Disclaimer

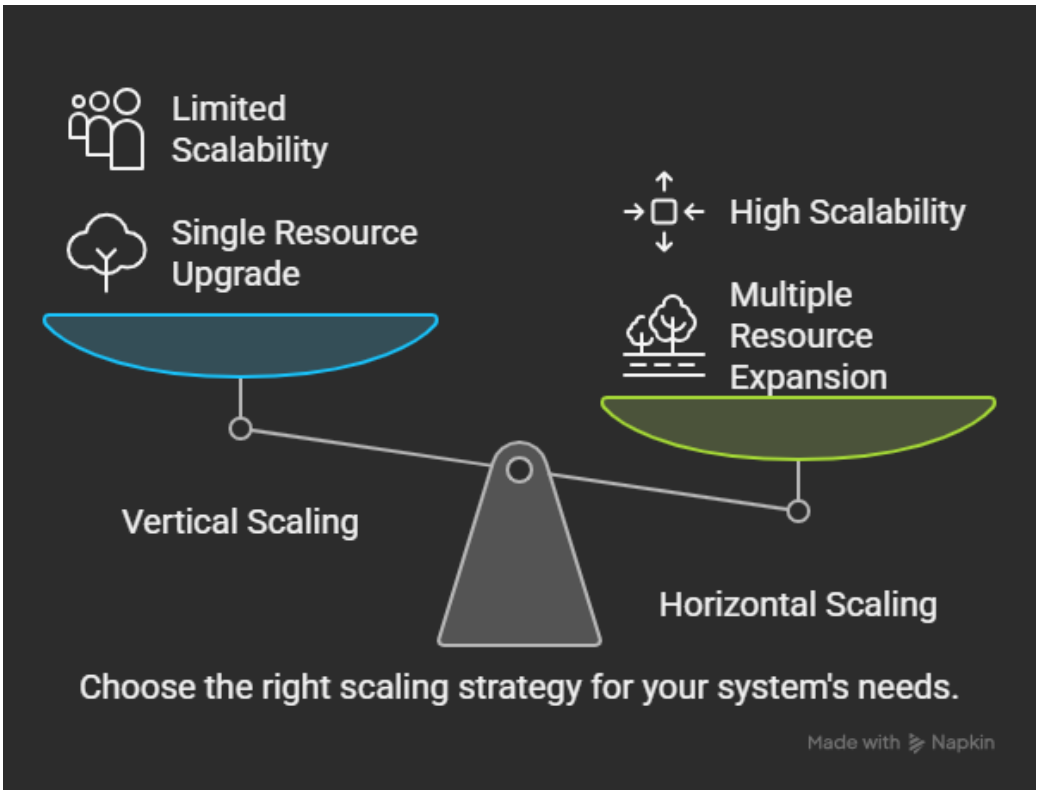
This document is intended for individuals or professional developers who are learning or revising system design concepts—particularly at the beginner level. These are my personal, handwritten notes (not AI-generated), so there may be occasional typos or mistakes. If you spot any, I'd truly appreciate it if you could let me know by emailing me at **ronakhingonia1@gmail.com**.

Happy learning! :)

Note: The content on *Page 10* was generated with assistance because the material was quite basic, but I included it to maintain the overall flow of the document.

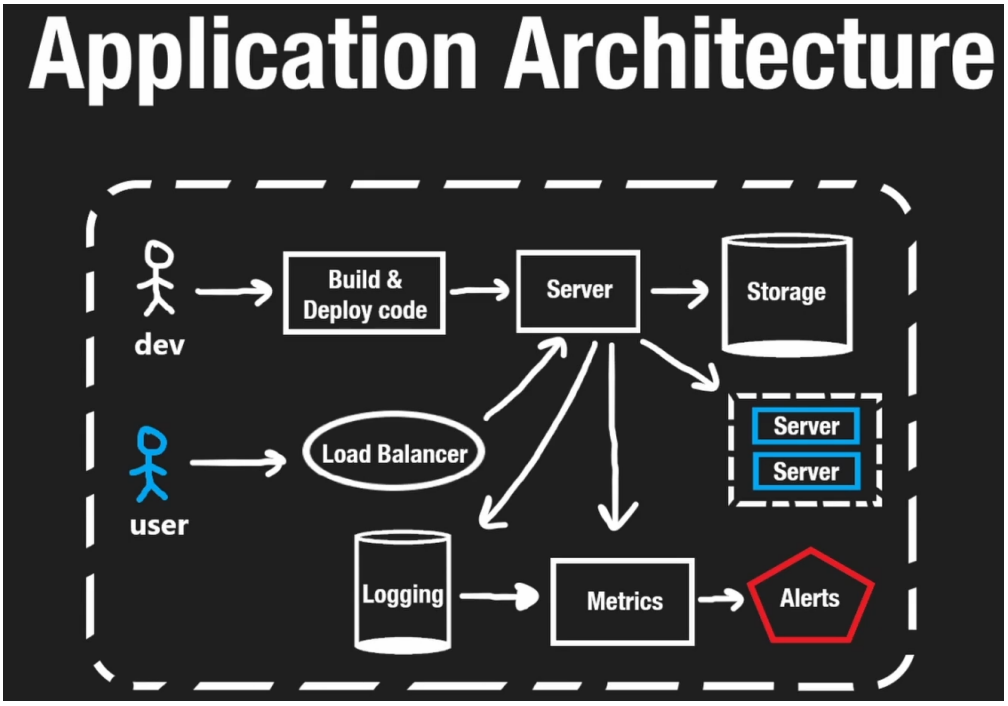
Application Architecture

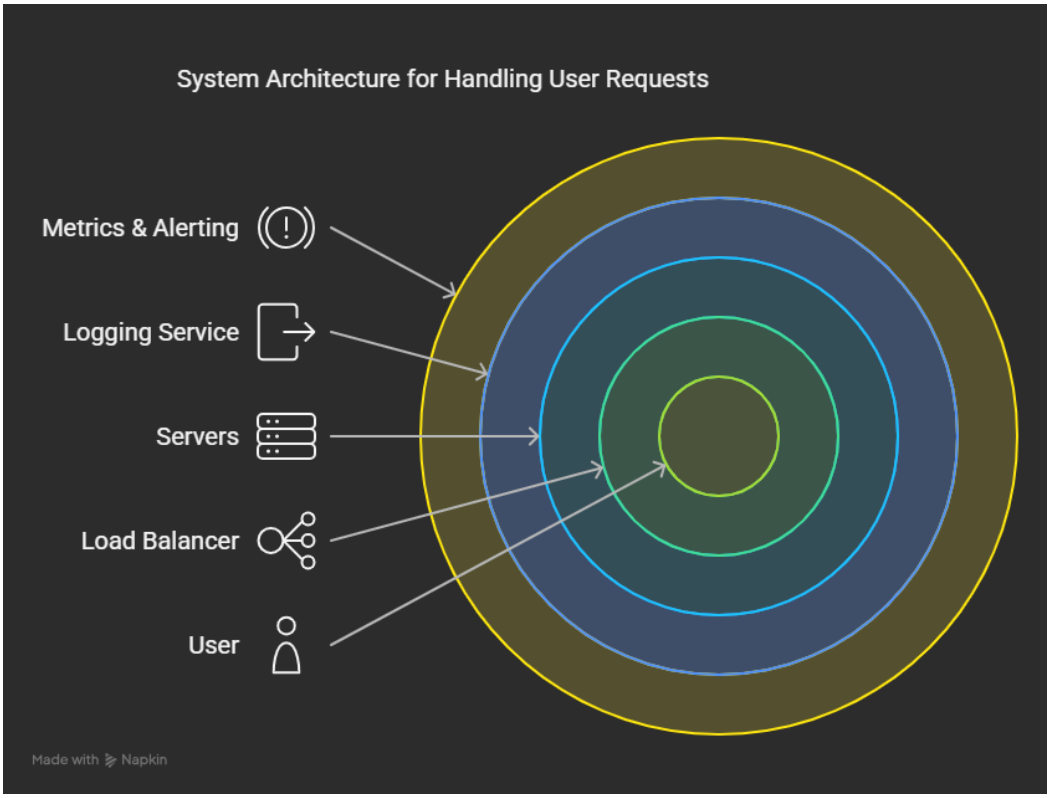
- What if we have a lot of users and they are all making request to our server at the same time. And our single computer (server) can't handle all of the requests on its own.
  - Maybe our CPU is not fast enough and we get a better CPU
  - Maybe it is a limitation with our RAM.
- To take a single resource like a server and making it better is called Vertical Scaling.
  - But computers have limitations. No matter how fast our CPU is, it has a limitation. It won't be able to handle infinite requests.
- So to make our system better, we can also use Horizontal Scaling. This is to actually take our server and make copies of it.
  - The benefit is that if we have more users, all of them don't have to go to a single server. They can talk to one of the other servers, this way we will be able to handle more requests at the same time.



- Now the problem we introduce when we have multiple servers handling requests, is when a user makes a request, how do we know which server that request should go to?
  - That's where the Load Balancer comes in.
  - It will forward the request to the server which has minimum amount of traffic so that each of the servers has a balanced amount of traffic.
- Now our servers don't have to run in isolation. Servers might actually be communicating with other servers as well.
  - For example, a E-commerce website can be communicating with other API like Stripe API to handle payments.
- Logging or print statements are used during local development to understand how code behaves. In production, servers are not running locally, so logging is essential to gain insights remotely.
  - Logs might not be stored directly on the server due to:
    - Storage limitations.
    - Lack of direct server interaction with logs.
    - Also, developers interact with the logs for debugging and monitoring.
  - An external Logging service is typically used to store and manage logs. Every time a user makes a request (successful or failed), the server logs the event to the external logging service.
- Now logs don't tell us the entire story. What if one of our servers isn't running due to hardware issue like storage full or faulty CPU.
  - To get that insight, we would have a Metrics service. This would provide us all of metrics that we care about on how are server is running.
  - Now some of metrics directly come from Logs since they are timestamped and response based.
- Now as developers, we have look at these Metrics to determine how the system is working. But if something goes wrong, we would have to manually go and look at the metrics and realize there is a mistake. Worse case - a user would tell us by emailing the issue since users will be first ones to know.

- As developers, we want to know immediately if something goes wrong. A push notification from the Metrics.
- To accomplish this, we have our metrics to feed that into a Alerting Service, which will notify us immediately as that metrics goes below the set threshold. And we will be notified probably at the same time as user.





Design Requirements

- Sometimes, system designing doesn't have a straight perfect solution. We have to weigh the trade offs and choose the better one.
- At the high level, no matter how complicated the system we design get, ultimately we are just:

1. Moving Data - In larger systems, data can move between larger machines that can be running in the same data center or can be located in different countries.

2. Store Data - Storing data can be very complicated. In context of DSA, we know that storing data in Array is different than in Binary-Tree. One of them is not necessarily better than the other, they are just different. Each of them is has own features and trade-offs.

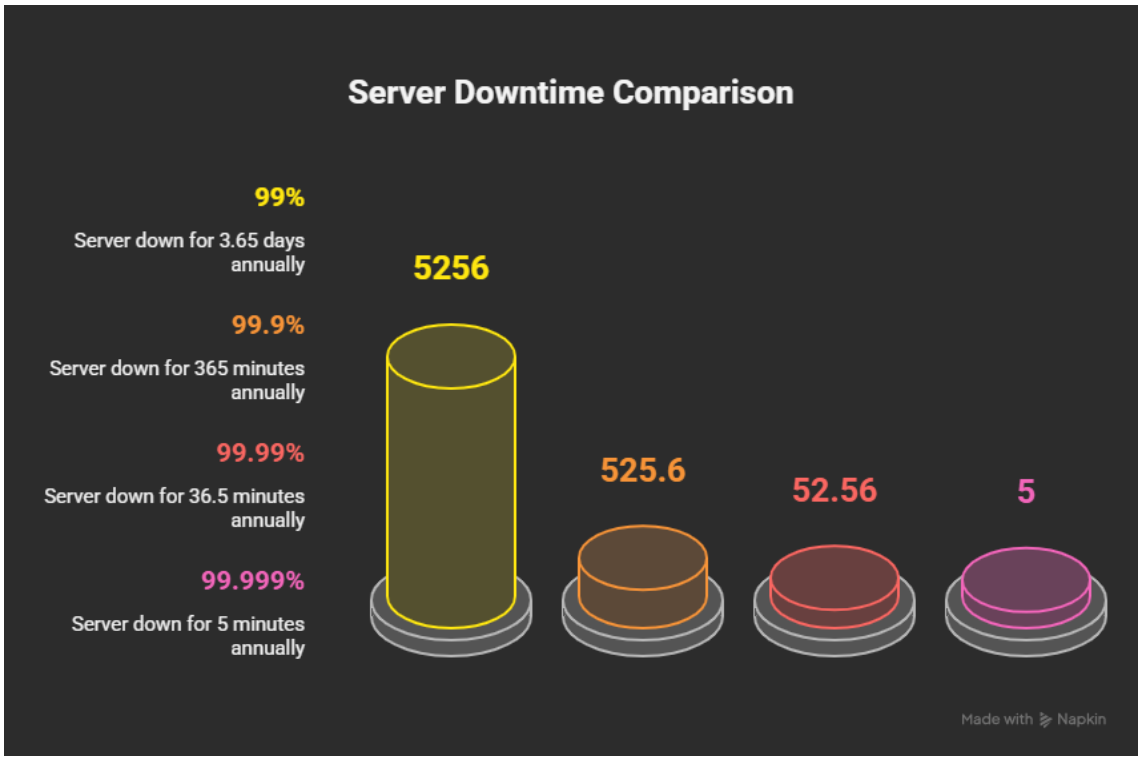
    - Storing data in larger systems actually follows some of the same ideas. But usually we are storing data in Data-bases, Blob stores, File system or Distributed file systems, etc. Like said before, they are just different with their own trade-offs.
    - If we were just storing a small amount of data, it really won't matter how we store it. But large data has to be stored in an efficient way. Which is again related to DSA where we learn how algorithms on different data-structures can be more efficient than other algorithms. Very important

3. Transform Data - Say we have server's response logs and we want to determine what % were successful and failed from them. So we are transforming the input data to get something we actually care about.

- When it comes to designing a large application, we want to be really careful how we design it. Because unlike a code script, bad design choices in our application architecture can be very difficult to correct later.
    - If you start with a wrong Database, you'll have to migrate all of that data from one DB to another DB, and at the same time re-write portions of your applications.

- But what exactly is a good design? We think in measure of trade-offs
- But how do we measure it? How do we know if something is good or bad?

1. Availability = uptime/(uptime+downtime) = uptime/total-time
    - For example if in a day, availability is 23hrs/24hrs i.e. 96%, that means if users were trying to make a request to our service, 96% of the times, they got a response.
    - Therefore, when we design a system we have our availability to be as high as possible, but it is very very difficult to achieve 100% availability. Theoretically, it is impossible since there can be a natural disaster near our data/server center.
    - Availability is measured in terms of '9'.



- Availability is often used to define an SLO (Service Level Objective) — a specific target for how reliable or available an application should be.
    - Example: If we're building a database, we might aim for 99.999% availability (also known as "five nines"), which allows for about 5 minutes of downtime per year.
- SLA (Service Level Agreement) is a broader term. It includes one or more SLOs and is a formal contract between a service provider and the customer.
    - Example: AWS might set an SLA for their database service that includes the SLO of 99.999% availability. If they fail to meet that level, they offer a partial refund as part of the agreement.

2. Reliability, Fault Tolerance and Redundancy:

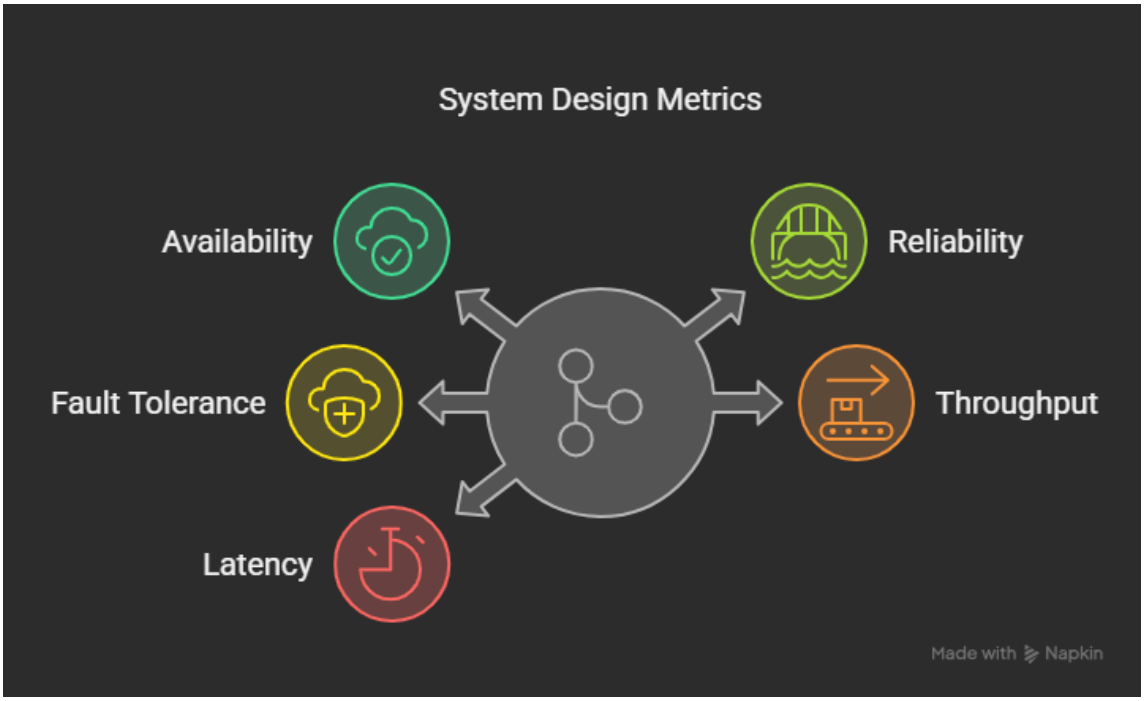
- If a user makes a request to our server and it responds, that means our server is available. But that doesn't necessarily mean our server is reliable.
  - Because if we just have a single server, its possible that server could crash.
  - Reliability is the probability that our system won't fail.
  - By adding another server into our system, we can increase the reliability. But we are also increasing availability of our system.
  - Because, For example, if we just had a single server, we could possibly have a lot of people making request to a single server. We could even have malicious users making request to our system to take our server down. This happens a lot using DoS and DDoS attacks. This would be a problem.
  - This could also be because the limited capacities of resources. We can increase availability by vertically scaling our servers, but this won't increase reliability.
  - Overall, this single server can go down for a thousand reasons. For a case like this, we would be happy if we had another server. This is why horizontal scaling has benefits compared to vertical.
- This is also called Fault Tolerance. If one portion of our system has a fault, then the system continues to operate successfully. That means our system is tolerant to failures of portions of the system (one server).
- We would also call this Redundancy since we a copy in our system that is not necessary. But this helps in an event of a failure.
  - It is best to have the copy in a different center say different part of the world. So our system is still reliable even in case of a natural disaster.

3. Through Put

- The amount of operations or data we can handle over some period of time.
- In context of commutating with a server, a user is making request to server and getting a response back. Through put here would be = requests per second
  - How many requests our server can handle per second. Say it is 100 req/sec
- So we can scale this similarly by vertical or horizontal scaling. But here horizontal scaling is complicated. For example horizontally scaling a DB, having data that is distributed and handling it gets very complicated. For DB = queries/second

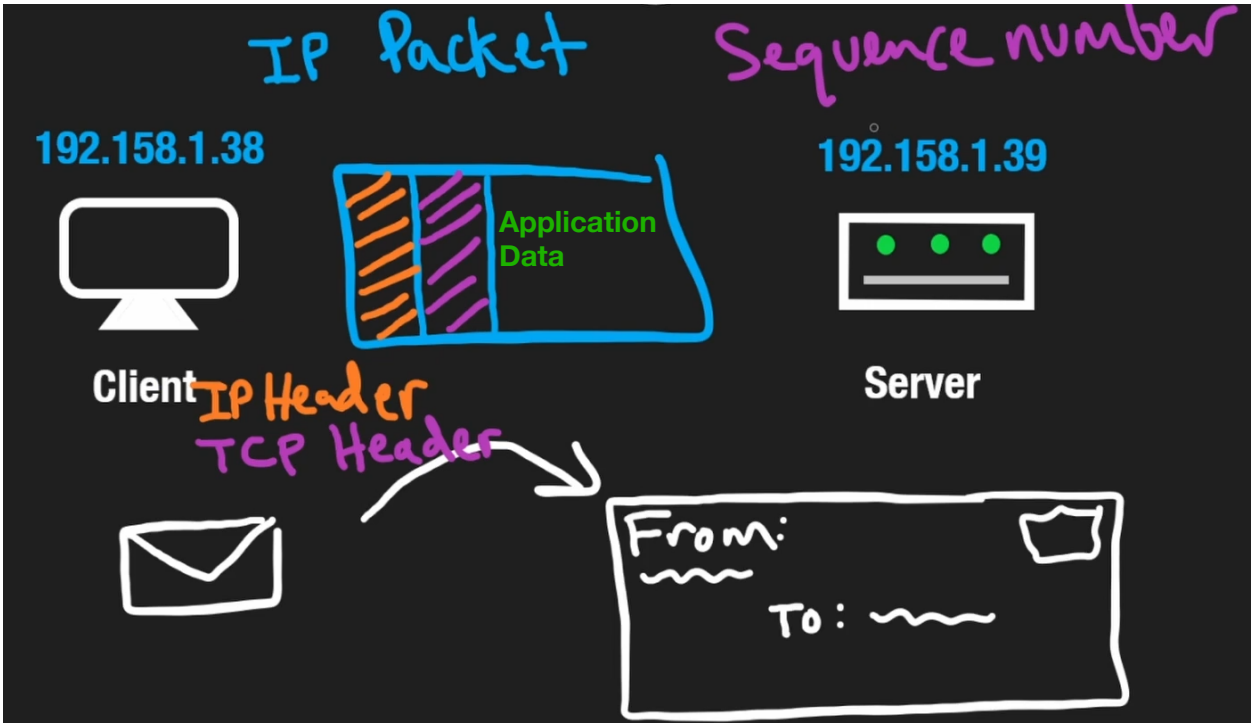
4. Latency

- This is commonly confused with Through Put.
- Latency is just some period of time it takes to an operation to complete.
- In context of user commutating with a server, Latency is amount of time is takes take operation (request to respond) to respond.
  - How long each individual request takes.
- Latency can be influenced by the network. For example, a user located far from the server might experience a 3-second latency, while a user closer to the server could have just 1 second of latency.
- However, latency isn't limited to network factors alone. At the hardware level, the CPU accesses data much faster from cache than from RAM, and faster from RAM than from disk storage.
- To reduce system latency, we can deploy servers across different parts of the world. This approach not only lowers latency but also improves availability, reliability, throughput, and fault tolerance.
- Another technique for reducing latency is using a Content Delivery Network (CDN), which will be discussed in detail in later chapters.



Networking Basics

- The rules of sending the data over the internet is called Internet Protocol (IP).
- The data we send is called a packet. A portion of each packet is reserved for the IP header, which contains metadata such as the sender's and receiver's IP addresses.
- When sending large amounts of data, multiple packets are transmitted. However, IP alone does not guarantee the correct sequence of these packets.
  - To solve this, TCP (Transmission Control Protocol) is added. TCP assigns a sequence number to each packet, allowing the receiver to correctly reorder them.
  - A TCP header contains metadata specific to the TCP connection, while the IP packet contains the entire IP header and TCP segment (TCP header + data).
  - When the server receives all the packets, it uses the sequence numbers to reassemble the original data in the correct order.
- As software engineers, we primarily focus on the actual data being transmitted, while networking protocols generally handle the details automatically.
- In a typical request, multiple protocols work together: IP at the network layer, TCP at the transport layer, and HTTP at the application layer.



TCP, UDP and HTTP(s)

- TCP is reliable, which is crucial because networks are not inherently reliable. When transmitting large amounts of data, some packets may fail to be sent or received. If a packet is lost during transmission, IP alone cannot handle the situation.
  - However, TCP solves this problem. If a packet is missing, TCP ensures that only the missing packets are resent to the receiver. This guarantees that the entire data will be correctly delivered, even if some packets are lost in transit. This is called Retransmission of lost packets.
  - This is possible because TCP establishes a two-way connection between both parties, allowing data to be sent in both directions. This connection is established through a three-way handshake.
    - However, establishing and maintaining a TCP connection can be expensive for the network. It requires sending additional data that isn't directly part of the actual communication, and the connection setup introduces latency. As a result, TCP connections take more time to establish compared to simpler protocols.
    - TCP is widely used because reliability is crucial for web communication. Most application layer protocols, like HTTP, SMTP, and WebSocket (WS), are built on top of TCP.
- UDP (User Datagram Protocol) is a connectionless protocol that does not guarantee delivery, order, or error checking, making it faster but less reliable than TCP. It's often used for applications where speed is more important than reliability, such as video streaming, online gaming, and voice calls.
  - Lost packets are lost forever and are not re-sent
- HTTPS uses SSL (Secure Sockets Layer) or TLS (Transport Layer Security) to secure communications over a network.
  - SSL came before TLS but is outdated; TLS is now the standard, though people still commonly say "SSL"
  - In the network stack, IP → TCP → HTTP/HTTPS. Plain HTTP is unencrypted and vulnerable to Man-in-the-Middle (MITM) attacks.
  - SSL/TLS encrypts HTTP traffic, securing it against interception and tampering.
  - HTTPS is the combination of HTTP with SSL/TLS encryption. It provides a reliable and secure way for application developers to protect data in transit.

Websockets, Polling and WebRTC

- WebSockets provide a persistent, full-duplex communication channel between client and server over a single TCP connection.
  - Unlike HTTP, which follows a request-response model, WebSockets allow real-time, two-way communication without repeatedly reopening connections.
  - In Twitch live chat, WebSockets enable instant message delivery: when a user sends a message, it's immediately broadcast to all connected viewers without delay.
  - Initial connection starts with an HTTP handshake, then upgrades to the WebSocket protocol with status 101.
  - After the upgrade, the connection stays open, minimizing latency and overhead.
  - WebSocket messages are lightweight compared to HTTP requests, making them ideal for high-frequency, real-time updates like chat, gaming, or stock tickers.
  - Security: WebSocket connections can be encrypted with WSS (WebSocket Secure), similar to HTTPS for HTTP.
- Polling is a technique where the client repeatedly sends requests to the server at regular intervals to check for new data.
  - Each request asks, "Is there any new data?" and the server responds, even if there's nothing new.
  - Polling creates a lot of unnecessary network traffic and server load, especially if updates are rare.
  - Latency is higher because the client only knows about new data after the next poll.
  - Example: In an old live chat app (before WebSockets), the client would send a request every 5 seconds to ask, "Any new messages?"
    - Even if no one had chatted, the server would still reply, wasting resources.
    - Instagram uses polling for within Stories and in comment streams on posts and reels.
- WebRTC (Web Real-Time Communication) is a technology that enables direct peer-to-peer communication between browsers or devices without needing an intermediate server for transmitting audio, video, or data; it is commonly used in video calls (like Google Meet), voice calls, and live data sharing with very low latency.

API Paradigms

1. REST (REpresentation State Transfer)

- It's not a protocol like HTTP itself, but rather a set of guidelines or standardization for designing web APIs using HTTP. It defines how clients and servers should communicate.
- **They are stateless:** each request from client to server must contain all the information needed to understand and process it. Or simply - meaning the server doesn't remember anything about the client's previous requests.
- Why "Representation State Transfer"?
  - "Representation" refers to how resources (e.g. videos, users, orders) are formatted when sent over HTTP (often JSON or XML).
  - "State Transfer" means each request carries the state needed for that operation, nothing extra is stored by the server.
- **Example: Listing Videos (GET request)**
  - Say we have a GET endpoint - https://yt.com/videos
  - This endpoint returns a list of videos (e.g., 10 by default). Now, suppose a user wants to see 10 more. There are two ways to design this:
    - Stateful Design (Not RESTful)
      - Server holds a session for each user (e.g. "this user has already seen items 1–10").
      - When the user asks for more videos, the server checks the session and sends the next 10.
      - Problem: This creates dependency on the server's memory, and it makes scaling difficult.
    - Stateless Design (RESTful)
      - The server doesn't remember the user or any previous request.
      - Client tells the server exactly which slice of data it wants on each request.
      - **For this, we use pagination.**

- **Pagination in REST APIs**
  - Instead of tracking the user's place on the server, we let the client tell the server what it needs.
  - First request:
    - `GET https://yt.com/videos?limit=10&offset=0`
  - Next request:
    - `GET https://yt.com/videos?limit=10&offset=10`
  - This way, the server just reads the parameters and returns the correct videos without needing to remember anything.
- Benefits of Stateless Design
  - Since the server doesn't manage user sessions, any request can go to any server.
  - This allows for horizontal scaling, adding more servers to handle more traffic.
  - All servers can access a central database, where persistent data (like video records) is stored.
- Important Notes
  - Stateless doesn't mean no data is stored. It just means the server doesn't store session-related data.
  - State can be stored on the client (like in cookies or local storage).
  - The client sends any needed information to the server with each request.

2. GraphQL

- This is not a protocol either and is built on top of HTTP, but it only uses HTTP's POST method for communication.
- Why POST?
  - GraphQL uses POST because it needs to send a query in the request body, and the GET method doesn't support a request body. This query tells the server exactly what data the client wants.
- In large applications using REST APIs, we often run into two main problems:

1. Overfetching

- Suppose we're showing a list of comments, and for each comment, we want:
  - *the comment text, user's name, user's profile picture*
- In REST, if you request user info for a comment, the user endpoint might return everything — name, profile picture, registration date, email, etc.
- With REST, solving this would mean adding custom logic on the server, and you'd have to do this every time your data needs change.
- GraphQL Solution:
  - With GraphQL, the client can ask for only the fields it needs — like just the name, profilePicture, and commentText.
  - The server returns exactly that, and nothing more.
  - This avoids overfetching, improves speed, and keeps things efficient.

2. Underfetching

- In REST, to display a video and its comments (with each comment showing the user's name and profile pic), you'd need to:
  - Fetch the video, Fetch its comments, Fetch user info for each comment
- That means multiple round trips to the server.
- GraphQL Solution:
  - GraphQL allows you to combine all these data requests into one.
  - With a single query, you can ask for the video, its comments, and the specific fields of each user.
  - It's all done through a single endpoint and in a single request, which makes things much simpler and faster.

- **Operations in GraphQL** - GraphQL has two main operations:
  - Query – to read data without changing anything on the server.
  - Mutation – to create, update, or delete data.
- **Caching Limitation:** Since GraphQL uses POST requests, which are not idempotent, caching is more difficult compared to REST APIs (which often use GET requests, which are easier to cache).

3. gRPC (Google Remote Procedure Call)

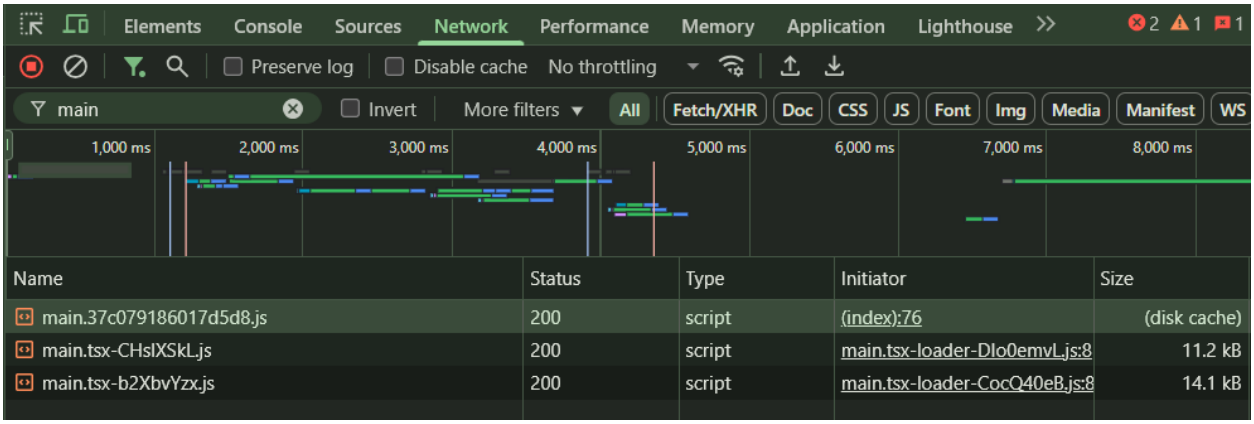
- gRPC is an open-source framework created by Google in 2016, built on HTTP/2.
- Why HTTP 2?
  - Because it needs some feature of http2 like Multiplexed streams, Header compression, Bi-directional streaming, etc.
  - Note : http2 makes websockets kind of obsolete. Since it has those features prebuild.
- Unfortunately, gRPC can't be used directly from a browser, because it requires low-level control over HTTP/2 that browsers don't allow.
  - To use gRPC in a browser, you need a special tool called a gRPC-Web proxy.
- **It's mainly used for server-to-server communication in systems** with multiple services (like micro services). Performance wise, it is objectively faster and more efficient then Rest APIs.
  - The main reason is, instead of sending raw JSON (JSON is just a string when send with Rest APIs), is actually sends information in Protocol Buffers.
  - These are schema-based objects and get converted into compact binary format before being sent. JSON is not a schema. But with protocol buffers you can define schema, exactly the type of data that a gRPC endpoint will return.
  - Since its a serialized into binary format. So response from server, we are sending a smaller amount of data when we would be sending using REST APIs.
- **Downsides of gRPC**:
  - Less standardization: It's newer than REST and doesn't have as many tools or libraries.
  - While they are much faster, When it comes to development, they have schemas which can be annoying. And with gRPC we need good tooling.
  - Harder to use in browsers, as explained earlier.
  - Also even though gRPC is build on top of HTTP which has status codes with an response. With gRPC we don't have that, rather we have error messages. gRPC doesn't really make use of status codes of http, you actually have our own custom error handling based on those error message you have defined on server side.
- gRPC doesn't use HTTP methods like GET, POST, etc. Instead it has verbs/actions

API Design

- **API design is crucial**, especially for **public-facing APIs**, because once an API is made public, its functionality shouldn't change in a way that breaks existing users.
- If we are, API should be **backwards compatible**.
  - Ex- To create a tweet, we have endpoint: POST https://api.twitter.com/v1.0/tweet
  - This might be used like this: - createTweet(userId, content)
  - Here, userId and content are required parameters.
  - Now, suppose we want to introduce a new parameter, parentId, which tells us if the tweet is a **reply or a retweet** of an existing tweet.
  - If parentId **is optional**, it's safe to add it because:
    - Existing users won't have to change anything.
    - Requests using only userId and content will still work. Backward compatible!
  - But if we make parentId **required**, then:
    - All current users must update their code.
    - Older clients will break. Not backward compatible!
- But sometimes, we need to redesign the API or introduce new required parameters. In such cases, we can't maintain backward compatibility. **This is where versioning comes in.**
  - Most public facing APIs with have version (v1.0) with them.
  - If we need major changes, we create a new version. Say v2.0
  - So now when we introduce new parameters, we will not break the older API version. Multiple versions of the API can coexist, serving both old and new users.
  - Older versions continue to work as they are.
  - New functionality is added in the new version.
- Also, in the case of GET requests, let's say we have an endpoint to fetch tweets of a user. We don't want to send all tweets at once, especially if the user has thousands of them. That would be inefficient and slow.
  - To solve this, we use pagination. The API return smaller, manageable responses, and clients can fetch more data through pagination tokens or next-page links.

Caching

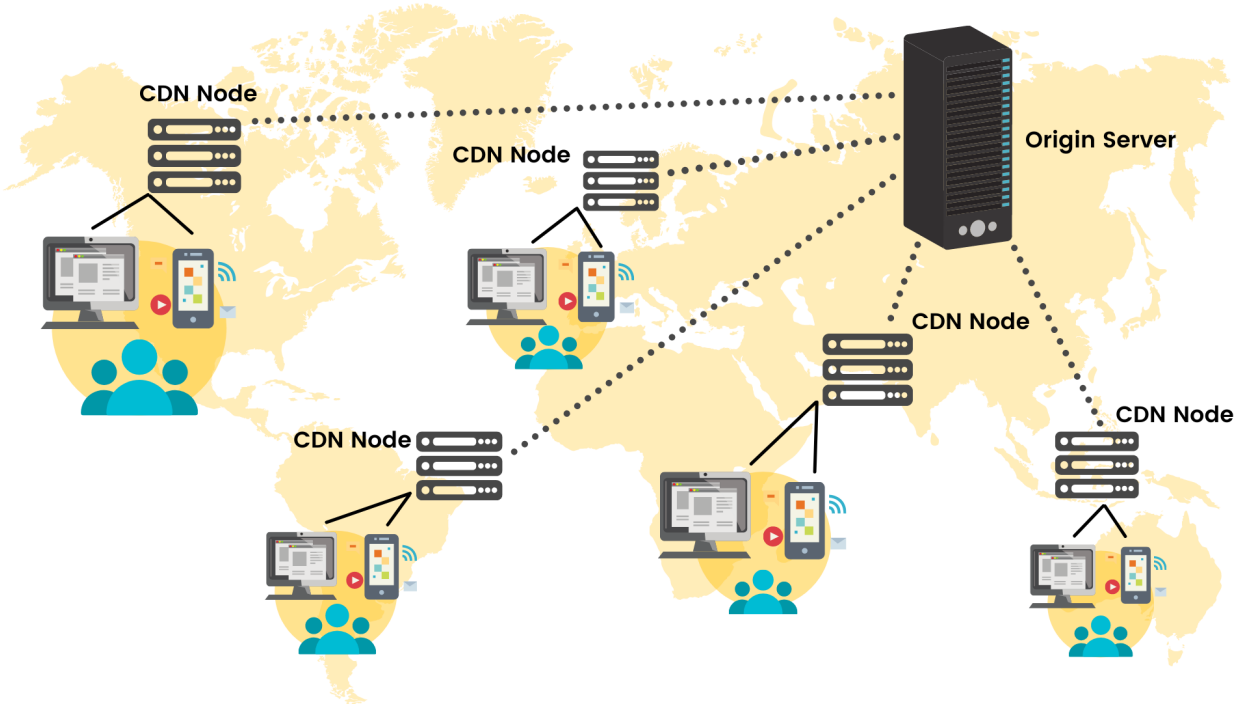
- Imagine you open a web page and look in your browser's Network tab. You notice that a JavaScript file (for example, main.xyz.js) is downloaded every time you visit—even though it never changes.
  - Since the file's contents stay the same, downloading it on every page load wastes time and bandwidth.
  - Instead, the browser can store (cache) that file locally. On your next visit, the browser serves it from its cache without asking the server. This reduces load on the server, cuts down your network usage, and makes the page load faster.



- Caching helps us speed things up significantly. It's a simple yet powerful technique that improves performance by avoiding repeated work.
- So far, we saw a basic client-side caching example using the browser. **But caching is also very useful on the server side**, where things can get more complex.
  - Imagine a platform like Twitter where users create a huge number of tweets.
  - When a tweet is posted, it's saved in a **database on disk** (slow, but permanent).
  - Now, suppose a user wants to read a tweet by its ID. Going to the database each time is slow, especially with millions of tweets.
  - That's where server-side caching comes in.
  - We can use a fast, in-memory key-value store like Redis for caching. Here's how it works:
    - User requests a tweet by ID.
    - Cache check: The application first looks in Redis (the "cache") in Memory/RAM.

        - Cache hit: If the tweet is there, it's returned immediately—super fast.
        - Cache miss: If it's not there, the application reads from the database on disk (slower) and then stores that tweet in Redis, so future reads are fast.
    - Why it helps?
        - Most tweets are rarely read. By caching only the popular ones (or tweets from heavy-traffic users), you serve those reads from memory.
        - This reduces the number of slow disk reads, improves overall throughput, and takes pressure off your database.
    - Now this ins't the complete picture because there are several different algorithms we can use in caching to accomplishing different things.
- The above simple strategy we talked about is called **Write-Around cache**.
    - We store the data in disk when the tweet is made and skip the cache entirely.
    - Only when someone actually reads that tweet does it get added to the cache.
    - This way, you don't fill your memory store with data that nobody ever reads.
- There are many caching strategies each with its own trade-offs.
    - Write-through caching where we immediately write it to memory and after that write it to primary storage.
        - Pros: Data in cache and database stay in sync.
        - Cons: Slower writes (two writes per operation).
    - Write-back caching: Writes go only to the cache (RAM/memory) at first. The cache later `flushes` (writes) data to the DB, either on eviction or at regular intervals.
        - Pros: Fast writes, fewer database operations. Faster response to request.
        - Cons: Risk of data loss if the cache/server crashes before flushing. Used in cases where loss of small amount of data is not a big deal.
    - **Eviction Policy:** Since memory is limited, we can't store everything in cache. When it's full and we want to add new data (like a tweet fetched from the database), we need to decide which existing data to remove:
        - FIFO: first in first out
        - LRU: least recently used (ideal for twitter case since LFU for old famous post can be higher but not much of use now)
        - LFU: least frequently used

Content Delivery Network (CDN)

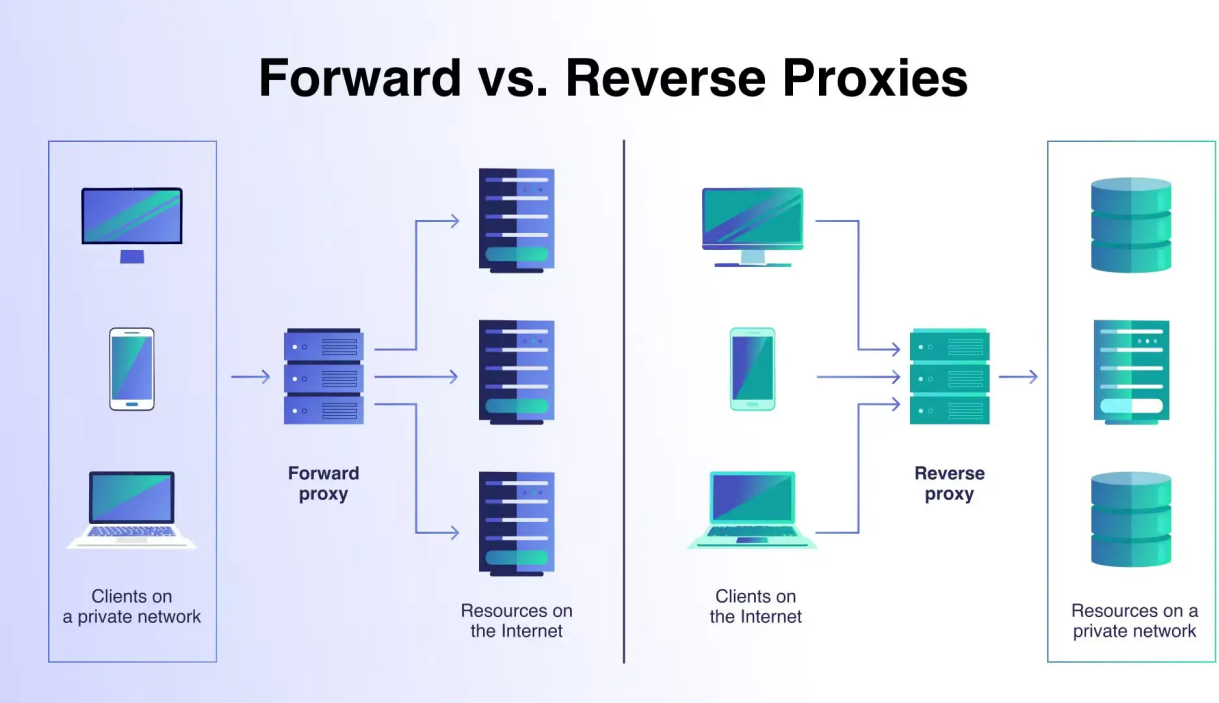


- A **CDN (Content Delivery Network)** is a system of servers located around the world that helps deliver content to users faster. It works by caching data closer to the user, so the user's request doesn't have to travel all the way to the original (central) server.
- Although diagrams often show just a few servers, in reality **CDNs have many servers globally**. This means most users are physically close to at least one CDN server, which speeds up loading times.
- The downside though is that we cannot put everything on CDN servers. CDN servers are pretty dumb. We can only put static content that is not changing. These servers don't run application code or handle dynamic requests.
  - Examples of static content that can be stored on a CDN:
    - JavaScript files (like the one talked above in caching section)
    - Images
    - Videos
    - CSS files
    - Fonts
- Having CDNs lowers the load on origin server and the distribution reduces the latency and increases availability and reliability.

- Now there are couple different types of CDN like Push, Pull, etc.
- Imagine you're building Twitter. Your origin server handles dynamic actions—creating tweets, deleting tweets, and so on. But profile pictures are just static images, so they're perfect for CDN hosting. Once uploaded, a profile picture only changes when the user swaps it out.
  - In a **Push CDN**, when a user uploads something like a profile picture:
    - It first gets stored on the origin server.
    - Then, we immediately send (push) that image to all CDN servers.
    - So when someone requests that profile picture, they'll get it quickly from the nearest CDN server.
    - Best for: Content that needs to be available everywhere immediately and doesn't change often.
  - In a **Pull CDN** setup:
    - The uploaded profile picture is saved only on the origin server.
    - When a user tries to view that profile:
      - Their request first goes to the closest CDN server.
      - The CDN checks if it already has the image (cache hit).
        - If yes, it sends the image right away.
        - If no, it's a cache miss.
          - The CDN then acts as a proxy—it fetches the image from the origin server on behalf of the client.
          - Once received, it caches the image and returns it to the user.
    - Next time someone nearby requests the same image, it will be served directly by the CDN (cache hit).
    - If no one from another region of the world ever requests that image, it never gets sent to all other CDN servers, saving bandwidth and storage.
  - **For a platform like Twitter** where not all content needs to be available globally at all times, **Pull CDN is better**. It's more efficient because it only stores content on CDN servers when needed.

Proxies & Load Balancers

- There are two main types of proxies: **Forward** & **Reverse**



  - Note - When someone says proxy that generally means forward proxy but not always and that can be a bit confusing.

- [Forward Proxy]: Imagine you're a user trying to visit a website. Normally, your request goes directly to the website's server. But sometimes, there's a **middle server**, called a **proxy**—that sits between you and the website.
  - This **proxy server** takes your request, sends it to the actual website, gets the response, and passes it back to you.
  - It **hides your real identity**, like your IP address, from the website you're visiting. So the website only sees the proxy, not you.
  - This is helpful when:
    - Your **ISP or country blocks** access to a certain website, but the proxy is allowed. You connect to the proxy, and the proxy gets the content for you.
    - A **school or office network** wants to control what people can access. All internet traffic goes through a proxy, and it can block certain websites.
    - You use a VPN, which is basically a type of proxy that also encrypts your connection and hides your location.

- **Reverse Proxy**: A **reverse proxy** is the opposite of a forward proxy. Instead of hiding the **client**, it hides the **destination server**.
  - As users, when we make a request, it first goes to a reverse proxy server. But this server isn't the one that handles our request—it simply forwards the request to the actual (backend) server that will do the real work. This request includes all the usual information like IP address, headers, etc.
  - After getting the response, it sends it back to you. **You never see or connect directly to the internal servers.**
    - So the client only knows about the reverse proxy, not the real server behind it.
    - On the other hand, the **actual server usually knows about the client,** because the reverse proxy forwards all that information.
  - This setup is useful for several reasons like:
    - It hides and protects the real servers from direct exposure to the public.
    - It can cache responses (store them temporarily), so repeated requests can be served faster without bothering the actual servers.
    - It can filter or block bad traffic before it reaches the internal servers.
    - It helps with SSL termination (handling HTTPS encryption/decryption so backend servers don't have to).
  - In short:
    - Forward proxy hides the client from the server.
    - Reverse proxy hides the server from the client.
  - **Examples of Reverse Proxies**
    - **CDNs (Content Delivery Networks)**
      - When we visit a website, we're often connecting to a CDN. The CDN might fetch the content from the origin server we never see. So, the CDN acts as a reverse proxy.
    - **Load Balancers**
      - Imagine a big service with multiple backend servers like amazon. When users make a request, it first hits the load balancer (a type of reverse proxy). The load balancer's job is to distribute traffic across the servers, so no single server gets overwhelmed.

Load Balancers: To evenly share the load, there are several strategies or algorithms.

1. **Round Robin**
   - Requests are sent to each server one by one, in a cycle.
   - Example: First request goes to Server A, second to B, third to C, and so on—then back to A.
   - A problem could be that one the server is less powerful then others. In that case we would have a variation i.e. Weighted Round Robin.
2. **Wighted Round Robin**
   - Used when some servers are more powerful than others.
   - Example: If Server A is twice as powerful as B and C, we can send more requests to A.
   - So, the traffic might split like 50% to A, 25% to B, and 25% to C.
3. **Least Connections**
   - Instead of cycling through servers, this method checks which server has the fewest active connections and sends the request there.
   - Useful when some requests take longer than others, so traffic is more balanced based on actual server load.
4. **Geo-based Load Balancing**
   - If servers are in different parts of the world, users can be connected to the server closest to them, which reduces delay and speeds up response time.
5. **Hash-based Load Balancing**
   - A field from the request (like the user's IP address or some content) is passed through a hash function to decide which server to send it to.
   - This can help in keeping the same user connected to the same server for a smoother experience (also known as session stickiness).

- **Types of Load Balancers: Layer 4 vs Layer 7**
  - There are two common types of load balancers based on the OSI model layers they operate on:

A. Layer 4 Load Balancer (Transport Layer)

- Works at a lower level of the network, the TCP (or UDP) layer.
- It decides how to distribute traffic based on things like IP address and ports.
- It does not inspect or understand the actual content of the request—just looks at the headers (like where the request is from and where it's going).
- Because it doesn't look at the actual content of the request, it's very fast and efficient.
- It can use simple methods like round-robin or location-based routing to forward requests.
- Cons: However, it's **not smart**—it can't see what kind of request is being made or what specific resource is being asked for.
- Pros:
  - Very fast and high-performance.
  - Good for basic traffic distribution.

B. Layer 7 Load Balancer (Application Layer)

- Works at the HTTP/HTTPS level, meaning it can see and understand the content of the request—like paths, headers, cookies, and payloads.
- This makes it much more **flexible** and **intelligent**.
- Example: Let's say you have 3 servers
  - Server 1 handles tweets,
  - Server 2 handles user profiles,
  - Server 3 handles authentication.
- With Layer 7, the load balancer can inspect the request, see that it's about tweets, and send it to Server 1. If the request is about login, it sends it to Server 3. This is called **content-based routing**.
- But Layer 7 does more work:
  - It has to decrypt HTTPS, read the content, and create a new request to the backend server.
  - So it sets up two connections: one between client and load balancer, and one between load balancer and server.

- Pros:
  - Highly flexible—great for smart routing based on application data.
  - Ideal when different servers serve different types of content.
- Cons:
  - Slower and more resource-heavy than Layer 4.

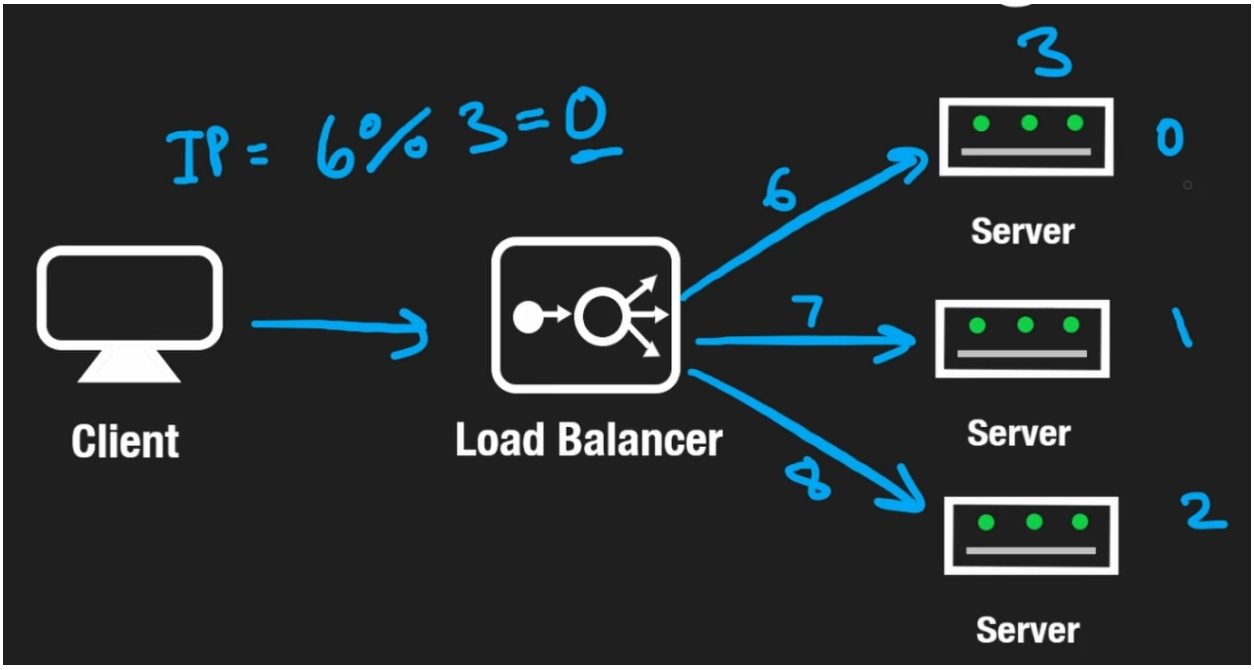
Feature	Layer 4 Load Balancer	Layer 7 Load Balancer
OSI Level	Transport (TCP/UDP)	Application (HTTP/HTTPS)
Content-aware?	No	Yes
Speed	Fast	Slower
Flexibility	Less flexible	More flexible
Use Case	Basic traffic routing	Smart routing (e.g., by URL)

  - Handles encryption, decryption, and more logic.

Now What Happens If the Load Balancer Fails?

- Even if you have multiple servers behind a load balancer, the load balancer itself can become a single point of failure.
- If it goes down, no traffic can reach your servers.
- Solutions:
  - Use replicas or multiple load balancers to avoid this issue.
  - You can set up:
    - Active-Active: multiple load balancers handling traffic at the same time.
    - Active-Passive: one main load balancer, and one backup ready to take over if the main one fails.
- In real-world setups:
  - Load balancers are usually very powerful and designed to handle huge amounts of traffic.

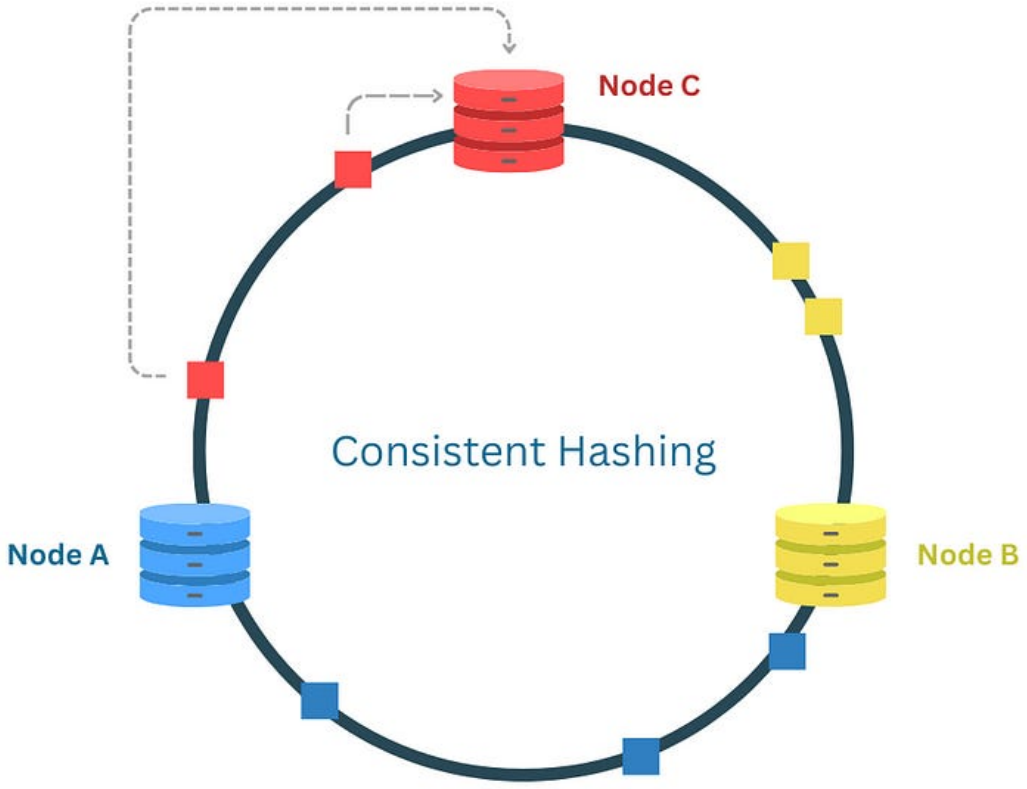
- They are often highly optimized because their main job is just forwarding requests, so they rarely get overloaded.



Consistent Hashing

- Suppose you have 3 servers (numbered 0, 1, 2) and you want to pick one based on a client's IP (treated here just as an integer).
- How it works:
  - Compute hash = IP % 3.
  - Send all requests from that IP to server hash.
- Example:
  - IP 6 ⇒ 6 % 3 = 0 ⇒ always goes to Server 0
  - IP 7 ⇒ 7 % 3 = 1 ⇒ always goes to Server 1
  - IP 8 ⇒ 8 % 3 = 2 ⇒ always goes to Server 2
- Why this can help:
  - If each server has its own small cache (e.g. Redis) and holds data only for the clients it sees, that cache stays "warm" for repeat requests from the same IP.
  - Unlike round-robin, the same IP always hits the same server, so cache hits stay high.

- If all your servers are truly stateless, it doesn't really matter which server handles a request, so consistent hashing offers no real benefit.
- But, What happens if one server crashes, is removed or is on maintenance?
- Say Server 2 goes down. Now we have only 2 servers left. That changes our hash:
  - Old: IP % 3
  - New: IP % 2
- So with this,
  - Earlier, 9 % 3 = 0 → server 0
  - Now, 9 % 2 = 1 → server 1
  - That means, IP 9 used to go to Server 0, Now it goes to Server 1. **All the cached data on Server 0 for user 9 is now wasted.**
- Even worse: **most users will be remapped** to different servers because the hashing logic changed.
- This is Where ***Consistent Hashing*** Helps.
  - Consistent hashing is designed so that when a server is added or removed, only a few users get remapped, not all of them.
  - It solves this remapping issue and makes server-specific caching more effective, even when your backend is dynamic.



- User9 will still go to server 0 with two servers left.

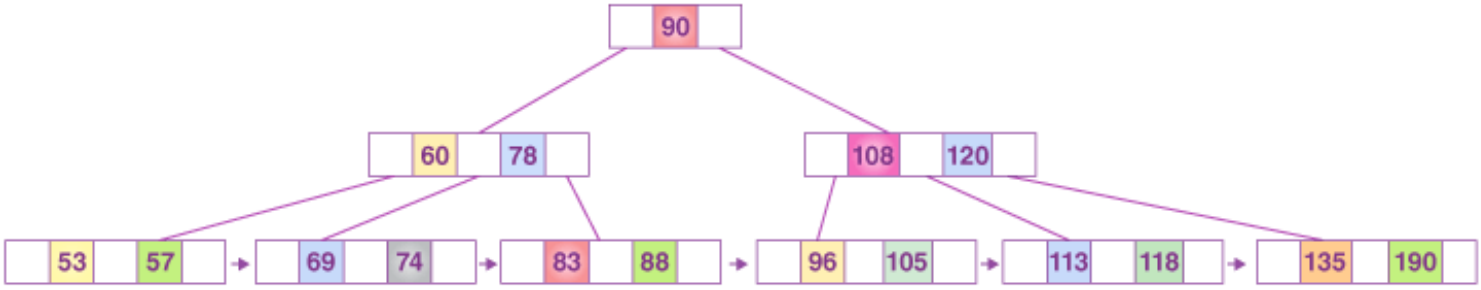
- **Consistent Hashing on a Circle (Ring)**: Think of the entire hash space as a circle (or ring).
  - We hash both servers and user requests into points on this circle.
  - When a request comes in, we find its position on the circle using a hash function (e.g., hash(IP) % 360, just as a simple example).
  - Then we go **clockwise on the circle until we hit a server**. That server handles the request.
  - Because the hash function (e.g. IP % 360 or a real-world SHA variant) never changes, the same IP always lands at the same spot on the circle—so it tends to hit the same server. Thus is Consistent.
- We tend place **servers at equal distances on the circle to balance traffic.**

- **What Happens When Servers Change?**
  - ***Removing a server***: Only the requests that mapped to that server's region get re-assigned (they "fall through" to the next server clockwise). Everybody else stays with their original server.
    - If a server is removed (e.g., server 3), its share of traffic moves to the next server in the circle (say, server 1).
    - This might cause some temporary imbalance (server 1 getting more traffic), but there are ways to fix or reduce this, like using virtual nodes (advanced topic).
  - ***Adding a server***: You insert it at a new point on the circle. Only the requests whose hashes now lie between that point and the next server clockwise switch over. All other mappings remain intact.
    - This is much better than reassigning all users like in the basic modulo approach.
  - This way, adding or removing one server touches only a small fraction of your traffic (about 1 / N of it), not everything.

- **When Is Consistent Hashing Useful?**
  - CDNs (Content Delivery Networks): to keep requests going to the same edge node
  - Databases: to divide and store data consistently across multiple DB nodes
    - In databases, each user's data might live on a specific node. When the user logs in, consistent hashing helps us route the request to exactly the right node.

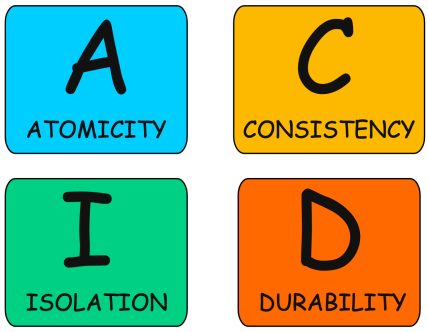
- It can get complex, but as developers you don't need to know every detail to use it effectively. Think of it as a tool in your toolkit.
- **Rendezvous (or Highest Random Weight):** Hashing is another method with the same goal: keep user-to-server mapping consistent.
  - It's often simpler and has its own use cases, but serves a similar purpose.
- *Summary of Key Concepts:*
  - Hash Key*:* Something unique like an IP address or user ID, used to identify a user/request
  - Hash Function*:* A good-quality function (e.g., SHA-1, MurmurHash) that maps keys to positions on the circle.
  - Nodes: The destinations — these could be servers, databases, or anything that handles user data.

- When we store a **large amount of data on disk**, we want to organize it in a way that makes **reading and writing** as fast as possible.
- A common data structure used in **Relational Databases (RDBMS)** for this purpose is the **B+ Tree**. It's similar to a Binary Tree, but instead of each node having just two children, a B+ Tree node can have **'m' children**. Also, instead of one key per node, each node holds **'m - 1' sorted keys**.
- Why do we use B+ Trees instead of regular binary trees?
  - In large systems, trees can grow very tall, and **searching** for data means going through several levels (from root to leaf).
  - By allowing **more children per node**, the tree becomes **shorter (less height)**, which means **fewer read/write operations** are needed to find data.
  - This improves performance, especially when the data is stored on slower media like disks.
- In a B+ Tree:
  - **Actual data is only stored in the leaf nodes**.
  - The non-leaf nodes only store keys that guide the search process, like a roadmap.
  - Leaf nodes are **linked together**, like a **linked list**, to support fast range queries.



- **Example**: Storing a phonebook, where each entry maps a name to a phone number.
  - We use a B+ Tree to organize the data by name, which acts as the index.
  - Indexes must be chosen wisely, they define how the data is ordered and searched.
  - Here, names make sense as the index because we often search alphabetically.
- **Indexes** are powerful because they maintain a **sorted order,** and B+ Trees are ideal for this. They help us quickly find the data we need without scanning everything.

- The vast majority of Relational DB are **ACID compliance** to ensure reliable and consistent data handling.



  - **A — Atomicity**: A transaction is all or nothing. If one part fails, the entire transaction fails, and no changes are made.
  - **C — Consistency**: The database moves from one valid state to another. Rules (like constraints) are always followed.
  - **I — Isolation**: Transactions don't interfere with each other. They act as if they are running alone.
  - **D — Durability**: Once a transaction is committed, the changes are permanent, even if the system crashes.

What Is a Database **Transaction**?

A transaction is a logical unit of work that can include multiple SQL statements (reads, inserts, updates, deletes).

- You start a transaction with `BEGIN`, run several statements, then finish with `COMMIT`
- Only after COMMIT do all the changes become permanent (durable). If you never commit, none of the changes stick.

- **Durability Example**: *Durability and Redis*
  - Redis, which stores data in **memory (RAM)**. It can act like a database (other than cache), but it's **not ACID-compliant**, especially when it comes to Durability.
  - Since RAM is volatile, data can be **lost on crash or restart**.
  - The trade-off is that Redis is **very fast**, but it **doesn't guarantee persistence** unless specifically configured with backup mechanisms.
  - On the other hand, **Relational Databases prioritize durability**, ensuring that once a change is saved, it stays saved.
- **Atomicity Example**: *Money Transfer*
  - Suppose we want to transfer ₹500 from Alice to Bob. This involves:
    - Subtract ₹500 from Alice's account
    - Add ₹500 to Bob's account
  - Here's how this might look in SQL inside a transaction:

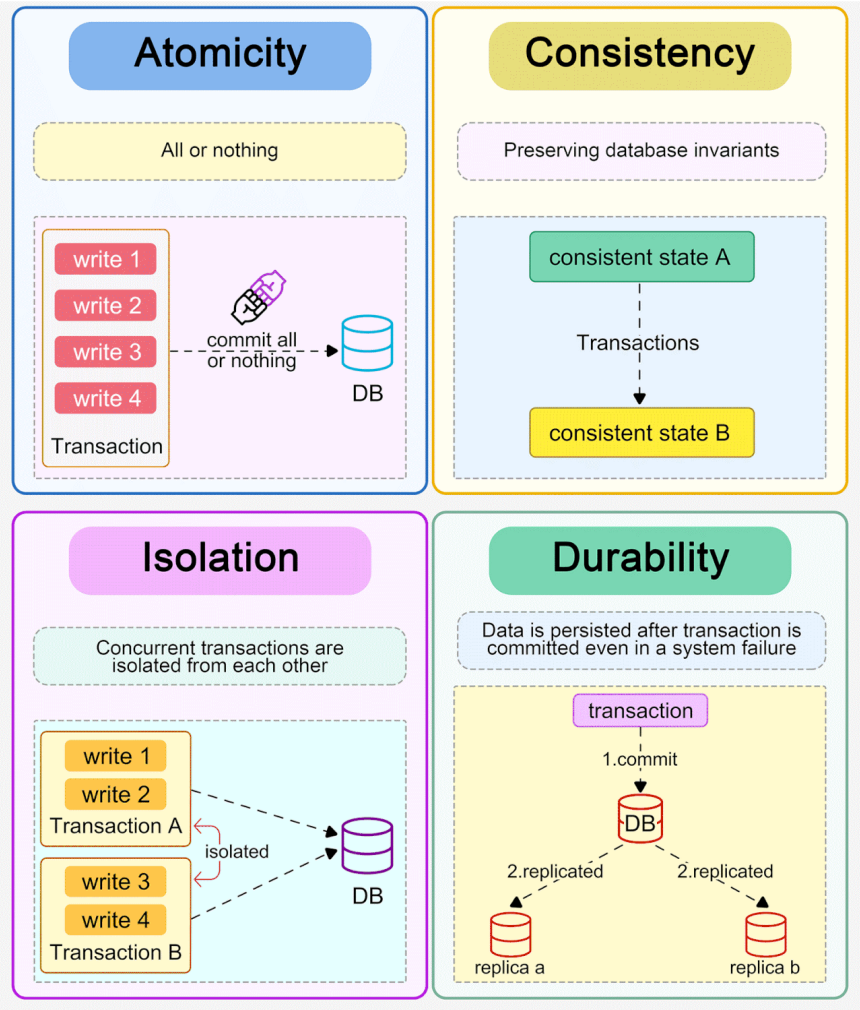
```sql
BEGIN;

UPDATE accounts SET balance = balance - 500 WHERE name = 'Alice';
UPDATE accounts SET balance = balance + 500 WHERE name = 'Bob';

COMMIT;
```

  - With Atomicity: Both steps succeed or fail together. If step 2 fails, step 1 is rolled back — **no money is lost.**
  - Without Atomicity: If step 1 succeeds but step 2 fails, ₹500 is removed from Alice but not added to Bob — **money is lost due to partial execution.**

- **Isolation** is important when multiple transactions are happening at the same time on the same database. It makes sure that one transaction doesn't affect another while it's still running.
  - If we don't have isolation, problems like dirty reads, non-repeatable reads, or phantom reads can happen. Let's understand this with a simple example.
- **Dirty Read Example**: *Alice and Bob*
  - Initial balances: Alice → ₹1000, Bob → ₹500
  - Transaction 1: Transfer ₹500 from Alice to Bob
  - Transaction 2: Add ₹200 to Alice's account
  - What can go wrong?
    - Transaction 1 deducts ₹500 from Alice (balance becomes ₹500) — not yet committed.
    - Transaction 2 reads that ₹500 and adds ₹200 → Alice has ₹700, and this gets committed.
    - Transaction 1 crashes.
  - Now the **final result** is: Alice has ₹700, Bob still has ₹500
  - But this is wrong — money is lost, and the **data is inconsistent**.
  - Transaction 2 read a value (₹500) that was not committed — this is called a **dirty read.**

- With proper **isolation**:
  - Transactions don't see uncommitted changes from others.
  - Transactions are executed in a controlled way, without overlapping in ways that cause errors.
  - So, even if transactions seem to run concurrently (which just means they overlap in time), isolation ensures the end result is consistent — as if they ran one after another.

- **Consistency** means the database always moves from one valid state to another, never violating its rules (like data types, foreign keys, or custom constraints).
  - It depends on Atomicity, Isolation, and Durability—only fully successful, isolated, and durable transactions can guarantee consistency.
  - Maintaining consistency is expensive, because the system must constantly check and enforce all rules before and after every transaction.
  - With consistency, we define rules (constraints, triggers, schemas), and our DBMS ensures we never reach an inconsistent state.



NoSQL (Not Only SQL)

- A better way to describe NoSQL databases is ***non-relational databases***. Unlike traditional relational databases, NoSQL databases don't follow a table-based structure with fixed relationships and joins. So, we can't use SQL-like queries to do complex joins or operations in the same way.

- Variations of NoSQL databases:

  1. Key-Value Store:

  - Examples: Redis, Memcached, etcd
  - These are the simplest type of NoSQL databases.
  - Data is stored as a pair: key → value.
  - Keys must be unique, but there's no relationship between different key-value pairs.
  - Most key-value stores (like Redis and Memcached) keep data in memory (RAM), not on disk.
  - Because of this, they are extremely fast for reading and writing data.
  - Downside: Data is usually not persisted, meaning it can be lost if the system crashes.
  - Common use: Used as a cache alongside a main database to speed up access to frequently used data.
  - Example: Caching hot product search results in an e-commerce site to reduce database load and latency.
    - Key: the exact search query (e.g. "search:wireless+headphones").
    - Value: the serialized list of "hot" product results (e.g. a JSON array of product IDs or small product objects).

  2. Document Store:

  - Example: MongoDB
  - These are more advanced than key-value stores.
  - Data is stored in the form of documents (usually JSON-like).
  - Each document is a nested set of key-value pairs and has its own primary key.

- No fixed schema – you don’t have to define fields in advance.
- This makes it flexible – you can store different types of data in the same collection. But they also support optional schema validation.
- You can add, remove, or change fields anytime.
- Use: Great for situations where your data shape can change over time.
- Example: You’re building a blog platform. Each blog post is a document.

3. Wide-Column Stores:

- Examples: Cassandra, BigTable
- Designed for massive scale, especially where you need to store lots of writes.
- Data is stored in rows and columns, but it's much more flexible than relational tables.
- You don’t always need a fixed schema, but you can have one if needed.
- Best used when you are writing a lot of data (e.g., time-series data).
- Downside: Not very ideal if you need to do a lot of reads or updates.
- Pros: Highly optimized for fast writes, especially across distributed systems.
- Example: You’re building a blog platform. Each blog post is a document.

4. Graph Databases:

- Examples: Neo4j, ArangoDB
- These are actually focused on relationships between data.
- Useful when data is connected in complex ways, like:
  - Social networks (followers, mutual friends)
  - Recommendation systems
- In relational databases, doing deep joins across large tables (like friends-of-friends) becomes slow and complex.
- Graph databases store this relational info natively, making such queries much faster and more natural.
- While they are a part of NoSQL, they do have a concept of relationships, so they're not fully non-relational.
- Example: You’re building a social media app. You want to represent who follows whom.

- The **biggest advantage of NoSQL databases is scalability.** They are designed to scale much more easily than traditional SQL (relational) databases.
- This is mainly because SQL databases follow **ACID properties**, which are great for keeping data reliable and consistent, but they also make it **harder to scale**, especially horizontally.

Scaling SQL Databases:

- Let's say we have one big SQL database handling tons of requests per second. The simplest way to make it faster is to **scale vertically**, meaning we upgrade the server (more CPU, memory, etc.). But this has limits.
- **Horizontal scaling** (adding more servers) is much more powerful, but it's very difficult with SQL databases because of the ACID rules. If we split one large database into two and store half the data on each server, now we have a new problem:
  - How do we know which server (or node) has the data we need?
  - What if a request goes to one node, but the data is actually on the other?
  - Also, SQL databases often rely on **joins and foreign keys**. So, if one part of the data is on Node A and another part is on Node B, and we need to combine (join) them, it becomes extremely complex. Maintaining consistency across different servers while still following ACID is very hard.

Why NoSQL handles this better:

- NoSQL databases **relax the ACID rules** to make scaling easier:
  - They might support atomic operations (some basic transaction-like behavior), but they often give up on full consistency.
  - They usually **don't support foreign keys** or complex joins.
  - They're often **schema-less**, meaning the structure of data is more flexible.
- Because of these design choices, **it's much easier to split data** across multiple nodes and **scale horizontally**. For example, document databases like MongoDB let you distribute data without worrying about complex relationships or joins.
- While NoSQL doesn't give you everything (like strong consistency or strict constraints), this trade-off is **intentional**. You give up some strict rules so that your system can **handle more data and more users at scale**—and that's the key benefit of NoSQL.

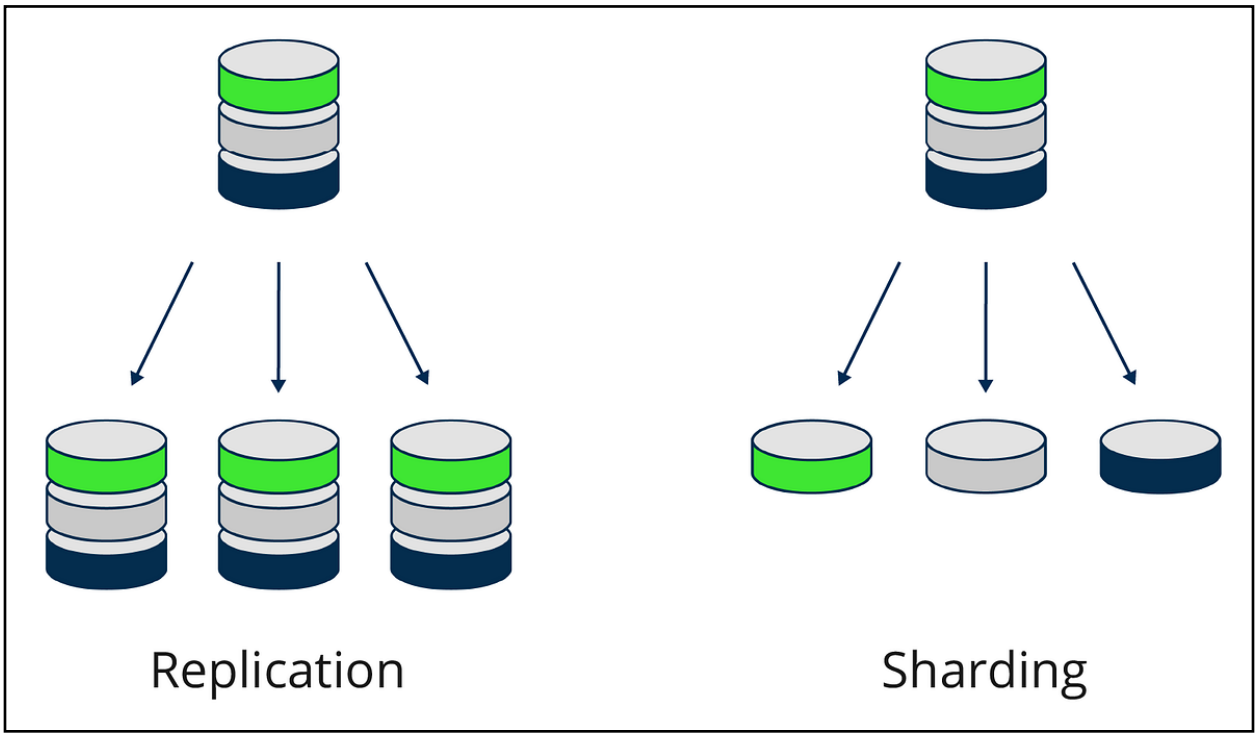
- While SQL databases focus on **ACID,** NoSQL databases follow a different model called **BaSE**, which stands for:
  - **Basically Available** – The system is generally available, even if not perfectly consistent.
  - **Soft State** – The state of the system may change over time, even without new input.
  - **Eventual Consistency** – The data will become consistent over time, but not immediately.
- BaSE isn't as well-known or strict as ACID, and in practice, it mostly comes down to **eventual consistency** — meaning that the system allows temporary inconsistencies, but will sync up eventually.
- One of the reasons NoSQL is popular is because it **scales well**. When a single database node gets too full or too slow, we can split the data across multiple nodes (horizontal scaling).
- However, instead of just splitting, NoSQL also uses **replication**—which means keeping multiple copies (replicas) of the same data on different nodes. For example, if you have Node A and Node B, both might store the same data.
- **Problem with replicas**: Let's say we write new data to Node A. To stay consistent, that same data needs to be written to Node B too. If that doesn't happen immediately, the replicas become **inconsistent**. So, if Node A has the latest update and Node B doesn't, a user reading from Node B will see **outdated data.**
- To solve this, NoSQL databases often use a **leader-follower** model:
  - One node is usually chosen as the leader (or primary).
  - Other nodes act as followers (or replicas).
  - All **writes** go to the leader. The leader is responsible for **pushing updates** to the followers, though there might be a **short delay**.
- This way, the leader makes sure the changes eventually reflect across all nodes. That's what **eventual consistency** is all about.
- **Example**: If you're viewing someone's *Instagram* profile and it shows 950 followers when they actually have 951, it's not a big deal. This minor lag is acceptable in many real-world scenarios. This is the tradeoff NoSQL makes: **slightly stale reads** in exchange for better performance and scalability.

Read vs Write Scalability

- NoSQL databases are optimized for **high read performance**. Even if only one node handles writes, **multiple replicas can handle reads**, which helps manage heavy read traffic.
- Think about an app like Instagram:
  - Most users are **scrolling (reading)**, not constantly posting (writing).
  - So it's fine to read from slightly outdated replicas if that lets the app handle millions of users smoothly.

Technically, SQL Can Replicate Too – But It's Harder

- SQL databases **can also replicate** and even **partition data (called sharding)**, but it's **much more complex** to manage compared to NoSQL.
  - In SQL, sharding and replication aren't built-in and require lots of manual setup.
  - NoSQL databases often **support sharding out of the box** (in-built), making it easier to scale horizontally.



Replication

- Let's say your application stores all of its data in a **single database** (SQL or NoSQL). At first, this might work fine. But as more users join and more requests come in, the database may get overwhelmed:
  - Too many simultaneous connections
  - Too much data to read and write
- To solve this, we use **replication** – creating copies of the database so we can spread out the load and handle more requests.

1. Leader-Follower (Master-Slave) Replication

- As talked before, Writes go only to the leader.
- One or more nodes are followers (replicas). They copy data from the leader and are used mainly for read operations.
- This setup helps scale read-heavy workloads (which is common in most apps).
- Note - Follower are also be used to further replicate to other followers offloading work from the leader node.
- Why Not Write to Followers?
  - If you let users write to a follower, that data won't go back to the leader, and the system ends up with **inconsistent data.**
- **Client Perspective:**
  - End users don't talk directly to the database. They send requests to your **application**, which then connects to the database. But from the app's view, it decides whether to read from the leader or a follower based on what kind of request it is.
- **Benefits of Leader-Follower Replication**:
  - Scales read operations – great for apps with lots of users reading data (which is the case in most of apps)
  - Improves availability and reliability – If the leader fails, one of the followers can take over (be promoted to leader).
- **Tradeoffs** around **consistency** and **latency**, depending on sync method.

Types of Replication: Synchronous vs Asynchronous

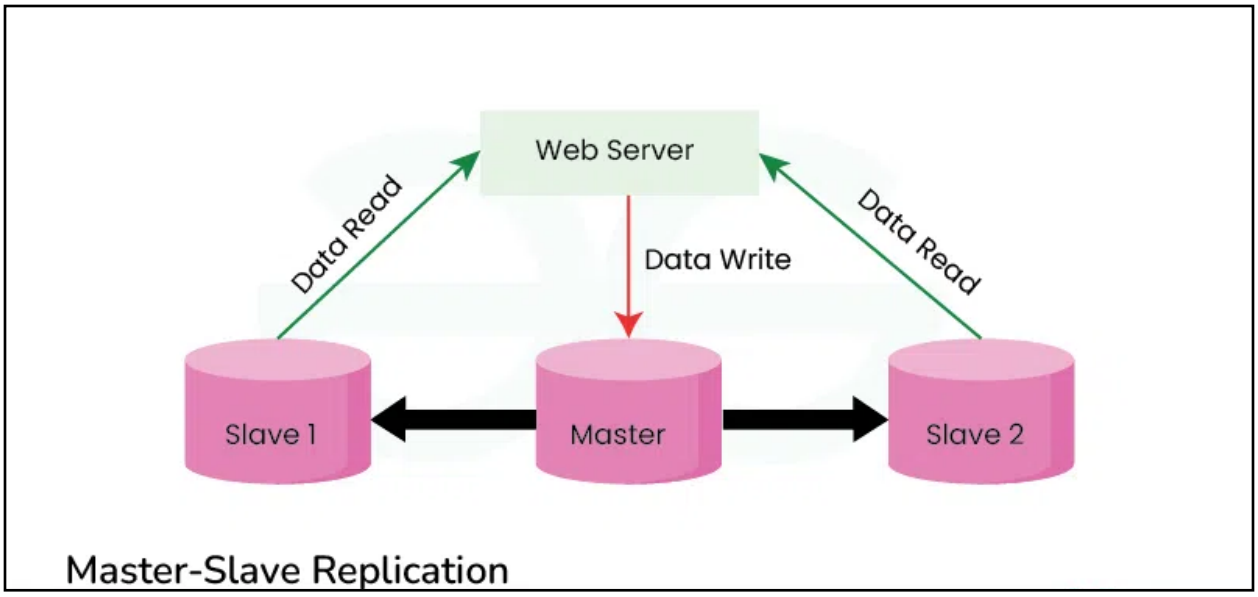
The main decision in replication is when the data gets copied. This leads to two strategies:

1. Asynchronous Replication (Async)

- As the name suggest, asynchronous is when we **don't have to it immediately.**
- Leader will accept the write/transaction immediately. And at some point, leader is going to take the this and replicate it to follower.
- We don't know how long it could take.
- It can be on schedule like every hour or it can be just few seconds later.
- Cons: **Risk of stale reads** – users reading from followers might get **outdated data**

2. Synchronous Replication (Sync)

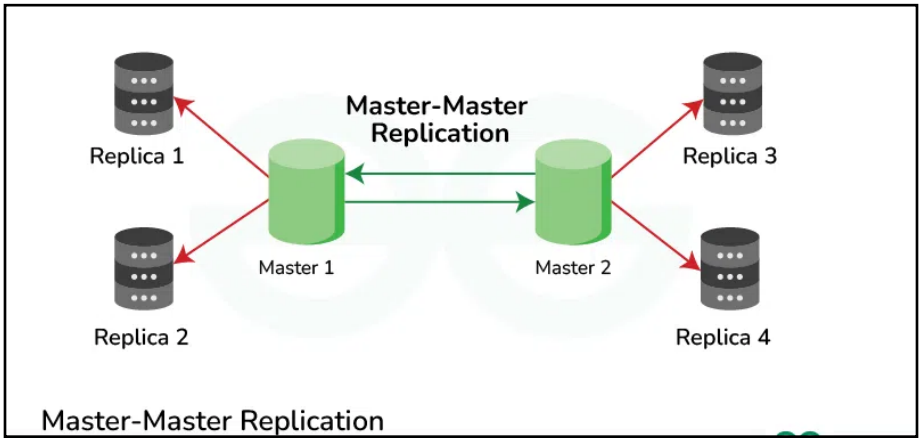
- In this, every time a user has a write/transaction to leader node, the **leader will stop immediately** and take it to the followers.
- **Only after replication is complete**, the write is considered done.
- Pros: **No stale reads**. All replicas have the same data instantly.
- Cons: Slower writes due to **higher latency**, especially if replicas are far away (e.g., in other countries)



Master-Slave Replication

2. Leader-Leader Replication

- Also called **Multi-Leader** or **Master-Master** replication, this setup allows nodes to:
  - **Accept writes** (not just one leader).
  - **Serve reads** like leader-follower systems.
- So now we can **scale both reads and writes** by adding more leader nodes. This is powerful — but also much **more complex**.
- So if we do async replication here, our data is going to be loosely consistent but it's definitely not going to be consistent.
- And if we do it synchronously, then we are going to see much higher latency depending on how many leaders we have and how they are geographically distributed around the world.
- **Why Use Multi-Leader Replication?**
  - The main reason is **geographical distribution.**
  - Imagine you have two leader databases, one in ***North America*** and one in ***Asia.***
    - Users in each region talk to the leader closest to them, reducing latency.
    - Each region handles its own traffic efficiently without depending on a single global leader.
  - Of course, the leaders can get out of sync, but that's acceptable in many cases.
    - The system can **sync them periodically** (e.g., every hour).
    - This works well if users in each region mostly stay within their region.
    - Even if a user travels from one region to another, the system will **eventually sync their data.**



Sharding

- Sharding is a technique used to handle large amounts of traffic and data in a database system. When a single database can't keep up with the number of reads and writes, and even **vertical scaling** (adding more power to a single machine) doesn't help anymore, we need a better solution.
- Replication (copying the same data to multiple machines) can help with read-heavy traffic, but it **doesn’t solve the problem of huge amounts of data**.
  - For example, imagine storing petabytes of data on a single machine—searching through all of that for each query would be slow.
  - Think about visiting Twitter and waiting 5–10 seconds just for your feed to load. **That’s not acceptable.**
- This is where **sharding** helps.
- Sharding means **splitting the actual data** into smaller, more manageable pieces and storing those pieces (called **shards**) on different machines. Each shard contains a portion of the data.
- ***For example***, if you have a table, you can divide the rows into two parts:
  - The first half of the rows goes into one shard (on its own machine).
  - The second half goes into another shard (on a different machine).
- It’s **important** that these **shards** **are on separate machines**, because putting them on the same machine wouldn’t really solve the performance issues.
- Now, when users make read or write requests:
  - The traffic is split between the shards.
  - Each shard handles only a **portion of the total data**, so **queries are faster**
  - The overall system can handle **more traffic.**
- So, sharding helps us solve:
  - **Scalability –** Traffic is split across multiple machines. Each shard can be scaled independently.
  - **Performance –** Each shard holds less data, so queries run faster.
  - **Handles Big Data –** You’re no longer limited by the size or power of a single machine.

Now how do we decide how to actually distribute the data? How do we decide what half of the data goes where?

- There are a couple of common strategies for this:

1. Range-Based Sharding

- In this method, we divide data based on **value ranges** of a certain column—this column is called the **shard key**.
- For example, if we use a user's **last name** as the shard key, we might say:
  - Users with last names starting from A–L go into Shard 1.
  - Users with last names from M–Z go into Shard 2.
- Another example might be splitting users based on gender:
  - Males go into one shard, females into another.
  - (Note: This is a simple discrete example rather than a range.)
- The shard key is typically something like a **primary key** or another frequently-used column.
- **Downsides:**
  - If the data isn't evenly distributed (e.g., more users have last names starting with "S"), one shard may end up with more load.
  - **Joins become complex or slow,** especially if you're trying to join data from two different shards. **Or it might not just work!**
  - **Related tables add complexity:**
    - If you shard multiple tables, and those tables are **linked** (e.g. by foreign keys), you now have to ensure related data ends up on the same shard.
    - This often requires **custom logic** in your application.
- **Pros:**
  - Simple to understand.
  - Easy to implement if your data is evenly distributed.

2. Hash-Based Sharding

- Instead of using value ranges, we use a **hash function** on the shard key to decide where data should go.
- For example, we hash the user ID, and based on the result, we assign it to a shard.
- A technique called consistent hashing is often used here. It's a special way of hashing that makes it easier to add or remove shards later without having to re-distribute all the data.
- Benefits:
  - Helps in distributing data more evenly across shards.
  - Avoids the "hotspot" issue where one shard has much more data or traffic than others.

Sharding in SQL vs. NoSQL Databases

- **SQL databases** (like MySQL, PostgreSQL) **don't support sharding by default**.
  - You have to implement the sharding logic yourself in your application.
  - Your app has to know:
    - How to divide data.
    - Where to find specific records.
    - Which shard to query or update.
  - **You lose many SQL features** like joins, foreign key constraints, and full ACID guarantees when data is spread across shards.
- **NoSQL databases** (like MongoDB, Cassandra, etc.) **have sharding by default**.
  - They often include automatic sharding support as part of the database system.
  - NoSQL systems **don't enforce strict relational rules**, so sharding works better with their design.
  - Data is often **eventually consistent** rather than immediately consistent, which is acceptable in many use cases.

CAP Theorem

- The **CAP Theorem** is a well-known concept in distributed databases, but it's often misunderstood and not as useful as many people think.
- **What is the CAP Theorem?**
  - CAP stands for:
    - **C – Consistency**
    - **A – Availability**
    - **P – Partition Tolerance**
- The theorem is only relevant to **distributed systems**, especially those with **replicated data**. It doesn't apply to simple databases running on a single server with no replicas.
- **When does it apply?**
  - CAP becomes important when you replicate your data across multiple machines—for example, in a **leader-follower** setup where changes in the leader are copied to the follower.
  - People often bring up CAP when talking about **NoSQL databases,** since they usually have replication built-in. However, trying to apply it to a **traditional SQL database with only one node** doesn't make sense—it's not a distributed system.

The Common Misunderstanding!!

- A lot of people think the CAP theorem says:
  - **"You can only choose two out of the three: Consistency, Availability, and Partition Tolerance."**
  - But that's not quite right.
- In reality:
  - **Partition Tolerance (P)** is always required in any real distributed system. Networks can and do fail, and the system must keep going despite those failures.
  - So the actual trade-off is between **Consistency (C) and Availability (A)—when a network partition happens.**

What is Partition Tolerance?

- Partition Tolerance means the system can keep running even if there's a **network failure**—that is, parts of the system can't talk to each other.
  - ***For example***, say you have a leader and a follower database. If the network between them goes down (a "partition"), the system is split. Each side may still be reachable by users, but they can't communicate with each other.
  - Now you have to choose:
    - Availability or
    - Consistency

So What Does the Theorem Actually Say?

In a distributed system that tolerates network partitions, you can choose either Consistency or Availability, but you can't guarantee both at the same time.

But What Do "Consistency" and "Availability" Actually Mean?

- One of the confusing things about the CAP theorem is that the terms Consistency and Availability don't always mean what people expect. That's where a lot of misunderstanding comes from.

1. Consistency

- In the context of CAP, **Consistency** means that **every read returns the most up-to-date data** across all replicas.
- Let's break that down:
  - Say someone writes data to the **leader** database.
  - Normally, the leader would send that update to the **follower** (replica).
  - But if there's a **network partition**, the leader and follower can't communicate.
  - If a user reads from the **follower**, they won't get the latest data—that's **not consistent.**
  - If we want to maintain **Consistency**, we must **prevent users from reading from replicas** that might have stale data.
- So, **Consistency** means no matter which node you read from, you get the latest, correct data.

2. Availability

- Availability in CAP doesn't mean "high uptime" like 99.99% availability.
- Instead, it means:
  - Every node that hasn't crashed must respond to requests.
- Even if a node has **stale or outdated data**, as long as it's up and running, it should still answer user requests.
- Here's the key point:
  - Let's say the follower is cut off from the leader due to a partition.
  - It still works fine, but its data might be old.
  - If we let it respond to users, we are choosing Availability over Consistency.
  - If we want Consistency, we might block that node from answering—even though it's not crashed.
- So in this case, we're choosing to sacrifice availability (of that node) in order to maintain consistency across the system.

Why You Can't Have Both During a Partition (**Final Verdict**)

- Now it's clear why CAP makes you choose between ***Availability and Consistency*** when a **Partition** happens:
  - If you want **every node** to keep responding, then some responses might be **inconsistent.**
  - If you want **only consistent data**, then you have to **block some nodes** from responding, which reduces availability.

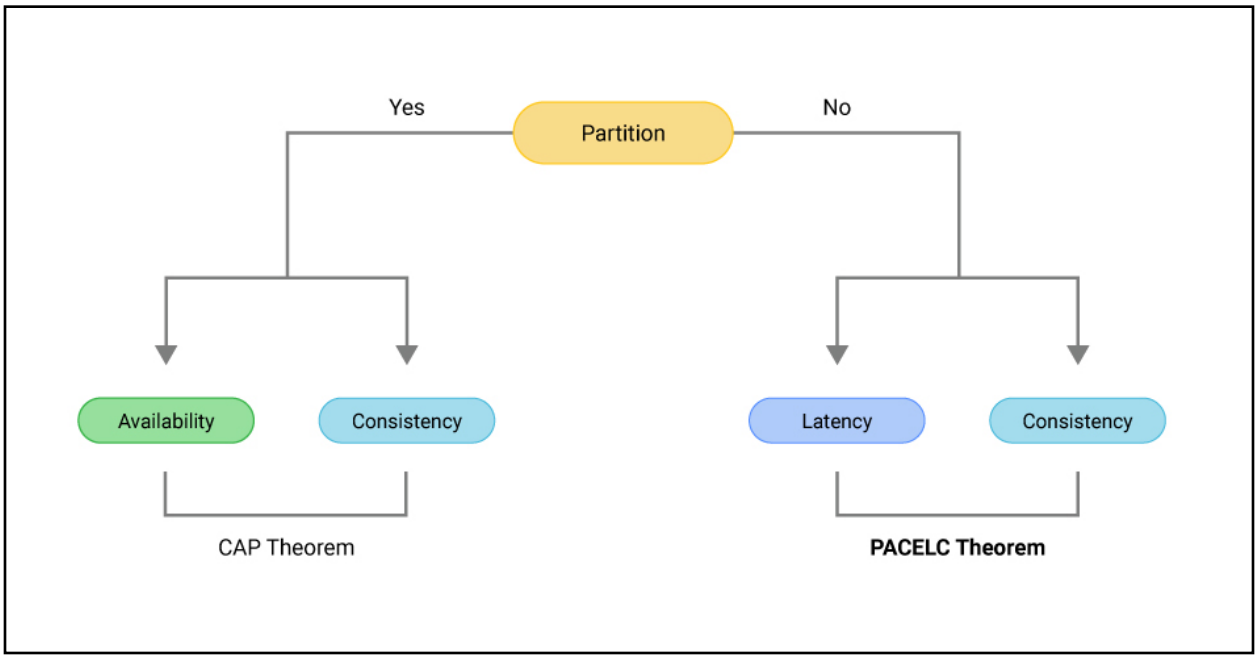
-> You either allow all nodes to serve requests (even if some are out-of-date), or you restrict access to only the consistent ones, sacrificing availability.

- That's the real trade-off in the CAP theorem and should be chosen based on situational needs.
- And sometimes, we don't even get two out of three—consistency or availability isn't guaranteed, and we might end up with neither one.

PACELC Theorem

- This theorem is actually a extension of CAP theorem. PAC is still the CAP theorem.

Given P, choose A or C. Else, favor Latency or Consistency



- The CAP theorem only talks about trade-offs **during a network partition**. But in real systems, partitions don't happen all the time—though they're always possible, since networks can fail at any time (like a router crashing, a data center going down, etc.).
- So what about when the system is running normally, with no partition?
  - Even in that case, we often **can't have both low latency and strong consistency.**

The Latency vs. Consistency Trade-off

- When you replicate data across databases (like from a leader to a follower), that replication **takes time.** *(We have talked about this in async vs sync replication)*
- So if a user tries to **read** from a replica, we face a decision:
  - Do we return the data immediately, even though it might be out of date?
    - That gives us **low latency** but **weaker consistency.**
  - Or do we **wait** for the data to be updated, and then return it?
    - That gives us **strong consistency,** but **higher latency.**

Object Storage

- Object storage is more similar to a file system than a database. In databases, organizing, filtering, and searching data is crucial. But object storage focuses more on storing and retrieving large chunks of data, not querying it.
- **Flat Structure (No Folders**)
  - In object storage, data is stored in a **flat structure**. Unlike traditional file systems where you have folders and subfolders, object storage has no real hierarchy.
  - Services like **AWS S3, Google Cloud Storage, and Azure Blob Storage** are common examples.
  - These services may give the illusion of folders (e.g., S3 shows "folders" in its interface), but behind the scenes, it's just part of the object's name (called the key). There's no actual directory structure—everything is stored in one flat space.

What Are Objects?

- In object storage, we store **objects**, also called **blobs (Binary Large Objects)**. These are usually large files like:
  - Images
  - Videos
  - Database backups
  - Any other big files you don't plan to edit
- You can **upload** (write) and **download** (read) these objects—but you can't update them. If you need a change, you have to upload a new version.
- This is a key tradeoff of object storage: **it's great for storing large files, but not for frequently changing data.**

How It Works

- Object storage is optimized for huge numbers of objects. Since it's flat, you can quickly retrieve any file using a unique key (similar to how a HashMap works).
  - Each object/file has a **globally unique name**.
  - In services like AWS S3, these names must be unique across the entire system, not just your project. For example, no two users can have the same S3 bucket name.

When to Use Object Storage

- Object storage is ideal when you need to store:
  - Images
  - Videos
  - Media files
  - Large backups
  - Anything you want to store for a long time but don't need to edit
- You *could* store these in a database, but it's not a good idea. Storing large files in a database:
  - Slows down performance
  - Uses more expensive storage
  - Makes querying inefficient
  - Adds unnecessary complexity
- Instead, use a database to store metadata about the files—like file name, tags, or upload date—but keep the actual files in object storage.

Accessing Files

- You don't need to use SQL or database queries to get files from object storage.
  - You access files via HTTP requests.
  - Just make a network call to the object's URL (like an S3 link).
  - The file's name (key) is all you need to fetch it.
  - You can control access with permissions, so only your app or certain users can access the files.
- From system design interview perspective, anytime you have large file, you'll mostly be using object storage for them.

Message Queues

- A message queue is a tool used in system design to help manage large amounts of tasks or "events" without overwhelming the application server.
  - It's a middleware or broker that decouples producers and consumers.

♦The Core Idea♦

- Sometimes, your app sends a lot of events (like user actions, logs, or data updates) to the server **very quickly** — more quickly than your server can handle.
- If the server tries to do everything at once, it can slow down or even crash.
- Instead of handling every event **immediately**, it's often better to put them in a line (a queue) and deal with them **one at a time** or in smaller batches. This is where a **message queue** comes in.
  - Note: Many MQ systems support parallel consumers for higher throughput.

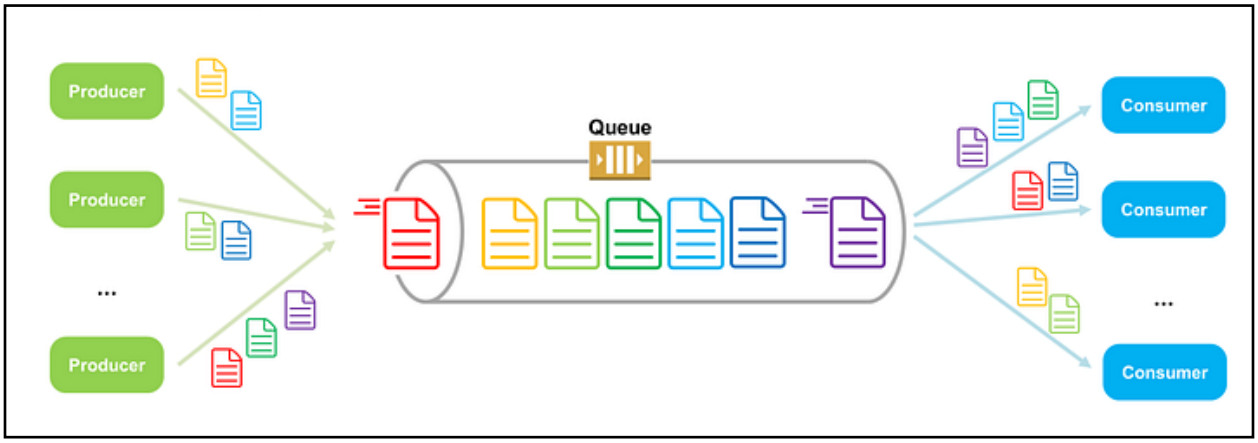
How It Works

1. Producer (Event Source): Any application or service component that creates and publishes messages to the queue, such as user actions, payment requests, or data updates.

2. Message Queue: A separate component that stores all the events safely, like a to-do list. It aims to hold messages reliably until the server is ready to process them.

3. Consumer (Processor): This is your application server. It reads the messages from the queue and handles them one by one (or in batches).

- Note: You can have multiple consumers for parallel processing. Messages can be acknowledged manually or automatically to support retries and ensure reliable processing.



Real-World Example: Processing Payments

- Imagine an online store:
  - Many users are buying items at the same time.
  - Each purchase creates a payment event that needs:
    - Payment processing
    - Saving the transaction
    - Logging and analytics
- All of these steps take time. If you try to do them immediately for every user at once, your server might get overloaded.
- **Solution:**
  - Each payment event is sent to a message queue.
  - The server reads one event at a time from the queue and processes it.
  - Even if 1,000 users check out at the same time, your app doesn't crash because the events are being processed in order and at a manageable pace.

Benefits of Using a Message Queue

- **Asynchronous Processing**: Tasks are done in the background, not instantly.
- **Scalability**: You can handle more events because you're not doing everything at once.
- **Decoupling**: The system creating the events doesn't need to know how or when they'll be processed.
- **Durability** (when enabled): Messages can be persisted to disk, ensuring they survive system crashes. However, this is configurable, some systems offer in-memory-only options for higher performance.
  - Some brokers (RabbitMQ, Kafka) offer disk persistence by default or via configuration.
  - Others (Redis Pub/Sub, in-memory queues) may lose messages unless durability is enabled.
- **Flexibility**: Messages can be processed in various orders - FIFO within partitions (Kafka), priority-based (RabbitMQ), or custom routing logic, depending on the system's capabilities.

- Message queues don't just hold messages — they also **manage the delivery process**. They make sure your server **gets the messages reliably,** and they keep trying if anything goes wrong. This makes your system more **robust, durable, and fault-tolerant.**

Message Delivery in Queues: Pull, Push, and Reliability

- There are two main ways your server can receive messages from the queue:

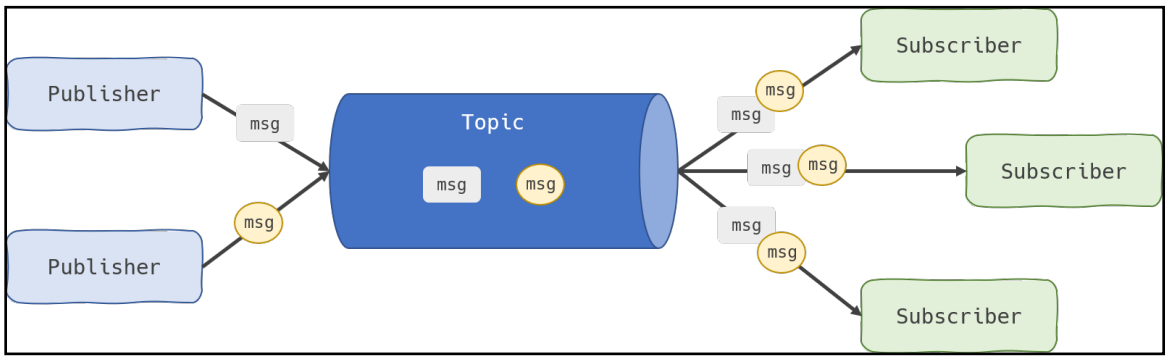
1. **Pull Method (Polling)**
   - The application server checks the queue regularly (every few seconds or minutes).
   - It asks: "Do you have any new messages?"
   - If the queue has something, the server takes it and processes it.
   - If not, the server waits and checks again later.
   - Think of this like checking your mailbox every hour to see if the mailman has delivered anything yet.
2. **Push Method**
   - Queue proactively sends messages to the application server as soon as they arrive
   - This can reduce latency compared to polling, but may increase complexity in handling back-pressure.

Acknowledgement (Ack): Did the Server Get the Message?

- Sometimes, even if the message is delivered, the **server may not be able to process it right away** — or something might go wrong (like a crash or error).
- To make sure nothing is lost, we use an "**acknowledgement**" system:
- **How It Works:**

1. The queue sends a message to the server.
2. The server **processes the message.**
3. The server then sends back an "**ack**" ("acknowledgement") to the queue saying:
   - "I got it. You can mark this as done."
4. If the server **doesn't ack** (maybe it failed, crashed, visibility timeout, or missed it):
   - The queue tries to **resend** the message.
   - It keeps retrying until it gets an ack — or until a maximum number of tries.

- Without acknowledgements:
  - If a message is lost or not processed correctly, it's gone forever.
  - That could lead to serious issues — like a lost payment or a missing update.
- With acknowledgements:
  - The system is more reliable. This typically ensures 'at-least-once' delivery.
  - You can be confident that no important message is lost.



- So far, we've looked at a basic setup:
  - One application → one message queue → one application server
- But real-world systems are often more complex. You might have:
  - Many apps sending messages
  - Many services needing to receive those messages
- This is where a system called **Pub/Sub (Publisher-Subscriber)** comes in.
- **Pub/Sub** extends the basic message queue pattern by allowing multiple consumers to receive copies of the same message, enabling broadcast communication patterns
  - A **publisher** is an app or service that sends messages (events).
  - A **subscriber** is an app or service that receives those messages.
  - Between them is a middle layer called a **Topic**.

How Pub/Sub Works

1. **Publishers** send messages (like payments, user actions, or analytics data).

2. Messages go to a **Topic** — a durable middle layer that stores and organizes these messages.

3. Each Topic can feed into one or more Subscriptions.

4. **Subscribers** (app servers or services) pull messages from subscriptions — not directly from the topic.

- This setup allows multiple services to receive the same message, or even different messages based on what they need.

Topics: Organizing Messages by Category

- Think of topics as labeled bins.
- Example:
  - Topic A: handles all payment messages
  - Topic B: handles all analytics messages
- Each topic stores only a specific type of message, which helps keep things clean and organized.

Subscriptions: Spreading Messages Out

- A topic can have multiple subscriptions.
- Each subscription is like a delivery route to a different service.
- A single message from a topic can be sent to multiple subscribers.

This is called **fan-out**:
One topic → many subscriptions → many receiving services

Mix & Match

- One service can **subscribe to multiple subscriptions or topics.**
- You can **replace or add services** without changing the whole system.
- The **message queue layer handles the logic**, so your actual apps stay **simple and flexible.**

Popular Tools

- Some common message queues and Pub/Sub systems:
  - **RabbitMQ** (open-source, queue-focused)
  - **Apache Kafka** (high-throughput, real-time streaming, distributed stream platform)
  - **Google Cloud Pub/Sub** (fully managed, cloud-native)
- They each have trade-offs in performance, complexity, and features — but they all support this kind of scalable message delivery.

Simple Analogy: The Event Bakery

- Imagine a bakery (your system) that takes custom orders.
  - Customers (apps) place orders (messages)
  - Each order goes into a box labeled with the type of cake (topics)
  - You have different delivery drivers (subscriptions), each picking up specific boxes:
    - One delivers to the kitchen (process the order)
    - One sends a copy to a storage room (backup)
    - One logs every order for statistics (analytics)
- If you need to change who delivers or add a new delivery route, you just change the subscriptions — not the bakery, not the customer, and not the order process.
- That's the power of message queues and Pub/Sub!

Real-World Example: Sending Notifications (Emails, SMS, Push Alerts)

- Imagine you're using an app like **Instagram or LinkedIn**. Here's what happens:
- You get a **notification** when:
  - Someone likes your post
  - Someone sends you a message
  - You get a new follower
- Each action here **triggers a notification,** and notifications can come in many forms:
  - Email
  - Push notification
  - SMS (text message)
- Imagine **millions of users** doing things at the same time. That's a lot of notifications!
- **The Problem**: If every user action immediately tried to send an email, push, and SMS in real-time, the servers would:
  - Get overloaded
  - Slow down
  - Possibly crash
- Also, what if the **email system** is temporarily down? You don't want to lose the notification.

- **The Solution:** Message Queues + Pub/Sub

  1. User likes a post

  2. The app sends a “like event” to a message queue

  3. The event goes into a topic called `notifications-topic`

  4. This topic has multiple subscriptions:

    - `email-subscription` → triggers email service
    - `sms-subscription` → triggers SMS service
    - `push-subscription` → triggers push notification service

  - Each service independently processes the message when it’s ready. The queue ensures:
    - No message is lost
    - If something fails, it retries
    - The original app stays fast and responsive

- **Analogy Revisited: The Notification Factory**
  - A user takes an action (e.g., likes a post)
  - That action is dropped into a "Notifications Box" (topic)
  - From that box:
    - One delivery truck sends an **email**
    - Another sends a **text message**
    - Another pushes an **alert** to your phone
  - Each delivery truck is **independent**, and you can add or remove trucks without changing how the box works.

MapReduce

- MapReduce is a programming model used for **processing large amounts of data (Big Data)**. It helps in splitting up big tasks across many machines to get the work done faster.
- Imagine you have personal data for millions (or even billions) of people — names, addresses, phone numbers, dates of birth, etc. You need to go through all of it and **remove sensitive information**. You could do this with one or two machines, but it would take a very long time.
  - If you want to do it faster, you need **multiple machines (or nodes)** working together. You split the job into smaller tasks, let each machine handle a part, and then combine the results. This is what MapReduce is all about — **splitting and processing large data in parallel.**

Two Main Types of Data Processing:

1. Batch Processing

- You already have all the data upfront.
- You process it in one go, like running a job to clean data or count something.
- Example 1: Removing personal fields from a large dataset of user profiles.
- Example 2: Counting how many times the word "Potter" appears in all the Harry Potter books.
- Can run once or at regular intervals (e.g., once a week).

2. Streaming Processing

- Data arrives continuously in real time.
- You don't have the entire dataset in advance.
- Example: Every time someone makes a payment, a transaction record comes in. You want to immediately remove the last name from the transaction.
- Streaming is useful when **data is being generated constantly,** like from payments, sensors, or live feeds.
- Since the data isn't static or complete, streaming is **usually more complex** to handle than batch processing.

Micro-Batch Processing

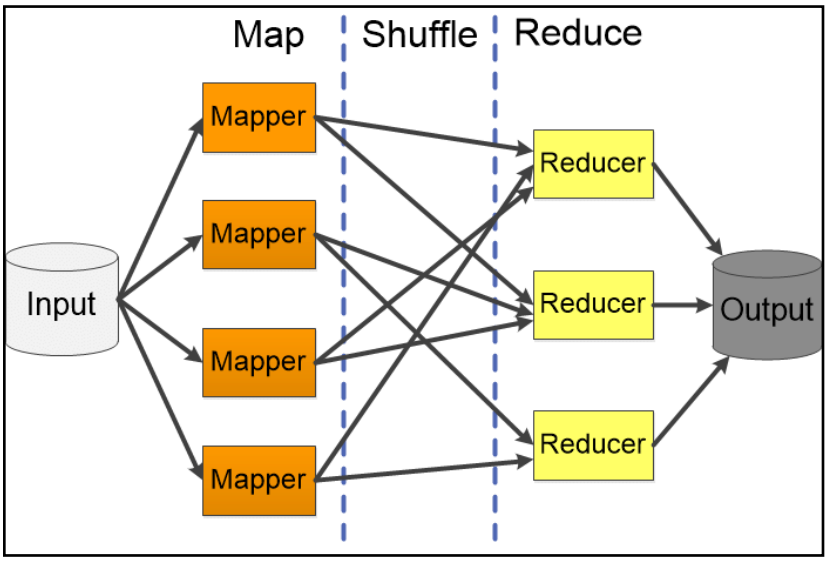
- Sometimes, instead of true streaming, systems **process small batches very quickly** (e.g., every 30 seconds). This is called **micro-batching**, and it mimics real-time processing.

History

- Google introduced MapReduce in 2004 to solve the problem of processing massive datasets using multiple machines.
- Although it's not commonly used anymore (even Google moved on), the core ideas are still relevant.

The Architecture

- You have some large input data, usually stored in a **distributed file system or object storage.**
- The data is split into parts and sent to different machines, called **workers**.
- Each worker processes a portion of the data.
- A **master (or leader)** machine (from one of the machines) coordinates everything:
  - It decides how to split the data.
  - It assigns tasks to each worker.
  - It manages the overall job execution.
- This setup allows the system to scale — adding more machines means faster processing.



- Let’s go back to our earlier example where we want to count words in all the Harry Potter books. Now, imagine we have 3 machines, and each machine is assigned one-third of the total pages.

Step 1: Map Phase

- Let’s goIn this phase, each machine (also called a worker) will:
  - Read its portion of the data (e.g., 1/3 of the pages).
  - Count how many times each word appears in its portion.
  - Produce key-value pairs where the key is the word and the value is the count (e.g., "the": 34).
- Each worker does this independently for their share of the data.
- Note: The map function is user-defined, meaning you (the programmer) write it based on what you want to achieve (like counting words, redacting names, etc.). You don’t control which data goes to which worker, but you do control how to process that data.

Step 2: Shuffle Phase (Shuffle-by-Key)

- After mapping, we now have many word-count pairs across all workers.
- The shuffle phase groups the same words together across all workers.
  - For example, all "the" word counts from each worker are sent to a single worker for final tallying.
- This ensures that each unique word and its counts — no matter where it appeared — are sent to the same machine.
- If we have millions of unique words but only 3 workers, each worker can handle around 1/3 of the words. The master/leader node decides who gets what, but we usually don’t worry about that part as developers.

Step 3: Reduce Phase

- Now each worker has a subset of all the unique words and their associated counts.
- Their task is to aggregate (sum up) the counts for each word.
- The final result — total count of each word — is then saved or written to some output storage.
- This completes a full **MapReduce** job.

Important Takeaways

- The MapReduce model gives you control over:
  - The map step: How you want to transform the input.
  - The reduce step: How you want to combine the results.
- You don't need to manage the infrastructure — things like splitting the data, assigning tasks, scaling with more machines — that's handled by the system.
- You focus only on writing efficient map and reduce functions.

Why Is It Called MapReduce?

- The terms come from programming:
  - map() — applies a function to a dataset (like JavaScript's array.map()).
  - reduce() — combines results (like array.reduce() in JS).
- It's a simple but powerful model for handling large-scale data.

Limitations of MapReduce

- You are limited to the map and reduce pattern.
- Not all problems naturally fit into key-value pairs.
- Complex workflows (e.g., chaining multiple steps together) can be hard to implement cleanly.
- You can write multiple MapReduce jobs and chain them, but this gets complicated fast.

Modern Alternatives

- **MapReduce is outdated**, though its the foundation for large-scale data processing.
- Hadoop was a major system built on MapReduce — still used but largely replaced.
- Apache Spark is now the go-to for modern batch processing: Faster, More flexible, Easier to work with, and **Can run MapReduce-like jobs**, but with better tools and APIs.
- **Spark also supports micro-batch processing** — small batch jobs run every few seconds to simulate real-time processing.
- For **true streaming (real-time data as it happens)**, a better tool is **Apache Flink** — it's designed specifically for real-time data pipelines.