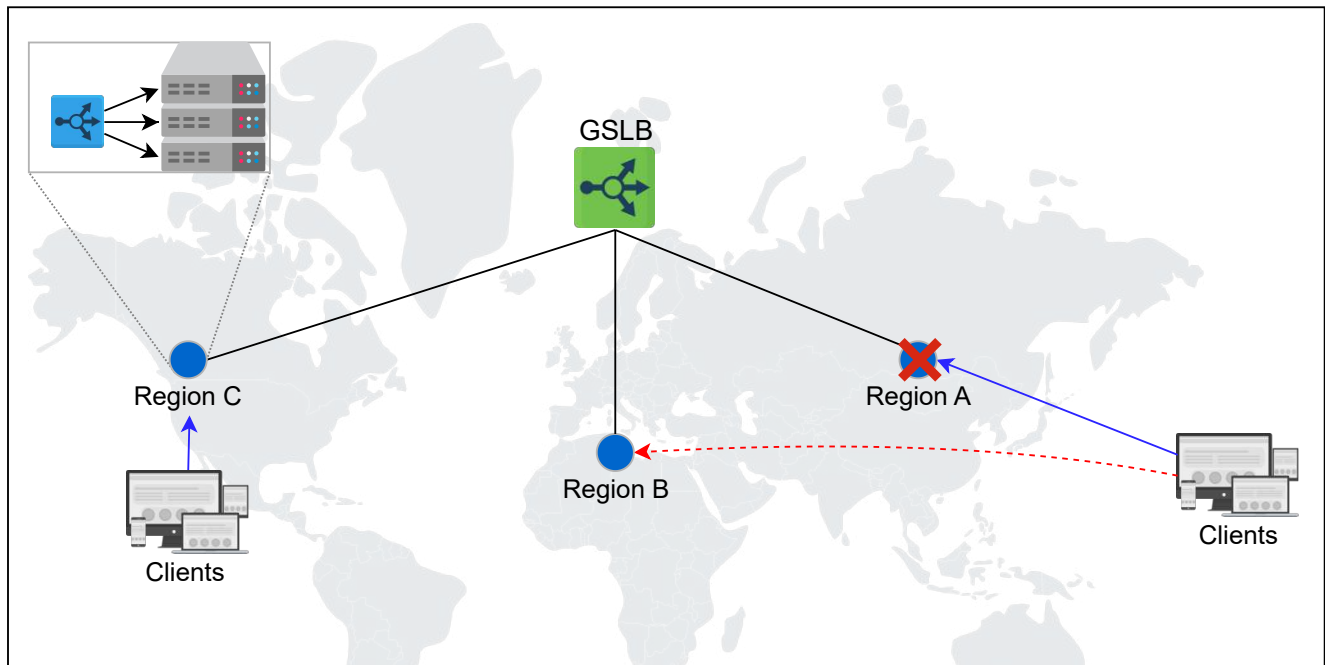
Load balancers were introduced in the 1990s as hardware devices. Hardware load balancers work as stand-alone devices and are quite expensive. Nonetheless, they have their performance benefits and are able to handle a lot of concurrent users. Configuration of hardware-based solutions is problematic because it requires additional human resources. Therefore, they aren't the go-to solutions even for large enterprises that can afford them. Availability can be an issue with hardware load balancers because additional hardware will be required to failover to in case of failures. Finally, hardware LBs can have higher maintenance/operational costs and compatibility issues making them less flexible. Not to mention that hardware LBs have vendor locks as well.

Software load balancers

Software load balancers are becoming increasingly popular because of their flexibility, programmability, and cost-effectiveness. That's all possible because they're implemented on commodity hardware. Software LBs scale well as requirements grow. Availability won't be an issue with software LBs because small additional costs are required to implement shadow load balancers on commodity hardware. Additionally, software LBs can provide predictive analysis that can help prepare for future traffic patterns.

Cloud load balancers

With the advent of the field of cloud computing, Load Balancers as a Service (LBaaS) has been introduced. This is where cloud owners provide load balancing services. Users pay according to their usage or the service-level agreement (SLA) with the cloud provider. Cloud-based LBs may not necessarily replace a local on-premise load balancing facility, but they can perform global traffic management between different zones. Primary advantages of such load balancers include ease of use, management, metered cost, flexibility in terms of usage, auditing, and monitoring services to improve business decisions. An example of how cloud-based LBs can provide GSLB is given below:



?

Tt





Types of Databases

Understand various types of databases and their use cases in system design.

We'll cover the following



- Relational databases
 - Why relational databases?
 - Flexibility
 - Reduced redundancy
 - Concurrency
 - Integration
 - Backup and disaster recovery
 - Drawback
 - Impedance mismatch
- Why non-relational (NoSQL) databases?
 - Types of NoSQL databases
 - Key-value database
 - Document database
 - Graph database
 - Columnar database
 - Drawbacks of NoSQL databases
 - Lack of standardization
 - Consistency
- Choose the right database
- Quiz



As we discussed earlier, databases are divided into two types: relational and non-relational. Let's discuss these types in detail.

Relational databases

Relational databases adhere to particular schemas before storing the data. The data stored in relational databases has prior structure. Mostly, this model organizes data into one or more relations (also called tables), with a unique key for each tuple (instance). Each entity of the data consists of instances and attributes, where instances are stored in rows, and the attributes of each instance are stored in columns. Since each tuple has a unique key, a tuple in one table can be linked to a tuple in other tables by storing the primary keys in other tables, generally known as foreign keys.

A Structure Query Language (SQL) is used for manipulating the database. This includes insertion, deletion, and retrieval of data.

There are various reasons for the popularity and dominance of relational databases, which include simplicity, robustness, flexibility, performance, scalability, and compatibility in managing generic data.

Relational databases provide the atomicity, consistency, isolation, and durability (ACID) properties to maintain the integrity of the database. ACID is a powerful abstraction that simplifies complex interactions with the data and hides many anomalies (like dirty reads, dirty writes, read skew, lost updates, write skew, and phantom reads) behind a simple transaction abort.

But ACID is like a big hammer by design so that it's generic enough for all t' ? problems. If some specific application only needs to deal with a few anomalies, there's a window of opportunity to use a custom solution for higher performance, though there is added complexity.

Let's discuss ACID in detail:



- **Atomicity:** A transaction is considered an atomic unit. Therefore, either all the statements within a transaction will successfully execute, or none of them will execute. If a statement fails within a transaction, it should be aborted and rolled back.



For example, if multiple users want to view a record from the database, it should return a similar result each time.

- **Isolation:** In the case of multiple transactions running concurrently, they shouldn't be affected by each other. The final state of the database should be the same as the transactions were executed sequentially.
- **Durability:** The system should guarantee that completed transactions will survive permanently in the database even in system failure events.

Various database management systems (DBMS) are used to define relational database schema along with other operations, such as to store, retrieve, and run SQL queries on data. Some of the popular DBMS are as follows:

- MySQL
- Oracle Database
- Microsoft SQL Server
- IBM DB2
- Postgres
- SQLite

Why relational databases?

Relational databases are the default choices of software professionals for structured data storage. There are a number of advantages to these databases. One of the greatest powers of the relational database is its abstractions of ACID transactions and related programming semantics. This



make it very convenient for the end-programmer to use a relational database. Let's revisit some important features of relational databases:

Flexibility

In the context of SQL, **data definition language (DDL)** provides us the flexibility to modify the database, including tables, columns, renaming the tables, and other changes. DDL even allows us to modify schema while other queries are happening and the database server is running.

Reduced redundancy

One of the biggest advantages of the relational database is that it eliminates data redundancy. The information related to a specific entity appears in one table while the relevant data to that specific entity appears in the other tables linked through foreign keys. This process is called normalization and has the additional benefit of removing an inconsistent dependency.

Concurrency

Concurrency is an important factor while designing an enterprise database. In such a case, the data is read and written by many users at the same time. We need to coordinate such interactions to avoid inconsistency in data—for example, the double booking of hotel rooms. Concurrency in a relational database is handled through transactional access to the data. As explained earlier, a transaction is considered an atomic operation, so it also works in error handling to either roll back or commit a transaction on successful execution.

Integration

The process of aggregating data from multiple sources is a common practice in enterprise applications. A common way to perform this aggregation is to integrate a shared database where multiple applications store their data.

?

Tr

☾

This way, all the applications can easily access each other's data while the concurrency control measures handle the access of multiple applications.

Backup and disaster recovery

Relational databases guarantee the state of data is consistent at any time.

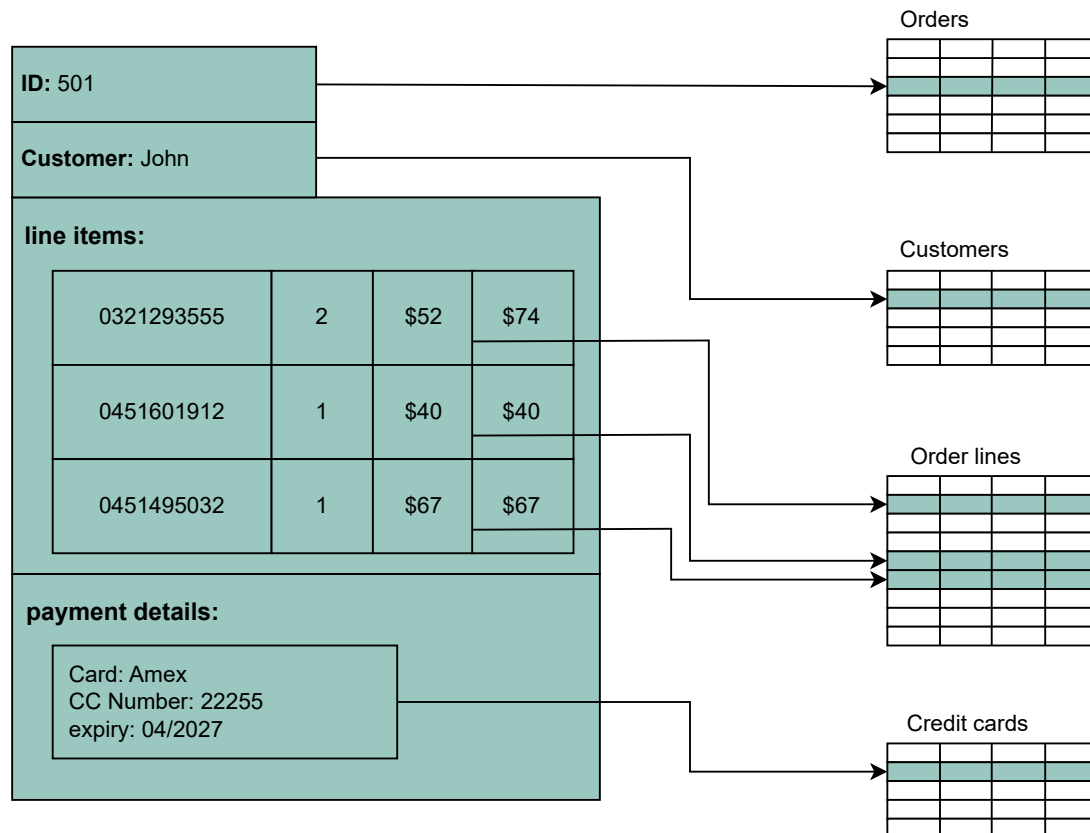
The export and import operations make backup and restoration easier. Most cloud-based relational databases perform continuous mirroring to avoid loss of data and make the restoration process easier and quicker.

Drawback

Impedance mismatch

Impedance mismatch is the difference between the relational model and the in-memory data structures. The relational model organizes data into a tabular structure with relations and tuples. SQL operation on this structured data yields relations aligned with relational algebra. However, it has some limitations. In particular, the values in a table take simple values that can't be a structure or a list. The case is different for in-memory, where a complex data structure can be stored. To make the complex structures compatible with the relations, we would need a translation of the data in light of relational algebra. So, the impedance mismatch requires translation between two representations, as denoted in the following figure:





A single aggregated value in the view is composed of several rows and tables in the relational database

Why non-relational (NoSQL) databases?

A NoSQL database is designed for a variety of data models to access and manage data. There are various types of NoSQL databases, which we'll explain in the next section. These databases are used in applications that require a large volume of semi-structured and unstructured data, low latency, and flexible data models. This can be achieved by relaxing some of the data consistency restrictions of other databases. Following are some characteristics of the NoSQL database:

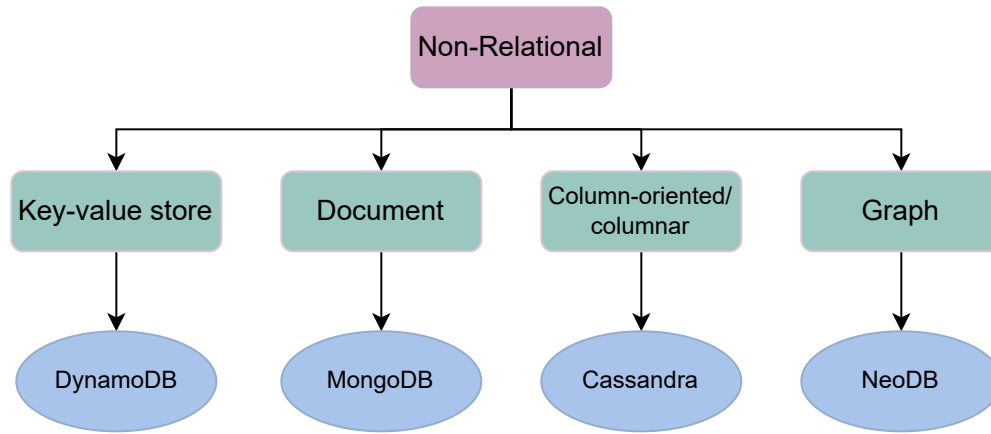
- **Simple design:** Unlike relational databases, NoSQL doesn't require dealing with the impedance mismatch—for example, storing all the employees' data in one document instead of multiple tables that require join operations. This strategy makes it simple and easier to write less code, debug, and maintain.

- **Horizontal scaling:** Primarily, NoSQL is preferred due to its ability to run databases on a large cluster. This solves the problem when the number of concurrent users increases. NoSQL makes it easier to scale out since the data related to a specific employee is stored in one document instead of multiple tables over nodes. NoSQL databases often spread data across multiple nodes and balance data and queries across nodes automatically. In case of a node failure, it can be transparently replaced without any application disruption.
- **Availability:** To enhance the availability of data, node replacement can be performed without application downtime. Most of the non-relational databases' variants support data replication to ensure high availability and disaster recovery.
- **Support for unstructured and semi-structured data:** Many NoSQL databases work with data that doesn't have schema at the time of database configuration or data writes. For example, document databases are structureless; they allow documents (JSON, XML, BSON, and so on) to have different fields. For example, one JSON document can have fewer fields than the other.
- **Cost:** Licenses for many RDBMSs are pretty expensive, while many NoSQL databases are open source and freely available. Similarly, some RDBMSs rely on costly proprietary hardware and storage systems, while NoSQL databases usually use clusters of cheap commodity servers.

NoSQL databases are divided into various categories based on the nature of the operations and features, including document store, columnar database, key-value store, and graph database. We'll discuss each of them along with their use cases from the system design perspective in the following sections.

Types of NoSQL databases

Various types of NoSQL databases are described below:



The types of NoSQL databases

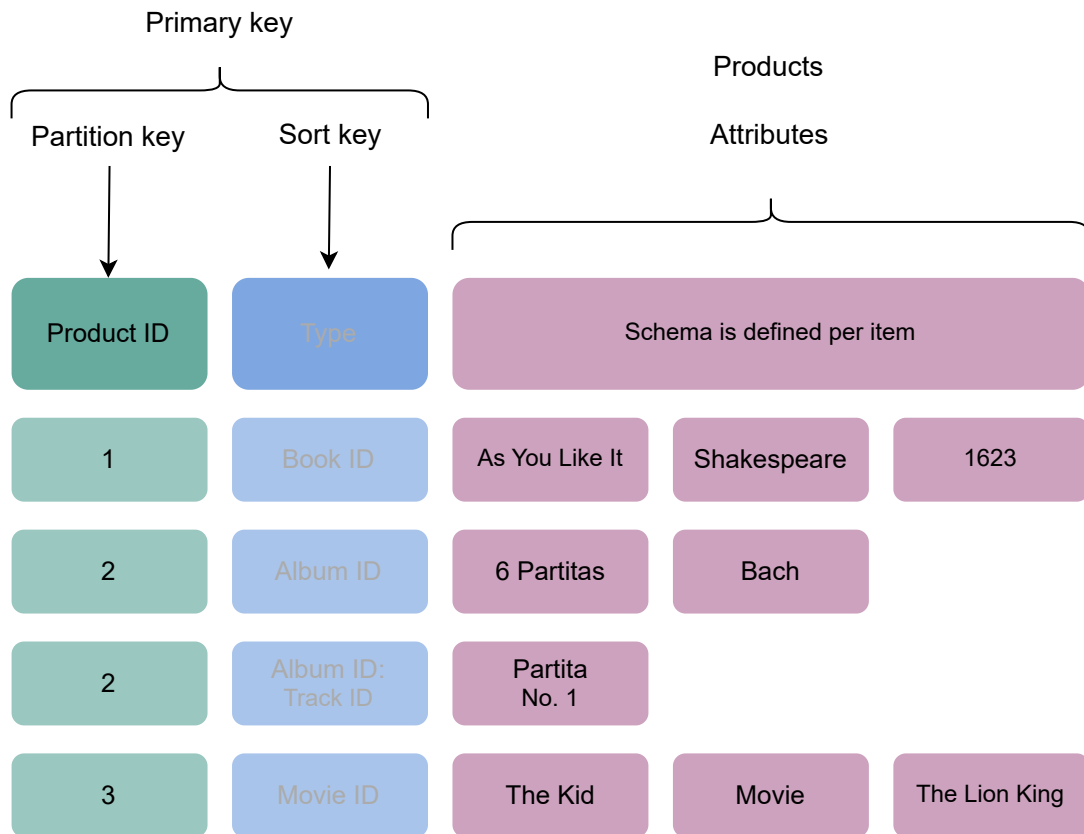
Key-value database

Key-value databases use key-value methods like hash tables to store data in key-value pairs. We can see this depicted in the figure a couple of paragraphs below. Here, the key serves as a unique or primary key, and the values can be anything ranging from simple scalar values to complex objects. These databases allow easy partitioning and horizontal scaling of the data. Some popular key-value databases include Amazon DynamoDB, Redis, and Memcached DB.

Use case: Key-value databases are efficient for session-oriented applications. Session oriented-applications, such as web applications, store users' data in the main memory or in a database during a session. This data may include user profile information, recommendations, targeted promotions, discounts, and more. A unique ID (a key) is assigned to each user's session for easy access and storage. Therefore, a better choice to store such data is the key-value database.

The following figure shows an example of a key-value database. The **Product ID** and **Type** of the item are collectively considered as the primary key. This is considered as a key for this key-value database. Moreover, the schema for

storing the item attributes is defined based on the nature of the item and the number of attributes it possesses.



Data stored in the form of key-value pair in DynamoDB, where the key is the combination of two attributes (Product ID and Type)

Document database

A **document database** is designed to store and retrieve documents in formats like XML, JSON, BSON, and so on. These documents are composed of a hierarchical tree data structure that can include maps, collections, and scalar values. Documents in this type of database may have varying structures and data. MongoDB and Google Cloud Firestore are examples of document databases. ?

Use case: Document databases are suitable for unstructured catalog data, **T** like JSON files or other complex structured hierarchical data. For example, in e-commerce applications, a product has thousands of attributes, which i. ☾

unfeasible to store in a relational database due to its impact on the reading performance. Here comes the role of a document database, which can efficiently store each attribute in a single file for easy management and faster reading speed. Moreover, it's also a good option for content management applications, such as blogs and video platforms. An entity required for the application is stored as a single document in such applications.

The following example shows data stored in a JSON document. This data is about a person. Various attributes are stored in the file, including **id**, **name**, **email**, and so on.

```
{  "id": 1001,
  "name": "Brown",
  "title": "Mr.",
  "email": "brown@anyEmail.com",
  "cell": "123-465-9999",
  "likes": [
    "designing",
    "cycling",
    "skiing"],
  "businesses": [
    { "name": "ABC co.",
      "partner": "Vike",
      "status": "Bankrupt",
      "date_founded": {
        "$date": "2021-12-10" } } ] }
```

A JSON file containing data of a businessman

Graph database

Graph databases use the graph data structure to store data, where nodes represent entities, and edges show relationships between entities. The organization of nodes based on relationships leads to interesting patterns between the nodes. This database allows us to store the data once and then interpret it differently based on relationships. Popular graph databases

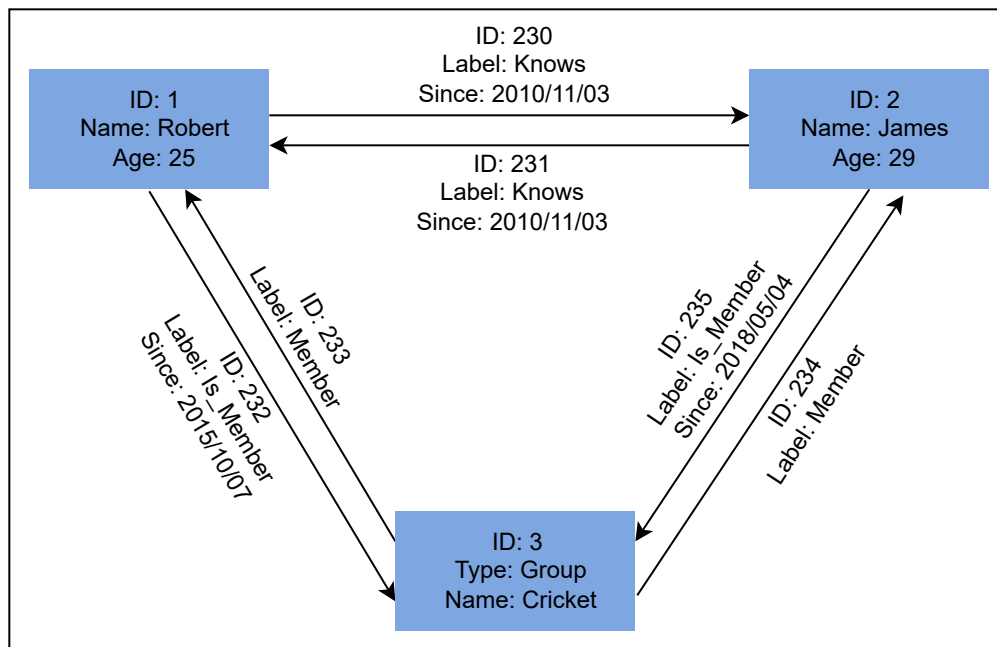
?

T

☾

include Neo4J, OrientDB, and InfiniteGraph. Graph data is kept in store files for persistent storage. Each of the files contains data for a specific part of the graph, such as nodes, links, properties, and so on.

In the following figure, some data is stored using a graph data structure in nodes connected to each other via edges representing relationships between nodes. Each node has some properties, like **Name**, **ID**, and **Age**. The node having **ID: 2** has the **Name** of **James** and **Age** of **29** years.



A graph consists of nodes and links. This graph captures entities and their relationships with each other

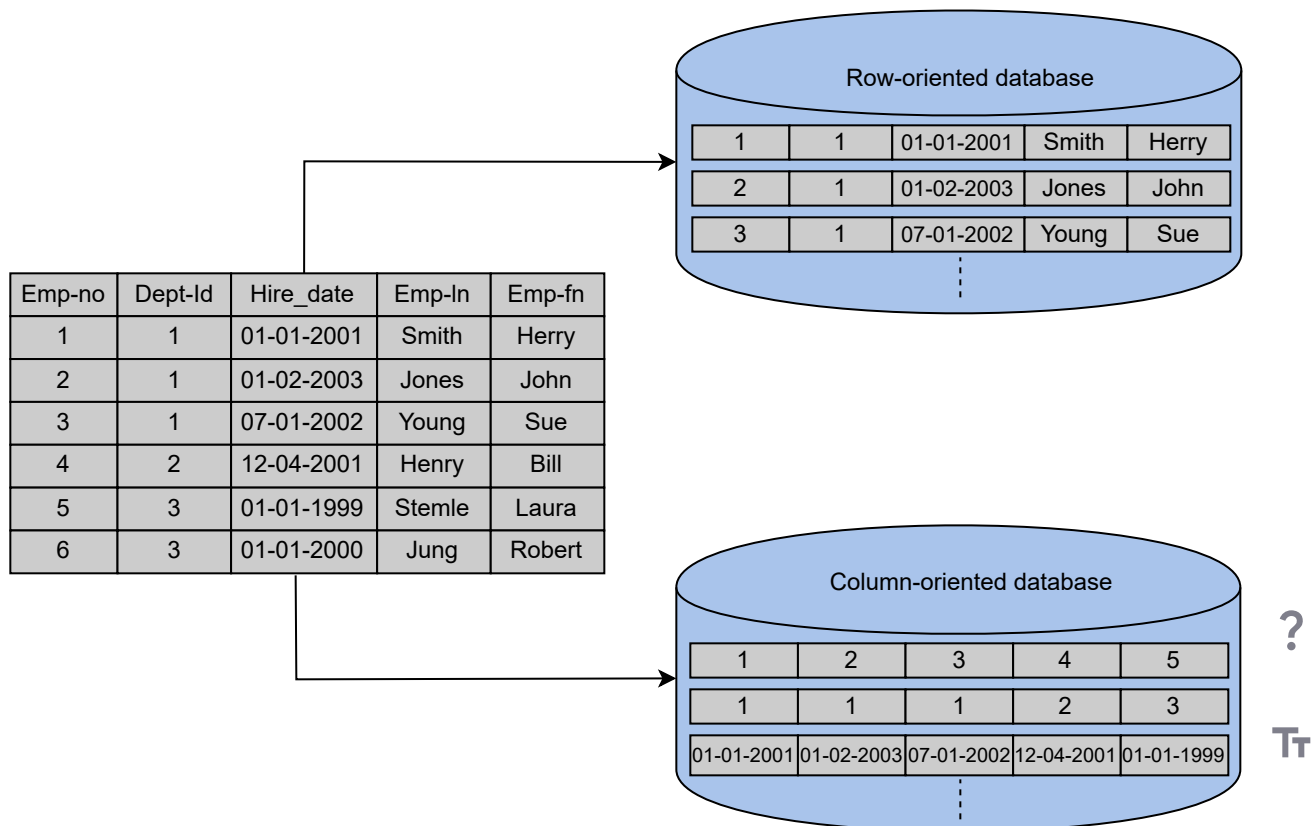
Use case: Graph databases can be used in social applications and provide interesting facts and figures among different kinds of users and their activities. The focus of graph databases is to store data and pave the way to drive analyses and decisions based on relationships between entities. The nature of graph databases makes them suitable for various applications, such as data regulation and privacy, machine learning research, financial services-based applications, and many more.

Columnar database

Columnar databases store data in columns instead of rows. They enable access to all entries in the database column quickly and efficiently. Popular columnar databases include Cassandra, HBase, Hypertable, and Amazon SimpleDB.

Use case: Columnar databases are efficient for a large number of aggregation and data analytics queries. It drastically reduces the disk I/O requirements and the amount of data required to load from the disk. For example, in applications related to financial institutions, there's a need to sum the financial transaction over a period of time. Columnar databases make this operation quicker by just reading the column for the amount of money, ignoring other attributes of customers.

The following figure shows an example of a columnar database, where data is stored in a column-oriented format. This is unlike relational databases, which store data in a row-oriented fashion:



Column-oriented and row-oriented database

Drawbacks of NoSQL databases

Lack of standardization

NoSQL doesn't follow any specific standard, like how relational databases follow relational algebra. Porting applications from one type of NoSQL database to another might be a challenge.

Consistency

NoSQL databases provide different products based on the specific trade-offs between consistency and availability when failures can happen. We won't have strong data integrity, like primary and referential integrities in a relational database. Data might not be strongly consistent but slowly converging using a weak model like eventual consistency.

Choose the right database

Various factors affect the choice of database to be used in an application. A comparison between the relational and non-relational databases is shown in the following table to help us choose:

Relational and Non-relational Databases

Relational Database	Non-relational Database
If the data to be stored is structured	If the data to be stored is unstructured
If ACID properties are required	If there's a need to serialize and deserialize data

?

Tt





Note: When NoSQL databases first came into being, they were drastically different to program and use as compared to traditional databases. Though, due to extensive research in academia and industry over the last many years, the programmer-facing differences between NoSQL and traditional stores are blurring. We might be using the same SQL constructs to talk to a NoSQL store and get a similar level of performance and consistency as a traditional store. Google's Cloud Spanner is one such database that's geo-replicated with automatic horizontal sharding ability and high-speed global snapshots of data.

Quiz

Test your knowledge of the different types of databases via a quiz.

1

Which database should we use when we have unstructured data and there's a need for high performance?

A) MongoDB

B) MySQL

C) Oracle

?

Tt





Data Replication

Understand the models through which data is replicated across several nodes.

We'll cover the following



- Replication
- Synchronous versus asynchronous replication
- Data replication models
 - Single leader/primary-secondary replication
 - Primary-secondary replication methods
 - Statement-based replication
 - Write-ahead log (WAL) shipping
 - Logical (row-based) log replication
 - Multi-leader replication
 - Conflict
 - Handle conflicts
 - Conflict avoidance
 - Last-write-wins
 - Custom logic
 - Multi-leader replication topologies
 - Peer-to-peer/leaderless replication
 - Quorums



Data is an asset for an organization because it drives the whole business. Data provides critical business insights into what's important and what needs to be changed. Organizations also need to securely save and serve



their clients' data on demand. Timely access to the required data under varying conditions (increasing reads and writes, disks and node failures, network and power outages, and so on) is required to successfully run an online business.

We need the following characteristics from our data store:

- Availability under faults (failure of some disk, nodes, and network and power outages).
- Scalability (with increasing reads, writes, and other operations).
- Performance (low latency and high throughput for the clients).

It's challenging, or even impossible, to achieve the above characteristics on a single node.

Replication

Replication refers to keeping multiple copies of the data at various nodes (preferably geographically distributed) to achieve availability, scalability, and performance. In this lesson, we assume that a single node is enough to hold our entire data. We won't use this assumption while discussing the partitioning of data in multiple nodes. Often, the concepts of replication and partitioning go together.

However, with many benefits, like availability, replication comes with its complexities. Replication is relatively simple if the replicated data doesn't require frequent changes. The main problem in replication arises when we have to maintain changes in the replicated data over time.

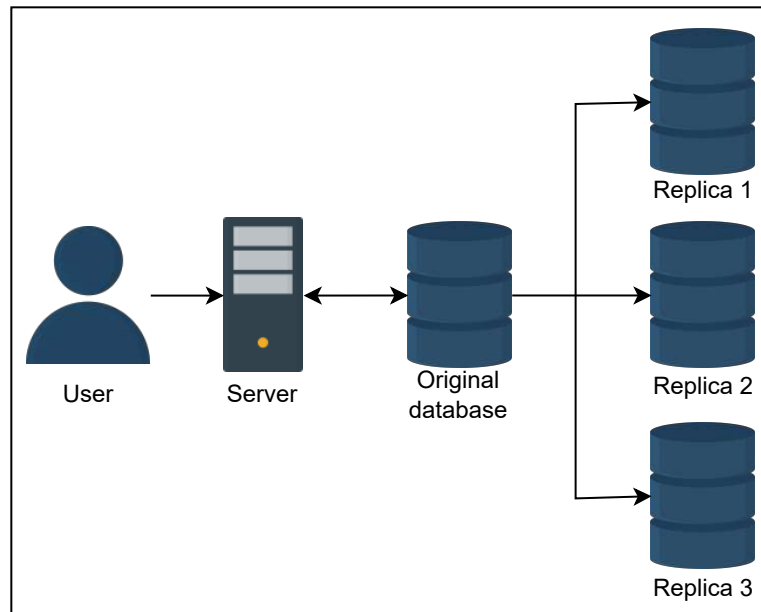
Additional complexities that could arise due to replication are as follows:

- How do we keep multiple copies of data consistent with each other?
- How do we deal with failed replica nodes?
- Should we replicate synchronously or asynchronously?



- How do we deal with replication lag in case of asynchronous replication?
- How do we handle concurrent writes?
- What consistency model needs to be exposed to the end programmers?

We'll explore the answer to these questions in this lesson.



Replication in action

Before we explain the different types of replication, let's understand the synchronous and asynchronous approaches of replication.

Synchronous versus asynchronous replication

There are two ways to disseminate changes to the replica nodes:

- Synchronous replication
- Asynchronous replication

In **synchronous replication**, the primary node waits for acknowledgments from secondary nodes about updating the data. After receiving acknowledgment from all secondary nodes, the primary node reports success to the client. Whereas in **asynchronous replication**, the primary

?

Tt

☾

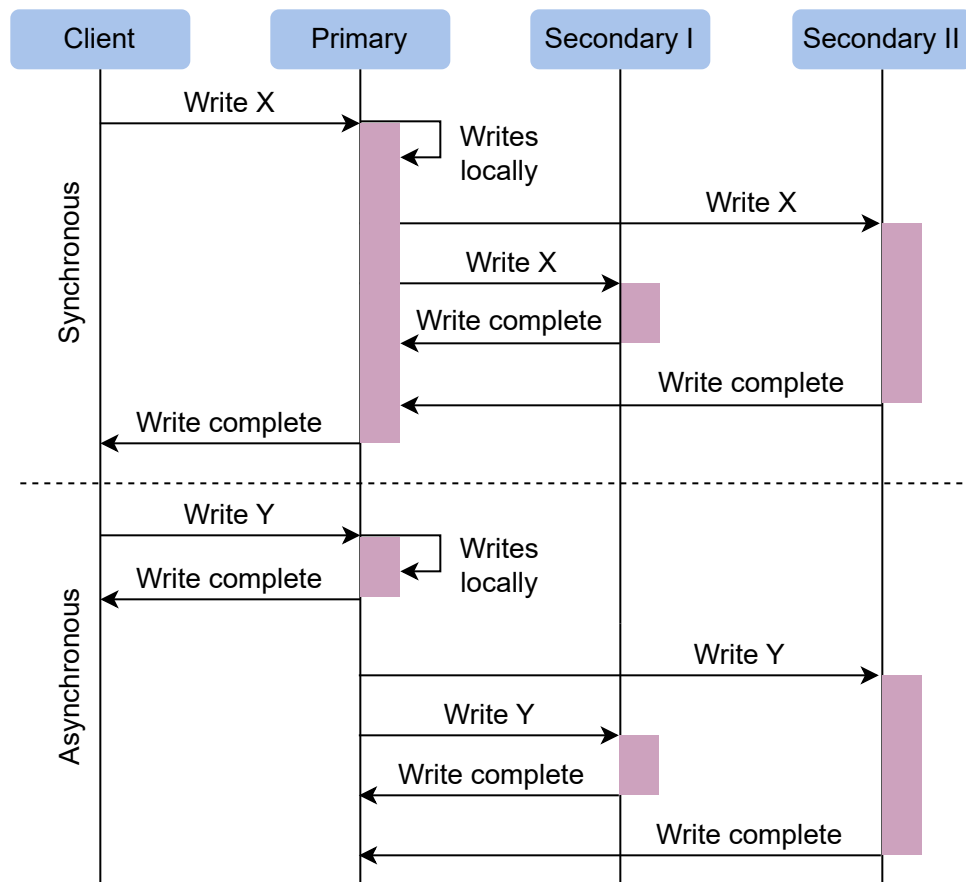
node doesn't wait for the acknowledgment from the secondary nodes and reports success to the client after updating itself.

The advantage of synchronous replication is that all the secondary nodes are completely up to date with the primary node. However, there's a disadvantage to this approach. If one of the secondary nodes doesn't acknowledge due to failure or fault in the network, the primary node would be unable to acknowledge the client until it receives the successful acknowledgment from the crashed node. This causes high latency in the response from the primary node to the client.

On the other hand, the advantage of asynchronous replication is that the primary node can continue its work even if all the secondary nodes are down. However, if the primary node fails, the writes that weren't copied to the secondary nodes will be lost.

The above paragraph explains a trade-off between data consistency and availability when different components of the system can fail.





Synchronous versus asynchronous replication

Data replication models

Now, let's discuss various mechanisms of data replication. In this section, we'll discuss the following models along with their strengths and weaknesses:

- Single leader or primary-secondary replication
- Multi-leader replication
- Peer-to-peer or leaderless replication

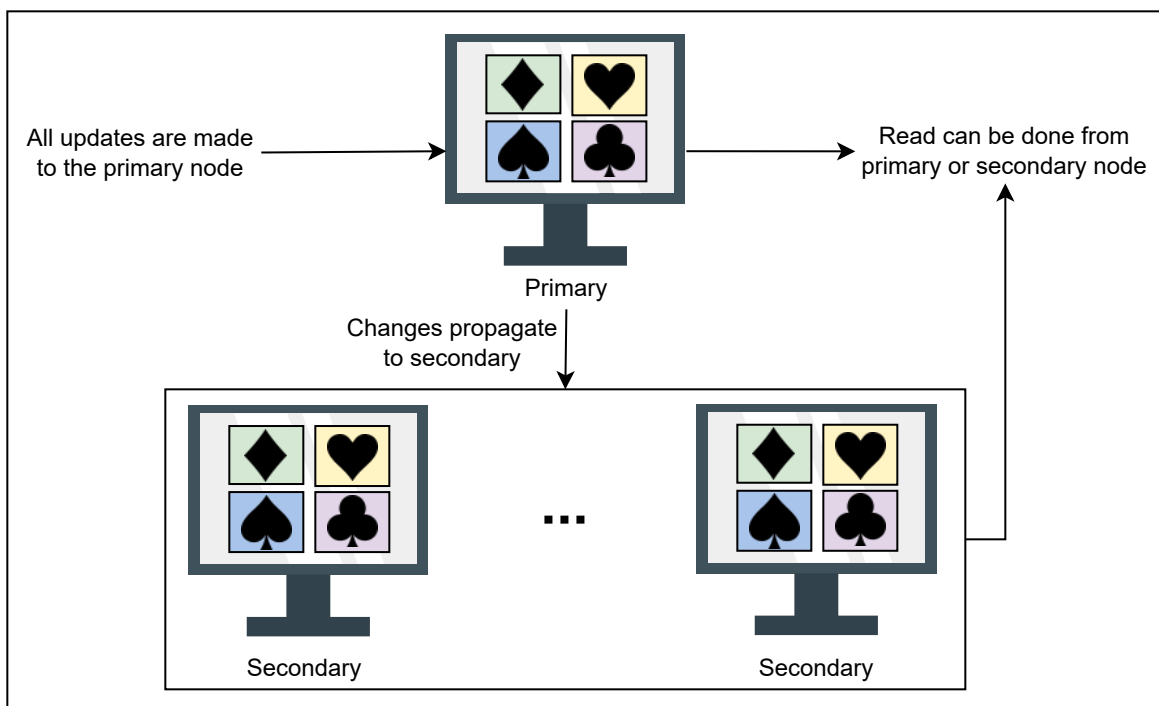
Single leader/primary-secondary replication

In **primary-secondary replication**, data is replicated across multiple nodes. One node is designated as the primary. It's responsible for processing any writes to data stored on the cluster. It also sends all the writes to the secondary nodes and keeps them in sync.

Primary-secondary replication is appropriate when our workload is read-heavy. To better scale with increasing readers, we can add more followers and distribute the read load across the available followers. However, replicating data to many followers can make a primary bottleneck. Additionally, primary-secondary replication is inappropriate if our workload is write-heavy.

Another advantage of primary-secondary replication is that it's read resilient. Secondary nodes can still handle read requests in case of primary node failure. Therefore, it's a helpful approach for read-intensive applications.

Replication via this approach comes with inconsistency if we use asynchronous replication. Clients reading from different replicas may see inconsistent data in the case of failure of the primary node that couldn't propagate updated data to the secondary nodes. So, if the primary node fails, any missed updates not passed on to the secondary nodes can be lost.



Primary-secondary data replication model where data is replicated from primary to secondary

Point to Ponder

Question

What happens when the primary node fails?

[Show Answer](#) ▼

Primary-secondary replication methods

There are many different replication methods in primary-secondary replication:

- Statement-based replication
- Write-ahead log (WAL) shipping
- Logical (row-based) log replication

Let's discuss each of them in detail.

Statement-based replication

In the **statement-based replication** approach, the primary node saves all statements that it executes, like insert, delete, update, and so on, and sends them to the secondary nodes to perform. This type of replication was used in MySQL before version 5.1. ?

This type of approach seems good, but it has its disadvantages. For example, any nondeterministic function (such as `NOW()`) might result in distinct writes on the follower and leader. Furthermore, if a write statement is dependent Tr

on a prior write, and both of them reach the follower in the wrong order, the outcome on the follower node will be uncertain.

Write-ahead log (WAL) shipping

In the **write-ahead log (WAL) shipping** approach, the primary node saves the query before executing it in a log file known as a write-ahead log file. It then uses these logs to copy the data onto the secondary nodes. This is used in PostgreSQL and Oracle. The problem with WAL is that it only defines data at a very low level. It's tightly coupled with the inner structure of the database engine, which makes upgrading software on the leader and followers complicated.

Logical (row-based) log replication

In the **logical (row-based) log replication** approach, all secondary nodes replicate the actual data changes. For example, if a row is inserted or deleted in a table, the secondary nodes will replicate that change in that specific table. The binary log records change to database tables on the primary node at the record level. To create a replica of the primary node, the secondary node reads this data and changes its records accordingly. Row-based replication doesn't have the same difficulties as WAL because it doesn't require information about data layout inside the database engine.

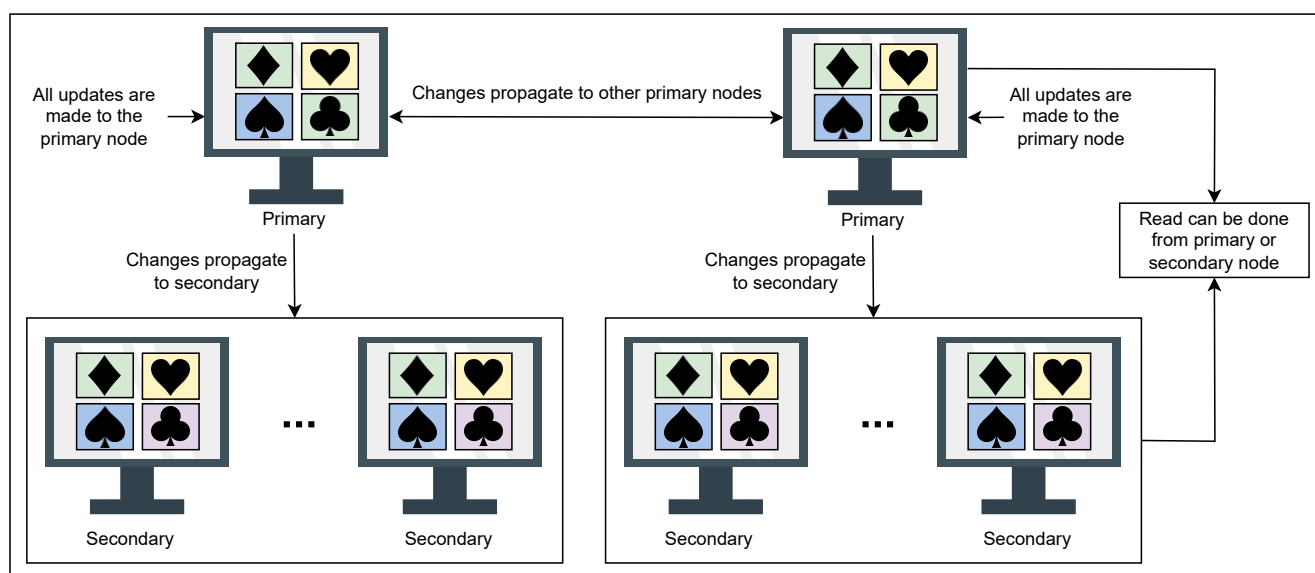
Multi-leader replication

As discussed above, single leader replication using asynchronous replication has a drawback. There's only one primary node, and all the writes have to go through it, which limits the performance. In case of failure of the primary node, the secondary nodes may not have the updated database. ?

Multi-leader replication is an alternative to single leader replication. There are multiple primary nodes that process the writes and send them to all other primary and secondary nodes to replicate. This type of replication is

used in databases along with external tools like the Tungsten Replicator for MySQL.

This kind of replication is quite useful in applications in which we can continue work even if we're offline—for example, a calendar application in which we can set our meetings even if we don't have access to the internet. Once we're online, it replicates its changes from our local database (our mobile phone or laptop acts as a primary node) to other nodes.



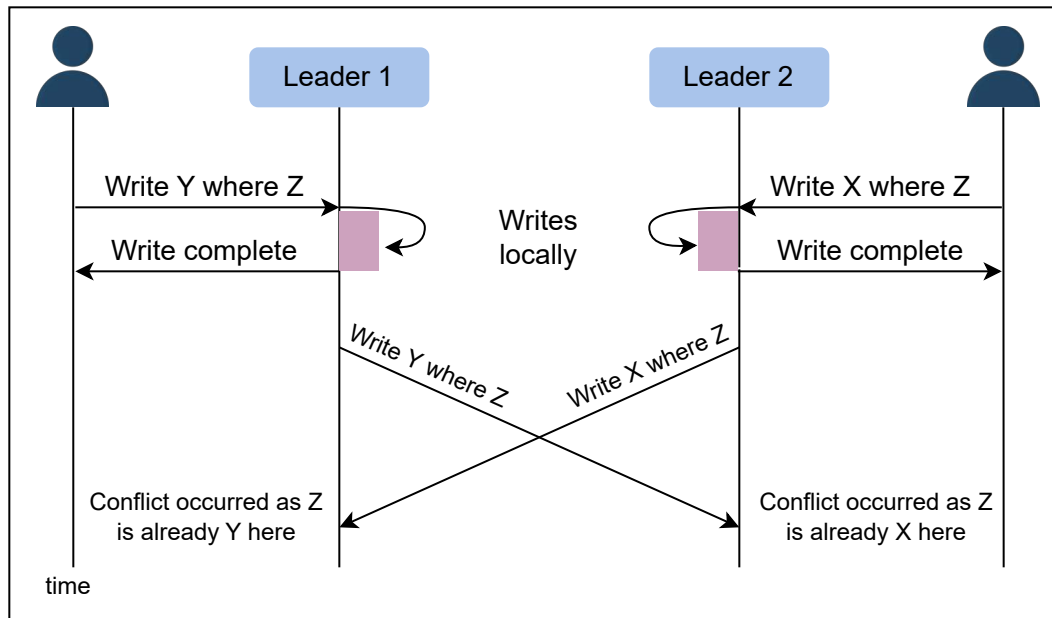
Multi-leader data replication model (all-to-all topology)

Conflict

Multi-leader replication gives better performance and scalability than single leader replication, but it also has a significant disadvantage. Since all the primary nodes concurrently deal with the write requests, they may modify the same data, which can create a conflict between them. For example, suppose the same data is edited by two clients simultaneously. In that case, their writes will be successful in their associated primary nodes, but when they reach the other primary nodes asynchronously, it creates a conflict.

Handle conflicts

Conflicts can result in different data at different nodes. These should be handled efficiently without losing any data. Let's discuss some of the approaches to handle conflicts:



Conflict of writes

Conflict avoidance

A simple strategy to deal with conflicts is to prevent them from happening in the first place. Conflicts can be avoided if the application can verify that all writes for a given record go via the same leader.

However, the conflict may still occur if a user moves to a different location and is now near a different data center. If that happens, we need to reroute the traffic. In such scenarios, the conflict avoidance approach fails and results in concurrent writes.

Last-write-wins

Using their local clock, all nodes assign a timestamp to each update. When a conflict occurs, the update with the latest timestamp is selected.

This approach can also create difficulty because the clock synchronization across nodes is challenging in distributed systems. There's clock skew that

can result in data loss.

Custom logic

In this approach, we can write our own logic to handle conflicts according to the needs of our application. This custom logic can be executed on both reads and writes. When the system detects a conflict, it calls our custom conflict handler.

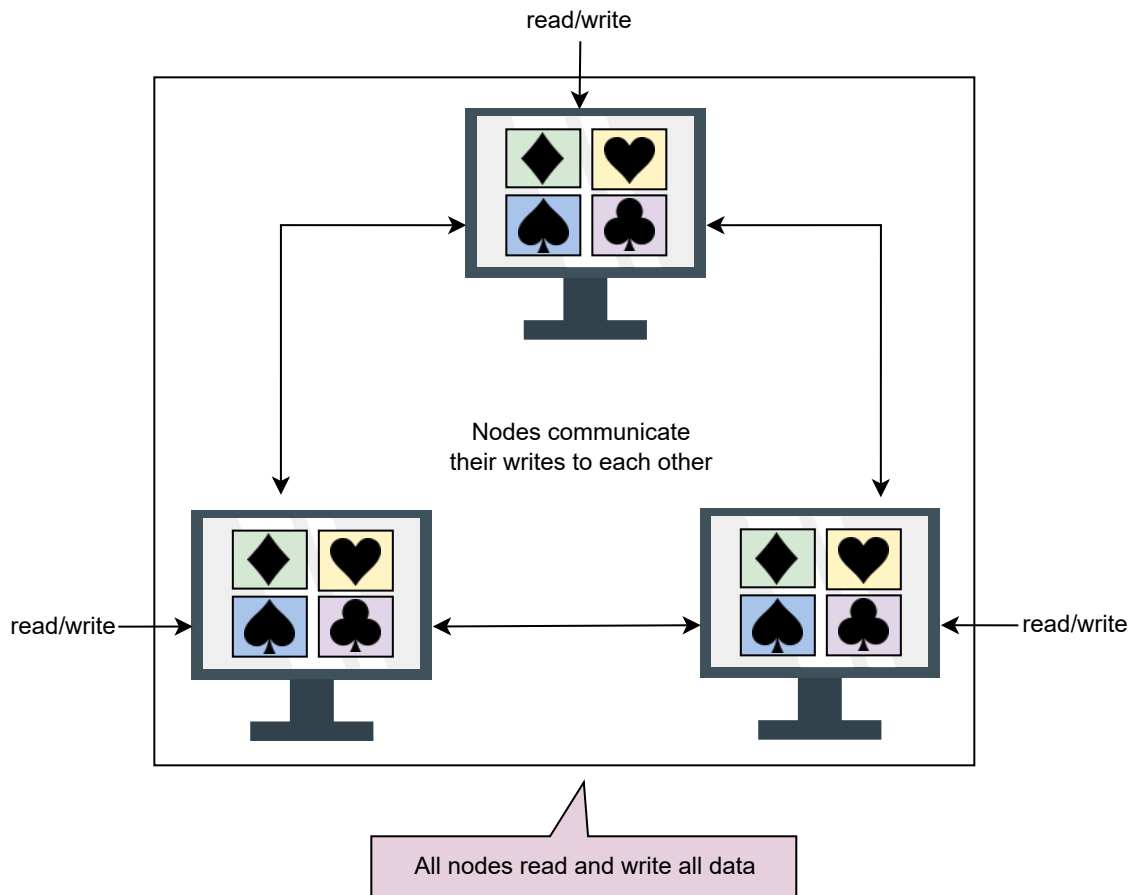
Multi-leader replication topologies

There are many topologies through which multi-leader replication is implemented, such as circular topology, star topology, and all-to-all topology. The most common is the all-to-all topology. In star and circular topology, there's again a similar drawback that if one of the nodes fails, it can affect the whole system. That's why all-to-all is the most used topology.

Peer-to-peer/leaderless replication

In primary-secondary replication, the primary node is a bottleneck and a single point of failure. Moreover, it helps to achieve read scalability but fails in providing write scalability. The **peer-to-peer replication** model resolves these problems by not having a single primary node. All the nodes have equal weightage and can accept reads and writes requests. Amazon popularized such a scheme in their DynamoDB data store.





Peer-to-peer data replication model where all nodes apply reads and writes to all the data

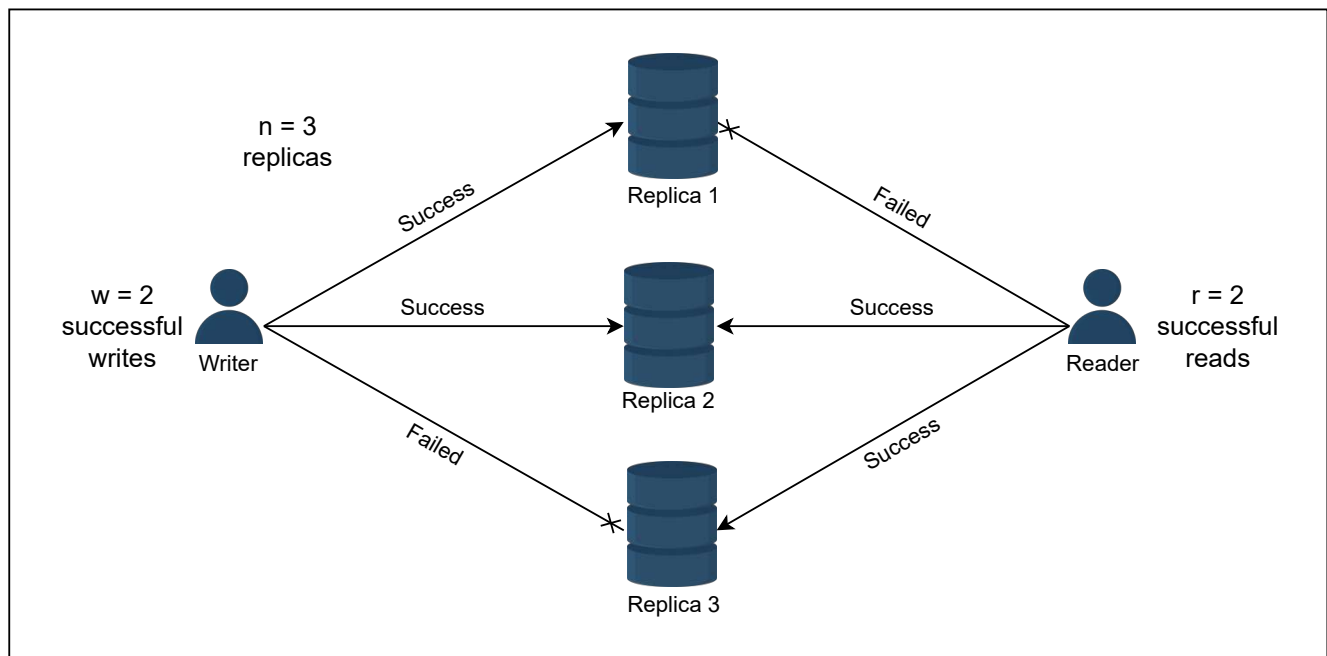
Like primary-secondary replication, this replication can also yield inconsistency. This is because when several nodes accept write requests, it may lead to concurrent writes. A helpful approach used for solving write-write inconsistency is called **quorums**.

Quorums

Let's suppose we have three nodes. If at least two out of three nodes are guaranteed to return successful updates, it means only one node has failed. This means that if we read from two nodes, at least one of them will have the updated version, and our system can continue working. ?

If we have n nodes, then every write must be updated in at least w nodes to be considered a success, and we must read from r nodes. We'll get an updated value from reading as long as $w + r > n$ because at least one of them will have the updated value. ☾

nodes must have an updated write from which we can read. Quorum reads and writes adhere to these r and w values. These n , w , and r are configurable in Dynamo-style databases.



Reader getting an updated value from replica 2

?

Tt





Data Partitioning

Learn about data partitioning models along with their pros and cons.

We'll cover the following



- Why do we partition data?
- Sharding
 - Vertical sharding
 - Horizontal sharding
 - Key-range based sharding
 - Advantages
 - Disadvantages
 - Hash-based sharding
 - Advantages
 - Disadvantages
 - Consistent hashing
 - Advantages of consistent hashing
 - Disadvantages of consistent hashing
- Rebalance the partitions
 - Avoid hash mod n
 - Fixed number of partitions
 - Dynamic partitioning
 - Partition proportionally to nodes
- Partitioning and secondary indexes
 - Partition secondary indexes by document
 - Partition secondary indexes by the term
- Request routing



- ZooKeeper
- Conclusion

Why do we partition data?

Data is an asset for any organization. Increasing data and concurrent read/write traffic to the data put scalability pressure on traditional databases. As a result, the latency and throughput are affected. Traditional databases are attractive due to their properties such as range queries, secondary indices, and transactions with the ACID properties.

At some point, a single node-based database isn't enough to tackle the load. We might need to distribute the data over many nodes but still export all the nice properties of relational databases. In practice, it has proved challenging to provide single-node database-like properties over a distributed database.

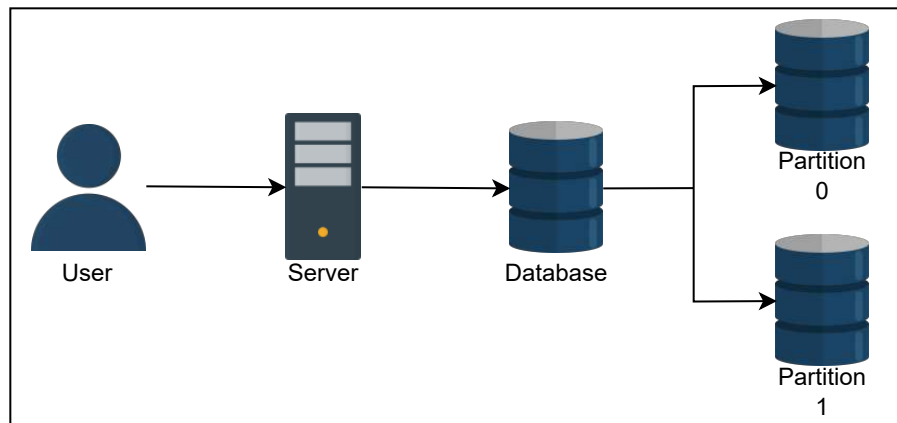
One solution is to move data to a NoSQL-like system. However, the historical codebase and its close cohesion with traditional databases make it an expensive problem to tackle.

Organizations might scale traditional databases by using a third-party solution. But often, integrating a third-party solution has its complexities. More importantly, there are abundant opportunities to optimize for the specific problem at hand and get much better performance than a general-purpose solution.

Data partitioning (or sharding) enables us to use multiple nodes where each node manages some part of the whole data. To handle increasing query rate and data amounts, we strive for balanced partitions and balanced read/write load.



We'll discuss different ways to partition data, related challenges, and their solutions in this lesson.



A database with two partitions to distribute the data and associated read/write load

Sharding

To divide load among multiple nodes, we need to partition the data by a phenomenon known as **partitioning** or **sharding**. In this approach, we split a large dataset into smaller chunks of data stored at different nodes on our network.

The partitioning must be balanced so that each partition receives about the same amount of data. If partitioning is unbalanced, the majority of queries will fall into a few partitions. Partitions that are heavily loaded will create a system bottleneck. The efficacy of partitioning will be harmed because a significant portion of data retrieval queries will be sent to the nodes that carry the highly congested partitions. Such partitions are known as hotspots. Generally, we use the following two ways to shard the data:

- Vertical sharding
- Horizontal sharding

?

Tt

☾

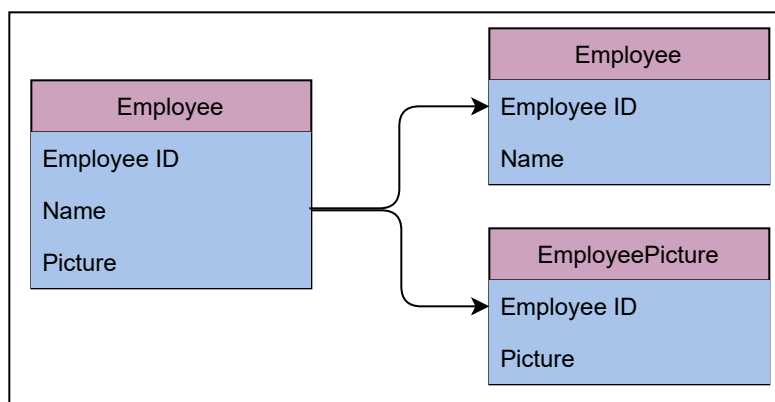
Vertical sharding

We can put different tables in various database instances, which might be running on a different physical server. We might break a table into multiple tables so that some columns are in one table while the rest are in the other. We should be careful if there are joins between multiple tables. We may like to keep such tables together on one shard.

Often, **vertical sharding** is used to increase the speed of data retrieval from a table consisting of columns with very wide text or a binary large object (blob). In this case, the column with large text or a blob is split into a different table.

As shown in the figure a couple paragraphs below, the **Employee** table is divided into two tables: a reduced **Employee** table and an **EmployeePicture** table. The **EmployeePicture** table has just two columns, **EmployeeID** and **Picture**, separated from the original table. Moreover, the primary key **EmployeeID** of the **Employee** table is added in both partitioned tables. This makes the data read and write easier, and the reconstruction of the table is performed efficiently.

Vertical sharding has its intricacies and is more amenable to manual partitioning, where stakeholders carefully decide how to partition data. In comparison, horizontal sharding is suitable to automate even under dynamic conditions.



Vertical partitioning

Note: Creating shards by moving specific tables of a database around is also a form of vertical sharding. Usually, those tables are put in the same shard because they often appear together in queries, for example, for joins. We will see an example of such a use-case [ahead in the course](#).

Horizontal sharding

At times, some tables in the databases become too big and affect read/write latency. **Horizontal sharding** or partitioning is used to divide a table into multiple tables by splitting data row-wise, as shown in the figure in the next section. Each partition of the original table distributed over database servers is called a **shard**. Usually, there are two strategies available:

- Key-range based sharding
- Hash based sharding

Key-range based sharding

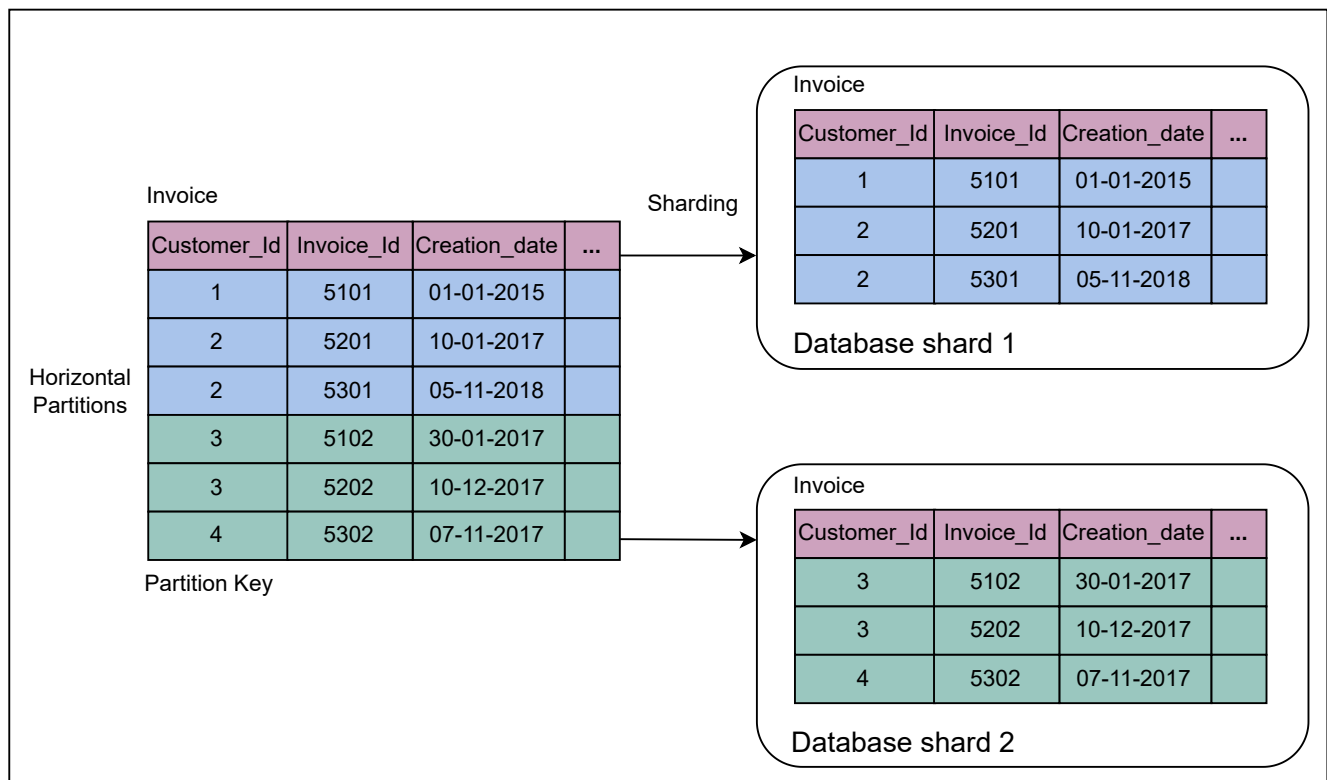
In the **key-range based sharding**, each partition is assigned a continuous range of keys.

In the following figure, horizontal partitioning on the **Invoice** table is performed using the key-range based sharding with **Customer_Id** as the partition key. The two different colored tables represent the partitions.

?

Tr





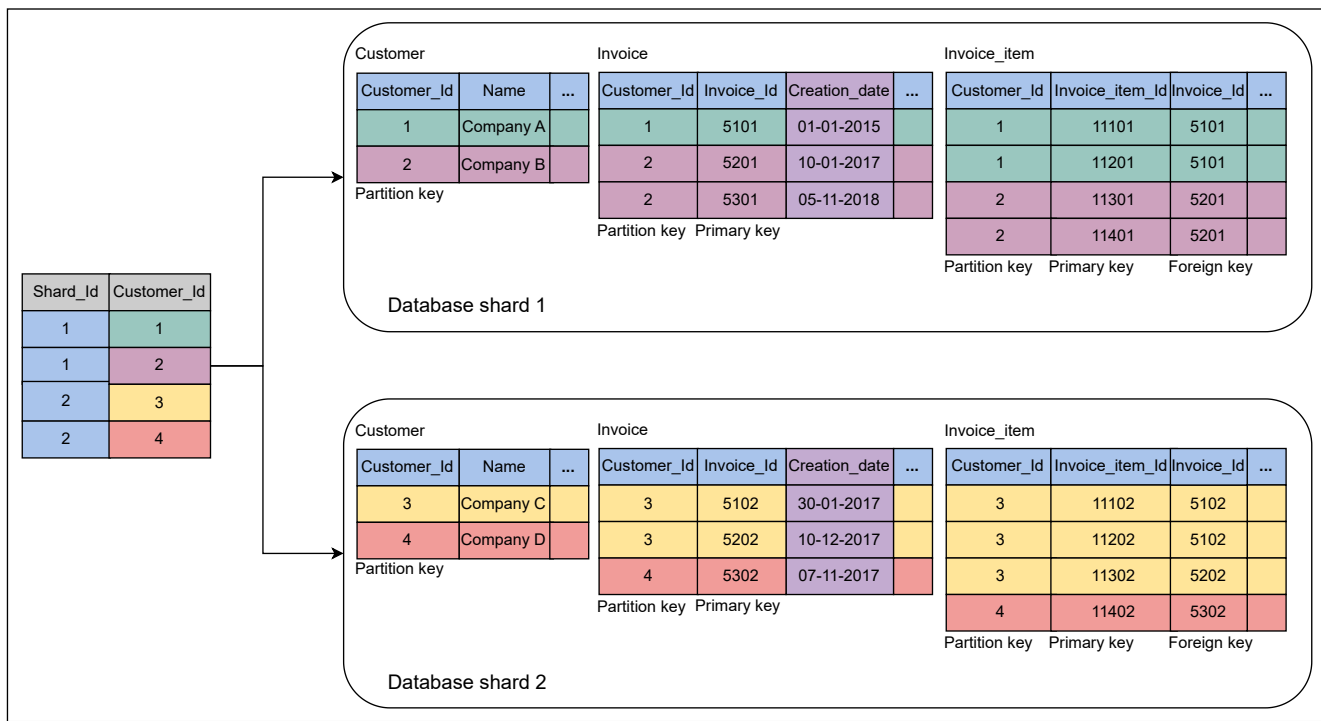
Horizontal partitioning

Sometimes, a database consists of multiple tables bound by foreign key relationships. In such a case, the horizontal partition is performed using the same partition key on all tables in a relation. Tables (or subtables) that belong to the same partition key are distributed to one database shard. The following figure shows that several tables with the same partition key are placed in a single database shard:

?

Tt





Horizontal partitioning on a set of tables

The basic design techniques used in multi-table sharding are as follows:

- There's a partition key in the **Customer** mapping table. This table resides on each shard and stores the partition keys used in the shard. Applications create a mapping logic between the partition keys and database shards by reading this table from all shards to make the mapping efficient. Sometimes, applications use advanced algorithms to determine the location of a partition key belonging to a specific shard.
- The partition key column, **Customer_Id**, is replicated in all other tables as a data isolation point. It has a trade-off between an impact on increased storage and locating the desired shards efficiently. Apart from this, it's helpful for data and workload distribution to different database shards. The data routing logic uses the partition key at the application tier to map queries specified for a database shard.
- Primary keys are unique across all database shards to avoid key collision during data migration among shards and the merging of data in the online analytical processing (OLAP) environment.

- The column `Creation_date` serves as the data consistency point, with an assumption that the clocks of all nodes are synchronized. This column is used as a criterion for merging data from all database shards into the global view when essential.

Advantages

- Using this method, the range-query-based scheme is easy to implement.
- Range queries can be performed using the partitioning keys, and those can be kept in partitions in sorted order.

Disadvantages

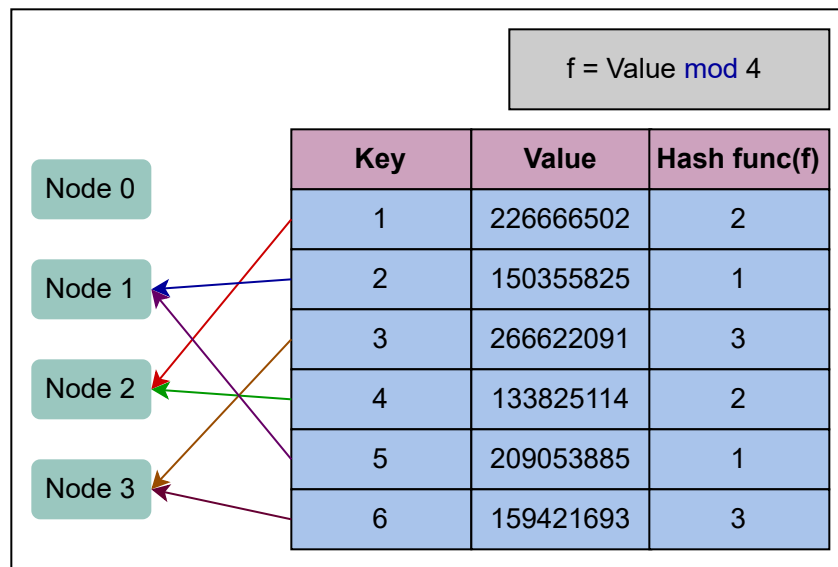
- Range queries can't be performed using keys other than the partitioning key.
- If keys aren't selected properly, some nodes may have to store more data due to an uneven distribution of the traffic.

Hash-based sharding

Hash-based sharding uses a hash-like function on an attribute, and it produces different values based on which attribute the partitioning is performed. The main concept is to use a hash function on the key to get a hash value and then mod by the number of partitions. Once we've found an appropriate hash function for keys, we may give each partition a range of hashes (rather than a range of keys). Any key whose hash occurs inside that range will be kept in that partition.

In the illustration below, we use a hash function of $Value \bmod n$. The n is the number of nodes, which is four. We allocate keys to nodes by checking the mod for each key. Keys with a mod value of 2 are allocated to node 2. Keys with a mod value of 1 are allocated to node 1. Keys with a mod value of 3 are allocated to node 3. Because there's no key with a mod value of 0, node 0 is left vacant.





Hash-based sharding

Advantages

- Keys are uniformly distributed across the nodes.

Disadvantages

- We can't perform range queries with this technique. Keys will be spread over all partitions.

Note: How many shards per database?

Empirically, we can determine how much each node can serve with acceptable performance. It can help us find out the maximum amount of data that we would like to keep on any one node. For example, if we find out that we can put a maximum of 50 GB of data on one node, we have the following:

Database size = 10 TB

Size of a single shard = 50 GB

Number of shards the database should be distributed in = $10 \text{ TB} / 50 \text{ GB} = 200 \text{ shards}$

?

Tt

C

Consistent hashing

Consistent hashing assigns each server or item in a distributed hash table a place on an abstract circle, called a ring, irrespective of the number of servers in the table. This permits servers and objects to scale without compromising the system's overall performance.

Advantages of consistent hashing

- It's easy to scale horizontally.
- It increases the throughput and improves the latency of the application.

Disadvantages of consistent hashing

- Randomly assigning nodes in the ring may cause non-uniform distribution.

Rebalance the partitions

Query load can be imbalanced across the nodes due to many reasons, including the following:

- The distribution of the data isn't equal.
- There's too much load on a single partition.
- There's an increase in the query traffic, and we need to add more nodes to keep up.

We can apply the following strategies to rebalance partitions.

Avoid hash mod n

Usually, we avoid the hash of a key for partitioning (we used such a scheme to explain the concept of hashing in simple terms earlier). The problem with the addition or removal of nodes in the case of *hashmodn* is that every node's partition number changes and a lot of data moves. For example, assume we have $hash(key) = 1235$. If we have five nodes at the start, the

key will start on node 1 ($1235 \bmod 5 = 0$). Now, if a new node is added, the key would have to be moved to node 6 ($1235 \bmod 6 = 5$), and so on. This moving of keys from one node to another makes rebalancing costly.

Fixed number of partitions

In this approach, the number of partitions to be created is fixed at the time when we set our database up. We create a higher number of partitions than the nodes and assign these partitions to nodes. So, when a new node is added to the system, it can take a few partitions from the existing nodes until the partitions are equally divided.

There's a downside to this approach. The size of each partition grows with the total amount of data in the cluster since all the partitions contain a small part of the total data. If a partition is very small, it will result in too much overhead because we may have to make a large number of small-sized partitions, each costing us some overhead. If the partition is very large, rebalancing the nodes and recovering from node failures will be expensive. It's very important to choose the right number of partitions. A fixed number of partitions is used in Elasticsearch, Riak, and many more.

Dynamic partitioning

In this approach, when the size of a partition reaches the threshold, it's split equally into two partitions. One of the two split partitions is assigned to one node and the other one to another node. In this way, the load is divided equally. The number of partitions adapts to the overall data amount, which is an advantage of dynamic partitioning.

However, there's a downside to this approach. It's difficult to apply dynamic rebalancing while serving the reads and writes. This approach is used in HBase and MongoDB.

Partition proportionally to nodes

In this approach, the number of partitions is proportionate to the number of nodes, which means every node has fixed partitions. In earlier approaches, the number of partitions was dependent on the size of the dataset. That isn't the case here. While the number of nodes remains constant, the size of each partition rises according to the dataset size. However, as the number of nodes increases, the partitions shrink. When a new node enters the network, it splits a certain number of current partitions at random, then takes one half of the split and leaves the other half alone. This can result in an unfair split. This approach is used by Cassandra and Ketama.

Point to Ponder

Question

Who performs the rebalancing? Is it automatic or manual?

Show Answer ▼

Partitioning and secondary indexes

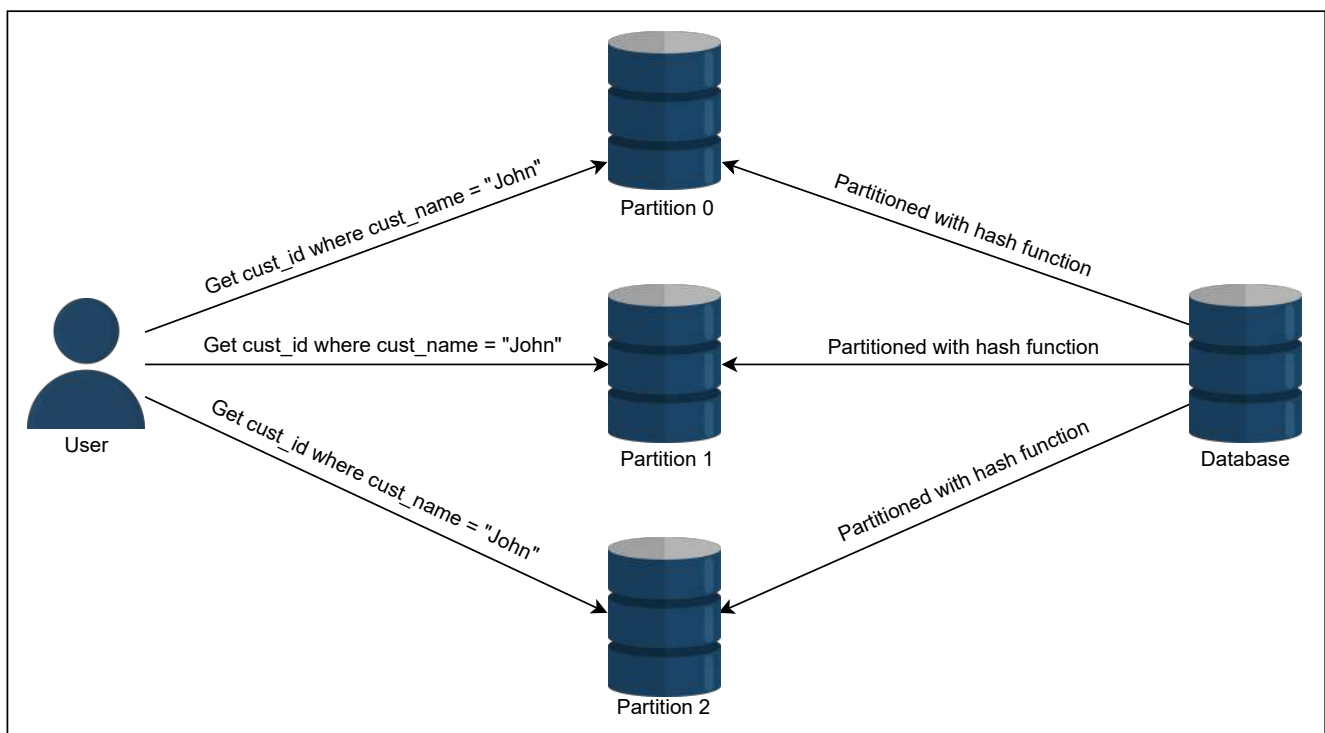
We've discussed key-value data model partitioning schemes in which the records are retrieved with primary keys. But what if we have to access the records through secondary indexes? Secondary indexes are the records that aren't identified by primary keys but are just a way of searching for some value. For example, the above [illustration of horizontal partitioning](#) contains the customer table, searching for all customers with the same creation year.

We can partition with secondary indexes in the following ways.

Partition secondary indexes by document

Each partition is fully independent in this indexing approach. Each partition has its secondary indexes covering just the documents in that partition. It's unconcerned with the data held in other partitions. If we want to write anything to our database, we need to handle that partition only containing the document ID we're writing. It's also known as the local index. In the illustration below, there are three partitions, each having its own identity and data. If we want to get all the customer IDs with the name **John**, we have to request from all partitions.

However, this type of querying on secondary indexes can be expensive. As a result of being restricted by the latency of a poor-performing partition, read query latencies may increase.



Partitioning secondary indexes by document

?

Partition secondary indexes by the term

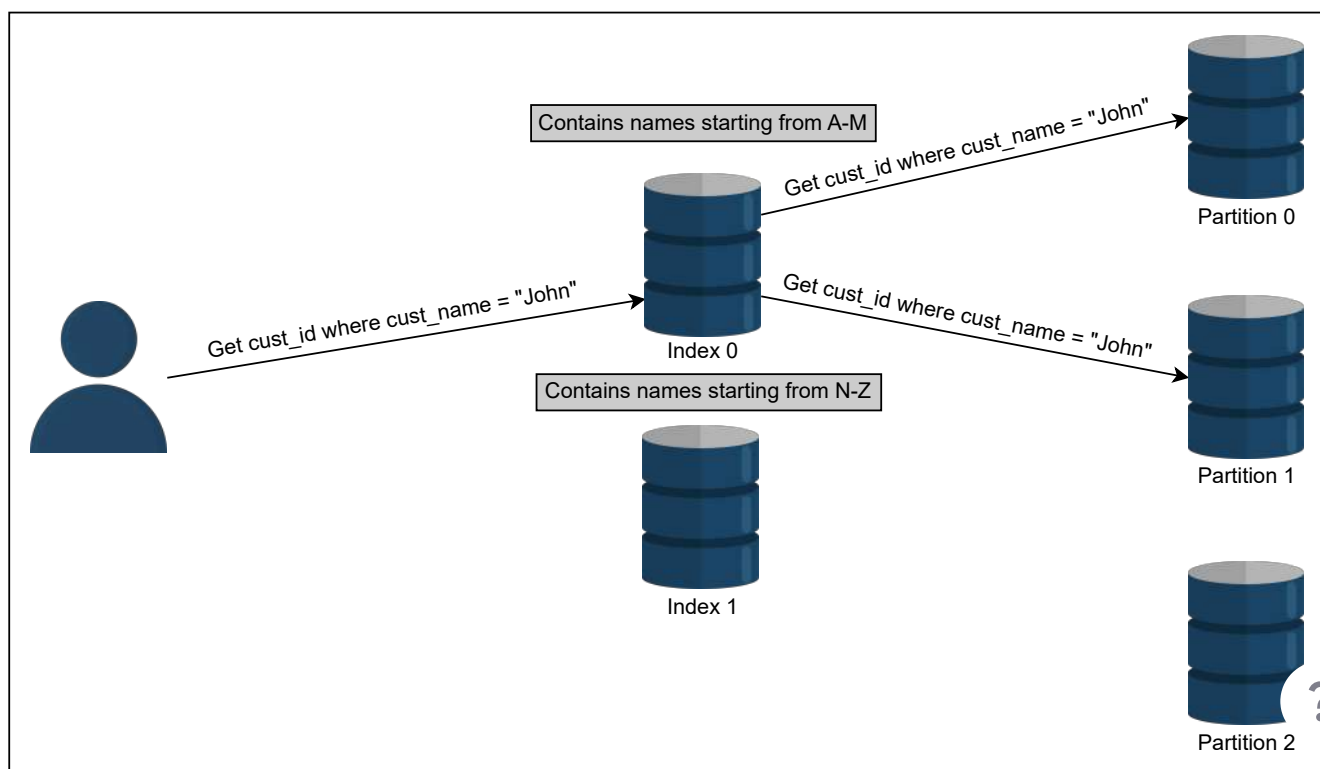
Tt

Instead of creating a secondary index for each partition (a local index), we can make a global index for secondary terms that encompasses data from all

partitions.

In the illustration below, we create indexes on names (the term on which we're partitioning) and store all the indexes for names on separated nodes. To get the `cust_id` of all the customers named `John`, we must determine where our term index is located. The `index 0` contains all the customers with names starting with "A" to "M." The `index 1` includes all the customers with names beginning with "N" to "Z." Because `John` lies in `index 0`, we fetch a list of `cust_id` with the name `John` from `index 0`.

Partitioning secondary indexes by the term is more read-efficient than partitioning secondary indexes by the document. This is because it only accesses the partition that contains the term. However, a single write in this approach affects multiple partitions, making the method write-intensive and complex.



Partitioning secondary indexes by term

Request routing

We've learned how to partition our data. However, one question arises here: How does a client know which node to connect to while making a request? The allocation of partitions to nodes varies after rebalancing. If we want to read a specific key, how do we know which IP address we need to connect to read?

This problem is also known as **service discovery**. Following are a few approaches to this problem:

- Allow the clients to request any node in the network. If that node doesn't contain the requested data, it forwards that request to the node that does contain the related data.
- The second approach contains a routing tier. All the requests are first forwarded to the routing tier, and it determines which node to connect to fulfill the request.
- The clients already have the information related to partitioning and which partition is connected to which node. So, they can directly contact the node that contains the data they need.

In all of these approaches, the main challenge is to determine how these components know about updates in the partitioning of the nodes.

ZooKeeper

To track changes in the cluster, many distributed data systems need a





Trade-offs in Databases

Learn when to use horizontal sharding instead of vertical sharding and vice versa.

We'll cover the following



- Which is the best database sharding approach?
 - Advantages and disadvantages of a centralized database
 - Advantages
 - Disadvantages
 - Advantages and disadvantages of a distributed database
 - Advantages
 - Disadvantages
 - Query optimization and processing speed in a distributed database
 - Parameters assumption
 - Possible approaches
 - Conclusion

Which is the best database sharding approach?

Both horizontal and vertical sharding involve adding resources to our computing infrastructure. Our business stakeholders must decide which is suitable for our organization. We must scale our resources accordingly for our organization and business to grow, to prevent downtime, and to reduce latency. We can scale these resources through a combination of adjustments to CPU, physical memory requirements, hard disk adjustments, and network bandwidth.

The following sections explain the pros and cons of no sharding versus sharding.

Advantages and disadvantages of a centralized database

Advantages

- Data maintenance, such as updating and taking backups of a centralized database, is easy.
- Centralized databases provide stronger consistency and ACID transactions than distributed databases.
- Centralized databases provide a much simpler programming model for the end programmers as compared to distributed databases.
- It's more efficient for businesses to have a small amount of data to store that can reside on a single node.

Disadvantages

- A centralized database can slow down, causing high latency for end users, when the number of queries per second accessing the centralized database is approaching single-node limits.
- A centralized database has a single point of failure. Because of this, its probability of not being accessible is much higher.

Advantages and disadvantages of a distributed database

Advantages

- It's fast and easy to access data in a distributed database because data is retrieved from the nearest database shard or the one frequently used.
- Data with different **levels of distribution transparency** ↴ can be stored in separate places. ?
- Intensive transactions consisting of queries can be divided into multiple optimized subqueries, which can be processed in a parallel fashion. Tr ↻

Disadvantages

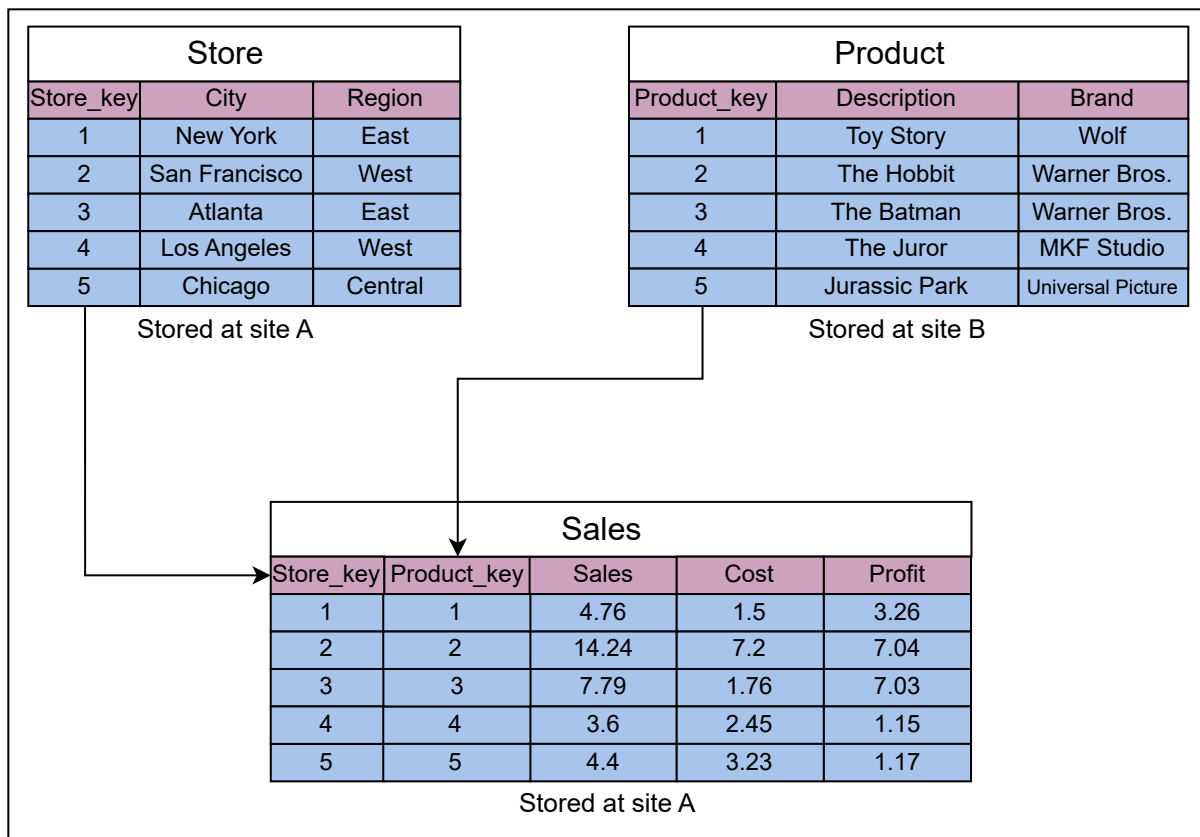
- Sometimes, data is required from multiple sites, which takes more time than expected.
- Relations are partitioned vertically or horizontally among different nodes. Therefore, operations such as joins need to reconstruct complete relations by carefully fetching data. These operations can become much more expensive and complex.
- It's difficult to maintain consistency of data across sites in the distributed database, and it requires extra measures.
- Updates and backups in distributed databases take time to synchronize data.

Query optimization and processing speed in a distributed database

A transaction in the distributed database depends on the type of query, number of sites (shards) involved, communication speed, and other factors, such as underline hardware and the type of database used. However, as an example, let's assume a query accessing three tables, **Store**, **Product**, and **Sales**, residing on different sites.

The number of attributes in each table is given in the following figure:





Database schema consisting of three tables: Store, Product, and Sales

Let's assume the distribution of both tables on different sites is the following:

- The **Store** table has 10,000 tuples stored at site A.
- The **Product** table has 100,000 tuples stored at site B.
- The **Sales** table has one million tuples stored at site A.

Now, assume that we need to process the following query:

```
Select Store_key from (Store JOIN Sales JOIN Product)
where Region= 'East' AND Brand='Wolf';
```

The above query performs the join operations on the **Store**, **Sales**, and **Product** tables and retrieves the **Store_key** values from the table generated by the result of join operations.

Next, assume every stored tuple is 200 bits long. That's equal to 25 Bytes.

Furthermore, estimated cardinalities of certain intermediate results are as

follows:

- The number of the **Wolf** brand is 10.
- The number of **East** region stores is 100,000.

Communication assumptions are the following:

- Data rate = 50M bits per second
- Access delay = 0.1 second

Parameters assumption

Before processing the query using different approaches, let's define some parameters:

a = Total access delay

b = Data rate

v = Total data volume

Now, let's compute the total communication time, T , according to the following formula:

$$T = a + \frac{v}{b}$$

Let's try the following possible approaches to execute the query.

Possible approaches

- Move the **Product** table to site A and process the query at A.

$$T = 0.1 + \frac{100,000 \times 200}{50,000,000} = 0.5 \text{ seconds}$$

Here, 0.1 is the access delay of the table on site A, and 100,000 is the number of tuples in the **Product** table. The size of each tuple in bits is

?

T

C

200, and 50,000,000 is the data rate. The 200 and 50,000,000 figures are the same for all of the following calculations.

- Move **Store** and **Sales** to site B and process the query at B:

$$T = 0.2 + \frac{(10,000 + 1,000,000) \times 200}{50,000,000} = 4.24 \text{ seconds}$$

Here, 0.2 is the access delay of the **Store** and **Product** tables. The numbers 10,000 and 1,000,000 are the number of tuples in the **Store** and **Product** tables, respectively.

- Restrict **Brand** at site B to **Wolf** (called selection) and move the result to site A:

$$T = 0.1 + \frac{10 \times 200}{50,000,000} \approx 0.1 \text{ seconds}$$

Here, 0.1 is the access delay of the **Product** table. The number of the **Wolf** brand is 10, hence the number of tuples.

When we compare the three approaches, the third approach provides us the least latency (0.1 seconds). This example shows that careful query optimization is also critical in the distributed database.

Conclusion

Data distribution (vertical and horizontal sharding) across multiple nodes aims to improve the following features, considering that the queries are optimized:

- Reliability (fault-tolerance)
- Performance
- Balanced storage capacity and dollar costs





Design of a Key-value Store

Learn about the functional and non-functional requirements and the API design of a key-value store.

We'll cover the following



- Requirements
 - Functional requirements
 - Non-functional requirements
- Assumptions
- API design
 - Data type

Requirements

Let's list the requirements of designing a key-value store to overcome the problems of traditional databases.

Functional requirements

The functional requirements are as follows:

- **Configurable service:** Some applications might have a tendency to trade strong consistency for higher availability. We need to provide a configurable service so that different applications could use a range of consistency models. We need tight control over the trade-offs between availability, consistency, cost-effectiveness, and performance.



- **Ability to always write:** The applications should always have the ability to write into the key-value storage. If the user wants strong consistency, this requirement might not always be fulfilled due to the implications of the CAP theorem.
- **Hardware heterogeneity:** The system shouldn't have distinguished nodes. Each node should be functionally able to do any task. Though servers can be heterogeneous, newer hardware might be more capable than older ones.

Non-functional requirements

The non-functional requirements are as follows:

- **Scalable:** Key-value stores should run on tens of thousands of servers distributed across the globe. Incremental scalability is highly desirable. We should add or remove the servers as needed with minimal to no disruption to the service availability. Moreover, our system should be able to handle an enormous number of users of the key-value store.
- **Available:** We need to provide continuous service, so availability is very important. This property is configurable. So, if the user wants strong consistency, we'll have less availability and vice versa.
- **Fault tolerance:** The key-value store should operate uninterrupted despite failures in servers or their components.

Point to Ponder

Question

Why do we need to run key-value stores on multiple servers?



[Show Answer](#) ▼

Assumptions

We'll assume the following to keep our design simple:

- The data centers hosting the service are trusted (non-hostile).
- All the required authentication and authorization are already completed.
- User requests and responses are relayed over HTTPS.

API design

Key-value stores, like ordinary hash tables, provide two primary functions, which are **get** and **put**.

Let's look at the API design.

The **get** function

The API call to get a value should look like this:

```
get(key)
```

We return the associated value on the basis of the parameter **key**. When data is replicated, it locates the object replica associated with a specific key that's hidden from the end user. It's done by the system if the store is configured with a weaker data consistency model. For example, in eventual consistency, there might be more than one value returned against a key.

?

Tt

☾

Parameter	Description
-----------	-------------



key	It's the key against which we want to get value .
-----	---

The put function

The API call to put the value into the system should look like this:

```
put(key, value)
```

It stores the value associated with the key. The system automatically determines where data should be placed. Additionally, the system often keeps metadata about the stored object. Such metadata can include the version of the object.

key	It's the key against which we have to store value .
value	It's the object to be stored against the key .

Point to Ponder

Question

We often keep hashes of the value (and at times, value + associated key) as metadata for data integrity checks. Should such a hash be taken after any data compression or encryption, or should it be taken before?



[Show Answer](#) 

Data type

The key is often a primary key in a key-value store, while the value can be any arbitrary binary data.

Note: Dynamo uses MD5 hashes on the key to generate a 128-bit identifier. These identifiers help the system determine which server node will be responsible for this specific key.

In the next lesson, we'll learn how to design our key-value store. First, we'll focus on adding scalability, replication, and versioning of our data to our system. Then, we'll ensure the functional requirements and make our system fault tolerant. We'll fulfill a few of our non-functional requirements first because implementing our functional requirements depends on the method chosen for scalability.

Note: This chapter is based on Dynamo, which is an influential work in the domain of key-value stores.

[← Back](#)[Next →](#)

System Design: The Key-value Store

Ensure Scalability and Replicat ☒ Mark as Completed





Ensure Scalability and Replication

Learn how consistent hashing enables scalability and how we replicate such partitioned data.

We'll cover the following



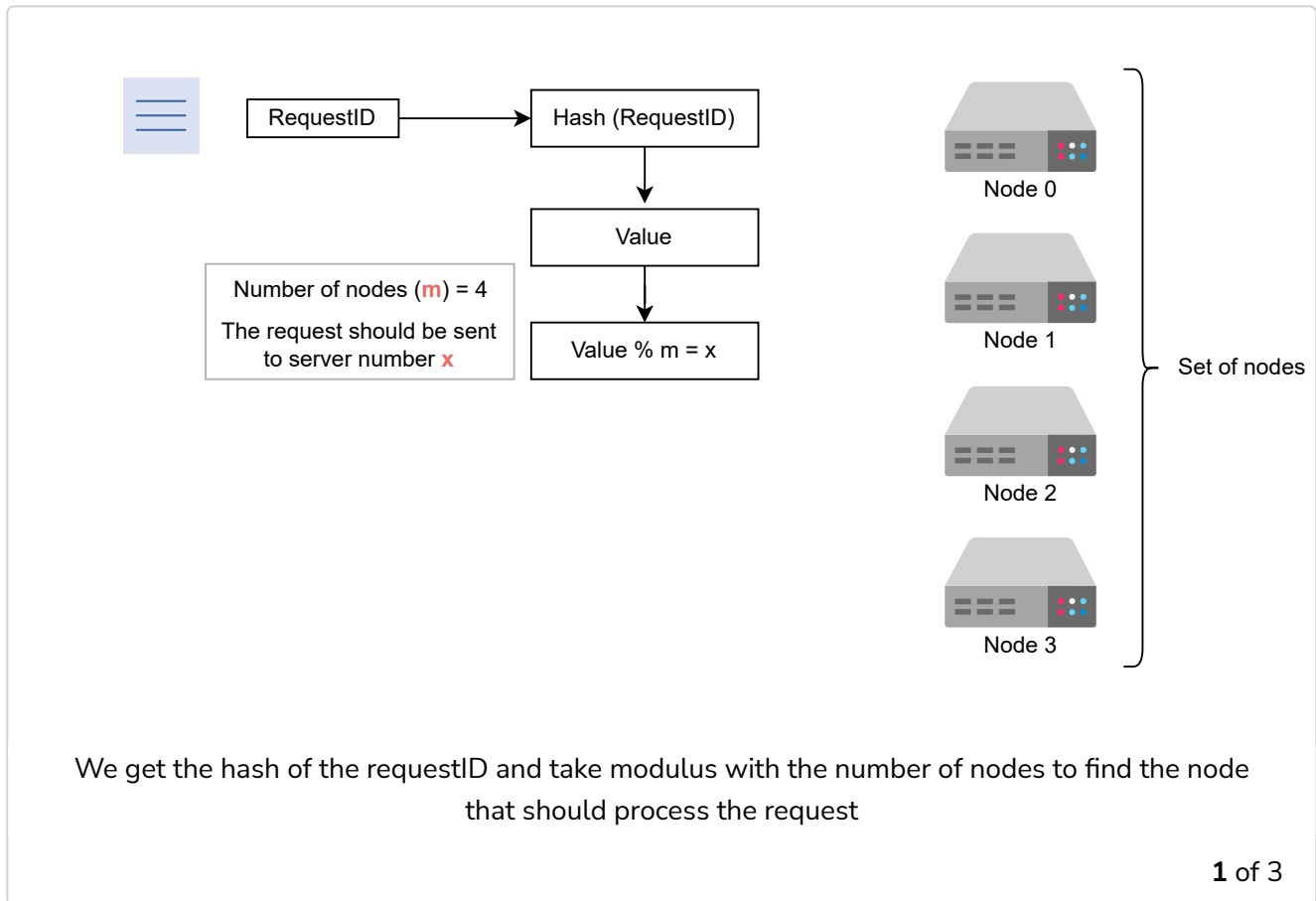
- Add scalability
 - Consistent hashing
 - Use virtual nodes
 - Advantages of virtual nodes
- Data replication
 - Primary-secondary approach
 - Peer-to-peer approach

Add scalability

Let's start with one of the core design requirements: scalability. We store key-value data in storage nodes. With a change in demand, we might need to add or remove storage nodes. It means we need to partition data over the nodes in the system to distribute the load across all nodes.

For example, let's consider that we have four nodes, and we want 25% of the requests to go to each node to balance the load equally. The traditional way to solve this is through the modulus operator. When a request comes in, we assign a request ID, calculate its hash, and find the remainder by taking the modulus with the number of nodes available. The remainder value is the node number, and we send the request to that node to process it.

The following slides explain this process:



We want to add and remove nodes with minimal change in our infrastructure. But in this method, when we add or remove a node, we need to move a lot of keys. This is inefficient. For example, node 2 is removed, and suppose for the same request ID, the new server to process a request will be node 1 because $10 \% 3 = 1$. Nodes hold information in their local caches, like keys and their values. So, we need to move that request's data to the next node that has to process the request. But this replication can be costly and can cause high latency.

Next, we'll learn how to copy data efficiently.

Point to Ponder

Question

Why didn't we use load balancers to distribute the requests to all nodes?

Show Answer ▼

Consistent hashing

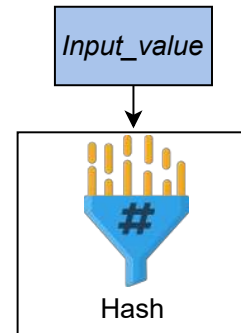
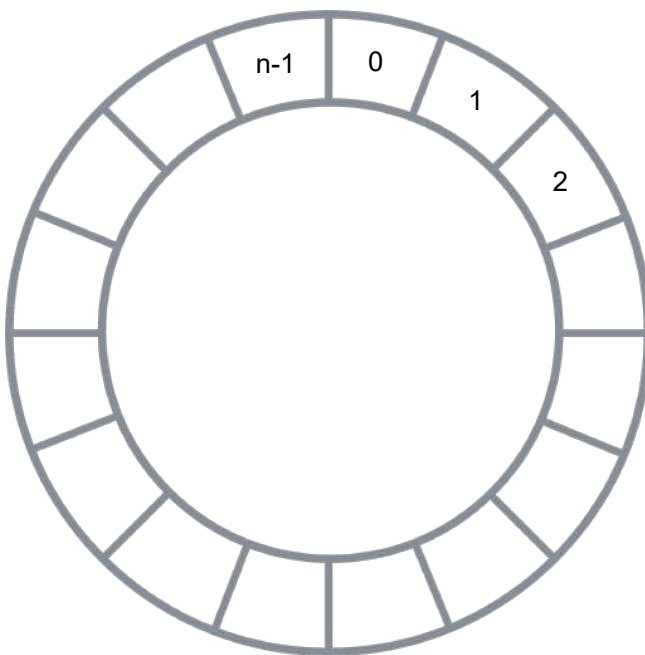
Consistent hashing is an effective way to manage the load over the set of nodes. In consistent hashing, we consider that we have a conceptual ring of hashes from 0 to $n - 1$, where n is the number of available hash values. We use each node's ID, calculate its hash, and map it to the ring. We apply the same process to requests. Each request is completed by the next node that it finds by moving in the clockwise direction in the ring.

Whenever a new node is added to the ring, the immediate next node is affected. It has to share its data with the newly added node while other nodes are unaffected. It's easy to scale since we're able to keep changes to our nodes minimal. This is because only a small portion of overall keys need to move. The hashes are randomly distributed, so we expect the load of requests to be random and distributed evenly on average on the ring.

?

Tt





Consider we have a conceptual ring of hashes from 0 to $n-1$, where n is the total number of hash values in the ring

1 of 14



The primary benefit of consistent hashing is that as nodes join or leave, it ensures that a minimal number of keys need to move. However, the request load isn't equally divided in practice. Any server that handles a large chunk of data can become a bottleneck in a distributed system. That node will receive a disproportionately large share of data storage and retrieval requests, reducing the overall system performance. As a result, these are referred to as hotspots.

?

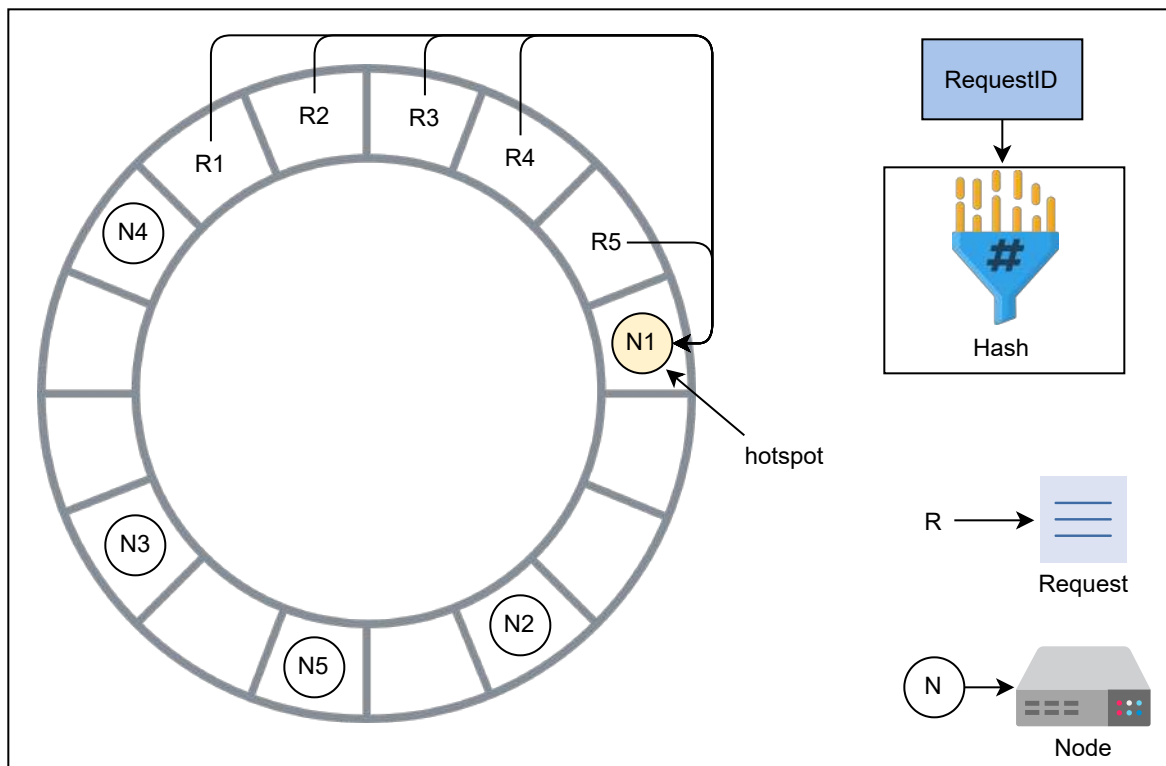
Tt

As shown in the figure below, most of the requests are between the N4 and N1 nodes. Now, N1 has to handle most of the requests compared to other

☾

nodes, and it has become a hotspot. That means non-uniform load distribution has increased load on a single server.

Note: It's a good exercise to think of possible solutions to the non-uniform load distribution before reading on.



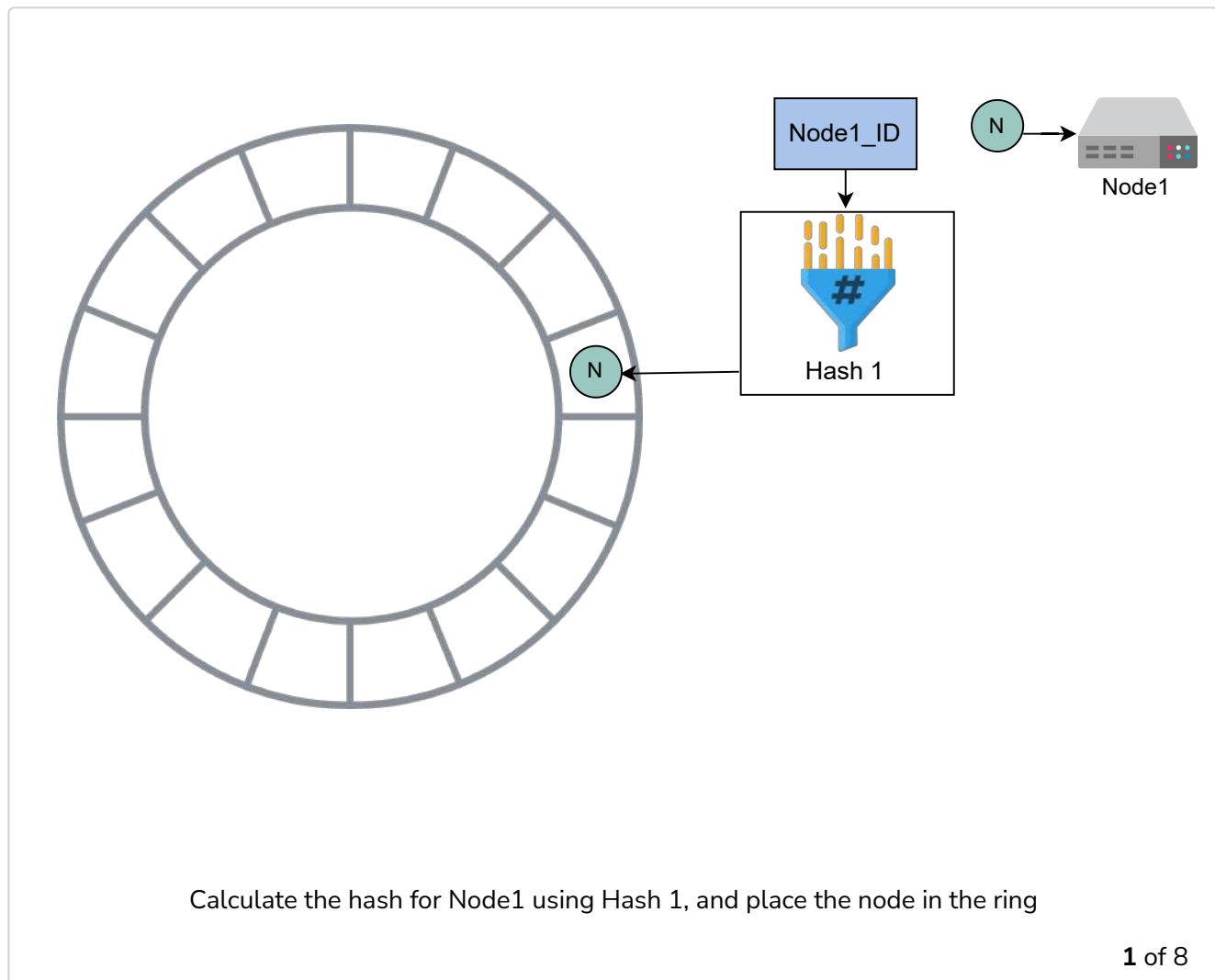
Non-uniform request distribution in the ring

Use virtual nodes

We'll use virtual nodes to ensure a more evenly distributed load across the nodes. Instead of applying a single hash function, we'll apply multiple hash functions onto the same key.

Let's take an example. Suppose we have three hash functions. For each node, we calculate three hashes and place them into the ring. For the request, we use only one hash function. Wherever the request lands onto the ring, it's processed by the next node found while moving in the clockwise direction. Each server has three positions, so the load of requests is more uniform.

Moreover, if a node has more hardware capacity than others, we can add more virtual nodes by using additional hash functions. This way, it'll have more positions in the ring and serve more requests.



Advantages of virtual nodes

Following are some advantages of using virtual nodes:

- If a node fails or does routine maintenance, the workload is uniformly distributed over other nodes. For each newly accessible node, the other nodes receive nearly equal load when it comes back online or is added to the system.

?

T

C

- It's up to each node to decide how many virtual nodes it's responsible for, considering the heterogeneity of the physical infrastructure. For example, if a node has roughly double the computational capacity as compared to the others, it can take more load.

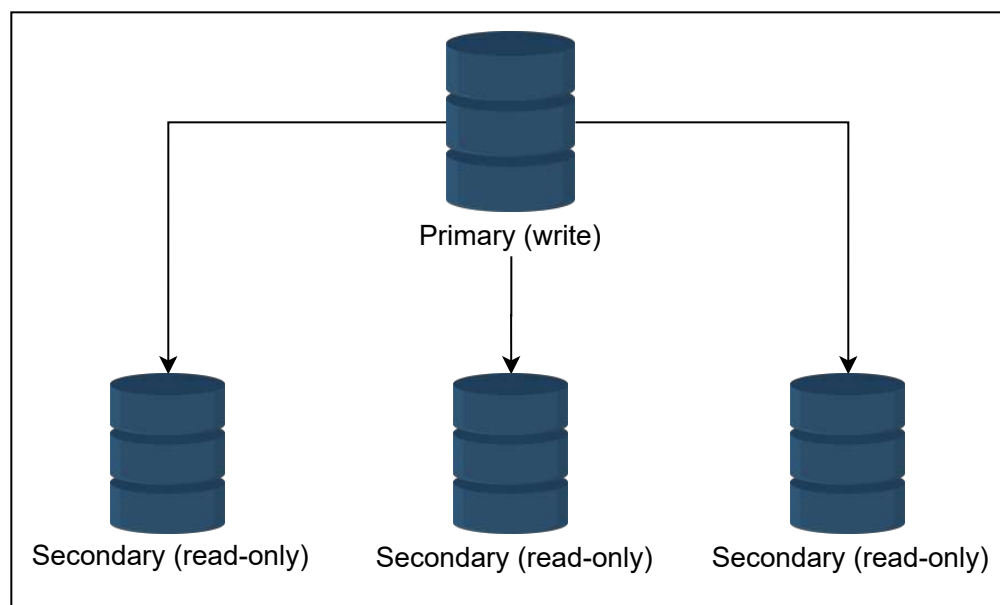
We've made the proposed design of key-value storage scalable. The next task is to make our system highly available.

Data replication

We have various methods to replicate the storage. It can be either a primary-secondary relationship or a peer-to-peer relationship.

Primary-secondary approach

In a **primary-secondary** approach, one of the storage areas is primary, and other storage areas are secondary. The secondary replicates its data from the primary. The primary serves the write requests while the secondary serves read requests. After writing, there's a lag for replication. Moreover, if the primary goes down, we can't write into the storage, and it becomes a single point of failure.



Primary-secondary approach

Point to Ponder

Question

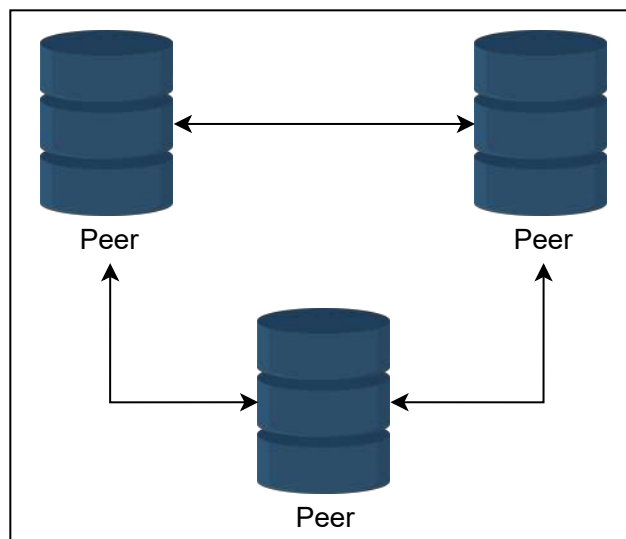
Does the primary-secondary approach fulfill the requirements of the key-value store that we defined in the System Design: The Key-value Store lesson?

Show Answer ▼

Peer-to-peer approach

In the **peer-to-peer** approach, all involved storage areas are primary, and they replicate the data to stay updated. Both read and write are allowed on all nodes. Usually, it's inefficient and costly to replicate in all n nodes.

Instead, three or five is a common choice for the number of storage nodes to be replicated.



?

Tt



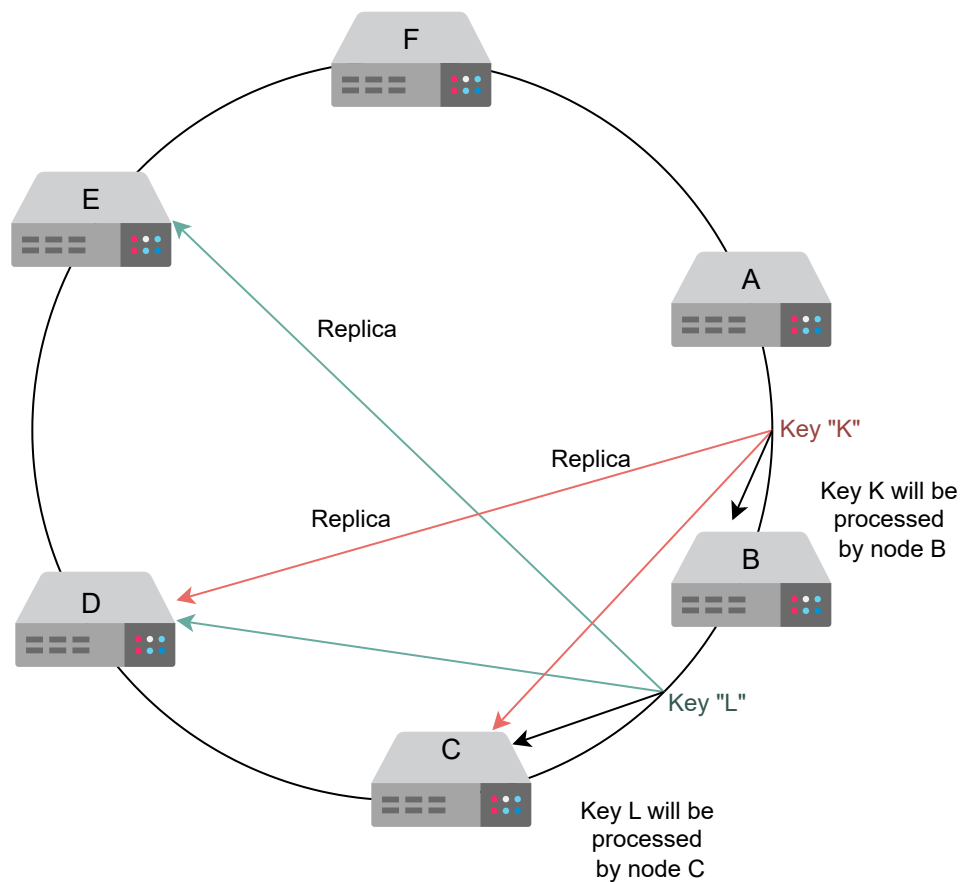
Peer-to-peer relationship

We'll use a peer-to-peer relationship for replication. We'll replicate the data on multiple hosts to achieve durability and high availability. Each data item will be replicated at n hosts, where n is a parameter configured per instance of the key-value store. For example, if we choose n to be 5, it means we want our data to be replicated to five nodes.

Each node will replicate its data to the other nodes. We'll call a node coordinator that handles read or write operations. It's directly responsible for the keys. A coordinator node is assigned the key "K." It's also responsible for replicating the keys to $n - 1$ successors on the ring (clockwise). These lists of successor virtual nodes are called preference lists. To avoid putting replicas on the same physical nodes, the preference list can skip those virtual nodes whose physical node is already in the list.

Let's consider the illustration given below. We have a replication factor, n , set to 3. For the key "K," the replication is done on the next three nodes: B, C, and D. Similarly, for key "L," the replication is done on nodes C, D, and E.





Replication in key-value store

Point to Ponder

Question

What is the impact of synchronous or asynchronous replication?

Show Answer ▼

?

Tt





Versioning Data and Achieving Configurability

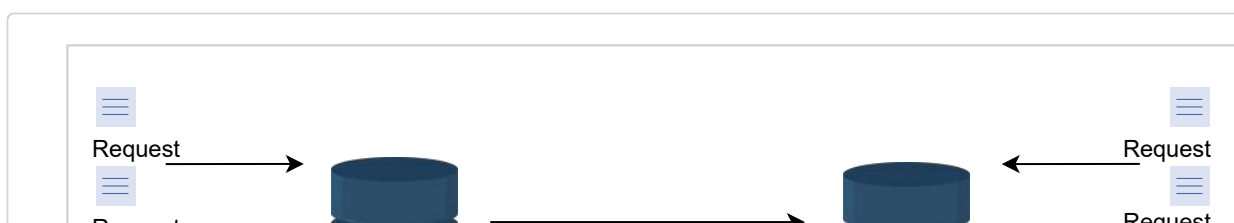
Learn how to resolve conflicts via versioning and how to make the key-value storage into a configurable service.

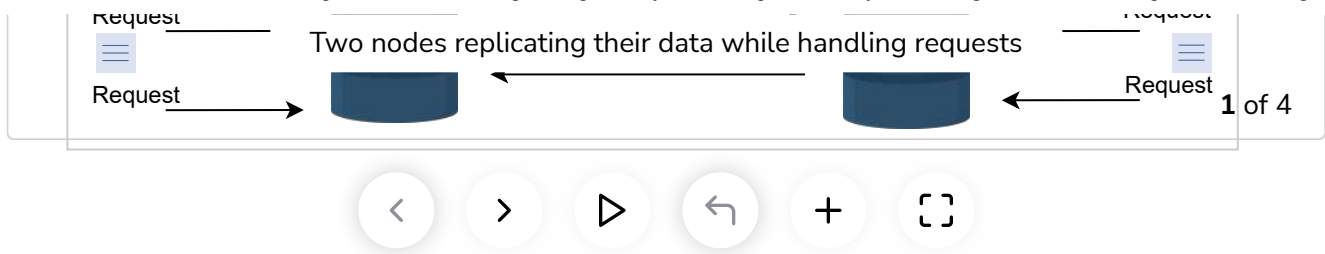
We'll cover the following

- Data versioning
 - Modify the API design
 - Vector clock usage example
 - Compromise with vector clocks limitations
- The get and put operations
- Usage of r and w

Data versioning

When network partitions and node failures occur during an update, an object's version history might become fragmented. As a result, it requires a reconciliation effort on the part of the system. It's necessary to build a way that explicitly accepts the potential of several copies of the same data so that we can avoid the loss of any updates. It's critical to realize that some failure scenarios can lead to multiple copies of the same data in the system. So, these copies might be the same or divergent. Resolving the conflicts among these divergent histories is essential and critical for consistency purposes. ?





To handle inconsistency, we need to maintain causality between the events. We can do this using the timestamps and update all conflicting values with the value of the latest request. But time isn't reliable in a distributed system, so we can't use it as a deciding factor.

Another approach to maintaining causality effectively is by using vector clocks. A **vector clock** is a list of (node, counter) pairs. There's a single vector clock for every version of an object. If two objects have different vector clocks, we're able to tell whether they're causally related or not (more on this in a bit). Unless one of the two changes is reconciled, the two are deemed at odds.

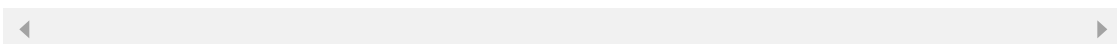
Modify the API design

We talked about how we can decide if two events are causally related or not using a vector clock value. For this, we need information about which node performed the operation before and what its vector clock value was. This is the context of an operation. So, we'll modify our API design as follows.

The API call to get a value should look like this:

```
get(key)
```

Parameter	Description
key	This is the key against which we want to get value .



We return an object or a collection of conflicting objects along with a **context**. The **context** holds encoded metadata about the object, including details such as the object's version.

The API call to put the value into the system should look like this:

```
put(key, context, value)
```

Parameter	Description
key	This is the key against which we have to store value .
context	This holds the metadata for each object.
value	This is the object that needs to be stored against the key .

The function finds the node where the value should be placed on the basis of the **key** and stores the value associated with it. The **context** is returned by the system after the **get** operation. If we have a list of objects in **context** that raises a conflict, we'll ask the client to resolve it.

To update an object in the key-value store, the client must give the **context**. We determine version information using a vector clock by supplying the **context** from a previous read operation. If the key-value store has access to several branches, it provides all objects at the leaf nodes, together with their respective version information in context, when processing a read request. ?
Reconciling disparate versions and merging them into a single new version is considered an update. T



Note: This process of resolving conflicts is comparable to how it's done in Git. If Git is able to merge multiple versions into one, merging is performed automatically. It's up to the client (the developer) to resolve conflicts manually if automatic conflict resolution is not possible. Along the same lines, our system can try automatic conflict resolution and, if not possible, ask the application to provide a final resolved value.

Vector clock usage example

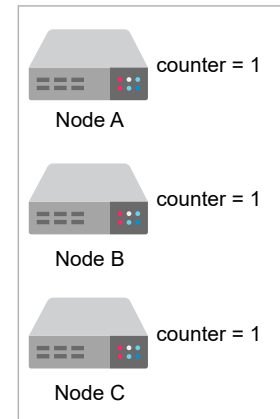
Let's consider an example. Say we have a write operation request. Node A handles the first version of the write request, $E1$; where E means event. The corresponding vector clock has node information and its counter—that is, $[A, 1]$. Node A handles another write for the same object on which the previous write was performed. So, for $E2$, we have $[A, 2]$. $E1$ is no longer required because $E2$ was updated on the same node. $E2$ reads the changes made by $E1$, and then new changes are made. Suppose a network partition happens. Now, the request is handled by two different nodes, B and C . The context with updated versions, which are $E3$, $E4$, and their related clocks, which are $([A, 2], [B, 1])$ and $([A, 2], [C, 1])$, are now in the system.

Suppose the network partition is repaired, and the client requests a write again, but now we have conflicts. The context $([A, 3], [B, 1], [C, 1])$ of the conflicts are returned to the client. After the client does reconciliation and A coordinates the write, we have $E5$ with the clock $([A, 4])$.

?

Tt





Let's suppose that we have three nodes. The vector clock counter is set to 1 for all of them

1 of 8



Compromise with vector clocks limitations

The size of vector clocks may increase if multiple servers write to the same object simultaneously. It's unlikely to happen in practice because writes are typically handled by one of the top n nodes in a preference list.

For example, if there are network partitions or multiple server failures, write requests may be processed by nodes not in the top n nodes in the preference list. As a result we can have a long version like this:

$([A, 10], [B, 4], [C, 1], [D, 2], [E, 1], [F, 3], [G, 5], [H, 7], [I, 2], [J, 2], [K, 1])$. It's a hassle to store and maintain such a long version history.

We can limit the size of the vector clock in these situations. We employ a clock truncation strategy to store a timestamp with each (node, counter) pair to show when the data item was last updated by the node. Vector clock pairs are purged when the number of (node, counter) pairs exceeds a predetermined threshold (let's say 10). Because the descendant linkages can't be precisely calculated, this truncation approach can lead to a lack of efficiency in reconciliation.

The **get** and **put** operations

One of our functional requirements is that the system should be configurable. We want to control the trade-offs between availability, consistency, cost-effectiveness, and performance. So, let's achieve configurability by implementing the basic **get** and **put** functions of the key-value store.

Every node can handle the **get** (read) and **put** (write) operations in our system. A node handling a read or write operation is known as a **coordinator**. The coordinator is the first among the top n nodes in the preference list.

There can be two ways for a client to select a node:

- We route the request to a generic load balancer.
- We use a partition-aware client library that routes requests directly to the appropriate coordinator nodes.

Both approaches have their benefits. The client isn't linked to the code in the first approach, whereas lower latency is achievable in the second. The latency is lower due to the reduced number of hops because the client can directly go to a specific server.

?

Tt



Let’s make our service configurable by having an ability where we can control the trade-offs between availability, consistency, cost-effectiveness, and performance. We can use a consistency protocol similar to those used in quorum systems.

Let’s take an example. Say n in the top n of the preference list is equal to 3. It means three copies of the data need to be maintained. We assume that nodes are placed in a ring. Say A, B, C, D, and E is the clockwise order of the nodes in that ring. If the write function is performed on node A, then the copies of that data will be placed on B and C. This is because B and C are the next nodes we find while moving in a clockwise direction of the ring.

Usage of r and w

Now, consider two variables, r and w . The r means the minimum number of nodes that need to be part of a successful read operation, while w is the minimum number of nodes involved in a successful write operation. So if $r = 2$, it means our system will read from two nodes when we have data stored in three nodes. We need to pick values of r and w such that at least one node is common between them. This ensures that readers could get the latest-written value. For that, we’ll use a quorum-like system by setting $r + w > n$.

The following table gives an overview of how the values of n , r , and w affect the speed of reads and writes:

?

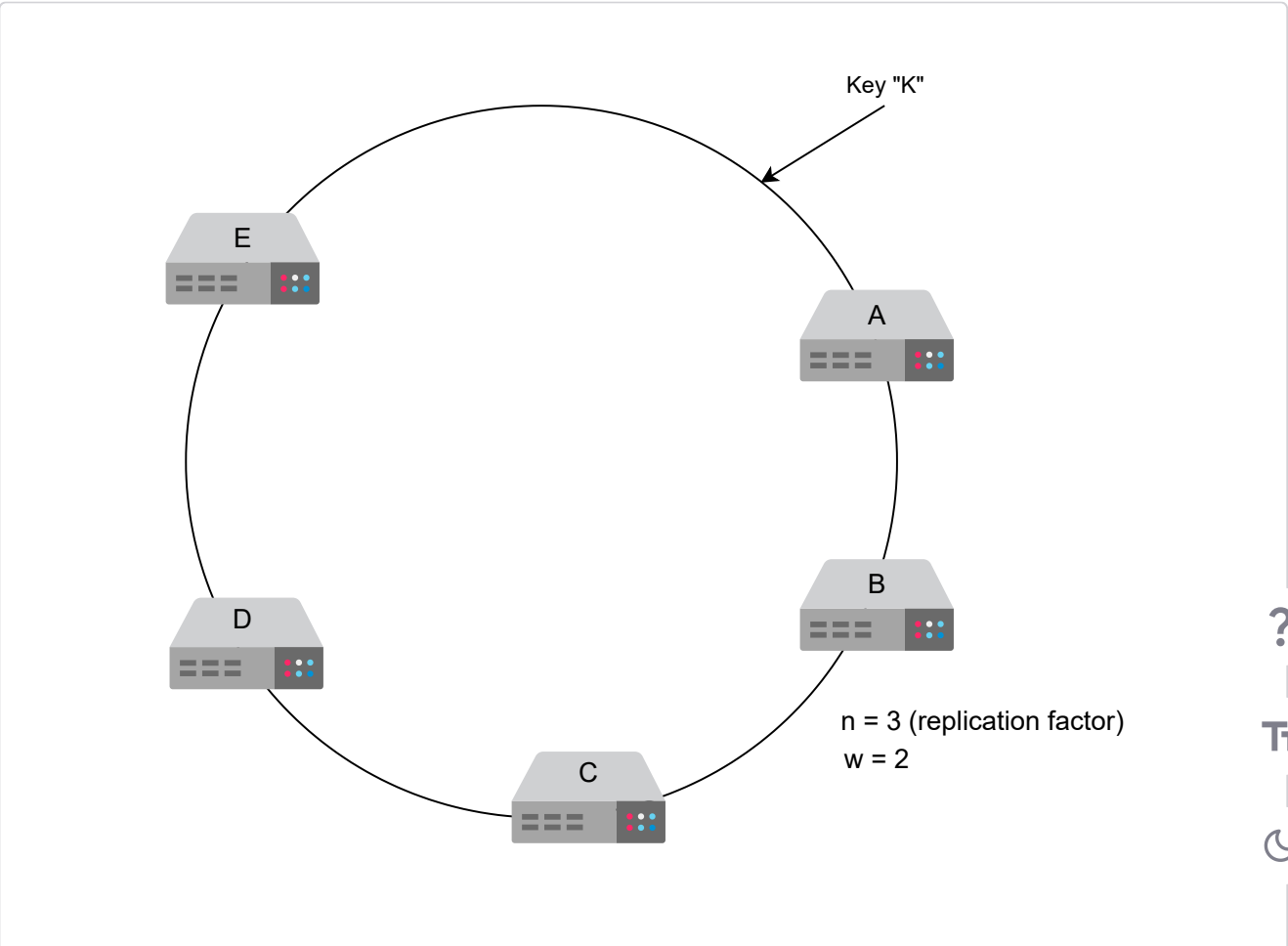
Tt

☾

Value Effects on Reads and Writes			
n	r	w	Description

3	2	1	It won't be allowed as it violates our consistency constraint $w > n$.
3	2	2	It will be allowed as it fulfills consistency.
3	3	1	It will provide speedy writes and slower reads as all readers need to go to all n replicas for every read.
3	1	3	It will provide speedy reads from any node as reads only need to go to one replica. Writes will be slower since we now need to write to all n replicas synchronously.

Let's say $n = 3$, which means we have three nodes where the data is copied to. Now, for $w = 2$, the operation makes sure to write in two nodes to make this request successful. For the third node, the data is updated asynchronously.



We have a replication factor of 3 and w is 2. The key "K" will be replicated to A, B, and C

1 of 3



In this model, the latency of a get operation is decided by the slowest of the r replicas. The reason is that for the larger value of r , we focus more on availability and compromise consistency.

The coordinator produces the vector clock for the new version and writes the new version locally upon receiving a `put()` request for a key. The coordinator sends n highest-ranking nodes with the updated version and a new vector clock. We consider a write successful if at least $w - 1$ nodes respond. Remember that the coordinate writes to itself first, so we get w writes in total.

Requests for a `get()` operation are made to the n highest-ranked reachable nodes in a preference list for a key. They wait for r answers before returning the results to the client. Coordinators return all dataset versions that they regard as unrelated if they get several datasets from the same source (divergent histories that need reconciliation). The conflicting versions are then merged, and the resulting key's value is rewritten to override the previous versions.

By now, we've fulfilled the scalability, availability, conflict-resolution, and configurable service requirements. The last requirement is to have a fault-tolerant system. Let's discuss how we'll achieve it in the next lesson.

?

← Back

Ensure Scalability and Replication

Next →

Enable Fault Tolerance and Failure Detec





Enable Fault Tolerance and Failure Detection

Learn how to make a key-value store fault tolerant and able to detect failure.

We'll cover the following



- Handle temporary failures
- Handle permanent failures
 - Anti-entropy with Merkle trees
- Promote membership in the ring to detect failures
- [Conclusion](#)

Handle temporary failures

Typically, distributed systems use a quorum-based approach to handle failures. A quorum is the minimum number of votes required for a distributed transaction to proceed with an operation. If a server is part of the consensus and is down, then we can't perform the required operation. It affects the availability and durability of our system.

We'll use a sloppy quorum instead of strict quorum membership. Usually, a leader manages the communication among the participants of the consensus. The participants send an acknowledgment after committing a successful write. Upon receiving these acknowledgments, the leader responds to the client. However, the drawback is that the participants are easily affected by the network outage. If the leader is temporarily down and the participants can't reach it, they declare the leader dead. Now, a new leader has to be reelected. Such frequent elections have a negative impact

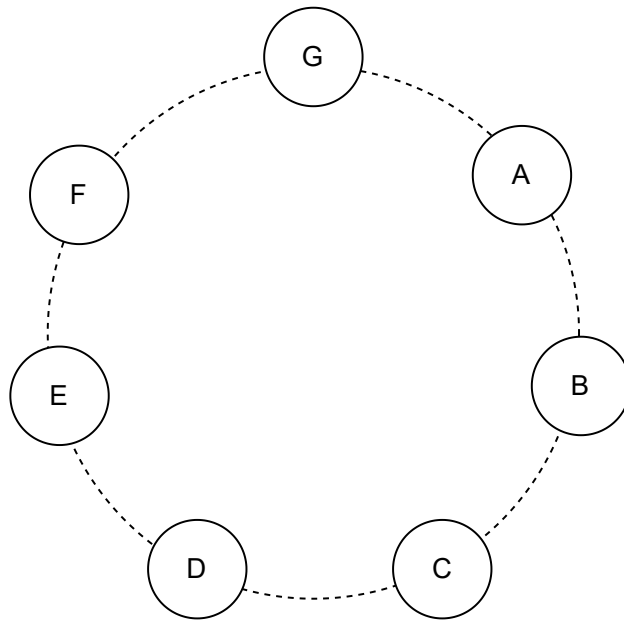


on performance because the system spends more time picking a leader than accomplishing any actual work.

In the sloppy quorum, the first n healthy nodes from the preference list handle all read and write operations. The n healthy nodes may not always be the first n nodes discovered when moving clockwise in the consistent hash ring.

Let's consider the following configuration with $n = 3$. If node A is briefly unavailable or unreachable during a write operation, the request is sent to the next healthy node from the preference list, which is node D in this case. It ensures the desired availability and durability. After processing the request, the node D includes a hint as to which node was the intended receiver (in this case, A). Once node A is up and running again, node D sends the request information to A so it can update its data. Upon completion of the transfer, D removes this item from its local storage without affecting the total number of replicas in the system.





Preference List= [A,D,C,B,G,E]

Suppose we have seven nodes in our ring and a preference list of the nodes

1 of 5



This approach is called a **hinted handoff**. Using it, we can ensure that reads and writes are fulfilled if a node faces temporary failure.

Note: A highly available storage system must handle data center failure due to power outages, cooling failures, network failures, or natural disasters. For this, we should ensure replication across the data centers. So, if one data center is down, we can recover it from the other.

?

Tt



Point to Ponder



Question

What are the limitations of using hinted handoff?

Show Answer ▼

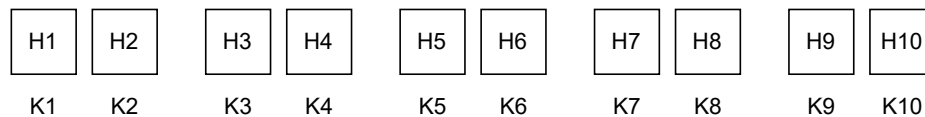
Handle permanent failures

In the event of permanent failures of nodes, we should keep our replicas synchronized to make our system more durable. We need to speed up the detection of inconsistencies between replicas and reduce the quantity of transferred data. We'll use Merkle trees for that.

In a **Merkle tree**, the values of individual keys are hashed and used as the leaves of the tree. There are hashes of their children in the parent nodes higher up the tree. Each branch of the Merkle tree can be verified independently without the need to download the complete tree or the entire dataset. While checking for inconsistencies across copies, Merkle trees reduce the amount of data that must be exchanged. There's no need for synchronization if, for example, the hash values of two trees' roots are the same and their leaf nodes are also the same. Until the process reaches the tree leaves, the hosts can identify the keys that are out of sync when the nodes exchange the hash values of children. The Merkle tree is a mechanism to implement anti-entropy, which means to keep all the data consistent. It reduces data transmission for synchronization and the number of discs accessed during the anti-entropy process.

The following slides explain how Merkle trees work:





Calculate the hashes for all keys. The hashes will be leaf nodes

1 of 14



Anti-entropy with Merkle trees

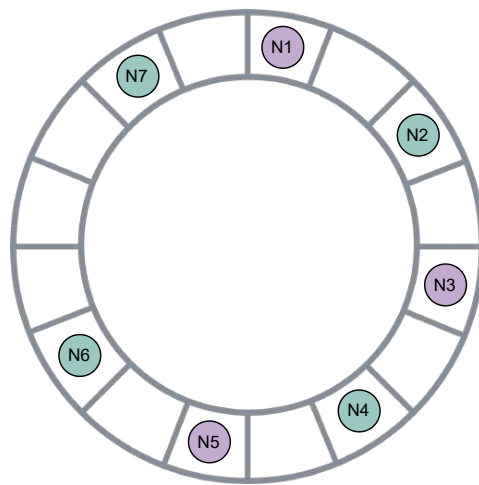
Each node keeps a distinct Merkle tree for the range of keys that it hosts for each virtual node. The nodes can determine if the keys in a given range are correct. The root of the Merkle tree corresponding to the common key ranges is exchanged between two nodes. We'll make the following comparison:

1. Compare the hashes of the root node of Merkle trees.
2. Do not proceed if they're the same.
3. Traverse left and right children using recursion. The nodes identify whether or not they have any differences and perform the necessary synchronization.



The following slides explain more about how Merkle trees work.

Note: We assume the ranges defined are hypothetical for illustration purposes.



Virtual nodes of Node A: [N1, N3, N5]
Virtual nodes of Node B: [N2, N4, N6, N7]

Let's suppose we have the virtual nodes A and B in the ring

1 of 9



The advantage of using Merkle trees is that each branch of the Merkle tree can be examined independently without requiring nodes to download the tree or the complete dataset. It reduces the quantity of data that must be exchanged for synchronization and the number of disc accesses that are required during the anti-entropy procedure.



The disadvantage is that when a node joins or departs the system, the tree's hashes are recalculated because multiple key ranges are affected.



We want our nodes to detect the failure of other nodes in the ring, so let's see how we can add it to our proposed design.

Promote membership in the ring to detect failures

The nodes can be offline for short periods, but they may also indefinitely go offline. We shouldn't rebalance partition assignments or fix unreachable replicas when a single node goes down because it's rarely a permanent departure. Therefore, the addition and removal of nodes from the ring should be done carefully.

Planned commissioning and decommissioning of nodes results in membership changes. These changes form history. They're recorded persistently on the storage for each node and reconciled among the ring members using a gossip protocol. A **gossip-based protocol** also maintains an eventually consistent view of membership. When two nodes randomly choose one another as their peer, both nodes can efficiently synchronize their persisted membership histories.

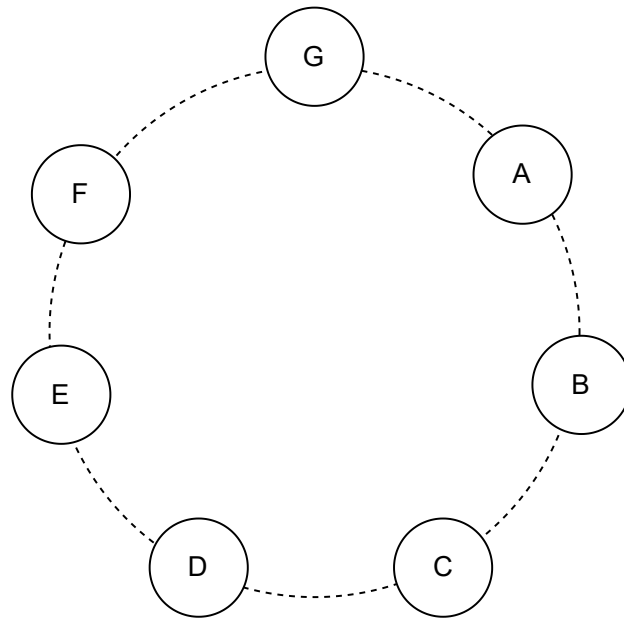
Let's learn how a gossip-based protocol works by considering the following example. Say node *A* starts up for the first time, and it randomly adds nodes *B* and *E* to its token set. The token set has virtual nodes in the consistent hash space and maps nodes to their respective token sets. This information is stored locally on the disk space of the node.

Now, node *A* handles a request that results in a change, so it communicates this to *B* and *E*. Another node, *D*, has *C* and *E* in its token set. It makes a change and tells *C* and *E*. The other nodes do the same process. This way, every node eventually knows about every other node's information. It's an efficient way to share information asynchronously, and it doesn't take up a lot of bandwidth.

?

T





A set of nodes in a ring

1 of 5



Points to Ponder

Question 1

Keeping in mind our consistent hashing approach, can the gossip-based protocol fail?

Show Answer ▼



Decentralized failure detection protocols use a gossip-based protocol that allows each node to learn about the addition or removal of other nodes. The join and leave methods of the explicit node notify the nodes about the permanent node additions and removals. The individual nodes detect temporary node failures when they fail to communicate with another node. If a node fails to communicate to any of the nodes present in its token set for the authorized time, then it communicates to the administrators that the node is dead.

Conclusion

A key-value store provides flexibility and allows us to scale the applications that have unstructured data. Web applications can use key-value stores to





Introduction to a CDN

Learn about CDNs, and formalize the requirements for a CDN design.

We'll cover the following



- Proposed solution
- Requirements
 - Functional requirements
 - Non-functional requirements
- Building blocks we will use

Proposed solution

The solution to all the problems discussed in the previous lesson is the **content delivery network** (CDN). A CDN is a group of geographically distributed proxy servers. A **proxy server** is an intermediate server between a client and the origin server. The proxy servers are placed on the network edge. As the network edge is close to the end users, the placement of proxy servers helps quickly deliver the content to the end users by reducing latency and saving bandwidth. A CDN has added intelligence on top of being a simple proxy server.

We can bring data close to the user by placing a small data center near the user and storing copies of the data there. CDN mainly stores two types of data: static and dynamic. A CDN primarily targets propagation delay by bringing the data closer to its users. CDN providers make the extra effort to have sufficient bandwidth available through the path and bring data closer



to the users (possibly within their ISP). They also try to reduce transmission and queuing delays because the ISP presumably has more bandwidth available within the autonomous system.

Let's look at the different ways CDN solves the discussed problems:

- **High latency:** CDN brings the content closer to end users. Therefore, it reduces the physical distance and latency.
- **Data-intensive applications:** Since the path to the data includes only the ISP and the nearby CDN components, there's no issue in serving a large number of users through a few CDN components in a specific area. As shown below, the origin data center will have to provide the data to local CDN components only once, whereas local CDN components can provide data to different users individually. No user will have to download their own copy of data from the origin servers.

Note: Various streaming protocols are used to deliver dynamic content by the CDN providers. For example, CDNs use the Real-time Messaging Protocol (RTMP), HTTP Live Streaming (HLS), Real-time Streaming Protocol (RTSP), and many more to deliver dynamic content.

- **Scarcity of data center resources:** A CDN is used to serve popular content. Due to this reason, most of the traffic is handled at the CDN instead of the origin servers. So, different local or distributed CDN components share the load on origin servers.

?

Tr





Dissemination of content to a geographically distributed CDN

Note: A few well-known CDN providers are Akamai, StackPath, Cloudflare, Rackspace, Amazon CloudFront, and Google Cloud CDN.

Point to Ponder

Question

Does a CDN cache all content from the origin server?

Show Answer ▼

?

Tt

☾

Requirements

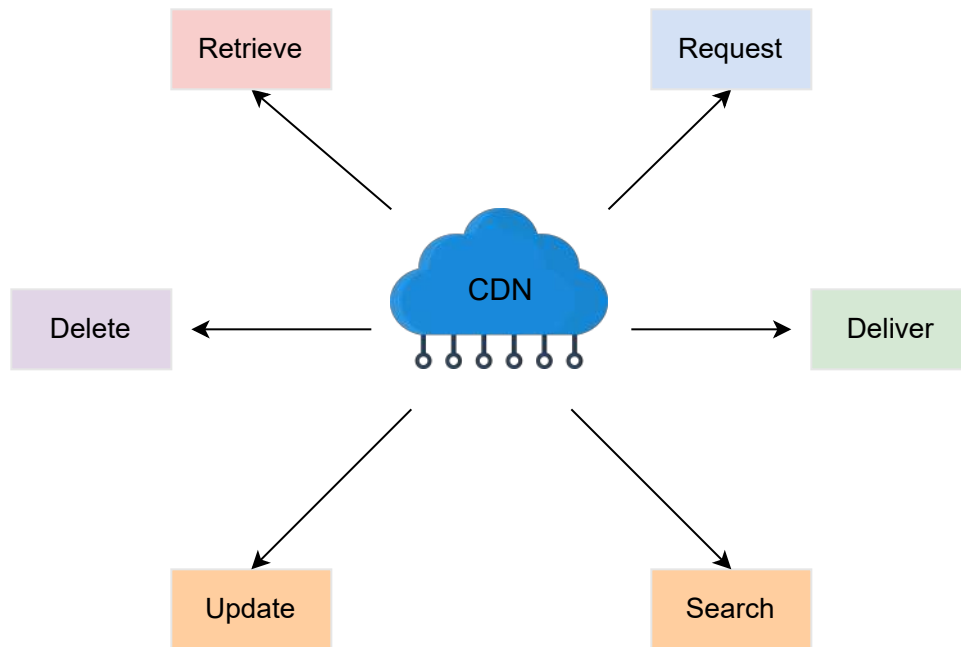
Let's look at the functional and non-functional requirements that we expect from a CDN.

Functional requirements

The following functional requirements will be a part of our design:

- **Retrieve:** Depending upon the type of CDN models, a CDN should be able to retrieve content from the origin servers. We'll cover CDN models in the coming lesson.
- **Request:** Content delivery from the proxy server is made upon the user's request. CDN proxy servers should be able to respond to each user's request in this regard.
- **Deliver:** In the case of the push model, the origin servers should be able to send the content to the CDN proxy servers.
- **Search:** The CDN should be able to execute a search against a user query for cached or otherwise stored content within the CDN infrastructure.
- **Update:** In most cases, content comes from the origin server, but if we run script in CDN, the CDN should be able to update the content within peer CDN proxy servers in a PoP.
- **Delete:** Depending upon the type of content (static or dynamic), it should be possible to delete cached entries from the CDN servers after a certain period.





Functional requirements of a CDN

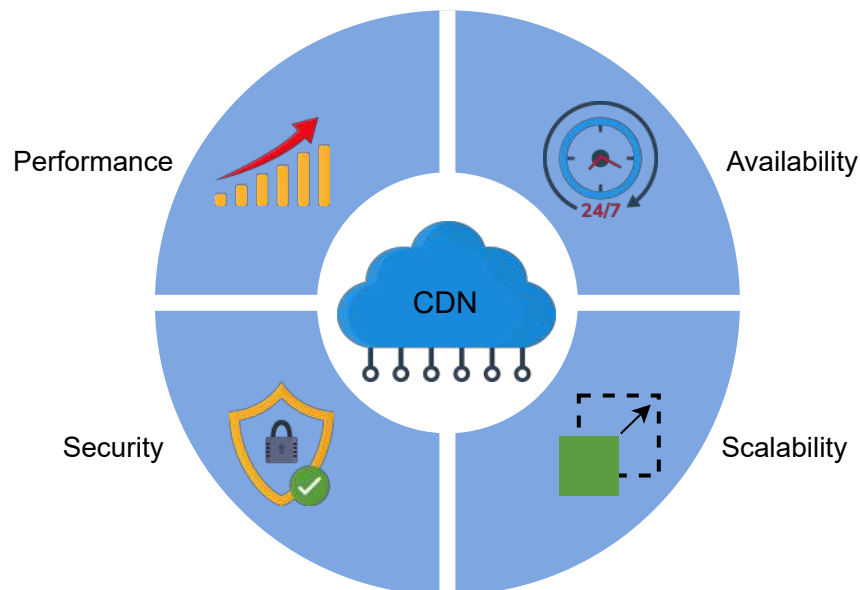
Non-functional requirements

- **Performance:** Minimizing latency is one of the core missions of a CDN. The proposed design should have minimum possible latency.
- **Availability:** CDNs are expected to be available at all times because of their effectiveness. Availability includes protection against attacks like DDoS.
- **Scalability:** An increasing number of users will request content from CDNs. Our proposed CDN design should be able to scale horizontally as the requirements increase.
- **Reliability and security:** Our CDN design should ensure no single point of failure. Apart from failures, the designed CDN must reliably handle massive traffic loads. Furthermore, CDNs should provide protection to hosted content from various attacks.

?

Tt

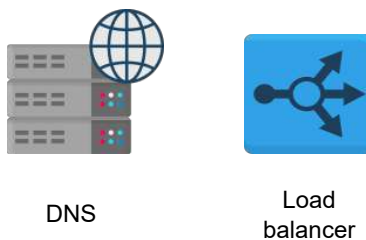




Non-functional requirements of CDN

Building blocks we will use

The design of a CDN utilizes the following building blocks:



The building blocks used in CDN design

?

Tt

