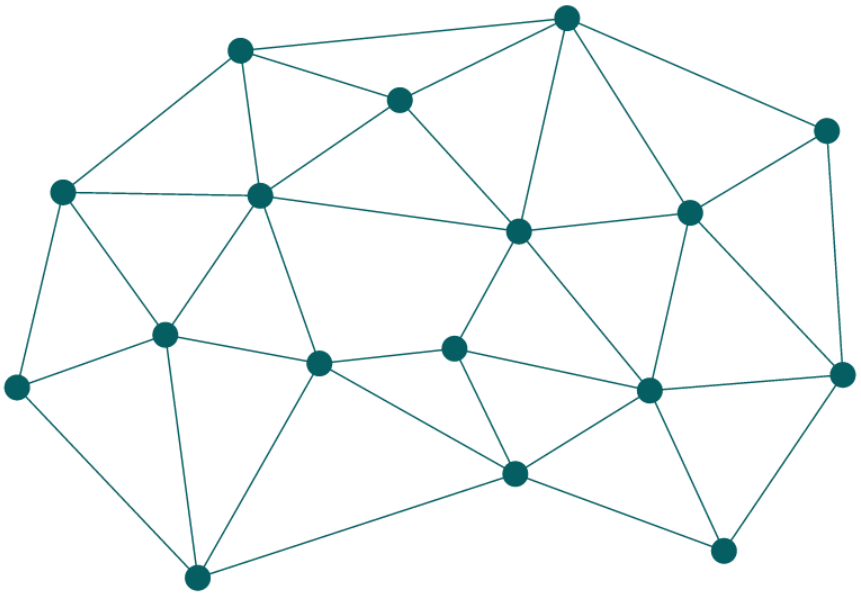
SYSTEM DESIGN



KARAN PRATAP SINGH

Table of Contents

Chapter V

Appendix

Preface

Welcome, I'm glad you’re here! System design interviews can be quite open-ended and intimidating as it is a very extensive topic. There can be multiple solutions to even the most basic system design problems. The objective of this book is to help you learn the fundamentals and explore the advanced topics of system design. This will also provide you with a reliable strategy to prepare for system design interviews.

Thank you for checking out this book, I hope it provides a positive learning experience.

License

What is System Design?

Before we start this course, let's talk about what even is system design.

System design is the process of defining the architecture, interfaces, and data for a system that satisfies specific requirements. System design meets the needs of your business or organization through coherent and efficient systems. It requires a systematic approach to building and engineering systems. A good system design requires us to think about everything, from infrastructure all the way down to the data and how it's stored.

Why is System Design so important?

System design helps us define a solution that meets the business requirements. It is one of the earliest decisions we can make when building a system. Often it is essential to think from a high level as these decisions are very difficult to correct later. It also makes it easier to reason about and manage architectural changes as the system evolves.

Chapter I

IP

An IP address is a unique address that identifies a device on the internet or a local network. IP stands for *"Internet Protocol"*, which is the set of rules governing the format of data sent via the internet or local network.

In essence, IP addresses are the identifier that allows information to be sent between devices on a network. They contain location information and make devices accessible for communication. The internet needs a way to differentiate between different computers, routers, and websites. IP addresses provide a way of doing so and form an essential part of how the internet works.

Versions

Now, let's learn about the different versions of IP addresses:

IPv4

The original Internet Protocol is IPv4 which uses a 32-bit numeric dot-decimal notation that only allows for around 4 billion IP addresses. Initially, it was more than enough but as internet adoption grew we needed something better.

Example: `102.22.192.181`

IPv6

IPv6 is a new protocol that was introduced in 1998. Deployment commenced in the mid-2000s and since the internet users have grown exponentially, it is still ongoing.

This new protocol uses 128-bit alphanumeric hexadecimal notation. This means that IPv6 can provide about ~340e+36 IP addresses. That's more than enough to meet the growing demand for years to come.

Example: `2001:0db8:85a3:0000:0000:8a2e:0370:7334`

Types

Let's discuss types of IP addresses:

Public

A public IP address is an address where one primary address is associated with your whole network. In this type of IP address, each of the connected devices has the same IP address.

Example: IP address provided to your router by the ISP.

Private

A private IP address is a unique IP number assigned to every device that connects to your internet network, which includes devices like computers, tablets, and smartphones, which are used in your household.

Example: IP addresses generated by your home router for your devices.

Static

A static IP address does not change and is one that was manually created, as opposed to having been assigned. These addresses are usually more expensive but are more reliable.

Example: They are usually used for important things like reliable geo-location services, remote access, server hosting, etc.

Dynamic

A dynamic IP address changes from time to time and is not always the same. It has been assigned by a Dynamic Host Configuration Protocol (DHCP) server. Dynamic IP addresses are the most common type of internet protocol address. They are cheaper to deploy and allow us to reuse IP addresses within a network as needed.

Example: They are more commonly used for consumer equipment and personal use.

OSI Model

The OSI Model is a logical and conceptual model that defines network communication used by systems open to interconnection and communication with other systems. The Open System Interconnection (OSI Model) also defines a logical network and effectively describes computer packet transfer by using various layers of protocols.

The OSI Model can be seen as a universal language for computer networking. It's based on the concept of splitting up a communication system into seven abstract layers, each one stacked upon the last.

Why does the OSI model matter?

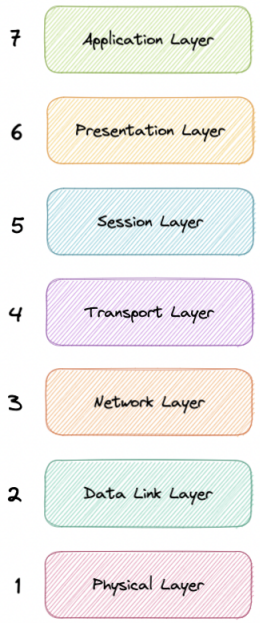
The Open System Interconnection (OSI) model has defined the common terminology used in networking discussions and documentation. This allows us to take a very complex communications process apart and evaluate its components.

While this model is not directly implemented in the TCP/IP networks that are most common today, it can still help us do so much more, such as:

- Make troubleshooting easier and help identify threats across the entire stack.
- Encourage hardware manufacturers to create networking products that can communicate with each other over the network.
- Essential for developing a security-first mindset.
- Separate a complex function into simpler components.

Layers

The seven abstraction layers of the OSI model can be defined as follows, from top to bottom:



Application

This is the only layer that directly interacts with data from the user. Software applications like web browsers and email clients rely on the application layer to initiate communication. But it should be made clear that client software applications are not part of the application layer, rather the application layer is responsible for the protocols and data manipulation that the software relies on to present meaningful data to the user. Application layer protocols include HTTP as well as SMTP.

Presentation

The presentation layer is also called the Translation layer. The data from the application layer is extracted here and manipulated as per the required format to transmit over the network. The functions of the presentation layer are translation, encryption/decryption, and compression.

Session

This is the layer responsible for opening and closing communication between the two devices. The time between when the communication is opened and closed is known as the session. The session layer ensures that the session stays open long enough to transfer all the data being exchanged, and then promptly closes the session in order to avoid wasting resources. The session layer also synchronizes data transfer with checkpoints.

Transport

The transport layer (also known as layer 4) is responsible for end-to-end communication between the two devices. This includes taking data from the session layer and breaking it up into chunks called segments before sending it to the Network layer (layer 3). It is also responsible for reassembling the segments on the receiving device into data the session layer can consume.

Network

The network layer is responsible for facilitating data transfer between two different networks. The network layer breaks up segments from the transport layer into smaller units, called packets, on the sender's device, and reassembles these packets on the receiving device. The network layer also finds the best physical path for the data to reach its destination this is known as routing. If the two devices communicating are on the same network, then the network layer is unnecessary.

Data Link

The data link layer is very similar to the network layer, except the data link layer facilitates data transfer between two devices on the same network. The data link layer takes packets from the network layer and breaks them into smaller pieces called frames.

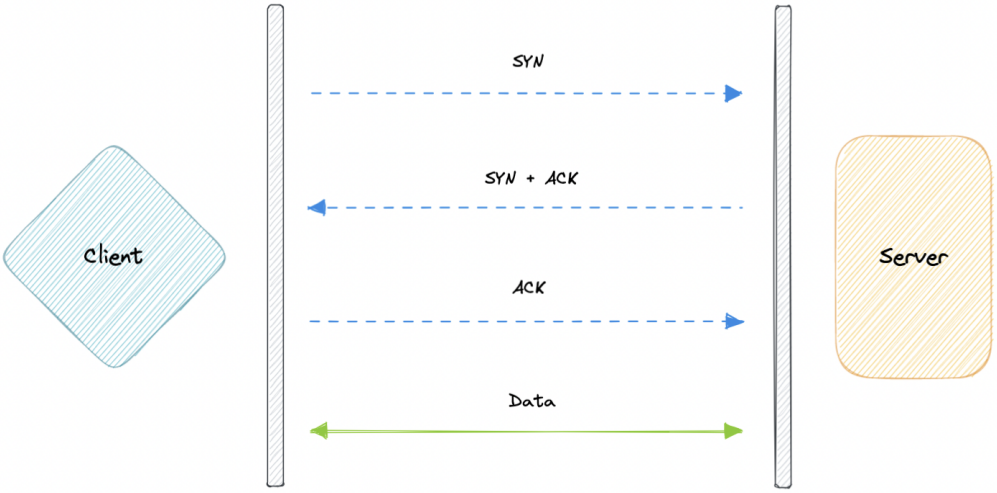
Physical

This layer includes the physical equipment involved in the data transfer, such as the cables and switches. This is also the layer where the data gets converted into a bit stream, which is a string of 1s and 0s. The physical layer of both devices must also agree on a signal convention so that the 1s can be distinguished from the 0s on both devices.

TCP and UDP

TCP

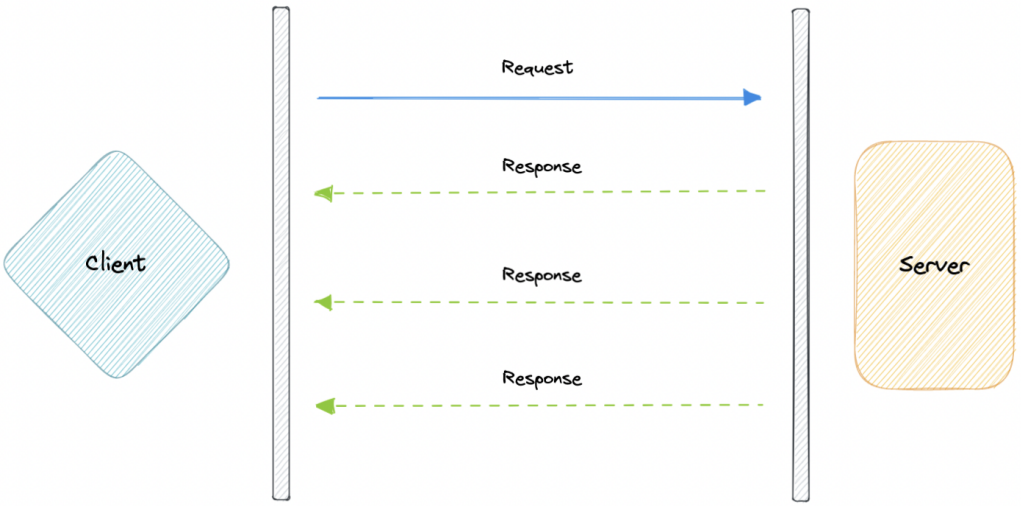
Transmission Control Protocol (TCP) is connection-oriented, meaning once a connection has been established, data can be transmitted in both directions. TCP has built-in systems to check for errors and to guarantee data will be delivered in the order it was sent, making it the perfect protocol for transferring information like still images, data files, and web pages.



But while TCP is instinctively reliable, its feedback mechanisms also result in a larger overhead, translating to greater use of the available bandwidth on the network.

UDP

User Datagram Protocol (UDP) is a simpler, connectionless internet protocol in which error-checking and recovery services are not required. With UDP, there is no overhead for opening a connection, maintaining a connection, or terminating a connection. Data is continuously sent to the recipient, whether or not they receive it.



It is largely preferred for real-time communications like broadcast or multicast network transmission. We should use UDP over TCP when we need the lowest latency and late data is worse than the loss of data.

TCP vs UDP

TCP is a connection-oriented protocol, whereas UDP is a connectionless protocol. A key difference between TCP and UDP is speed, as TCP is comparatively slower than UDP. Overall, UDP is a much faster, simpler, and more efficient protocol, however, retransmission of lost data packets is only possible with TCP.

TCP provides ordered delivery of data from user to server (and vice versa), whereas UDP is not dedicated to end-to-end communications, nor does it check the readiness of the receiver.

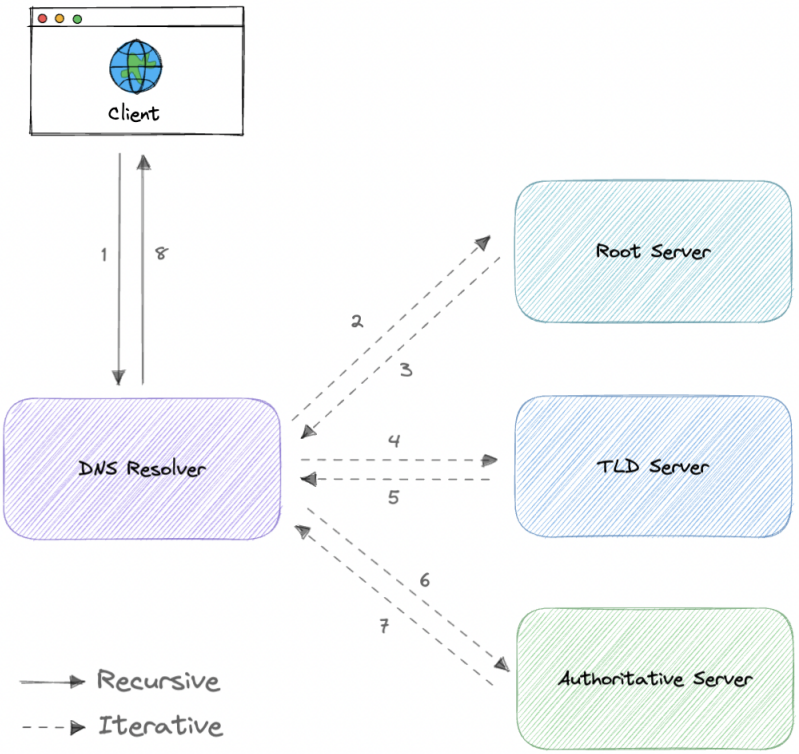
Feature	TCP	UDP
Connection	Requires an established connection	Connectionless protocol
Guaranteed delivery	Can guarantee delivery of data	Cannot guarantee delivery of data
Re-transmission	Re-transmission of lost packets is possible	No re-transmission of lost packets
Speed	Slower than UDP	Faster than TCP
Broadcasting	Does not support broadcasting	Supports broadcasting
Use cases	HTTPS, HTTP, SMTP, POP, FTP, etc	Video streaming, DNS, VoIP, etc

Domain Name System (DNS)

Earlier we learned about IP addresses that enable every machine to connect with other machines. But as we know humans are more comfortable with names than numbers. It's easier to remember a name like `google.com` than something like `122.250.192.232`.

This brings us to Domain Name System (DNS) which is a hierarchical and decentralized naming system used for translating human-readable domain names to IP addresses.

How DNS works



DNS lookup involves the following eight steps:

1. A client types example.com into a web browser, the query travels to the internet, and is received by a DNS resolver.
2. The resolver then recursively queries a DNS root nameserver.
3. The root server responds to the resolver with the address of a Top-Level Domain (TLD).
4. The resolver then makes a request to the `.com` TLD.
5. The TLD server then responds with the IP address of the domain's nameserver, example.com.
6. Lastly, the recursive resolver sends a query to the domain's nameserver.
7. The IP address for example.com is then returned to the resolver from the nameserver.
8. The DNS resolver then responds to the web browser with the IP address of the domain requested initially.

Once the IP address has been resolved, the client should be able to request content from the resolved IP address. For example, the resolved IP may return a webpage to be rendered in the browser.

Server types

Now, let's look at the four key groups of servers that make up the DNS infrastructure.

DNS Resolver

A DNS resolver (also known as a DNS recursive resolver) is the first stop in a DNS query. The recursive resolver acts as a middleman between a client and a DNS nameserver. After receiving a DNS query from a web client, a recursive resolver will either respond with cached data, or send a request to a root nameserver, followed by another request to a TLD nameserver, and then one last request to an authoritative nameserver. After receiving a response from the authoritative nameserver containing the requested IP address, the recursive resolver then sends a response to the client.

DNS root server

A root server accepts a recursive resolver's query which includes a domain name, and the root nameserver responds by directing the recursive resolver to a TLD nameserver, based on the extension of that domain (`.com`, `.net`, `.org`, etc.). The root nameservers are overseen by a nonprofit called the Internet Corporation for Assigned Names and Numbers (ICANN).

There are 13 DNS root nameservers known to every recursive resolver. Note that while there are 13 root nameservers, that doesn't mean that there are only 13 machines in the root nameserver system. There are 13 types of root nameservers, but there are multiple copies of each one all over the world, which use Anycast routing to provide speedy responses.

TLD nameserver

A TLD nameserver maintains information for all the domain names that share a common domain extension, such as `.com`, `.net`, or whatever comes after the last dot in a URL.

Management of TLD nameservers is handled by the Internet Assigned Numbers Authority (IANA), which is a branch of ICANN. The IANA breaks up the TLD servers into two main groups:

- **Generic top-level domains**: These are domains like `.com`, `.org`, `.net`, `.edu`, and `.gov`.
- **Country code top-level domains**: These include any domains that are specific to a country or state. Examples include `.uk`, `.us`, `.ru`, and `.jp`.

Authoritative DNS server

The authoritative nameserver is usually the resolver's last step in the journey for an IP address. The authoritative nameserver contains information specific to the domain name it serves (e.g. google.com) and it can provide a recursive resolver with the IP address of that server found in the DNS A record, or if the domain has a CNAME

record (alias) it will provide the recursive resolver with an alias domain, at which point the recursive resolver will have to perform a whole new DNS lookup to procure a record from an authoritative nameserver (often an A record containing an IP address). If it cannot find the domain, returns the NXDOMAIN message.

Query Types

There are three types of queries in a DNS system:

Recursive

In a recursive query, a DNS client requires that a DNS server (typically a DNS recursive resolver) will respond to the client with either the requested resource record or an error message if the resolver can't find the record.

Iterative

In an iterative query, a DNS client provides a hostname, and the DNS Resolver returns the best answer it can. If the DNS resolver has the relevant DNS records in its cache, it returns them. If not, it refers the DNS client to the Root Server or another Authoritative Name Server that is nearest to the required DNS zone. The DNS client must then repeat the query directly against the DNS server it was referred.

Non-recursive

A non-recursive query is a query in which the DNS Resolver already knows the answer. It either immediately returns a DNS record because it already stores it in a local cache, or queries a DNS Name Server which is authoritative for the record, meaning it definitely holds the correct IP for that hostname. In both cases, there is no need for additional rounds of queries (like in recursive or iterative queries). Rather, a response is immediately returned to the client.

Record Types

DNS records (aka zone files) are instructions that live in authoritative DNS servers and provide information about a domain including what IP address is associated with that domain and how to handle requests for that domain.

These records consist of a series of text files written in what is known as *DNS syntax*. DNS syntax is just a string of characters used as commands that tell the DNS server what to do. All DNS records also have a *"TTL"*, which stands for time-to-live, and indicates how often a DNS server will refresh that record.

There are more record types but for now, let's look at some of the most commonly used ones:

- **A (Address record)**: This is the record that holds the IP address of a domain.
- **AAAA (IP Version 6 Address record)**: The record that contains the IPv6 address for a domain (as opposed to A records, which stores the IPv4 address).
- **CNAME (Canonical Name record)**: Forwards one domain or subdomain to another domain, does NOT provide an IP address.
- **MX (Mail exchanger record)**: Directs mail to an email server.
- **TXT (Text Record)**: This record lets an admin store text notes in the record. These records are often used for email security.
- **NS (Name Server records)**: Stores the name server for a DNS entry.
- **SOA (Start of Authority)**: Stores admin information about a domain.
- **SRV (Service Location record)**: Specifies a port for specific services.
- **PTR (Reverse-lookup Pointer records)**: Provides a domain name in reverse lookups.
- **CERT (Certificate record)**: Stores public key certificates.

Subdomains

A subdomain is an additional part of our main domain name. It is commonly used to logically separate a website into sections. We can create multiple subdomains or child domains on the main domain.

For example, `blog.example.com` where `blog` is the subdomain, `example` is the primary domain and .com is the top-level domain (TLD). Similar examples can be `support.example.com` or `careers.example.com`.

DNS Zones

A DNS zone is a distinct part of the domain namespace which is delegated to a legal entity like a person, organization, or company, who is responsible for maintaining the DNS zone. A DNS zone is also an administrative function, allowing for granular control of DNS components, such as authoritative name servers.

DNS Caching

A DNS cache (sometimes called a DNS resolver cache) is a temporary database, maintained by a computer's operating system, that contains records of all the recent visits and attempted visits to websites and other internet domains. In other words, a DNS cache is just a memory of recent DNS lookups that our computer can quickly refer to when it's trying to figure out how to load a website.

The Domain Name System implements a time-to-live (TTL) on every DNS record. TTL specifies the number of seconds the record can be cached by a DNS client or server. When the record is stored in a cache, whatever TTL value came with it gets stored as well. The server continues to update the TTL of the record stored in the cache, counting down every second. When it hits zero, the record is deleted or purged from the cache. At that point, if a query for that record is received, the DNS server has to start the resolution process.

Reverse DNS

A reverse DNS lookup is a DNS query for the domain name associated with a given IP address. This accomplishes the opposite of the more commonly used forward DNS lookup, in which the DNS system is queried to return an IP address. The process of reverse resolving an IP address uses PTR records. If the server does not have a PTR record, it cannot resolve a reverse lookup.

Reverse lookups are commonly used by email servers. Email servers check and see if an email message came from a valid server before bringing it onto their network. Many email servers will reject messages from any server that does not support reverse lookups or from a server that is highly unlikely to be legitimate.

Note: Reverse DNS lookups are not universally adopted as they are not critical to the normal function of the internet.

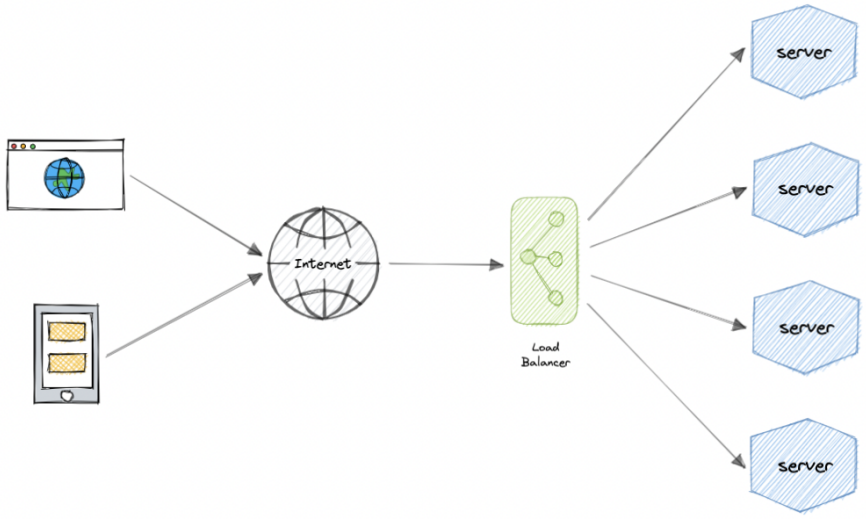
Examples

These are some widely used managed DNS solutions:

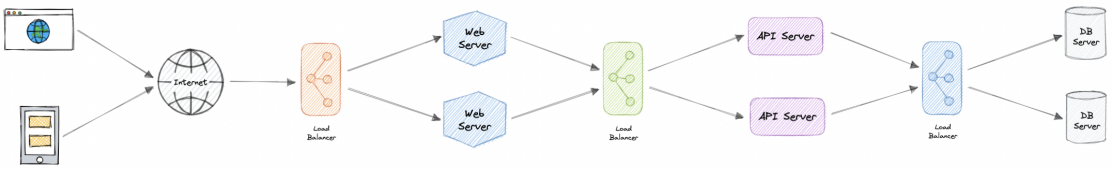
- Route53
- Cloudflare DNS
- Google Cloud DNS
- Azure DNS
- NS1

Load Balancing

Load balancing lets us distribute incoming network traffic across multiple resources ensuring high availability and reliability by sending requests only to resources that are online. This provides the flexibility to add or subtract resources as demand dictates.



For additional scalability and redundancy, we can try to load balance at each layer of our system:



But why?

Modern high-traffic websites must serve hundreds of thousands, if not millions, of concurrent requests from users or clients. To cost-effectively scale to meet these high volumes, modern computing best practice generally requires adding more servers.

A load balancer can sit in front of the servers and route client requests across all servers capable of fulfilling those requests in a manner that maximizes speed and capacity utilization. This ensures that no single server is overworked, which could degrade performance. If a single server goes down, the load balancer redirects traffic to the remaining online servers. When a new server is added to the server group, the load balancer automatically starts sending requests to it.

Workload distribution

This is the core functionality provided by a load balancer and has several common variations:

- **Host-based**: Distributes requests based on the requested hostname.
- **Path-based**: Using the entire URL to distribute requests as opposed to just the hostname.
- **Content-based**: Inspects the message content of a request. This allows distribution based on content such as the value of a parameter.

Layers

Generally speaking, load balancers operate at one of the two levels:

Network layer

This is the load balancer that works at the network's transport layer, also known as layer 4. This performs routing based on networking information such as IP addresses

and is not able to perform content-based routing. These are often dedicated hardware devices that can operate at high speed.

Application layer

This is the load balancer that operates at the application layer, also known as layer 7. Load balancers can read requests in their entirety and perform content-based routing. This allows the management of load based on a full understanding of traffic.

Types

Let's look at different types of load balancers:

Software

Software load balancers usually are easier to deploy than hardware versions. They also tend to be more cost-effective and flexible, and they are used in conjunction with software development environments. The software approach gives us the flexibility of configuring the load balancer to our environment's specific needs. The boost in flexibility may come at the cost of having to do more work to set up the load balancer. Compared to hardware versions, which offer more of a closed-box approach, software balancers give us more freedom to make changes and upgrades.

Software load balancers are widely used and are available either as installable solutions that require configuration and management or as a managed cloud service.

Hardware

As the name implies, a hardware load balancer relies on physical, on-premises hardware to distribute application and network traffic. These devices can handle a large volume of traffic but often carry a hefty price tag and are fairly limited in terms of flexibility.

Hardware load balancers include proprietary firmware that requires maintenance and updates as new versions, and security patches are released.

DNS

DNS load balancing is the practice of configuring a domain in the Domain Name System (DNS) such that client requests to the domain are distributed across a group of server machines.

Unfortunately, DNS load balancing has inherent problems limiting its reliability and efficiency. Most significantly, DNS does not check for server and network outages, or errors. It always returns the same set of IP addresses for a domain even if servers are down or inaccessible.

Routing Algorithms

Now, let's discuss commonly used routing algorithms:

- **Round-robin**: Requests are distributed to application servers in rotation.
- **Weighted Round-robin**: Builds on the simple Round-robin technique to account for differing server characteristics such as compute and traffic handling capacity using weights that can be assigned via DNS records by the administrator.
- **Least Connections**: A new request is sent to the server with the fewest current connections to clients. The relative computing capacity of each server is factored into determining which one has the least connections.
- **Least Response Time**: Sends requests to the server selected by a formula that combines the fastest response time and fewest active connections.
- **Least Bandwidth**: This method measures traffic in megabits per second (Mbps), sending client requests to the server with the least Mbps of traffic.
- **Hashing**: Distributes requests based on a key we define, such as the client IP address or the request URL.

Advantages

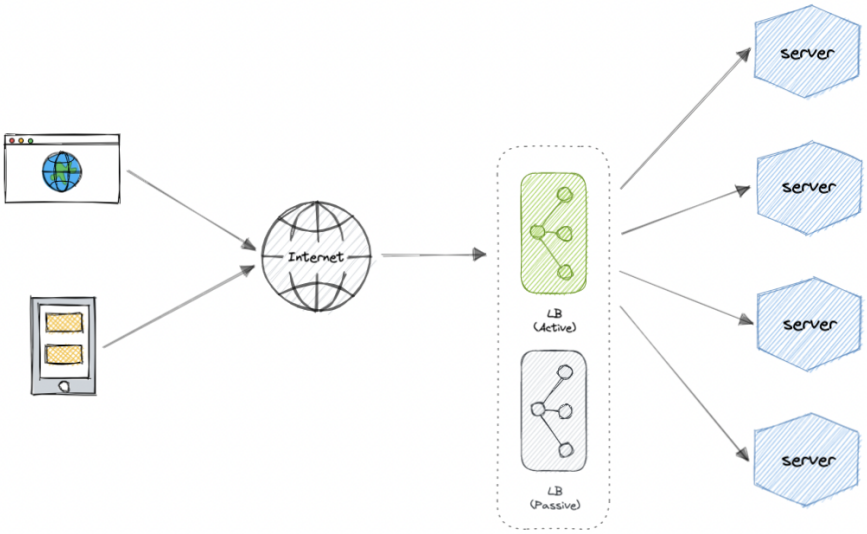
Load balancing also plays a key role in preventing downtime, other advantages of load balancing include the following:

- Scalability
- Redundancy
- Flexibility
- Efficiency

Redundant load balancers

As you must've already guessed, the load balancer itself can be a single point of failure. To overcome this, a second or `N` number of load balancers can be used in a cluster mode.

And, if there's a failure detection and the *active* load balancer fails, another *passive* load balancer can take over which will make our system more fault-tolerant.



Features

Here are some commonly desired features of load balancers:

- **Autoscaling**: Starting up and shutting down resources in response to demand conditions.
- **Sticky sessions**: The ability to assign the same user or device to the same resource in order to maintain the session state on the resource.
- **Healthchecks**: The ability to determine if a resource is down or performing poorly in order to remove the resource from the load balancing pool.
- **Persistence connections**: Allowing a server to open a persistent connection with a client such as a WebSocket.
- **Encryption**: Handling encrypted connections such as TLS and SSL.
- Certificates: Presenting certificates to a client and authentication of client certificates.
- **Compression**: Compression of responses.
- **Caching:** An application-layer load balancer may offer the ability to cache responses.
- **Logging**: Logging of request and response metadata can serve as an important audit trail or source for analytics data.
- **Request tracing**: Assigning each request a unique id for the purposes of logging, monitoring, and troubleshooting.
- **Redirects**: The ability to redirect an incoming request based on factors such as the requested path.
- **Fixed response**: Returning a static response for a request such as an error message.

Examples

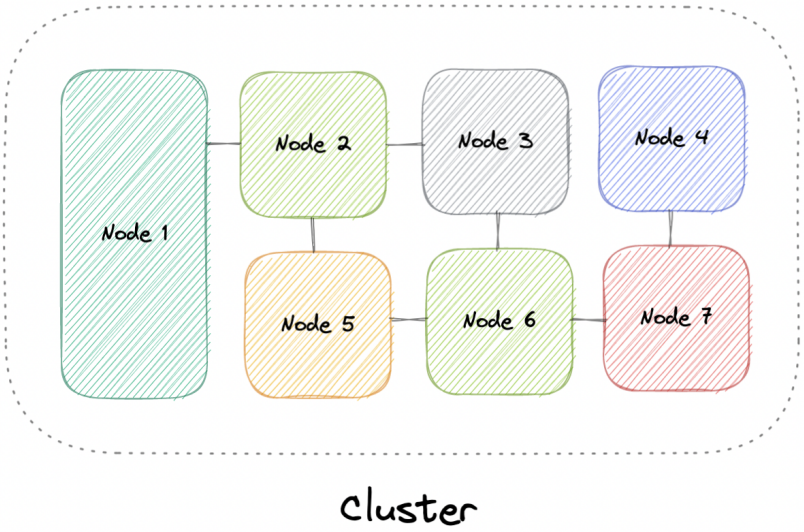
Following are some of the load balancing solutions commonly used in the industry:

- Amazon Elastic Load Balancing
- Azure Load Balancing
- GCP Load Balancing
- DigitalOcean Load Balancer
- Nginx
- HAProxy

Clustering

At a high level, a computer cluster is a group of two or more computers, or nodes, that run in parallel to achieve a common goal. This allows workloads consisting of a high number of individual, parallelizable tasks to be distributed among the nodes in the cluster. As a result, these tasks can leverage the combined memory and processing power of each computer to increase overall performance.

To build a computer cluster, the individual nodes should be connected to a network to enable internode communication. The software can then be used to join the nodes together and form a cluster. It may have a shared storage device and/or local storage on each node.



Typically, at least one node is designated as the leader node and acts as the entry point to the cluster. The leader node may be responsible for delegating incoming work to the other nodes and, if necessary, aggregating the results and returning a response to the user.

Ideally, a cluster functions as if it were a single system. A user accessing the cluster should not need to know whether the system is a cluster or an individual machine.

Furthermore, a cluster should be designed to minimize latency and prevent bottlenecks in node-to-node communication.

Types

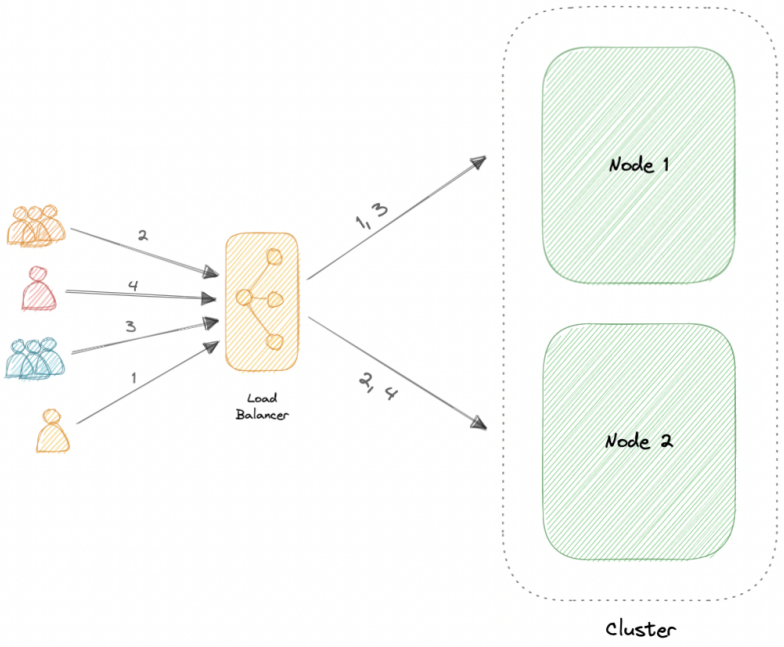
Computer clusters can generally be categorized into three types:

- Highly available or fail-over
- Load balancing
- High-performance computing

Configurations

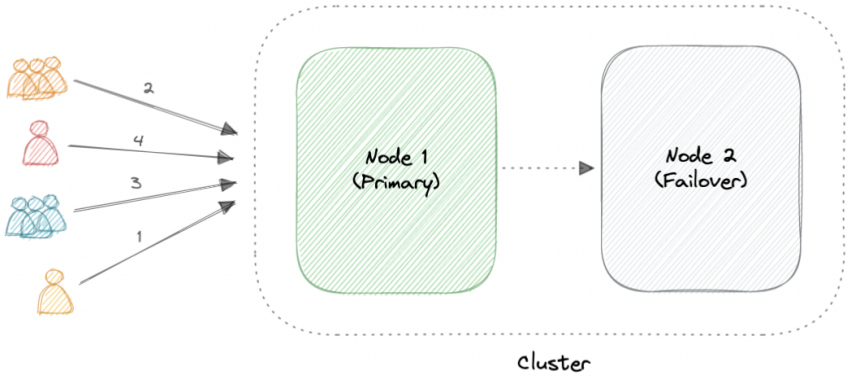
The two most commonly used high availability (HA) clustering configurations are active-active and active-passive.

Active-Active



An active-active cluster is typically made up of at least two nodes, both actively running the same kind of service simultaneously. The main purpose of an active-active cluster is to achieve load balancing. A load balancer distributes workloads across all nodes to prevent any single node from getting overloaded. Because there are more nodes available to serve, there will also be an improvement in throughput and response times.

Active-Passive



Like the active-active cluster configuration, an active-passive cluster also consists of at least two nodes. However, as the name *active-passive* implies, not all nodes are going to be active. For example, in the case of two nodes, if the first node is already active, then the second node must be passive or on standby.

Advantages

Four key advantages of cluster computing are as follows:

- High availability
- Scalability
- Performance
- Cost-effective

Load balancing vs Clustering

Load balancing shares some common traits with clustering, but they are different processes. Clustering provides redundancy and boosts capacity and availability. Servers in a cluster are aware of each other and work together toward a common purpose. But with load balancing, servers are not aware of each other. Instead, they react to the requests they receive from the load balancer.

We can employ load balancing in conjunction with clustering, but it also is applicable in cases involving independent servers that share a common purpose such as to run a website, business application, web service, or some other IT resource.

Challenges

The most obvious challenge clustering presents is the increased complexity of installation and maintenance. An operating system, the application, and its dependencies must each be installed and updated on every node.

This becomes even more complicated if the nodes in the cluster are not homogeneous. Resource utilization for each node must also be closely monitored, and logs should be aggregated to ensure that the software is behaving correctly.

Additionally, storage becomes more difficult to manage, a shared storage device must prevent nodes from overwriting one another and distributed data stores have to be kept in sync.

Examples

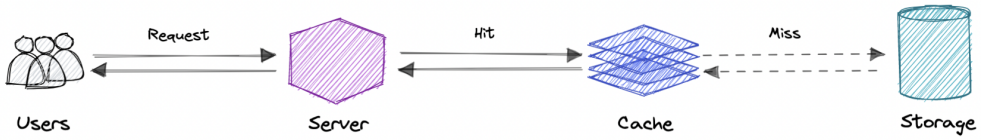
Clustering is commonly used in the industry, and often many technologies offer some sort of clustering mode. For example:

- Containers (e.g. Kubernetes, Amazon ECS)
- Databases (e.g. Cassandra, MongoDB)
- Cache (e.g. Redis)

Caching

"There are only two hard things in Computer Science: cache invalidation and naming things."

- Phil Karlton



A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer. Trading off capacity for speed, a cache typically stores a subset of data transiently, in contrast to databases whose data is usually complete and durable.

Caches take advantage of the locality of reference principle *"recently requested data is likely to be requested again"*.

Caching and Memory

Like a computer's memory, a cache is a compact, fast-performing memory that stores data in a hierarchy of levels, starting at level one, and progressing from there sequentially. They are labeled as L1, L2, L3, and so on. A cache also gets written if requested, such as when there has been an update and new content needs to be saved to the cache, replacing the older content that was saved.

No matter whether the cache is read or written, it's done one block at a time. Each block also has a tag that includes the location where the data was stored in the cache. When data is requested from the cache, a search occurs through the tags to

find the specific content that's needed in level one (L1) of the memory. If the correct data isn't found, more searches are conducted in L2.

If the data isn't found there, searches are continued in L3, then L4, and so on until it has been found, then, it's read and loaded. If the data isn't found in the cache at all, then it's written into it for quick retrieval the next time.

Cache hit and Cache miss

Cache hit

A cache hit describes the situation where content is successfully served from the cache. The tags are searched in the memory rapidly, and when the data is found and read, it's considered a cache hit.

Cold, Warm, and Hot Caches

A cache hit can also be described as cold, warm, or hot. In each of these, the speed at which the data is read is described.

A hot cache is an instance where data was read from the memory at the *fastest* possible rate. This happens when the data is retrieved from L1.

A cold cache is the *slowest* possible rate for data to be read, though, it's still successful so it's still considered a cache hit. The data is just found lower in the memory hierarchy such as in L3, or lower.

A warm cache is used to describe data that's found in L2 or L3. It's not as fast as a hot cache, but it's still faster than a cold cache. Generally, calling a cache warm is used to express that it's slower and closer to a cold cache than a hot one.

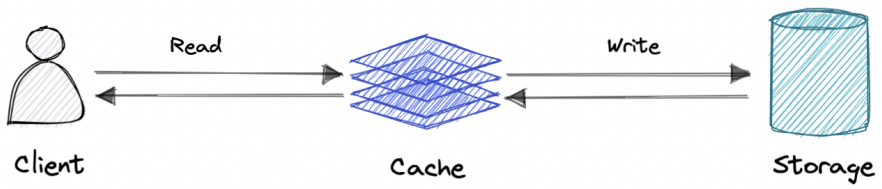
Cache miss

A cache miss refers to the instance when the memory is searched and the data isn't found. When this happens, the content is transferred and written into the cache.

Cache Invalidation

Cache invalidation is a process where the computer system declares the cache entries as invalid and removes or replaces them. If the data is modified, it should be invalidated in the cache, if not, this can cause inconsistent application behavior. There are three kinds of caching systems:

Write-through cache

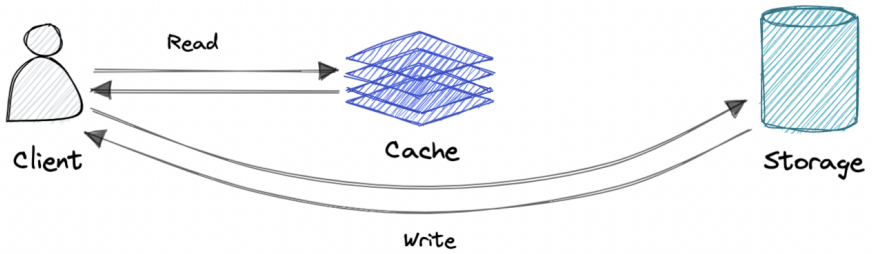


Data is written into the cache and the corresponding database simultaneously.

Pro: Fast retrieval, complete data consistency between cache and storage.

Con: Higher latency for write operations.

Write-around cache

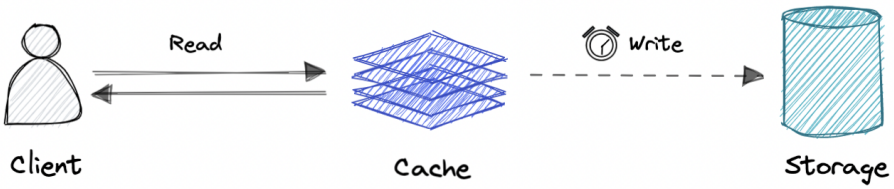


Where write directly goes to the database or permanent storage, bypassing the cache.

Pro: This may reduce latency.

Con: It increases cache misses because the cache system has to read the information from the database in case of a cache miss. As a result, this can lead to higher read latency in the case of applications that write and re-read the information quickly. Read happen from slower back-end storage and experiences higher latency.

Write-back cache



Where the write is only done to the caching layer and the write is confirmed as soon as the write to the cache completes. The cache then asynchronously syncs this write to the database.

Pro: This would lead to reduced latency and high throughput for write-intensive applications.

Con: There is a risk of data loss in case the caching layer crashes. We can improve this by having more than one replica acknowledging the write in the cache.

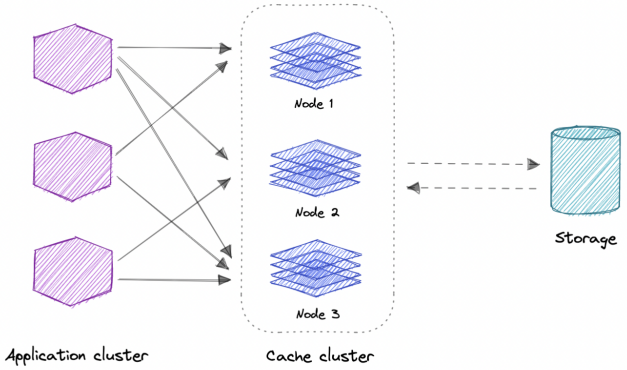
Eviction policies

Following are some of the most common cache eviction policies:

- **First In First Out (FIFO)**: The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.
- **Last In First Out (LIFO)**: The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
- **Least Recently Used (LRU)**: Discards the least recently used items first.
- **Most Recently Used (MRU)**: Discards, in contrast to LRU, the most recently used items first.

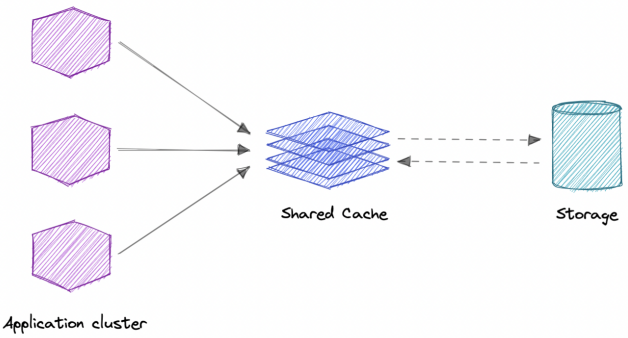
- **Least Frequently Used (LFU)**: Counts how often an item is needed. Those that are used least often are discarded first.
- **Random Replacement (RR)**: Randomly selects a candidate item and discards it to make space when necessary.

Distributed Cache



A distributed cache is a system that pools together the random-access memory (RAM) of multiple networked computers into a single in-memory data store used as a data cache to provide fast access to data. While most caches are traditionally in one physical server or hardware component, a distributed cache can grow beyond the memory limits of a single computer by linking together multiple computers.

Global Cache



As the name suggests, we will have a single shared cache that all the application nodes will use. When the requested data is not found in the global cache, it's the responsibility of the cache to find out the missing piece of data from the underlying data store.

Use cases

Caching can have many real-world use cases such as:

- Database Caching
- Content Delivery Network (CDN)
- Domain Name System (DNS) Caching
- API Caching

When not to use caching?

Let's also look at some scenarios where we should not use cache:

- Caching isn't helpful when it takes just as long to access the cache as it does to access the primary data store.
- Caching doesn't work as well when requests have low repetition (higher randomness), because caching performance comes from repeated memory access patterns.
- Caching isn't helpful when the data changes frequently, as the cached version gets out of sync, and the primary data store must be accessed every time.

It's important to note that a cache should not be used as permanent data storage. They are almost always implemented in volatile memory because it is faster, and thus should be considered transient.

Advantages

Below are some advantages of caching:

- Improves performance
- Reduce latency
- Reduce load on the database
- Reduce network cost
- Increase Read Throughput

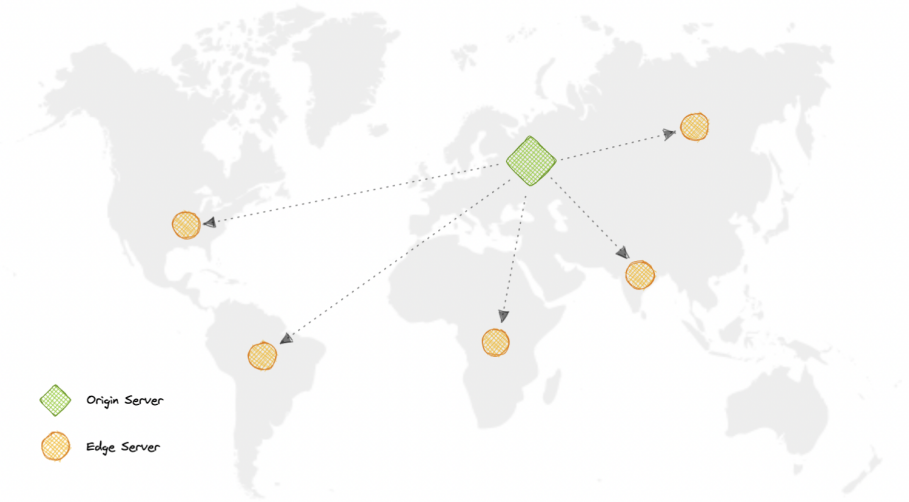
Examples

Here are some commonly used technologies for caching:

- Redis
- Memcached
- Amazon Elasticache
- Aerospike

Content Delivery Network (CDN)

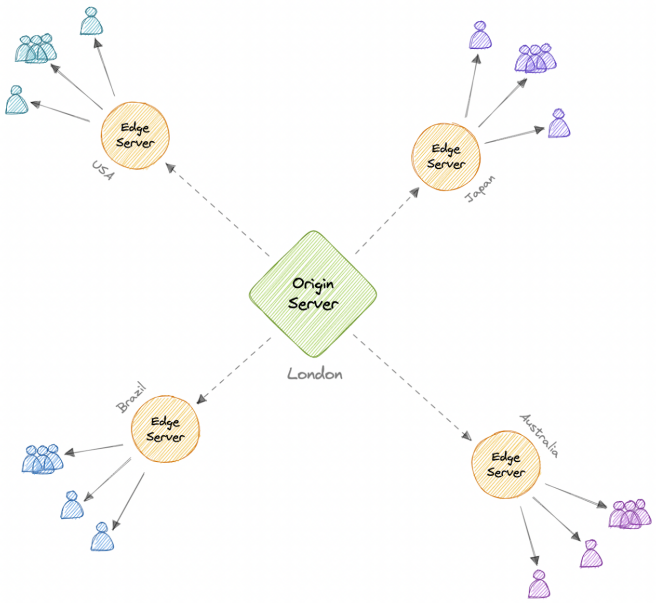
A content delivery network (CDN) is a geographically distributed group of servers that work together to provide fast delivery of internet content. Generally, static files such as HTML/CSS/JS, photos, and videos are served from CDN.



Why use a CDN?

Content Delivery Network (CDN) increases content availability and redundancy while reducing bandwidth costs and improving security. Serving content from CDNs can significantly improve performance as users receive content from data centers close to them and our servers do not have to serve requests that the CDN fulfills.

How does a CDN work?



In a CDN, the origin server contains the original versions of the content while the edge servers are numerous and distributed across various locations around the world.

To minimize the distance between the visitors and the website's server, a CDN stores a cached version of its content in multiple geographical locations known as edge locations. Each edge location contains several caching servers responsible for content delivery to visitors within its proximity.

Once the static assets are cached on all the CDN servers for a particular location, all subsequent website visitor requests for static assets will be delivered from these edge servers instead of the origin, thus reducing the origin load and improving scalability.

For example, when someone in the UK requests our website which might be hosted in the USA, they will be served from the closest edge location such as the London edge location. This is much quicker than having the visitor make a complete request to the origin server which will increase the latency.

Types

CDNs are generally divided into two types:

Push CDNs

Push CDNs receive new content whenever changes occur on the server. We take full responsibility for providing content, uploading directly to the CDN, and rewriting URLs to point to the CDN. We can configure when content expires and when it is updated. Content is uploaded only when it is new or changed, minimizing traffic, but maximizing storage.

Sites with a small amount of traffic or sites with content that isn't often updated work well with push CDNs. Content is placed on the CDNs once, instead of being re-pulled at regular intervals.

Pull CDNs

In a Pull CDN situation, the cache is updated based on request. When the client sends a request that requires static assets to be fetched from the CDN if the CDN doesn't have it, then it will fetch the newly updated assets from the origin server and populate its cache with this new asset, and then send this new cached asset to the user.

Contrary to the Push CDN, this requires less maintenance because cache updates on CDN nodes are performed based on requests from the client to the origin server. Sites with heavy traffic work well with pull CDNs, as traffic is spread out more evenly with only recently-requested content remaining on the CDN.

Disadvantages

As we all know good things come with extra costs, so let's discuss some disadvantages of CDNs:

- **Extra charges**: It can be expensive to use a CDN, especially for high-traffic services.
- **Restrictions**: Some organizations and countries have blocked the domains or IP addresses of popular CDNs.
- **Location**: If most of our audience is located in a country where the CDN has no servers, the data on our website may have to travel further than without using any CDN.

Examples

Here are some widely used CDNs:

- Amazon CloudFront
- Google Cloud CDN
- Cloudflare CDN
- Fastly

Proxy

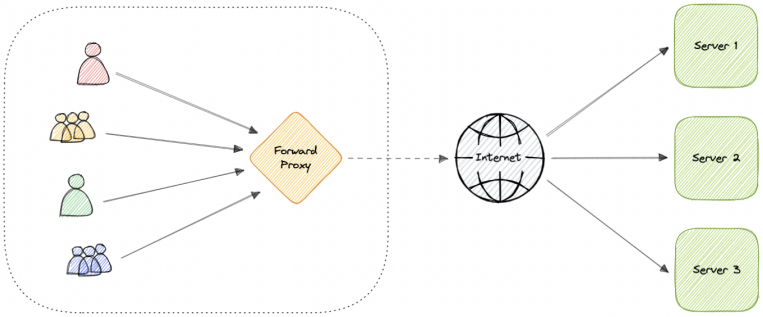
A proxy server is an intermediary piece of hardware/software sitting between the client and the backend server. It receives requests from clients and relays them to the origin servers. Typically, proxies are used to filter requests, log requests, or sometimes transform requests (by adding/removing headers, encrypting/decrypting, or compression).

Types

There are two types of proxies:

Forward Proxy

A forward proxy, often called a proxy, proxy server, or web proxy is a server that sits in front of a group of client machines. When those computers make requests to sites and services on the internet, the proxy server intercepts those requests and then communicates with web servers on behalf of those clients, like a middleman.



Advantages

Here are some advantages of a forward proxy:

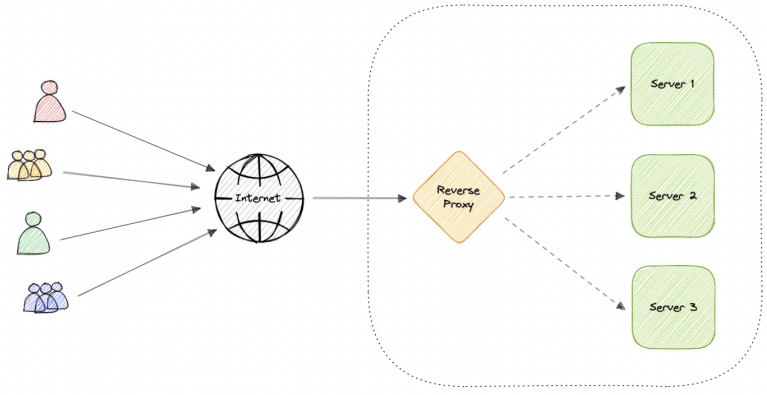
- Block access to certain content
- Allows access to geo-restricted content
- Provides anonymity
- Avoid other browsing restrictions

Although proxies provide the benefits of anonymity, they can still track our personal information. Setup and maintenance of a proxy server can be costly and requires configurations.

Reverse Proxy

A reverse proxy is a server that sits in front of one or more web servers, intercepting requests from clients. When clients send requests to the origin server of a website, those requests are intercepted by the reverse proxy server.

The difference between a forward and reverse proxy is subtle but important. A simplified way to sum it up would be to say that a forward proxy sits in front of a client and ensures that no origin server ever communicates directly with that specific client. On the other hand, a reverse proxy sits in front of an origin server and ensures that no client ever communicates directly with that origin server.



Introducing reverse proxy results in increased complexity. A single reverse proxy is a single point of failure, configuring multiple reverse proxies (i.e. a failover) further increases complexity.

Advantages

Here are some advantages of using a reverse proxy:

- Improved security
- Caching
- SSL encryption
- Load balancing
- Scalability and flexibility

Load balancer vs Reverse Proxy

Wait, isn't reverse proxy similar to a load balancer? Well, no as a load balancer is useful when we have multiple servers. Often, load balancers route traffic to a set of servers serving the same function, while reverse proxies can be useful even with just one web server or application server. A reverse proxy can also act as a load balancer but not the other way around.

Examples

Below are some commonly used proxy technologies:

- Nginx
- HAProxy
- Traefik
- Envoy

Availability

Availability is the time a system remains operational to perform its required function in a specific period. It is a simple measure of the percentage of time that a system, service, or machine remains operational under normal conditions.

The Nine's of availability

Availability is often quantified by uptime (or downtime) as a percentage of time the service is available. It is generally measured in the number of 9s.

$$Availability = \frac{Uptime}{(Uptime+Downtime)}$$

If availability is 99.00% available, it is said to have "2 nines" of availability, and if it is 99.9%, it is called "3 nines", and so on.

Availability (Percent)	Downtime (Year)	Downtime (Month)	Downtime (Week)
90% (one nine)	36.53 days	72 hours	16.8 hours
99% (two nines)	3.65 days	7.20 hours	1.68 hours
99.9% (three nines)	8.77 hours	43.8 minutes	10.1 minutes
99.99% (four nines)	52.6 minutes	4.32 minutes	1.01 minutes
99.999% (five nines)	5.25 minutes	25.9 seconds	6.05 seconds
99.9999% (six nines)	31.56 seconds	2.59 seconds	604.8 milliseconds

99.99999% (seven nines)	3.15 seconds	263 milliseconds	60.5 milliseconds
99.999999% (eight nines)	315.6 milliseconds	26.3 milliseconds	6 milliseconds
99.9999999% (nine nines)	31.6 milliseconds	2.6 milliseconds	0.6 milliseconds

Availability in Sequence vs Parallel

If a service consists of multiple components prone to failure, the service's overall availability depends on whether the components are in sequence or in parallel.

Sequence

Overall availability decreases when two components are in sequence.

$$Availability\ (Total)\ =\ Availability\ (Foo)\ *\ Availability\ (Bar)$$

For example, if both `Foo` and `Bar` each had 99.9% availability, their total availability in sequence would be 99.8%.

Parallel

Overall availability increases when two components are in parallel.

$$Availability\ (Total)\ =\ 1\ -\ (1\ -\ Availability\ (Foo))\ *\ (1\ -\ Availability\ (Bar))$$

For example, if both `Foo` and `Bar` each had 99.9% availability, their total availability in parallel would be 99.9999%.

Availability vs Reliability

If a system is reliable, it is available. However, if it is available, it is not necessarily reliable. In other words, high reliability contributes to high availability, but it is possible to achieve high availability even with an unreliable system.

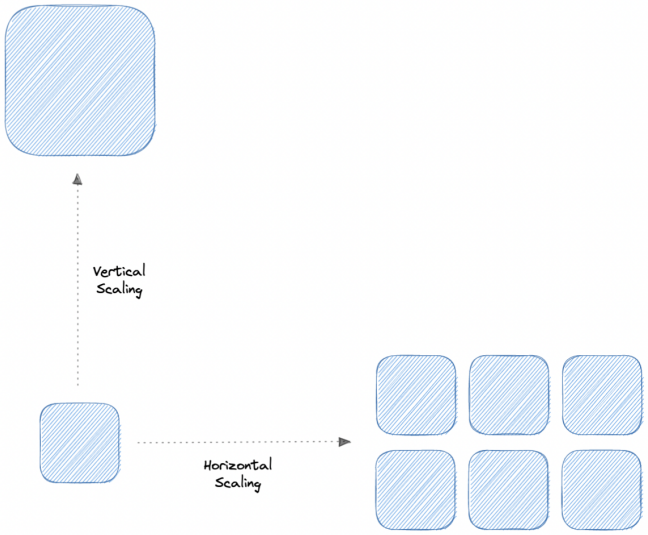
High availability vs Fault Tolerance

Both high availability and fault tolerance apply to methods for providing high uptime levels. However, they accomplish the objective differently.

A fault-tolerant system has no service interruption but a significantly higher cost, while a highly available system has minimal service interruption. Fault-tolerance requires full hardware redundancy as if the main system fails, with no loss in uptime, another system should take over.

Scalability

Scalability is the measure of how well a system responds to changes by adding or removing resources to meet demands.



Let's discuss different types of scaling:

Vertical scaling

Vertical scaling (also known as scaling up) expands a system's scalability by adding more power to an existing machine. In other words, vertical scaling refers to improving an application's capability via increasing hardware capacity.

Advantages

- Simple to implement
- Easier to manage
- Data consistent

Disadvantages

- Risk of high downtime
- Harder to upgrade
- Can be a single point of failure

Horizontal scaling

Horizontal scaling (also known as scaling out) expands a system's scale by adding more machines. It improves the performance of the server by adding more instances to the existing pool of servers, allowing the load to be distributed more evenly.

Advantages

- Increased redundancy
- Better fault tolerance
- Flexible and efficient
- Easier to upgrade

Disadvantages

- Increased complexity
- Data inconsistency
- Increased load on downstream services

Storage

Storage is a mechanism that enables a system to retain data, either temporarily or permanently. This topic is mostly skipped over in the context of system design, however, it is important to have a basic understanding of some common types of storage techniques that can help us fine-tune our storage components. Let's discuss some important storage concepts:

RAID

RAID (Redundant Array of Independent Disks) is a way of storing the same data on multiple hard disks or solid-state drives (SSDs) to protect data in the case of a drive failure.

There are different RAID levels, however, and not all have the goal of providing redundancy. Let's discuss some commonly used RAID levels:

- **RAID 0**: Also known as striping, data is split evenly across all the drives in the array.
- **RAID 1**: Also known as mirroring, at least two drives contains the exact copy of a set of data. If a drive fails, others will still work.
- **RAID 5**: Striping with parity. Requires the use of at least 3 drives, striping the data across multiple drives like RAID 0, but also has a parity distributed across the drives.
- **RAID 6**: Striping with double parity. RAID 6 is like RAID 5, but the parity data are written to two drives.
- **RAID 10**: Combines striping plus mirroring from RAID 0 and RAID 1. It provides security by mirroring all data on secondary drives while using striping across each set of drives to speed up data transfers.

Comparison

Let's compare all the features of different RAID levels:

Features	RAID 0	RAID 1	RAID 5	RAID 6	RAID 10
Description	Striping	Mirroring	Striping with Parity	Striping with double parity	Striping and Mirroring
Minimum Disks	2	2	3	4	4
Read Performance	High	High	High	High	High
Write Performance	High	Medium	High	High	Medium
Cost	Low	High	Low	Low	High
Fault Tolerance	None	Single-drive failure	Single-drive failure	Two-drive failure	Up to one disk failure in each sub-array
Capacity Utilization	100%	50%	67%-94%	50%-80%	50%

Volumes

Volume is a fixed amount of storage on a disk or tape. The term volume is often used as a synonym for the storage itself, but it is possible for a single disk to contain more than one volume or a volume to span more than one disk.

File storage

File storage is a solution to store data as files and present it to its final users as a hierarchical directories structure. The main advantage is to provide a user-friendly solution to store and retrieve files. To locate a file in file storage, the complete path of the file is required. It is economical and easily structured and is usually found on hard drives, which means that they appear exactly the same for the user and on the hard drive.

Example: Amazon EFS, Azure files, Google Cloud Filestore, etc.

Block storage

Block storage divides data into blocks (chunks) and stores them as separate pieces. Each block of data is given a unique identifier, which allows a storage system to place the smaller pieces of data wherever it is most convenient.

Block storage also decouples data from user environments, allowing that data to be spread across multiple environments. This creates multiple paths to the data and allows the user to retrieve it quickly. When a user or application requests data from a block storage system, the underlying storage system reassembles the data blocks and presents the data to the user or application

Example: Amazon EBS.

Object Storage

Object storage, which is also known as object-based storage, breaks data files up into pieces called objects. It then stores those objects in a single repository, which can be spread out across multiple networked systems.

Example: Amazon S3, Azure Blob Storage, Google Cloud Storage, etc.

NAS

A NAS (Network Attached Storage) is a storage device connected to a network that allows storage and retrieval of data from a central location for authorized network users. NAS devices are flexible, meaning that as we need additional storage, we can add to what we have. It's faster, less expensive, and provides all the benefits of a public cloud on-site, giving us complete control.

HDFS

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. It has many similarities with existing distributed file systems.

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks, all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance.

Chapter II

Databases and DBMS

What is a Database?

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a Database Management System (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database.

What is DBMS?

A database typically requires a comprehensive database software program known as a Database Management System (DBMS). A DBMS serves as an interface between the database and its end-users or programs, allowing users to retrieve, update, and manage how the information is organized and optimized. A DBMS also facilitates oversight and control of databases, enabling a variety of administrative operations such as performance monitoring, tuning, and backup and recovery.

Components

Here are some common components found across different databases:

Schema

The role of a schema is to define the shape of a data structure, and specify what kinds of data can go where. Schemas can be strictly enforced across the entire database, loosely enforced on part of the database, or they might not exist at all.

Table

Each table contains various columns just like in a spreadsheet. A table can have as meager as two columns and upwards of a hundred or more columns, depending upon the kind of information being put in the table.

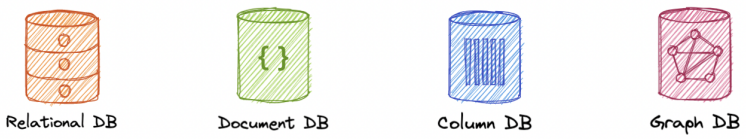
Column

A column contains a set of data values of a particular type, one value for each row of the database. A column may contain text values, numbers, enums, timestamps, etc.

Row

Data in a table is recorded in rows. There can be thousands or millions of rows in a table having any particular information.

Types



Below are different types of databases:

- SQL
- NoSQL
    - Document
    - Key-value
    - Graph
    - Timeseries
    - Wide column
    - Multi-model

SQL and NoSQL databases are broad topics and will be discussed separately in SQL databases and NoSQL databases. Learn how they compare to each other in SQL vs NoSQL databases.

Challenges

Some common challenges faced while running databases at scale:

- **Absorbing significant increases in data volume**: The explosion of data coming in from sensors, connected machines, and dozens of other sources.
- **Ensuring data security**: Data breaches are happening everywhere these days, it's more important than ever to ensure that data is secure but also easily accessible to users.
- **Keeping up with demand**: Companies need real-time access to their data to support timely decision-making and to take advantage of new opportunities.
- **Managing and maintaining the database and infrastructure**: As databases become more complex and data volumes grow, companies are faced with the expense of hiring additional talent to manage their databases.
- **Removing limits on scalability**: A business needs to grow if it's going to survive, and its data management must grow along with it. But it's very difficult to predict how much capacity the company will need, particularly with on-premises databases.
- **Ensuring data residency, data sovereignty, or latency requirements**: Some organizations have use cases that are better suited to run on-premises. In those cases, engineered systems that are pre-configured and pre-optimized for running the database are ideal.

SQL Databases

A SQL (or relational) database is a collection of data items with pre-defined relationships between them. These items are organized as a set of tables with columns and rows. Tables are used to hold information about the objects to be represented in the database. Each column in a table holds a certain kind of data and a field stores the actual value of an attribute. The rows in the table represent a collection of related values of one object or entity.

Each row in a table could be marked with a unique identifier called a primary key, and rows among multiple tables can be made related using foreign keys. This data can be accessed in many different ways without re-organizing the database tables themselves. SQL databases usually follow the ACID consistency model.

Materialized views

A materialized view is a pre-computed data set derived from a query specification and stored for later use. Because the data is pre-computed, querying a materialized view is faster than executing a query against the base table of the view. This performance difference can be significant when a query is run frequently or is sufficiently complex.

It also enables data subsetting and improves the performance of complex queries that run on large data sets which reduces network loads. There are other uses of materialized views, but they are mostly used for performance and replication.

N+1 query problem

The N+1 query problem happens when the data access layer executes N additional SQL statements to fetch the same data that could have been retrieved when executing the primary SQL query. The larger the value of N, the more queries will be executed, the larger the performance impact.

This is commonly seen in GraphQL and ORM (Object-Relational Mapping) tools and can be addressed by optimizing the SQL query or using a data loader that batches consecutive requests and makes a single data request under the hood.

Advantages

Let's look at some advantages of using relational databases:

- Simple and accurate
- Accessibility
- Data consistency
- Flexibility

Disadvantages

Below are the disadvantages of relational databases:

- Expensive to maintain
- Difficult schema evolution
- Performance hits (join, denormalization, etc.)
- Difficult to scale due to poor horizontal scalability

Examples

Here are some commonly used relational databases:

- PostgreSQL
- MySQL
- MariaDB
- Amazon Aurora

NoSQL Databases

NoSQL is a broad category that includes any database that doesn't use SQL as its primary data access language. These types of databases are also sometimes referred to as non-relational databases. Unlike in relational databases, data in a NoSQL database doesn't have to conform to a pre-defined schema. NoSQL databases follow BASE consistency model.

Below are different types of NoSQL databases:

Document

A document database (also known as a document-oriented database or a document store) is a database that stores information in documents. They are general-purpose databases that serve a variety of use cases for both transactional and analytical applications.

Advantages

- Intuitive and flexible
- Easy horizontal scaling
- Schemaless

Disadvantages

- Schemaless
- Non-relational

Examples

- MongoDB
- Amazon DocumentDB
- CouchDB

Key-value

One of the simplest types of NoSQL databases, key-value databases save data as a group of key-value pairs made up of two data items each. They're also sometimes referred to as a key-value store.

Advantages

- Simple and performant
- Highly scalable for high volumes of traffic
- Session management
- Optimized lookups

Disadvantages

- Basic CRUD
- Values can't be filtered
- Lacks indexing and scanning capabilities
- Not optimized for complex queries

Examples

- Redis
- Memcached
- Amazon DynamoDB
- Aerospike

Graph

A graph database is a NoSQL database that uses graph structures for semantic queries with nodes, edges, and properties to represent and store data instead of tables or documents.

The graph relates the data items in the store to a collection of nodes and edges, the edges representing the relationships between the nodes. The relationships allow data in the store to be linked together directly and, in many cases, retrieved with one operation.

Advantages

- Query speed
- Agile and flexible
- Explicit data representation

Disadvantages

- Complex
- No standardized query language

Use cases

- Fraud detection
- Recommendation engines
- Social networks
- Network mapping

Examples:

- Neo4j
- ArangoDB
- Amazon Neptune
- JanusGraph

Time series

A time-series database is a database optimized for time-stamped, or time series, data.

Advantages

- Fast insertion and retrieval
- Efficient data storage

Use cases

- IoT data
- Metrics analysis
- Application monitoring
- Understand financial trends

Examples

- InfluxDB
- Apache Druid

Wide column

Wide column databases, also known as wide column stores, are schema-agnostic. Data is stored in column families, rather than in rows and columns.

Advantages:

- Highly scalable, can handle petabytes of data
- Ideal for real-time big data applications

Disadvantages:

- Expensive
- Increased write time

Use cases:

- Business analytics
- Attribute-based data storage

Examples:

- BigTable
- Apache Cassandra
- ScyllaDB

Multi-model

Multi-model databases combine different database models (i.e. relational, graph, key-value, document, etc.) into a single, integrated backend. This means they can accommodate various data types, indexes, queries, and store data in more than one model.

Advantages:

- Flexibility
- Suitable for complex projects
- Data consistent

Disadvantages:

- Complex
- Less mature

Examples:

- ArangoDB
- Azure Cosmos DB
- Couchbase

SQL vs NoSQL Databases

In the world of databases, there are two main types of solutions, SQL (relational) and NoSQL (non-relational) databases. Both of them differ in the way they were built, the kind of information they store, and how they store it. Relational databases are structured and have predefined schemas while non-relational databases are unstructured, distributed, and have a dynamic schema.

High-level differences

Here are some high-level differences between SQL and NoSQL:

Storage

SQL stores data in tables, where each row represents an entity and each column represents a data point about that entity.

NoSQL databases have different data storage models such as key-value, graph, document, etc.

Schema

In SQL, each record conforms to a fixed schema, meaning the columns must be decided and chosen before data entry and each row must have data for each column. The schema can be altered later, but it involves modifying the database using migrations.

Whereas in NoSQL, schemas are dynamic. Fields can be added on the fly, and each *record* (or equivalent) doesn't have to contain data for each *field*.

Querying

SQL databases use SQL (structured query language) for defining and manipulating the data, which is very powerful.

In a NoSQL database, queries are focused on a collection of documents. Different databases have different syntax for querying.

Scalability

In most common situations, SQL databases are vertically scalable, which can get very expensive. It is possible to scale a relational database across multiple servers, but this is a challenging and time-consuming process.

On the other hand, NoSQL databases are horizontally scalable, meaning we can add more servers easily to our NoSQL database infrastructure to handle large traffic. Any cheap commodity hardware or cloud instances can host NoSQL databases, thus making it a lot more cost-effective than vertical scaling. A lot of NoSQL technologies also distribute data across servers automatically.

Reliability

The vast majority of relational databases are ACID compliant. So, when it comes to data reliability and a safe guarantee of performing transactions, SQL databases are still the better bet.

Most of the NoSQL solutions sacrifice ACID compliance for performance and scalability.

Reasons

As always we should always pick the technology that fits the requirements better. So, let's look at some reasons for picking SQL or NoSQL based database:

For SQL:

- Structured data with strict schema
- Relational data
- Need for complex joins
- Transactions
- Lookups by index are very fast

For NoSQL:

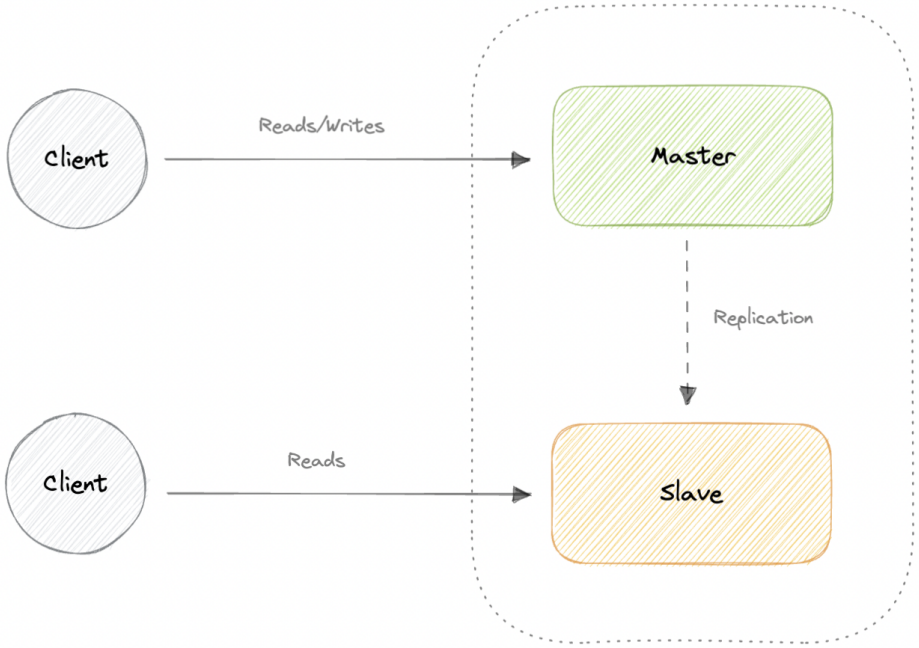
- Dynamic or flexible schema
- Non-relational data
- No need for complex joins
- Very data-intensive workload
- Very high throughput for IOPS

Database Replication

Replication is a process that involves sharing information to ensure consistency between redundant resources such as multiple databases, to improve reliability, fault-tolerance, or accessibility.

Master-Slave Replication

The master serves reads and writes, replicating writes to one or more slaves, which serve only reads. Slaves can also replicate additional slaves in a tree-like fashion. If the master goes offline, the system can continue to operate in read-only mode until a slave is promoted to a master or a new master is provisioned.



Advantages

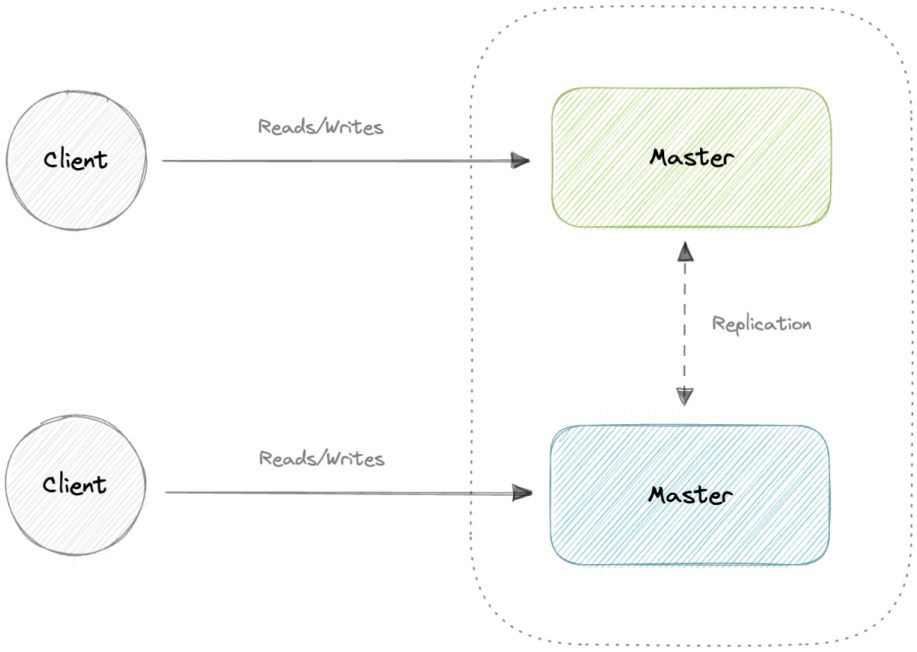
- Backups of the entire database of relatively no impact on the master.
- Applications can read from the slave(s) without impacting the master.
- Slaves can be taken offline and synced back to the master without any downtime.

Disadvantages

- Replication adds more hardware and additional complexity.
- Downtime and possibly loss of data when a master fails.
- All writes also have to be made to the master in a master-slave architecture.
- The more read slaves, the more we have to replicate, which will increase replication lag.

Master-Master Replication

Both masters serve reads/writes and coordinate with each other. If either master goes down, the system can continue to operate with both reads and writes.



Advantages

- Applications can read from both masters.
- Distributes write load across both master nodes.
- Simple, automatic, and quick failover.

Disadvantages

- Not as simple as master-slave to configure and deploy.
- Either loosely consistent or have increased write latency due to synchronization.
- Conflict resolution comes into play as more write nodes are added and as latency increases.

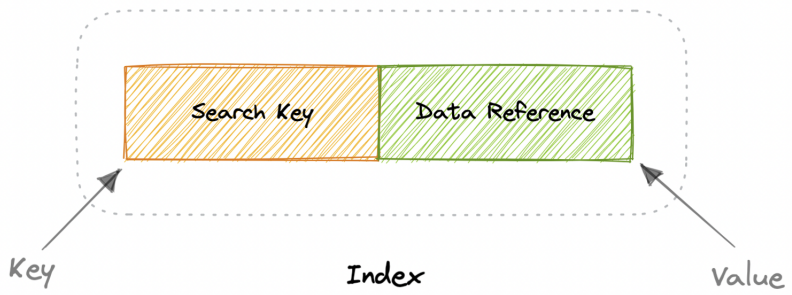
Synchronous vs Asynchronous replication

The primary difference between synchronous and asynchronous replication is how the data is written to the replica. In synchronous replication, data is written to primary storage and the replica simultaneously. As such, the primary copy and the replica should always remain synchronized.

In contrast, asynchronous replication copies the data to the replica after the data is already written to the primary storage. Although the replication process may occur in near-real-time, it is more common for replication to occur on a scheduled basis and it is more cost-effective.

Indexes

Indexes are well known when it comes to databases, they are used to improve the speed of data retrieval operations on the data store. An index makes the trade-offs of increased storage overhead, and slower writes (since we not only have to write the data but also have to update the index) for the benefit of faster reads. Indexes are used to quickly locate data without having to examine every row in a database table. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access to ordered records.

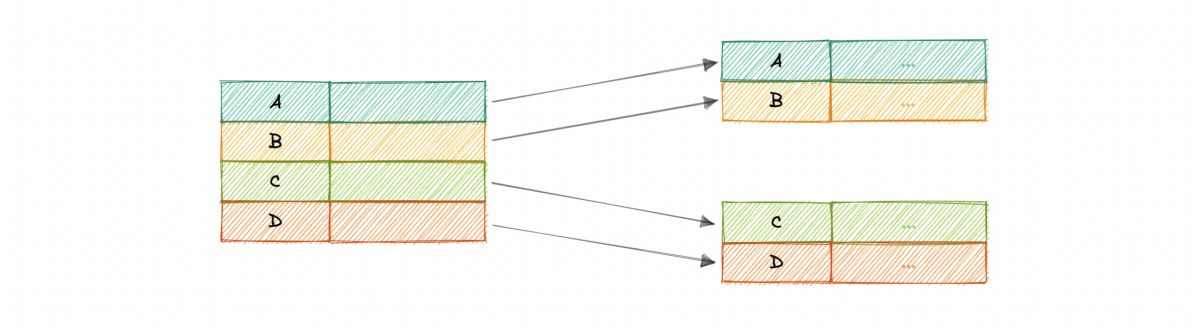


An index is a data structure that can be perceived as a table of contents that points us to the location where actual data lives. So when we create an index on a column of a table, we store that column and a pointer to the whole row in the index. Indexes are also used to create different views of the same data. For large data sets, this is an excellent way to specify different filters or sorting schemes without resorting to creating multiple additional copies of the data.

One quality that database indexes can have is that they can be dense or sparse. Each of these index qualities comes with its own trade-offs. Let's look at how each index type would work:

Dense Index

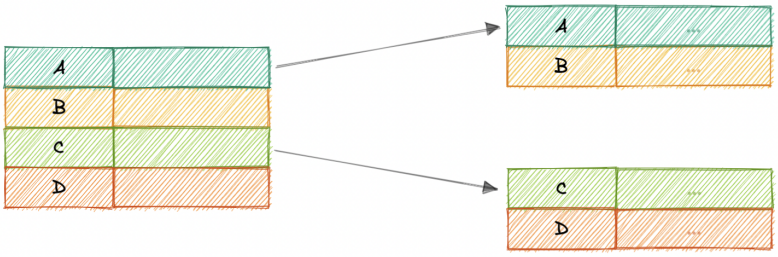
In a dense index, an index record is created for every row of the table. Records can be located directly as each record of the index holds the search key value and the pointer to the actual record.



Dense indexes require more maintenance than sparse indexes at write-time. Since every row must have an entry, the database must maintain the index on inserts, updates, and deletes. Having an entry for every row also means that dense indexes will require more memory. The benefit of a dense index is that values can be quickly found with just a binary search. Dense indexes also do not impose any ordering requirements on the data.

Sparse Index

In a sparse index, records are created only for some of the records.



Sparse indexes require less maintenance than dense indexes at write-time since they only contain a subset of the values. This lighter maintenance burden means that inserts, updates, and deletes will be faster. Having fewer entries also means that the index will use less memory. Finding data is slower since a scan across the page typically follows the binary search. Sparse indexes are also optional when working with ordered data.

Normalization and Denormalization

Terms

Before we go any further, let's look at some commonly used terms in normalization and denormalization.

Keys

Primary key: Column or group of columns that can be used to uniquely identify every row of the table.

Composite key: A primary key made up of multiple columns.

Super key: Set of all keys that can uniquely identify all the rows present in a table.

Candidate key: Attributes that identify rows uniquely in a table.

Foreign key: It is a reference to a primary key of another table.

Alternate key: Keys that are not primary keys are known as alternate keys.

Surrogate key: A system-generated value that uniquely identifies each entry in a table when no other column was able to hold properties of a primary key.

Dependencies

Partial dependency: Occurs when the primary key determines some other attributes.

Functional dependency: It is a relationship that exists between two attributes, typically between the primary key and non-key attribute within a table.

Transitive functional dependency: Occurs when some non-key attribute determines some other attribute.

Anomalies

Database anomaly happens when there is a flaw in the database due to incorrect planning or storing everything in a flat database. This is generally addressed by the process of normalization.

There are three types of database anomalies:

Insertion anomaly: Occurs when we are not able to insert certain attributes in the database without the presence of other attributes.

Update anomaly: Occurs in case of data redundancy and partial update. In other words, a correct update of the database needs other actions such as addition, deletion, or both.

Deletion anomaly: Occurs where deletion of some data requires deletion of other data.

Example

Let's consider the following table which is not normalized:

ID	Name	Role	Team
1	Peter	Software Engineer	A
2	Brian	DevOps Engineer	B
3	Hailey	Product Manager	C
4	Hailey	Product Manager	C
5	Steve	Frontend Engineer	D

Let's imagine, we hired a new person "John" but they might not be assigned a team immediately. This will cause an *insertion anomaly* as the team attribute is not yet present.

Next, let's say Hailey from Team C got promoted, to reflect that change in the database, we will need to update 2 rows to maintain consistency which can cause an *update anomaly*.

Finally, we would like to remove Team B but to do that we will also need to remove additional information such as name and role, this is an example of a *deletion anomaly*.

Normalization

Normalization is the process of organizing data in a database. This includes creating tables and establishing relationships between those tables according to rules designed both to protect the data and to make the database more flexible by eliminating redundancy and inconsistent dependency.

Why do we need normalization?

The goal of normalization is to eliminate redundant data and ensure data is consistent. A fully normalized database allows its structure to be extended to accommodate new types of data without changing the existing structure too much. As a result, applications interacting with the database are minimally affected.

Normal forms

Normal forms are a series of guidelines to ensure that the database is normalized. Let's discuss some essential normal forms:

1NF

For a table to be in the first normal form (1NF), it should follow the following rules:

- Repeating groups are not permitted.
- Identify each set of related data with a primary key.
- Set of related data should have a separate table.
- Mixing data types in the same column is not permitted.

2NF

For a table to be in the second normal form (2NF), it should follow the following rules:

- Satisfies the first normal form (1NF).
- Should not have any partial dependency.

3NF

For a table to be in the third normal form (3NF), it should follow the following rules:

- Satisfies the second normal form (2NF).
- Transitive functional dependencies are not permitted.

BCNF

Boyce-Codd normal form (or BCNF) is a slightly stronger version of the third normal form (3NF) used to address certain types of anomalies not dealt with by 3NF as originally defined. Sometimes it is also known as the 3.5 normal form (3.5NF).

For a table to be in the Boyce-Codd normal form (BCNF), it should follow the following rules:

- Satisfied the third normal form (3NF).
- For every functional dependency X → Y, X should be the super key.

There are more normal forms such as 4NF, 5NF, and 6NF but we won't discuss them here. Check out this amazing video that goes into detail.

In a relational database, a relation is often described as *"normalized"* if it meets the third normal form. Most 3NF relations are free of insertion, update, and deletion anomalies.

As with many formal rules and specifications, real-world scenarios do not always allow for perfect compliance. If you decide to violate one of the first three rules of normalization, make sure that your application anticipates any problems that could occur, such as redundant data and inconsistent dependencies.

Advantages

Here are some advantages of normalization:

- Reduces data redundancy.
- Better data design.
- Increases data consistency.
- Enforces referential integrity.

Disadvantages

Let's look at some disadvantages of normalization:

- Data design is complex.
- Slower performance.
- Maintenance overhead.
- Require more joins.

Denormalization

Denormalization is a database optimization technique in which we add redundant data to one or more tables. This can help us avoid costly joins in a relational database. It attempts to improve read performance at the expense of some write performance. Redundant copies of the data are written in multiple tables to avoid expensive joins.

Once data becomes distributed with techniques such as federation and sharding, managing joins across the network further increases complexity. Denormalization might circumvent the need for such complex joins.

Note: Denormalization does not mean reversing normalization.

Advantages

Let's look at some advantages of denormalization:

- Retrieving data is faster.
- Writing queries is easier.
- Reduction in number of tables.
- Convenient to manage.

Disadvantages

Below are some disadvantages of denormalization:

- Expensive inserts and updates.
- Increases complexity of database design.
- Increases data redundancy.
- More chances of data inconsistency.

ACID and BASE Consistency Models

Let's discuss the ACID and BASE consistency models.

ACID

The term ACID stands for Atomicity, Consistency, Isolation, and Durability. ACID properties are used for maintaining data integrity during transaction processing.

In order to maintain consistency before and after a transaction relational databases follow ACID properties. Let us understand these terms:

Atomic

All operations in a transaction succeed or every operation is rolled back.

Consistent

On the completion of a transaction, the database is structurally sound.

Isolated

Transactions do not contend with one another. Contentious access to data is moderated by the database so that transactions appear to run sequentially.

Durable

Once the transaction has been completed and the writes and updates have been written to the disk, it will remain in the system even if a system failure occurs.

BASE

With the increasing amount of data and high availability requirements, the approach to database design has also changed dramatically. To increase the ability to scale and at the same time be highly available, we move the logic from the database to separate servers. In this way, the database becomes more independent and focused on the actual process of storing data.

In the NoSQL database world, ACID transactions are less common as some databases have loosened the requirements for immediate consistency, data freshness, and accuracy in order to gain other benefits, like scale and resilience.

BASE properties are much looser than ACID guarantees, but there isn't a direct one-for-one mapping between the two consistency models. Let us understand these terms:

Basic Availability

The database appears to work most of the time.

Soft-state

Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.

Eventual consistency

The data might not be consistent immediately but eventually, it becomes consistent. Reads in the system are still possible even though they may not give the correct response due to inconsistency.

ACID vs BASE Trade-offs

There's no right answer to whether our application needs an ACID or a BASE consistency model. Both the models have been designed to satisfy different

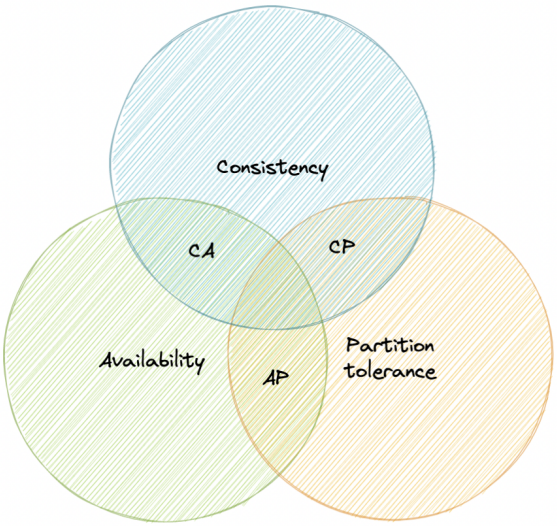
requirements. While choosing a database we need to keep the properties of both the models and the requirements of our application in mind.

Given BASE's loose consistency, developers need to be more knowledgeable and rigorous about consistent data if they choose a BASE store for their application. It's essential to be familiar with the BASE behavior of the chosen database and work within those constraints.

On the other hand, planning around BASE limitations can sometimes be a major disadvantage when compared to the simplicity of ACID transactions. A fully ACID database is the perfect fit for use cases where data reliability and consistency are essential.

CAP Theorem

CAP theorem states that a distributed system can deliver only two of the three desired characteristics Consistency, Availability, and Partition tolerance (CAP).



Let's take a detailed look at the three distributed system characteristics to which the CAP theorem refers.

Consistency

Consistency means that all clients see the same data at the same time, no matter which node they connect to. For this to happen, whenever data is written to one node, it must be instantly forwarded or replicated across all the nodes in the system before the write is deemed "successful".

Availability

Availability means that any client making a request for data gets a response, even if one or more nodes are down.

Partition tolerance

Partition tolerance means the system continues to work despite message loss or partial failure. A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network. Data is sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages.

Consistency-Availability Tradeoff

We live in a physical world and can't guarantee the stability of a network, so distributed databases must choose Partition Tolerance (P). This implies a tradeoff between Consistency (C) and Availability (A).

CA database

A CA database delivers consistency and availability across all nodes. It can't do this if there is a partition between any two nodes in the system, and therefore can't deliver fault tolerance.

Example: PostgreSQL, MariaDB.

CP database

A CP database delivers consistency and partition tolerance at the expense of availability. When a partition occurs between any two nodes, the system has to shut down the non-consistent node until the partition is resolved.

Example: MongoDB, Apache HBase.

AP database

An AP database delivers availability and partition tolerance at the expense of consistency. When a partition occurs, all nodes remain available but those at the wrong end of a partition might return an older version of data than others. When the

partition is resolved, the AP databases typically re-syncs the nodes to repair all inconsistencies in the system.

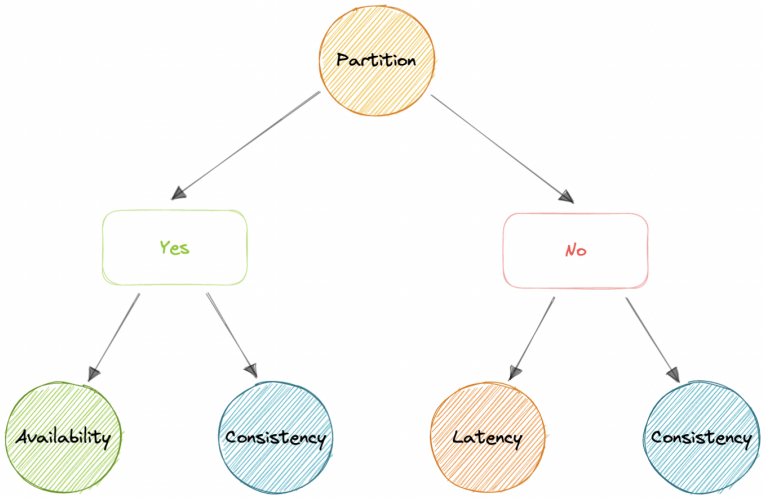
Example: Apache Cassandra, CouchDB.

PACELC Theorem

The PACELC theorem is an extension of the CAP theorem. The CAP theorem states that in the case of network partitioning (P) in a distributed system, one has to choose between Availability (A) and Consistency (C).

PACELC extends the CAP theorem by introducing latency (L) as an additional attribute of a distributed system. The theorem states that else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C).

The PACELC theorem was first described by Daniel J. Abadi.



PACELC theorem was developed to address a key limitation of the CAP theorem as it makes no provision for performance or latency.

For example, according to the CAP theorem, a database can be considered Available if a query returns a response after 30 days. Obviously, such latency would be unacceptable for any real-world application.

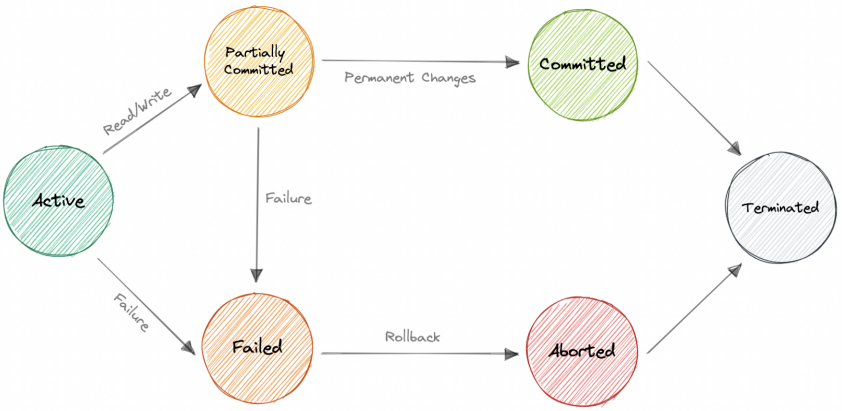
Transactions

A transaction is a series of database operations that are considered to be a *"single unit of work"*. The operations in a transaction either all succeed, or they all fail. In this way, the notion of a transaction supports data integrity when part of a system fails. Not all databases choose to support ACID transactions, usually because they are prioritizing other optimizations that are hard or theoretically impossible to implement together.

Usually, relational databases support ACID transactions, and non-relational databases don't (there are exceptions).

States

A transaction in a database can be in one of the following states:



Active

In this state, the transaction is being executed. This is the initial state of every transaction.

Partially Committed

When a transaction executes its final operation, it is said to be in a partially committed state.

Committed

If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

Failed

The transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.

Aborted

If any of the checks fail and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are aborted.

The database recovery module can select one of the two operations after a transaction aborts:

- Restart the transaction
- Kill the transaction

Terminated

If there isn't any roll-back or the transaction comes from the *committed state*, then the system is consistent and ready for a new transaction and the old transaction is terminated.

Distributed Transactions

A distributed transaction is a set of operations on data that is performed across two or more databases. It is typically coordinated across separate nodes connected by a network, but may also span multiple databases on a single server.

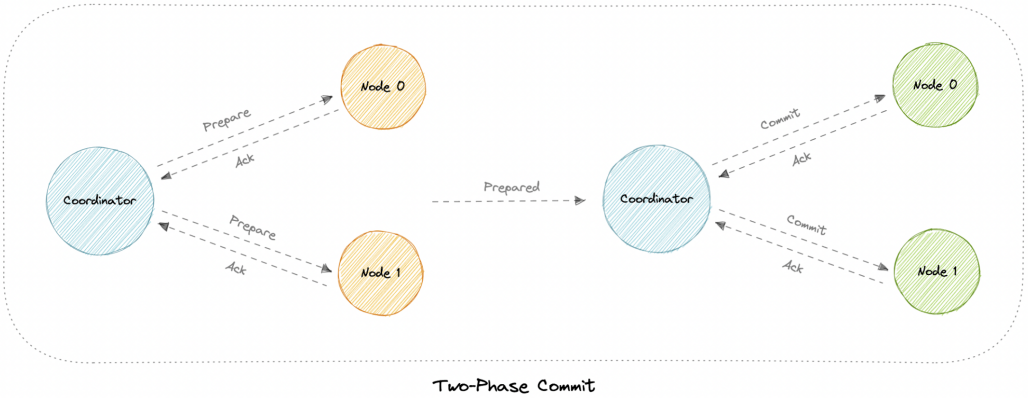
Why do we need distributed transactions?

Unlike an ACID transaction on a single database, a distributed transaction involves altering data on multiple databases. Consequently, distributed transaction processing is more complicated, because the database must coordinate the committing or rollback of the changes in a transaction as a self-contained unit.

In other words, all the nodes must commit, or all must abort and the entire transaction rolls back. This is why we need distributed transactions.

Now, let's look at some popular solutions for distributed transactions:

Two-Phase commit



Two-Phase Commit

The two-phase commit (2PC) protocol is a distributed algorithm that coordinates all the processes that participate in a distributed transaction on whether to commit or abort (roll back) the transaction.

This protocol achieves its goal even in many cases of temporary system failure and is thus widely used. However, it is not resilient to all possible failure configurations, and in rare cases, manual intervention is needed to remedy an outcome.

This protocol requires a coordinator node, which basically coordinates and oversees the transaction across different nodes. The coordinator tries to establish the consensus among a set of processes in two phases, hence the name.

Phases

Two-phase commit consists of the following phases:

Prepare phase

The prepare phase involves the coordinator node collecting consensus from each of the participant nodes. The transaction will be aborted unless each of the nodes responds that they're *prepared*.

Commit phase

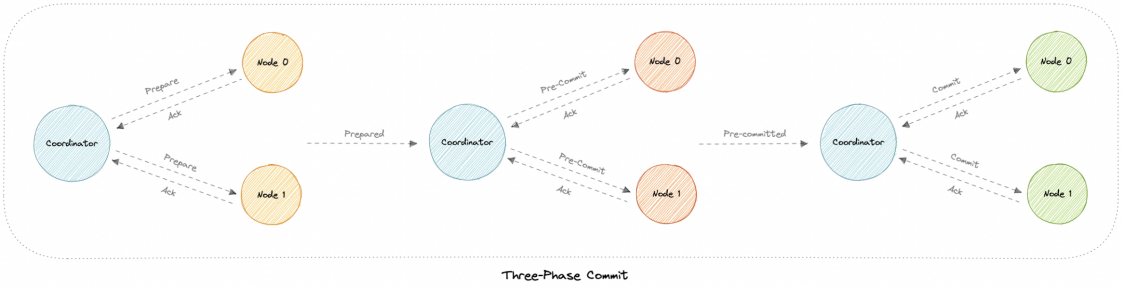
If all participants respond to the coordinator that they are *prepared*, then the coordinator asks all the nodes to commit the transaction. If a failure occurs, the transaction will be rolled back.

Problems

Following problems may arise in the two-phase commit protocol:

- What if one of the nodes crashes?
- What if the coordinator itself crashes?
- It is a blocking protocol.

Three-phase commit



Three-Phase Commit

Three-phase commit (3PC) is an extension of the two-phase commit where the commit phase is split into two phases. This helps with the blocking problem that occurs in the two-phase commit protocol.

Phases

Three-phase commit consists of the following phases:

Prepare phase

This phase is the same as the two-phase commit.

Pre-commit phase

Coordinator issues the pre-commit message and all the participating nodes must acknowledge it. If a participant fails to receive this message in time, then the transaction is aborted.

Commit phase

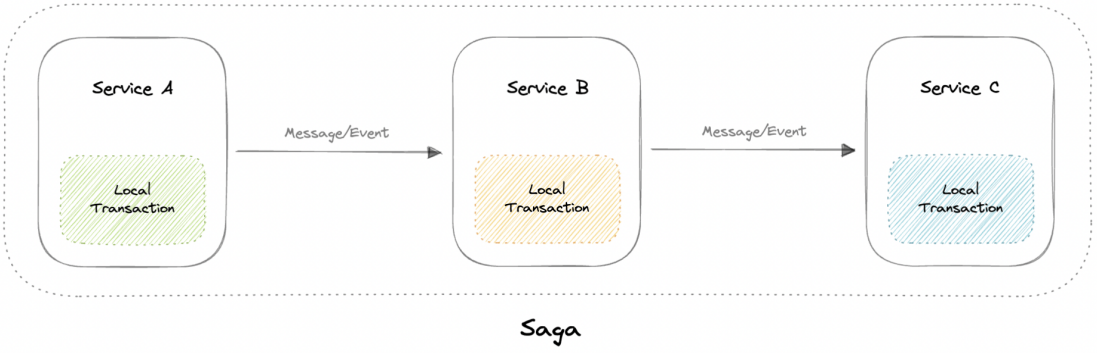
This step is also similar to the two-phase commit protocol.

Why is the Pre-commit phase helpful?

The pre-commit phase accomplishes the following:

- If the participant nodes are found in this phase, that means that *every* participant has completed the first phase. The completion of prepare phase is guaranteed.
- Every phase can now time out and avoid indefinite waits.

Sagas



A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

Coordination

There are two common implementation approaches:

- **Choreography**: Each local transaction publishes domain events that trigger local transactions in other services.

- **Orchestration**: An orchestrator tells the participants what local transactions to execute.

Problems

- The Saga pattern is particularly hard to debug.
- There's a risk of cyclic dependency between saga participants.
- Lack of participant data isolation imposes durability challenges.
- Testing is difficult because all services must be running to simulate a transaction.

Sharding

Before we discuss sharding, let's talk about data partitioning:

Data Partitioning

Data partitioning is a technique to break up a database into many smaller parts. It is the process of splitting up a database or a table across multiple machines to improve the manageability, performance, and availability of a database.

Methods

There are many different ways one could use to decide how to break up an application database into multiple smaller DBs. Below are two of the most popular methods used by various large-scale applications:

Horizontal Partitioning (or Sharding)

In this strategy, we split the table data horizontally based on the range of values defined by the *partition key*. It is also referred to as ***database sharding***.

Vertical Partitioning

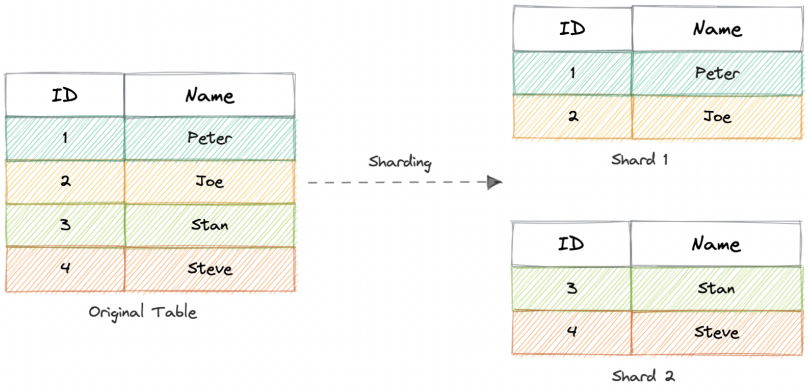
In vertical partitioning, we partition the data vertically based on columns. We divide tables into relatively smaller tables with few elements, and each part is present in a separate partition.

In this tutorial, we will specifically focus on sharding.

What is sharding?

Sharding is a database architecture pattern related to *horizontal partitioning*, which is the practice of separating one table's rows into multiple different tables, known as *partitions* or *shards*. Each partition has the same schema and columns, but also a

subset of the shared data. Likewise, the data held in each is unique and independent of the data held in other partitions.



The justification for data sharding is that, after a certain point, it is cheaper and more feasible to scale horizontally by adding more machines than to scale it vertically by adding powerful servers. Sharding can be implemented at both application or the database level.

Partitioning criteria

There are a large number of criteria available for data partitioning. Some most commonly used criteria are:

Hash-Based

This strategy divides the rows into different partitions based on a hashing algorithm rather than grouping database rows based on continuous indexes.

The disadvantage of this method is that dynamically adding/removing database servers becomes expensive.

List-Based

In list-based partitioning, each partition is defined and selected based on the list of values on a column rather than a set of contiguous ranges of values.

Range Based

Range partitioning maps data to various partitions based on ranges of values of the partitioning key. In other words, we partition the table in such a way that each partition contains rows within a given range defined by the partition key.

Ranges should be contiguous but not overlapping, where each range specifies a non-inclusive lower and upper bound for a partition. Any partitioning key values equal to or higher than the upper bound of the range are added to the next partition.

Composite

As the name suggests, composite partitioning partitions the data based on two or more partitioning techniques. Here we first partition the data using one technique, and then each partition is further subdivided into sub-partitions using the same or some other method.

Advantages

But why do we need sharding? Here are some advantages:

- **Availability**: Provides logical independence to the partitioned database, ensuring the high availability of our application. Here individual partitions can be managed independently.
- **Scalability**: Proves to increase scalability by distributing the data across multiple partitions.
- **Security**: Helps improve the system's security by storing sensitive and non-sensitive data in different partitions. This could provide better manageability and security to sensitive data.

- **Query Performance**: Improves the performance of the system. Instead of querying the whole database, now the system has to query only a smaller partition.
- **Data Manageability**: Divides tables and indexes into smaller and more manageable units.

Disadvantages

- **Complexity**: Sharding increases the complexity of the system in general.
- **Joins across shards**: Once a database is partitioned and spread across multiple machines it is often not feasible to perform joins that span multiple database shards. Such joins will not be performance efficient since data has to be retrieved from multiple servers.
- **Rebalancing**: If the data distribution is not uniform or there is a lot of load on a single shard, in such cases, we have to rebalance our shards so that the requests are as equally distributed among the shards as possible.

When to use sharding?

Here are some reasons why sharding might be the right choice:

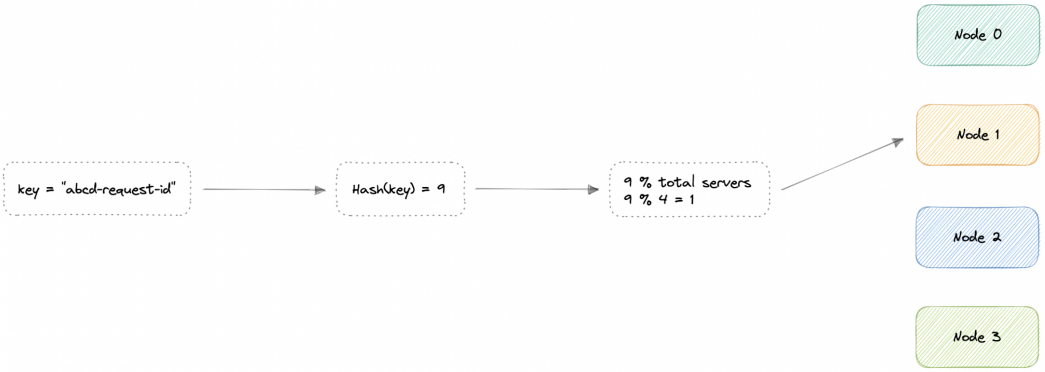
- Leveraging existing hardware instead of high-end machines.
- Maintain data in distinct geographic regions.
- Quickly scale by adding more shards.
- Better performance as each machine is under less load.
- When more concurrent connections are required.

Consistent Hashing

Let's first understand the problem we're trying to solve.

Why do we need this?

In traditional hashing-based distribution methods, we use a hash function to hash our partition keys (i.e. request ID or IP). Then if we use the modulo against the total number of nodes (server or databases). This will give us the node where we want to route our request.



$$Hash(key_1) \rightarrow H_1 \bmod N = Node_0$$
$$Hash(key_2) \rightarrow H_2 \bmod N = Node_1$$
$$Hash(key_3) \rightarrow H_3 \bmod N = Node_2$$
$$\dots$$
$$Hash(key_n) \rightarrow H_n \bmod N = Node_{n-1}$$

Where,

`key`: Request ID or IP.

`H`: Hash function result.

`N`: Total number of nodes.

`Node`: The node where the request will be routed.

The problem with this is if we add or remove a node, it will cause N to change, meaning our mapping strategy will break as the same requests will now map to a different server. As a consequence, the majority of requests will need to be redistributed which is very inefficient.

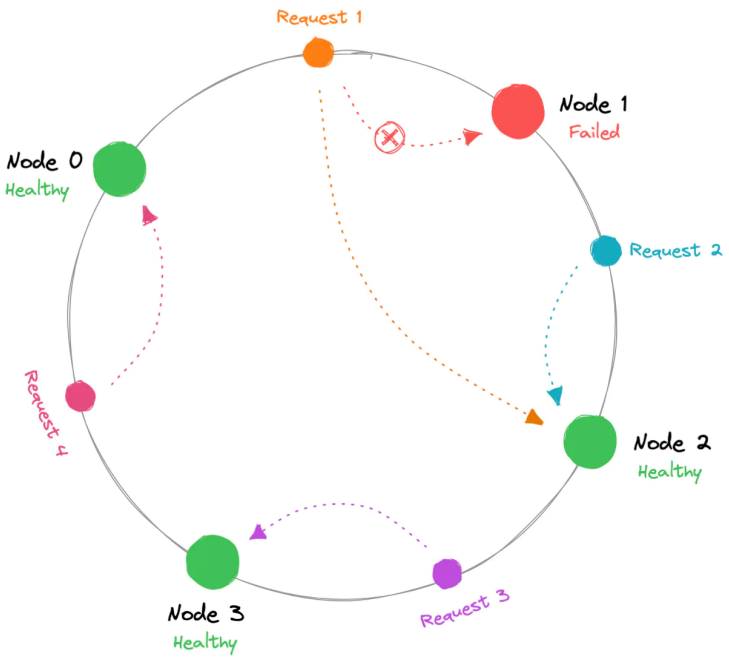
We want to uniformly distribute requests among different nodes such that we should be able to add or remove nodes with minimal effort. Hence, we need a distribution scheme that does not depend directly on the number of nodes (or servers), so that, when adding or removing nodes, the number of keys that need to be relocated is minimized.

Consistent hashing solves this horizontal scalability problem by ensuring that every time we scale up or down, we do not have to re-arrange all the keys or touch all the servers.

Now that we understand the problem, let's discuss consistent hashing in detail.

How does it work

Consistent Hashing is a distributed hashing scheme that operates independently of the number of nodes in a distributed hash table by assigning them a position on an abstract circle, or hash ring. This allows servers and objects to scale without affecting the overall system.



Using consistent hashing, only `K/N` data would require re-distributing.

$$R = K / N$$

Where,

`R`: Data that would require re-distribution.

`K`: Number of partition keys.

`N`: Number of nodes.

The output of the hash function is a range let's say `0...m-1` which we can represent on our hash ring. We hash the requests and distribute them on the ring depending on what the output was. Similarly, we also hash the node and distribute them on the same ring as well.

$$Hash(key_1) = P_1$$
$$Hash(key_2) = P_2$$
$$Hash(key_3) = P_3$$
$$\dots$$
$$Hash(key_n) = P_{m-1}$$

Where,

`key`: Request/Node ID or IP.

`P`: Position on the hash ring.

`m`: Total range of the hash ring.

Now, when the request comes in we can simply route it to the closest node in a clockwise (can be counterclockwise as well) manner. This means that if a new node is added or removed, we can use the nearest node and only a *fraction* of the requests need to be re-routed.

In theory, consistent hashing should distribute the load evenly however it doesn't happen in practice. Usually, the load distribution is uneven and one server may end up handling the majority of the request becoming a *hotspot*, essentially a bottleneck for the system. We can fix this by adding extra nodes but that can be expensive.

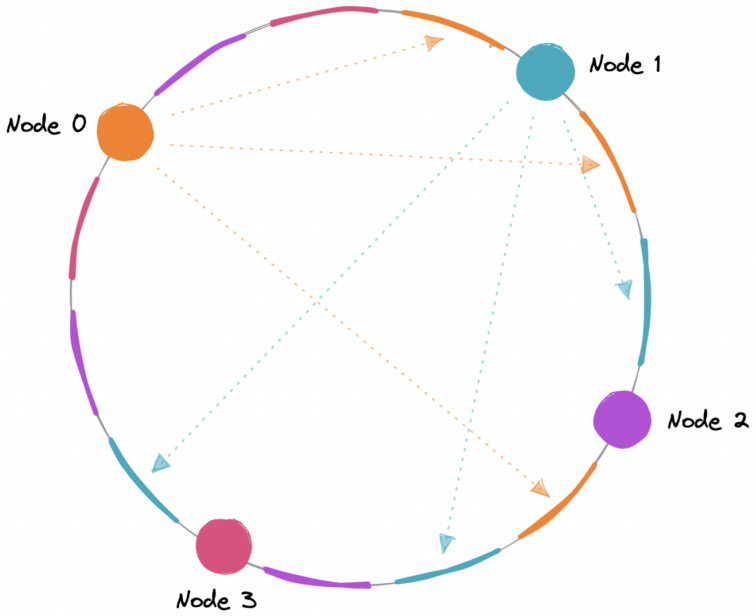
Let's see how we can address these issues.

Virtual Nodes

In order to ensure a more evenly distributed load, we can introduce the idea of a virtual node, sometimes also referred to as a VNode.

Instead of assigning a single position to a node, the hash range is divided into multiple smaller ranges, and each physical node is assigned several of these smaller ranges. Each of these subranges is considered a VNode. Hence, virtual nodes are

basically existing physical nodes mapped multiple times across the hash ring to minimize changes to a node's assigned range.



For this, we can use k number of hash functions.

$$Hash_1(key_1) = P_1$$
$$Hash_2(key_2) = P_2$$
$$Hash_3(key_3) = P_3$$
$$\dots$$
$$Hash_k(key_n) = P_{m-1}$$

Where,

`key`: Request/Node ID or IP.

`k`: Number of hash functions.

`P`: Position on the hash ring.

`m`: Total range of the hash ring.

As VNodes help spread the load more evenly across the physical nodes on the cluster by diving the hash ranges into smaller subranges, this speeds up the re-balancing process after adding or removing nodes. This also helps us reduce the probability of hotspots.

Data Replication

To ensure high availability and durability, consistent hashing replicates each data item on multiple `N` nodes in the system where the value `N` is equivalent to the *replication factor*.

The replication factor is the number of nodes that will receive the copy of the same data. In eventually consistent systems, this is done asynchronously.

Advantages

Let's look at some advantages of consistent hashing:

- Makes rapid scaling up and down more predictable.
- Facilitates partitioning and replication across nodes.
- Enables scalability and availability.
- Reduces hotspots.

Disadvantages

Below are some disadvantages of consistent hashing:

- Increases complexity.
- Cascading failures.
- Load distribution can still be uneven.
- Key management can be expensive when nodes transiently fail.

Examples

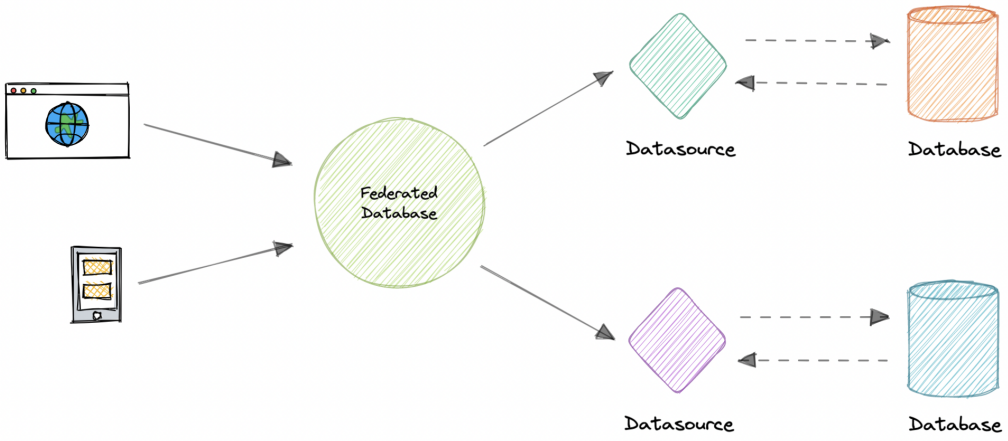
Let's look at some examples where consistent hashing is used:

- Data partitioning in Apache Cassandra.
- Load distribution across multiple storage hosts in Amazon DynamoDB.

Database Federation

Federation (or functional partitioning) splits up databases by function. The federation architecture makes several distinct physical databases appear as one logical database to end-users.

All of the components in a federation are tied together by one or more federal schemas that express the commonality of data throughout the federation. These federated schemas are used to specify the information that can be shared by the federation components and to provide a common basis for communication among them.



Federation also provides a cohesive, unified view of data derived from multiple sources. The data sources for federated systems can include databases and various other forms of structured and unstructured data.

Characteristics

Let's look at some key characteristics of a federated database:

- **Transparency**: Federated database masks user differences and implementations of underlying data sources. Therefore, the users do not need to be aware of where the data is stored.
- **Heterogeneity**: Data sources can differ in many ways. A federated database system can handle different hardware, network protocols, data models, etc.
- **Extensibility**: New sources may be needed to meet the changing needs of the business. A good federated database system needs to make it easy to add new sources.
- **Autonomy**: A Federated database does not change existing data sources, interfaces should remain the same.
- **Data integration**: A federated database can integrate data from different protocols, database management systems, etc.

Advantages

Here are some advantages of federated databases:

- Flexible data sharing.
- Autonomy among the database components.
- Access heterogeneous data in a unified way.
- No tight coupling of applications with legacy databases.

Disadvantages

Below are some disadvantages of federated databases:

- Adds more hardware and additional complexity.
- Joining data from two databases is complex.
- Dependence on autonomous data sources.
- Query performance and scalability.