If you have any questions or suggestions regarding the notes, please feel free to reach out to me on

Sanjay Yadav Phone: 8310206130

https://www.linkedin.com/in/yadav-sanjay/

https://www.youtube.com/@GeekySanjay

https://github.com/geeky-sanjay/

Load Balancing - 1 - June 5

- Minimum Viable Product (MVP)
- Module Introduction
- Why do we need distributed system
- What happens to the user's data when a user does not logout ?
- DNS and ICANN
- how a URL is different from the IP address or how they are related?
- What is reason of keeping layered design.
- Do root DNS sync up with ICANN servers often
- How can people globally and uniquely identify a domain name
- Will it take longer for you to connect to Google DNS because you are overwriting the default DNS?
- Distribution of load
- Is IP assigned by ISP , get added in ICANN Domain
- So if we consider the cost, we are getting better configuration in horizontal.... Then where we should use vertical

Important Link

HLD Curriculum

https://docs.google.com/spreadsheets/d/1Zed_C4BHrF2LFer1ZVpQivIU947KvWnRzR9EL1iNv_k/edit

Class Summary - HLD Basics & Consistent Hashing

https://docs.google.com/document/d/1wY6rSGJwqa9uxAsoYZUQ0FumV3Sp5kRBCknUQPT3gcM/edit

Domain extensions (.com .us .in .org)

https://en.wikipedia.org/wiki/List_of_Internet_top-level_domains
https://www.godaddy.com/resources/skills/most-common-domain-extensions

Google DNS

https://developers.google.com/speed/public-dns

Domain Name Service - reference material

https://www.namecheap.com/support/knowledgebase/article.aspx/319/2237/how-can-i-set-up-an-a-address-record-for-my-domain/
https://support.dnsimple.com/articles/differences-between-a-cname-alias-url/
https://datatracker.ietf.org/meeting/108/materials/slides-108-tdd-sessb-dns-deep-dive-part-1-pdf-00
https://www.cloudflare.com/learning/dns/what-is-dns/

Case Study of De.licio.us

De.licio.us was a bookmarking website created by Joshua in 2003. At that time, the internet speed was around 2kbps. For context, YouTube came in 2005, AWS (Amazon Web Services) in 2006, and Chrome around 2008.

De.licio.us allows users to save their favorite websites as bookmarks and access them later by logging into the site. It was very popular back then because there was no system like Chrome's bookmark synchronization, making it very useful for people to save their favorite websites.

Minimum Viable Product (MVP)

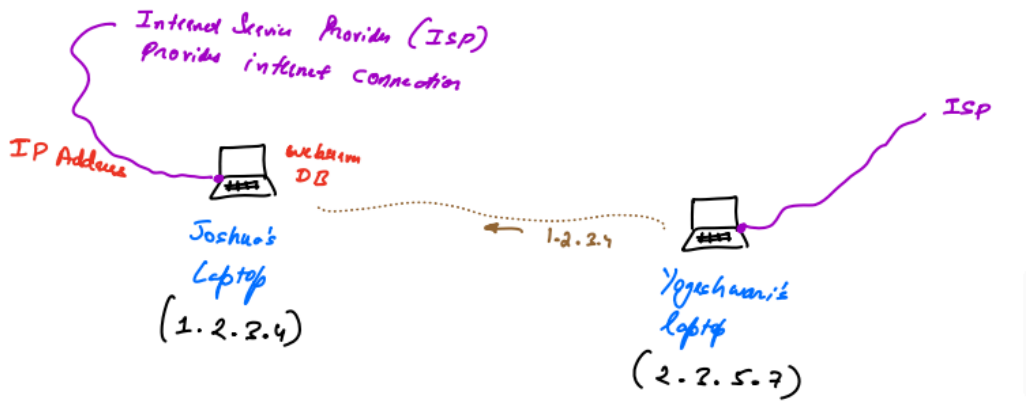
When starting a project, it's a good idea to begin with the basics, known as a "Minimum Viable Product (MVP)." This is like a Proof of Concept, Blueprint, or Prototype of the product.

- Product means it should solve a problem.
- Viable means people should be able to use it or see a demo of it.
- Minimal means any feature that is not critical should not be implemented. We only develop essential features needed to use it.

Minimum Requirements for De.licio.us

- **User Registration and Login**: Users should be able to register and log in (authentication and authorization). Note that logging out is not required for the MVP, as its absence won't affect the system's functionality.
- **Add Bookmarks**: Users can add bookmarks using an API. For example, a POST request to "/bookmarks" with a payload like {URL, userId, payload}. The userId can be provided by the user either through a header or a token. The API will return an acknowledgment indicating success or failure.
- **View Bookmarks**: Users should be able to view their own bookmarks (authorization). This can be done with a GET request to "/bookmarks/" and userId as the payload. The response will be a list of bookmarks for that user.

How Can People Use My Website from Their Laptops?



To allow others to access your website, you need to follow these steps:

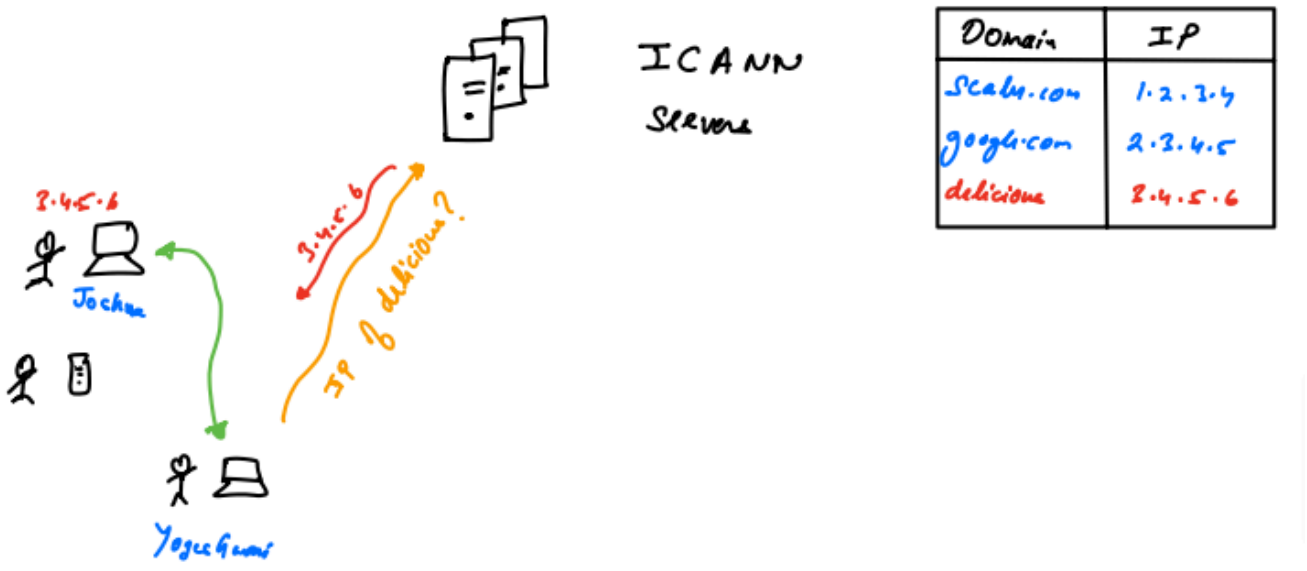
1. **Connect to the Internet:** Ensure your laptop is connected to the internet.
2. **Provide Internet Access:** The other users also need to have an internet connection. Internet service providers (ISPs) like Jio, BSNL, Vodafone, Act, Verizon, Hayai, AT&T, Comcast, and Starlink give us an IP (Internet Protocol) address. This IP address is unique on the internet and allows others to find your system online.
3. **Understanding IP Addresses:** Every device on the internet has a unique IP address. Because there are many users, creating a unique IP for each is complex. To handle this, terms like static vs. dynamic IP, subnetting, and public vs. private IP address are used.
4. **Connecting to a Device:** To connect to any device on the internet, you need to know its IP address. For example, if system one wants to connect to system two, system one must enter system two's IP address in the browser. This IP address is the public address.

When You Enter a Domain Name/URL, How Does the Browser Know Which IP to Connect To?

People don't remember IP addresses easily, so we use domain names instead. Here's how the process works:

1. **Buy a Domain Name:** Purchase a domain name from a domain provider like GoDaddy, Namecheap, or Hostinger.
2. **Configure Your IP Address:** Link your domain name to your IP address through the domain provider's settings.
3. **ICANN Servers:** The configuration you set up is sent to ICANN servers. ICANN is a non-profit organization that maintains all domain records, associating each domain name with its corresponding IP address.

ICANN -> ICANN stands for the "Internet Corporation for Assigned Names and Numbers". It's a non-profit organization responsible for coordinating the mai ntenance and procedures of several databases related to the domain name and IP address of the Internet, ensuring the network's stable and secure operation. In simple terms, ICANN makes sure that when you type a website address into your browser, the internet knows exactly where to send your request. It keeps everything running smoothly by managing the unique identifiers that allow computers to find and communicate with each other on the internet.



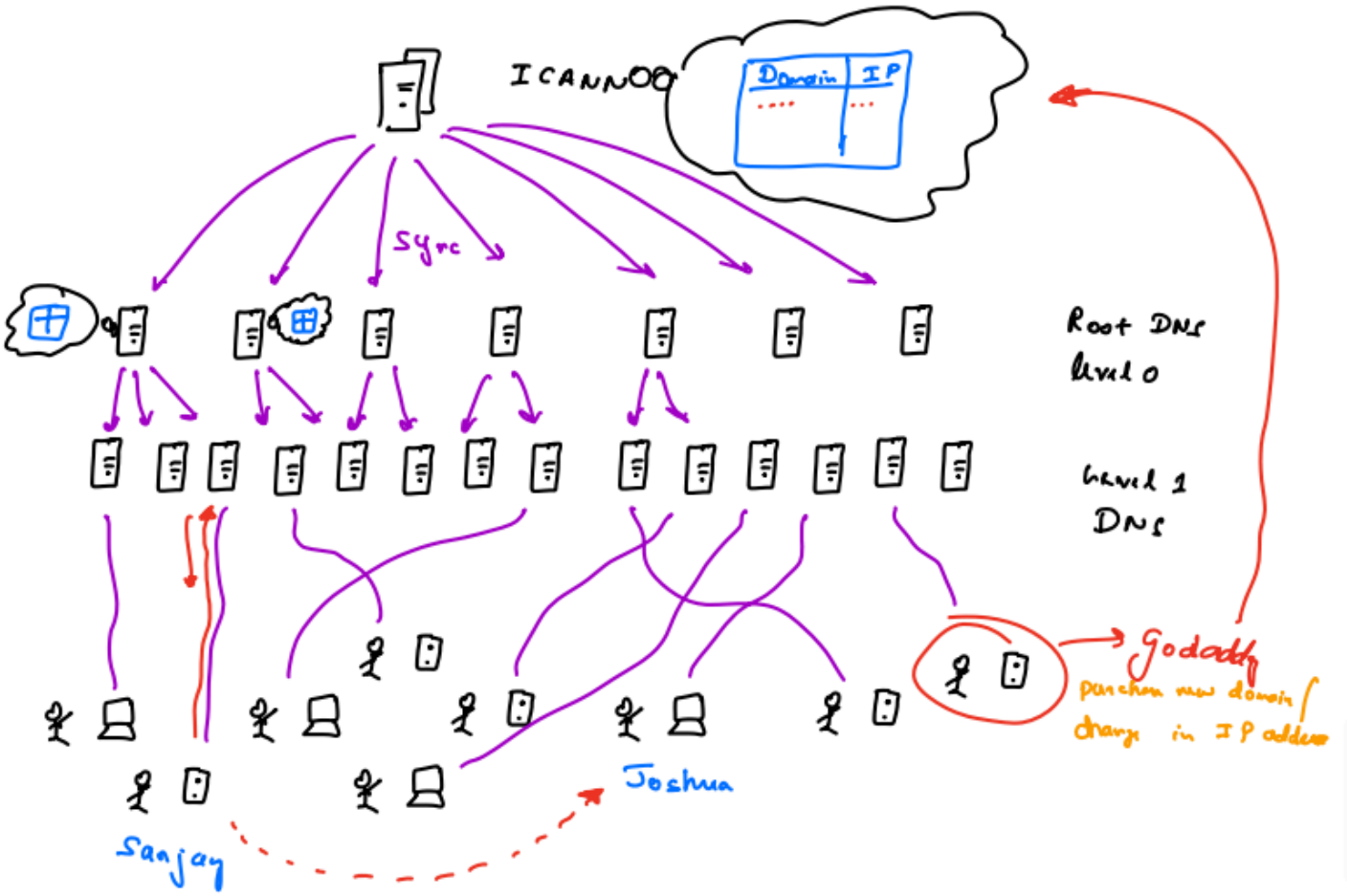
Here's an illustration to explain the process:

Yogeshwari wants to connect to Joshua's laptop. She types Joshua's domain name into her browser, which then sends a request to an ICANN server. The ICANN server looks up the domain name in its DNS records and returns Joshua's IP address to Yogeshwari's browser. With this IP address, Yogeshwari's browser can now connect to Joshua's laptop.

Issues with the Above Design

1. **Bottleneck:** The current design faces a bottleneck issue where the server receives too many requests, leading to slow performance. With billions of people and trillions of devices connected to various websites, the load on servers can become overwhelming. If a trillion devices were to hit the ICANN server, it could easily crash due to the excessive load.
2. **Single Point of Failure:** Another issue is the single point of failure. If the ICANN server were to go down, it would disrupt the entire internet service. This means that if the ICANN server experiences any technical issues or maintenance, it could lead to a widespread outage, affecting internet access for countless users and devices.

How DNS Works and How It Helps Solve Both Issues



As illustrated, the solution to this issue is DNS. Here's how it works:

1. You register your domain with GoDaddy, which communicates with the ICANN server to register your website.
2. When someone registers a website, GoDaddy sends this information to the ICANN server, creating an entry in the ICANN database.
3. The information then propagates to various levels of DNS servers, starting from root DNS servers, which are organized in a hierarchical, multi-level structure.
4. Level 0 DNS servers sync with the ICANN server and internally maintain this mapping. These servers also sync with higher-level DNS servers, such as level 1 DNS servers, and so on.
5. Users connect to different DNS servers, ensuring there is no single point of failure. If one server is down, users can connect to another DNS server.
6. This layered design prevents bottlenecks because millions of users are distributed across thousands of DNS servers. Each server is connected to its top layer, which in turn is connected to higher-level layers, rather than all servers being directly connected to the ICANN server.

Who Maintains the DNS Servers?

DNS servers are maintained by large organizations that rely heavily on the internet, such as Google, Microsoft, Facebook, Amazon, Netflix, military entities, government agencies, financial markets, universities, and Internet Service Providers (ISPs).

Every ISP has its own DNS servers. When you receive a router from your ISP, it is typically pre-configured with the ISP's DNS address. However, you can override this DNS IP setting by changing the TCP/IP configuration on your router..

How to Change DNS Settings Directly from Network Settings on Your Computer

Google DNS
https://developers.google.com/speed/public-dns

Changing the DNS settings directly from your network settings can improve your internet speed, security, and reliability. Here's how to do it on Windows, macOS, and Linux:

For Windows:

1. **Open Network Settings:**
   - Press `Win + R` to open the Run dialog box.
   - Type `ncpa.cpl` and press Enter to open the Network Connections window.
2. **Select Your Network Adapter:**
   - Right-click on your active network connection (e.g., "Wi-Fi" or "Ethernet") and select "Properties."
3. **Access TCP/IPv4 Settings:**
   - In the properties window, scroll down and select "Internet Protocol Version 4 (TCP/IPv4)," then click "Properties."
4. **Enter DNS Server Addresses:**
   - Select "Use the following DNS server addresses."

    - Enter the preferred DNS server address (e.g., `8.8.8.8` for Google's DNS) in the "Preferred DNS server" field.
    - Enter the alternate DNS server address (e.g., `8.8.4.4` for Google's DNS) in the "Alternate DNS server" field.
5. **Save Changes:**
    - Click "OK" to save the changes.
    - Close the remaining windows.

For macOS:

1. **Open Network Preferences:**
    - Click on the Apple menu and select "System Preferences."
    - Click on "Network."
2. **Select Your Network Connection:**
    - Select your active network connection (e.g., "Wi-Fi" or "Ethernet") from the left sidebar.
    - Click "Advanced."
3. **Enter DNS Server Addresses:**
    - Go to the "DNS" tab.
    - Click the "+" button at the bottom of the DNS Servers list to add a new DNS server.
    - Enter the preferred DNS server address (e.g., `8.8.8.8` for Google's DNS) and press Enter.
    - Repeat for the alternate DNS server address (e.g., `8.8.4.4` for Google's DNS).
4. **Save Changes:**
    - Click "OK" to save the DNS settings.
    - Click "Apply" in the Network preferences window.

For Linux (Ubuntu):

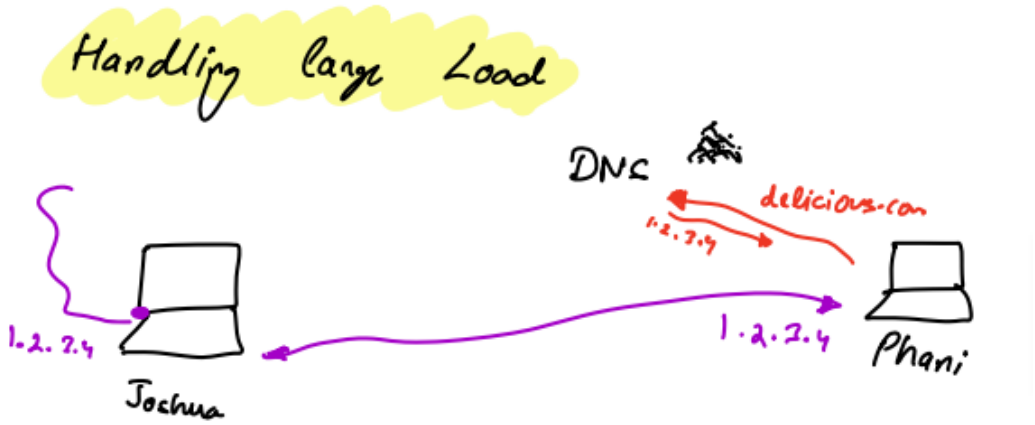
1. **Open Network Settings:**
    - Click on the network icon in the system tray and select "Settings" or "Network Settings."
2. **Select Your Network Connection:**
    - Choose your active network connection (e.g., "Wi-Fi" or "Wired").
3. **Edit DNS Settings:**
    - Click the gear icon next to your active connection to open settings.
    - Go to the "IPv4" tab.
    - Select "Automatic (DHCP)" under "Method" and then select "Additional DNS Servers."
    - Enter the preferred DNS server address (e.g., `8.8.8.8` for Google's DNS).
    - Enter the alternate DNS server address (e.g., `8.8.4.4` for Google's DNS), separated by a comma.
4. **Save Changes:**
    - Click "Apply" to save the changes.

Example DNS Server Addresses:

- **Google Public DNS**: 8.8.8.8 (Preferred), 8.8.4.4 (Alternate)
- **Cloudflare DNS**: 1.1.1.1 (Preferred), 1.0.0.1 (Alternate)
- **OpenDNS**: 208.67.222.222 (Preferred), 208.67.220.220 (Alternate)

By following these steps, you can directly change the DNS settings on your computer to ensure a more reliable and secure internet experience.

Handling Large Load ->



Right now, if someone wants to connect to Joshua's laptop, they send a request to the DNS. The DNS then gives them Joshua's IP address, and they connect to his laptop. However, there are many problems with handling a large load this way:

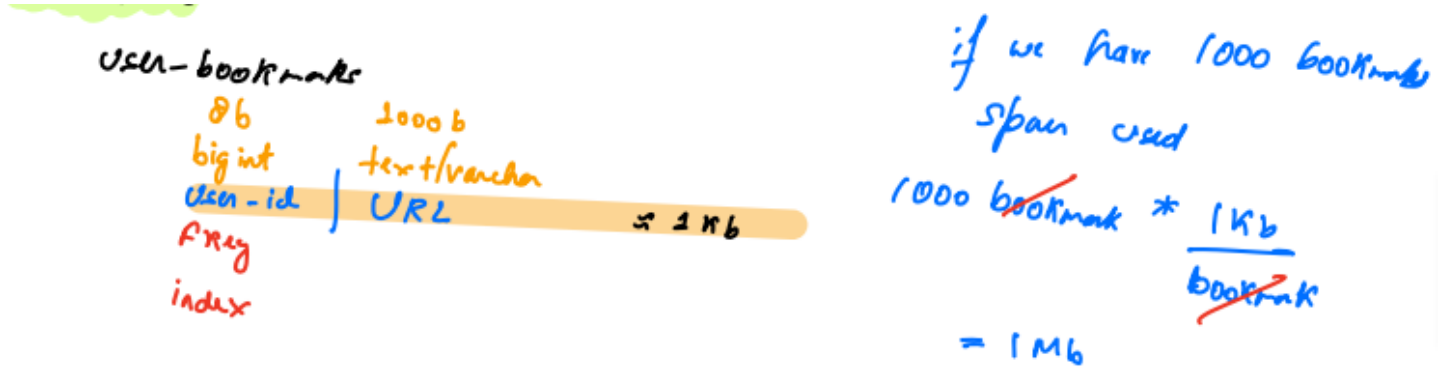
- If Joshua turns off his laptop, the website goes down temporarily.
- If Joshua is playing a game or downloading a video, the website will be slow because both the game and the website need CPU resources and internet bandwidth.

Solution:

- Joshua can set up a dedicated server. A server is any machine on a network or the internet that provides services. It's like a big machine without input or output devices that we can communicate with using SSH. It is only used to serve the website instead of personal use. It has code to run the web server, database, and an operating system like Linux.

DataBase for user-bookmarks

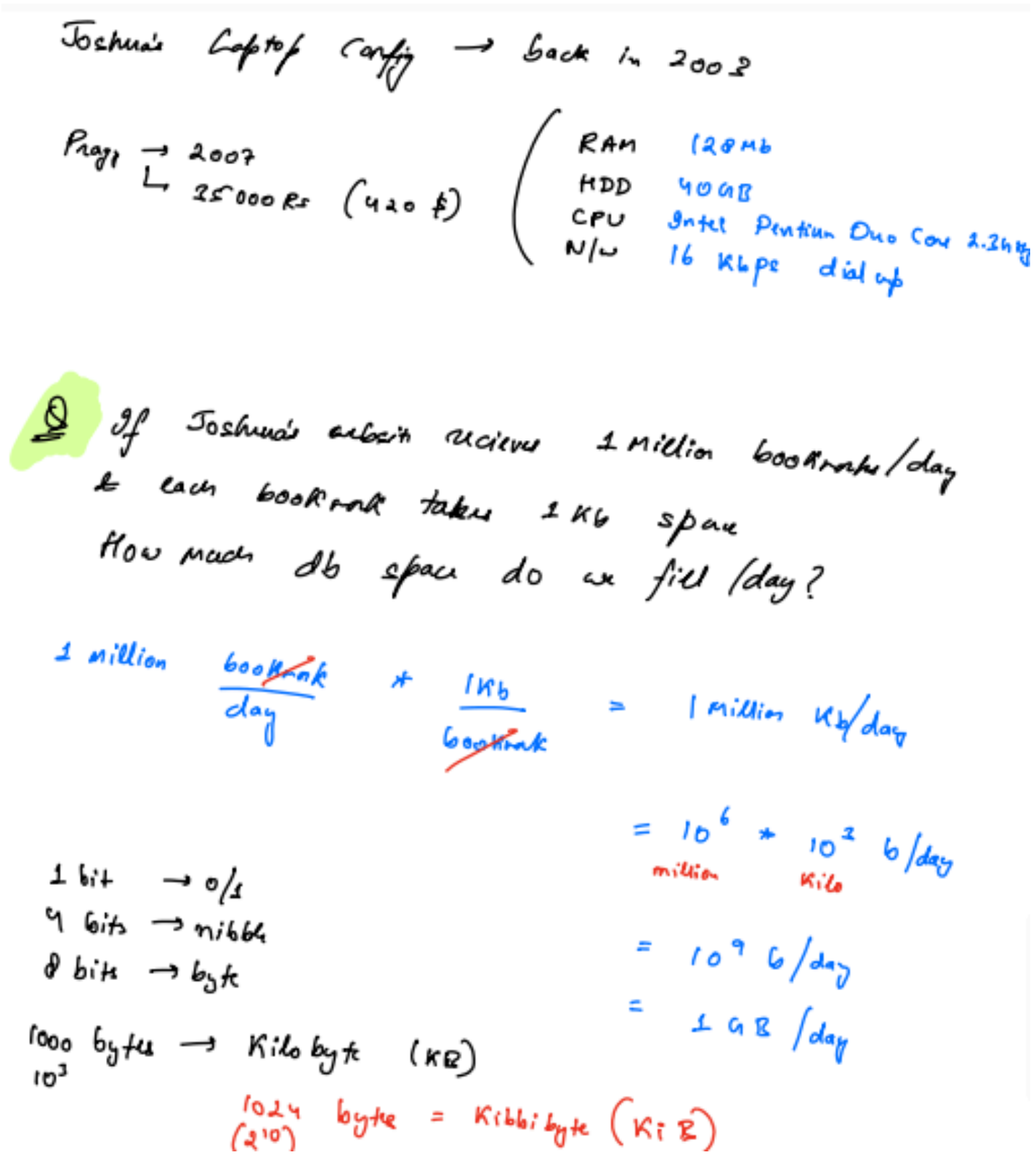
Next, we will work on the database for the user-bookmarks website, considering the MVP (Minimum Viable Product).



We will create a table that keeps track of user IDs and URLs. For user IDs, a big integer is enough, and for URLs, we will use the text data type because URLs can be very large in size. We assume each URL is about 1 KB in size. So, if a user has 1,000 bookmarks, the total size will be around 1 MB.

Required Resources ->

Let's assume Joshua's system is from back in 2003. We will calculate how much space is required if millions of users visit daily and how he can manage the server resources, as shown below.

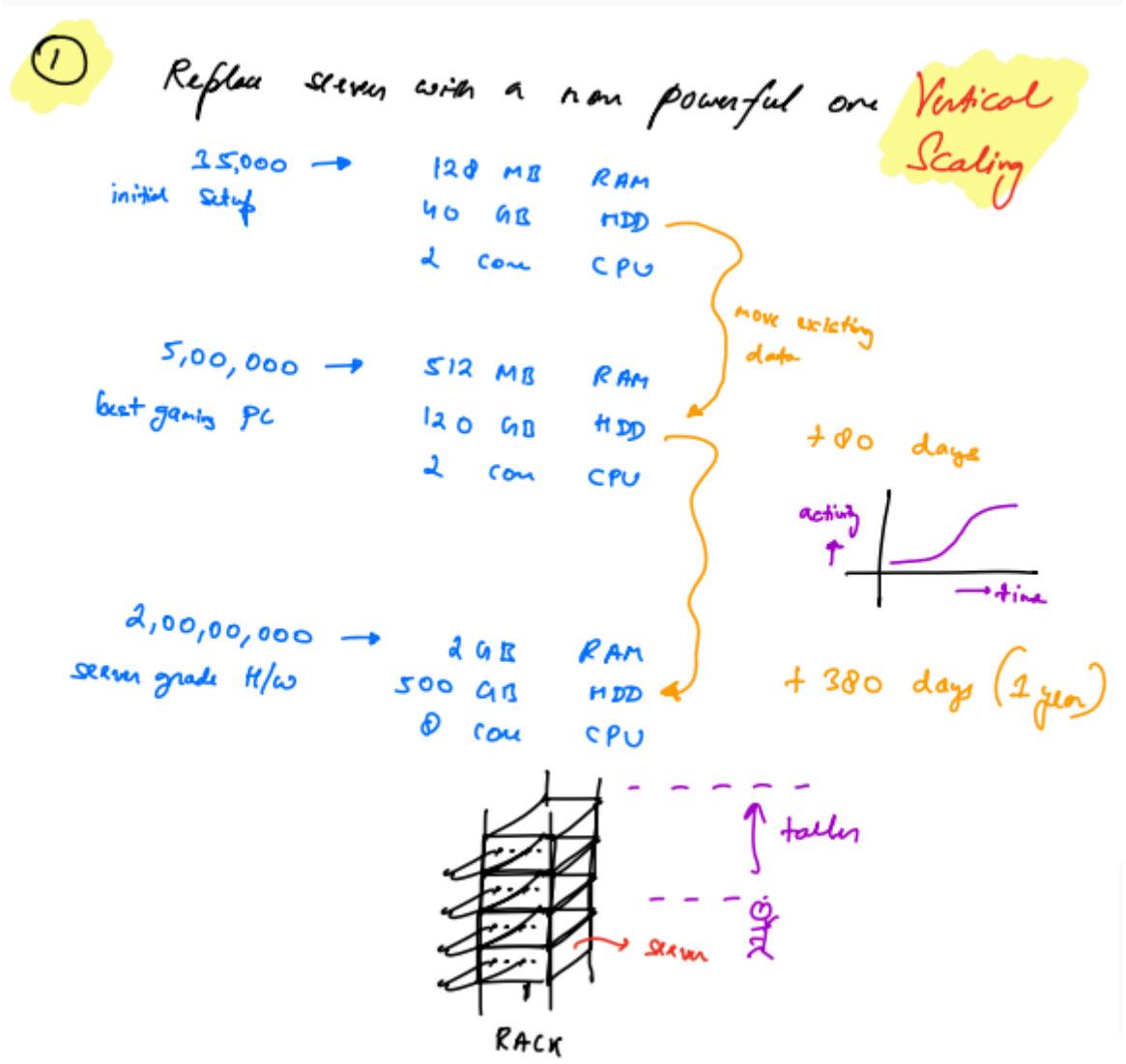


Let's assume the website is very popular now, with 1 million users visiting daily. The space needed to store bookmarks each day is 1 GB. With the current configuration, the disk will be full in 40 days. Then, Joshua will have to replace this machine with a higher-end machine.

Vertical scaling ->

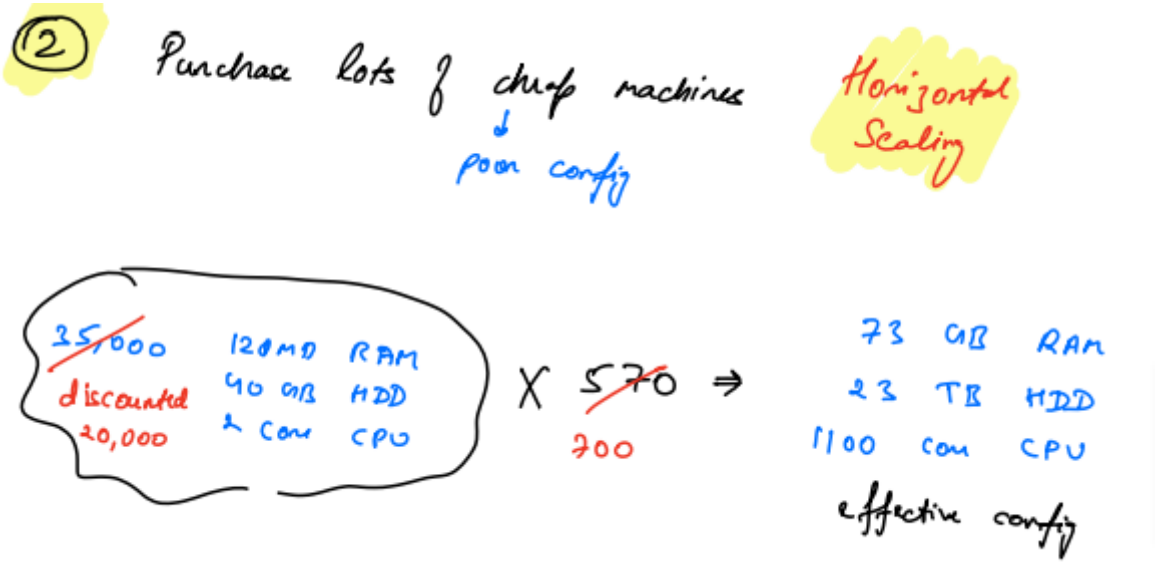
Vertical scaling, also known as "scaling up," means increasing the capacity of a single machine or server to handle more load. This is done by adding more resources, such as CPU, memory, or storage, to the existing machine.

In the context of Joshua's situation, if the website becomes very popular with 1 million users daily, and 1 GB of space is needed each day for bookmarks, vertical scaling would involve upgrading his current machine. This means he would replace his machine with a higher-end one that has more disk space, more memory, and a faster CPU to manage the increased load and storage requirements.



Horizontal scaling -> Horizontal scaling, also known as "scaling out," means adding more machines or servers to handle increased load. Instead of upgrading a single machine, you distribute the workload across multiple machines.

In Joshua's situation, if the website becomes very popular with 1 million users daily, and 1 GB of space is needed each day for bookmarks, horizontal scaling would involve adding more servers to share the load with the same chip-configured machine. Each server would handle a portion of the user requests and store part of the bookmarks. This way, the load is balanced across multiple machines, improving performance and ensuring the system can handle more traffic without running out of resources quickly.



Difference between Vertical and Horizontal scaling

Vertical Scaling:

- Replace the current system with a more powerful one.
- Limited to the technology available today.
- Very simple; just spend more money.
- Required for CPU-bound tasks that need a single CPU, like video processing (rendering video), AI, and ML. For example, ChatGPT-4 requires a 1.7 trillion parameter model, which needs around 7 TB of storage and must be loaded into RAM and on GPUs.

Horizontal Scaling:

- Buy lots of smaller, cheaper systems. It is cost-effective.
- Can add an infinite number of machines as needed.
- Extremely difficult and creates many problems.

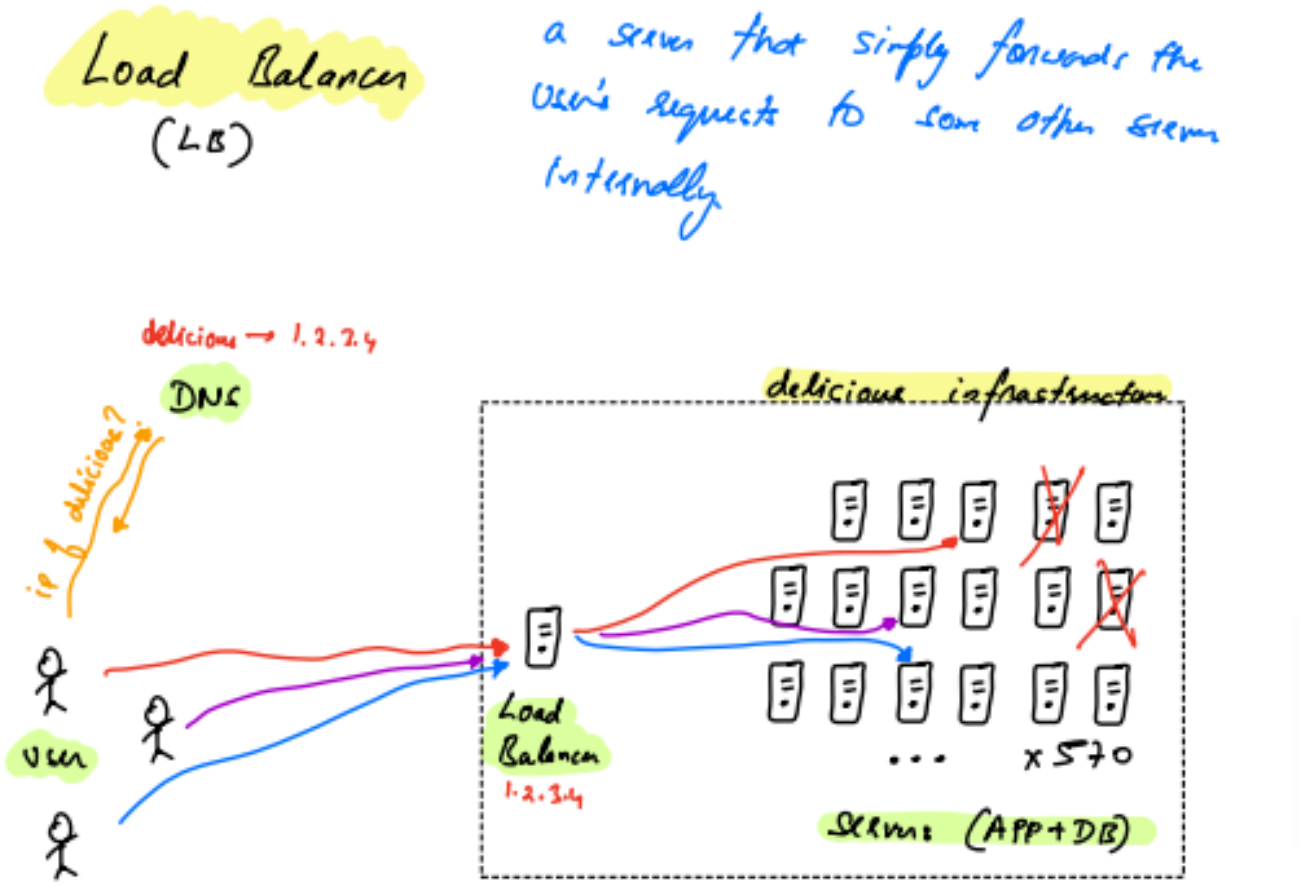
Load balancer -> A load balancer is a device or software that distributes incoming network traffic across multiple servers to ensure no single server becomes overwhelmed. It helps improve the availability and reliability of applications by spreading the load evenly.

Key Points:

- **Distributes Traffic:** Spreads incoming requests across multiple servers.
- **Improves Performance:** Prevents any single server from becoming a bottleneck, enhancing overall performance.
- **Increases Reliability:** If one server fails, the load balancer redirects traffic to other servers, ensuring the application remains available.
- **Scalability:** Makes it easier to add or remove servers as needed without affecting the overall system.

In Joshua's case, if his website becomes popular and he implements horizontal scaling by adding more servers, a load balancer would help manage the traffic efficiently.

Let's assume Delicious uses over 570 cheaper servers, each holding the app and database. When a user connects through DNS, the question arises: which server should the DNS connect to? To provide a unified view of all servers to the user and avoid exposing all systems, we use one server as a load balancer. We add the IP address of this load balancer to the DNS. The user connects to the load balancer via DNS, and the load balancer forwards the request to another server.



Consistent Hashing - June 7

- Revision of last class
- how does the load balancer know the new server is added when checking for health?
- isn't ICAN server then a single point of failure?
- Different domain registrars can have same IP address?
- how loadbalancer know the new server is added when chek for health?
- Does "cluster" mean the group of horizontally scaled machines?
- Can LB use combination of heartbeat + health check both?
- How will DNS know if one of our Load balancers is Down
- So, Each IP address will have multiple machines with one HOT machine, the rest are backup?
- the IP comes from wire , is it public or private IP address ?
- isn't this cause performance issue or like for one request multiple servers are busy so it will increase cost ?
- I'm case of hot rollover , does LB do a heartbeat/healthcheck with just the current active/hot server, or all servers connected to the wire?
- In hot copy, do we need another system to send or identify which is hotcopy?
- now the DNS server will be abottle neck right. Lots of request will hit DNS server right?
- DNS has LB's IP registered against it right?
- Does the load balancer maintain a list of the private IPs of the servers?
- Does load balancer doesn't get overloaded?
- Why are we discussing sharding
- how will the LB know that a user is in which server? will it check in all the servers?
- Ideal value of number of hash functions to use? (value of K i mean)
- When some Server goes down .. how is directed.. hash function
- If the server got down then? how do we handle
- Where does API Gateway comes in picture here?

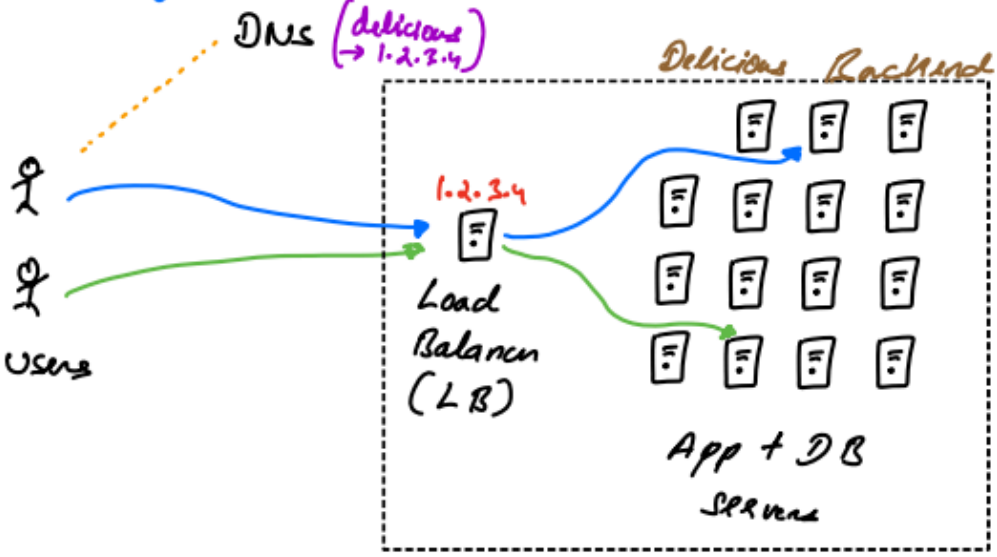
Revision of last class

- what & why of HLD
- Case study - delicious
  - made in 2003
  - bookmarking service
- how are websites made available online
  - ISP
  - domain
  - DNS

Root DNs (7) like o

- Scale issue - run out of resources
  HDD / CPU / NW / RAM
- Scale Vertically / Scale Horizontally
  Scale up / Scale out

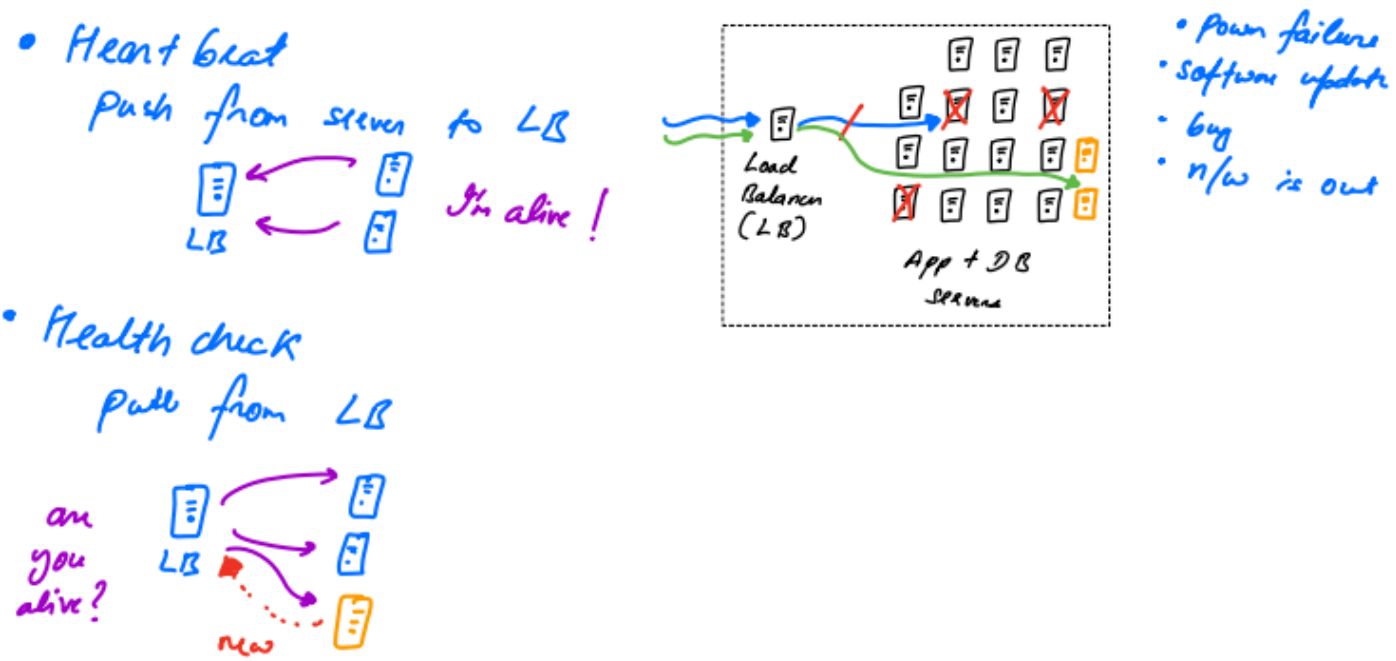
- Need for having a Load Balancer



① provide a unified view of the system to
- the end user
- to the DNS

② to distribute the load evenly across all servers (no server should get overwhelmed)

how does the load balancer know the new server is added when checking for health?



The load balancer knows what servers are available in the system through a process called health checking and Heartbeat. This involves regularly monitoring the status and performance of each server in the server pool.

Key Points:

- **Health Checks:** The load balancer sends periodic requests to each server to check if it's responding properly.
- **Response Evaluation:** It evaluates the responses received from each server to determine if it's healthy and capable of handling requests.
- **Status Updates:** Based on the health check results, the load balancer maintains an updated list of available servers.
- **Dynamic Adjustment:** If a server fails or becomes overloaded, the load balancer removes it from the pool of available servers until it becomes healthy again.
- **Scalability:** As servers are added or removed from the system, the load balancer dynamically adjusts its list of available servers.

Heartbeat:

Heartbeat is a mechanism used in computing to monitor the status and availability of servers within a system. It involves servers sending regular signals, known as heartbeats, to a central monitoring system. These signals indicate that the servers are operational and functioning as expected. By monitoring the receipt of these heartbeats, administrators can ensure the continuous operation of their systems. If a server fails to send a heartbeat within the expected timeframe, it may indicate a problem with the server, such as a failure or outage, prompting further investigation and remedial action.

Health Check:

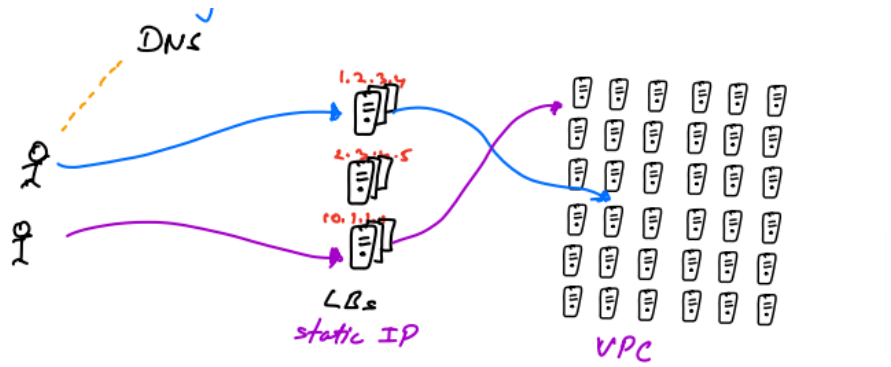
Health checks are procedures performed by load balancers to assess the status and performance of servers in a distributed system. These checks involve sending requests to individual servers and evaluating their responses to determine their health and availability. The criteria for evaluating server health typically include factors such as response time, server errors, and other performance indicators. If a server fails a health check, it is temporarily removed from the pool of available servers, ensuring that incoming traffic is only directed to healthy and operational servers. Health checks play a crucial role in maintaining the reliability and stability of distributed systems by ensuring that resources are efficiently utilized and potential issues are promptly addressed.

Isn't the Loadbalancer itself a bottleneck or a single-point failure?

1. **Load balancers don't perform complex tasks like computing or database queries**.
   If we compare app servers and load balancers, app servers have to do a lot of work: they deserialize the request, handle authorization, fetch data, process the data, generate a response, and then serialize the response. They have to repeat this process for every single request. On the other hand, load balancers don't need to deserialize requests because everything they need comes in the header. They use very fast routing algorithms to find the right server immediately. While a typical app server might handle 1,000 requests per second, a load balancer with the same machine can handle 100,000 requests per second. **Still with 5 billion users Load Balancer is the Bottlenack!!**

2. **Horizontally Scaling the Load Balancer:** We can horizontally scale load balancers by adding more of them. To manage multiple load balancers, we don't need another load balancer. Instead, we use DNS to manage them. DNS can store multiple IP addresses for a single domain, acting as a load balancer for the load balancers.

   For example, if Delicious's website has multiple IP addresses like [1.2.3.4, 2.3.4.5, 10.1.1.1], the DNS can store all these addresses. A typical DNS might return a random IP address from the list or the entire list, allowing the client's browser to choose a random one. Geo DNS can also be used, picking the IP address geographically closest to the user.



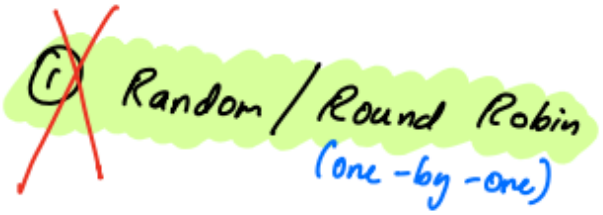
3. **Hot copy rollover mechanism**

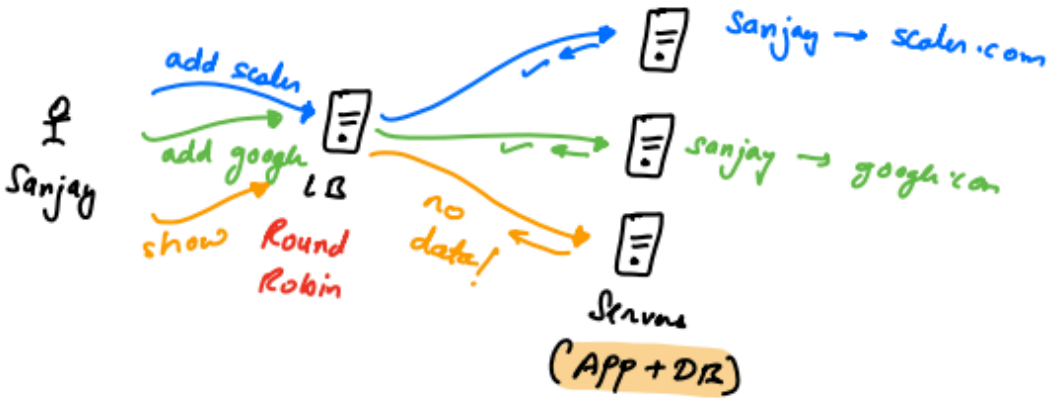
The hot copy rollover mechanism is a strategy used to ensure data availability and reliability by maintaining a constantly updated duplicate (hot copy) of critical data. When the primary data source becomes unavailable due to failure or maintenance, the system can instantly switch over to the hot copy without significant downtime.
This means we have multiple copies of every load balancer, and we connect all machines with a single IP address because an IP belongs to the network, not to a single machine. If we take one main cable and split it into multiple cables, all those cables will share the same IP address and connect to multiple systems. If something goes wrong with the primary system, a backup system (hot system) becomes active immediately without any downtime.

Note: Request and Response are mostly (exception - websocket connections) routed via Load Balancer

Routing Algorithm: It decides which server each request should be forwarded to.

Random / Round Robin (one by one):





Let's consider using a Round Robin approach with one load balancer and three servers. The load balancer sends requests to the servers one by one. For example, if

Sanjay bookmarks "Scaler," the request goes to the first server, and it acknowledges it. If Sanjay then bookmarks "Google," the request goes to the second server. But when Sanjay tries to retrieve his bookmarks, the request might go to the third server, which doesn't have his data, resulting in a "no data" response. This creates a very bad user experience.

Round Robin doesn't work well here because the data is stateful, and we need to ensure that each user is routed to the same server that holds their data. **The user should always be redirected to the server that holds their data to maintain consistency.**

Sharding: First, we need to determine how to distribute the data. We will do this using sharding, which involves splitting the data across multiple machines.

Sharding is a method of splitting and storing data across multiple databases or servers. It helps distribute the load and improve performance by dividing a large dataset into smaller, more manageable pieces called "shards."

Here's how sharding works:

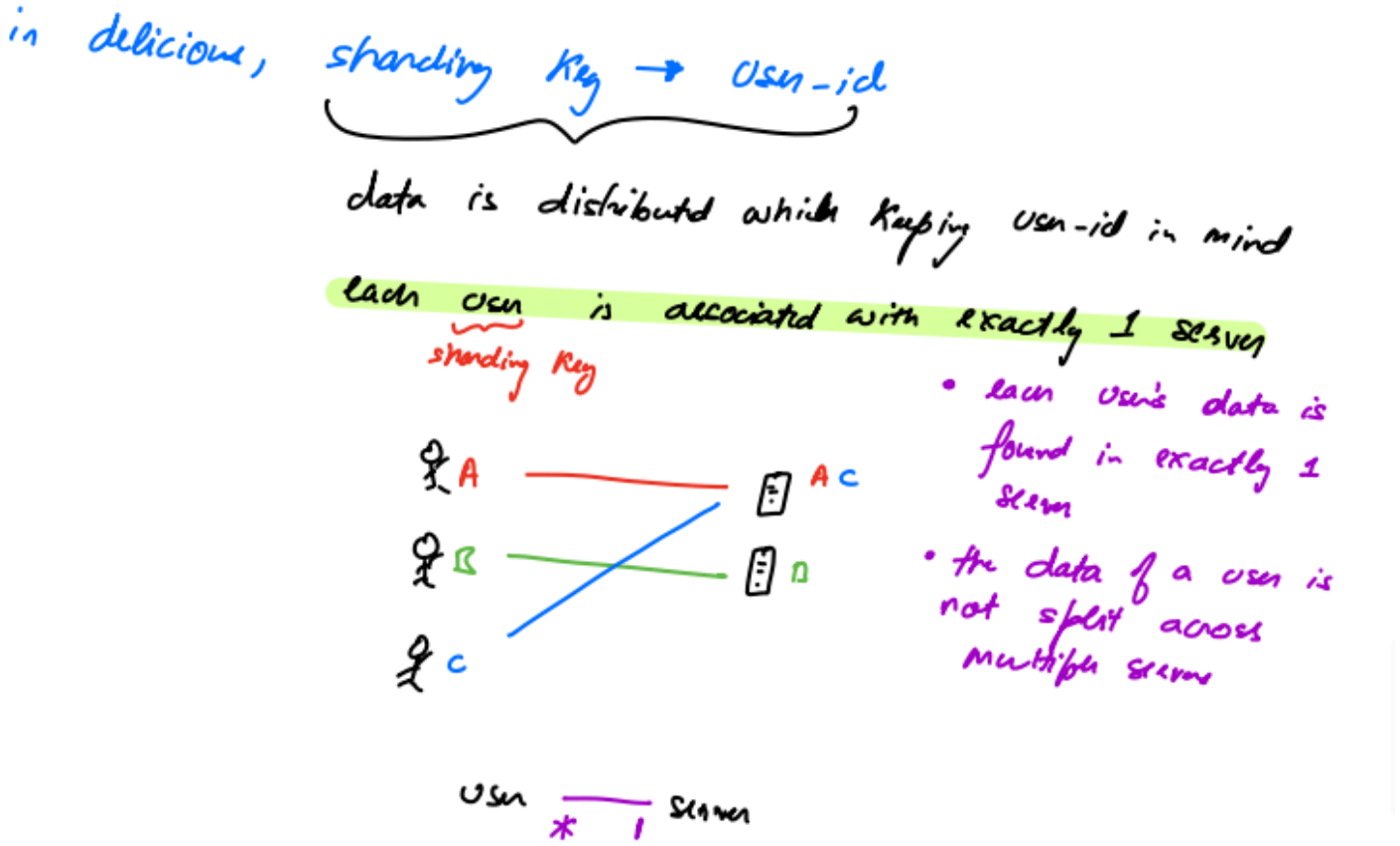
1. Data Partitioning: The data is divided into shards based on a specific key, such as user ID or a range of values. Each shard holds a subset of the total data.
2. Distribution: Each shard is stored on a different server or database, which helps balance the load and prevents any single server from being overwhelmed.
3. Routing Requests: When a request is made, a routing mechanism determines which shard contains the needed data and directs the request to the appropriate server.

Example Scenario: Consider a user-bookmarks website. Instead of storing all bookmarks in one database, we can shard the data based on user IDs:

- Users with IDs 1-1000 are stored in Shard 1 on Server 1.
- Users with IDs 1001-2000 are stored in Shard 2 on Server 2.
- Users with IDs 2001-3000 are stored in Shard 3 on Server 3.

If a user with ID 1500 wants to add or retrieve a bookmark, the system routes the request to Shard 2 on Server 2. This ensures requests are handled efficiently and that the load is evenly distributed.

Sharding always follows certain criteria, like a sharding key. **In the case of Delicious**, the sharding key is user_id. Data is distributed with user_id in mind, ensuring that each user is associated with exactly one server. While the same server can handle multiple users, each user is uniquely associated with only one server. **if we have multiple sharding key then 1 server associate with one sharding key.**



If user1's data is on server1 (determined by sharding using the sharding key), then which server should handle user1's request?

The sharding key determines two important aspects:

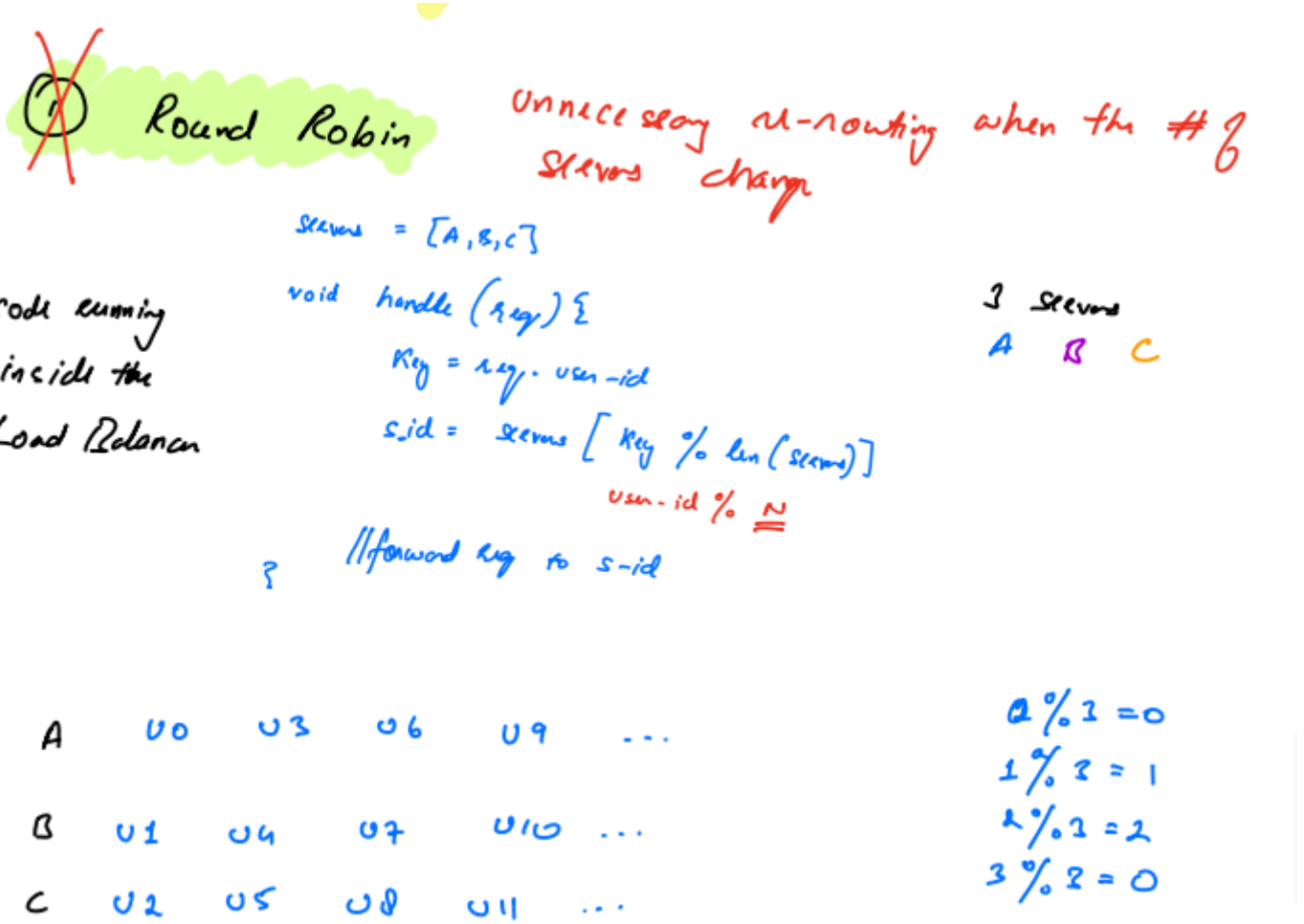
1. How data is distributed.
2. How routing should be handled.

Sharding techniques ->

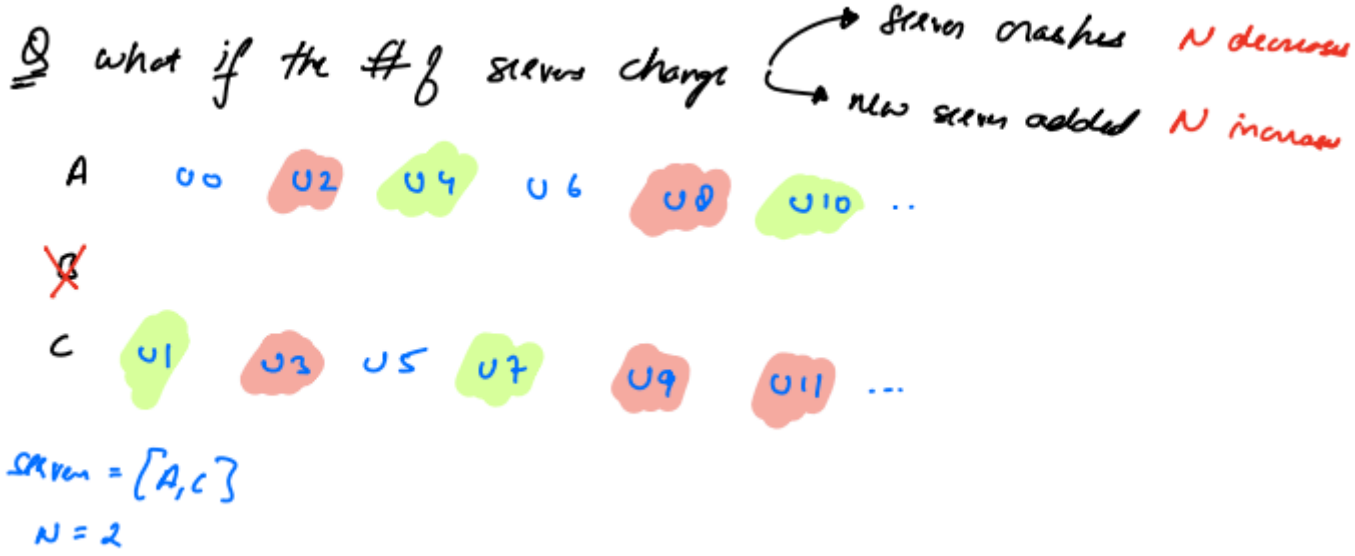
Sharding techniques refer to the various methods used to partition and distribute data across multiple servers or databases in a sharded system. These techniques ensure efficient data management and scalability.

Round Robin ->

Let's assume we have some code written in a Load Balancer. We have 3 servers, which we initialize as an array of servers: A, B, and C. We also have one method, handle, which returns void and accepts a request. From the request, we get the user_id as a key. To get the server_id, we take the modulus of user_id using the number of servers and forward the request to the resulting server_id. Every time, it will forward the request to the corresponding server based on the modulus result.



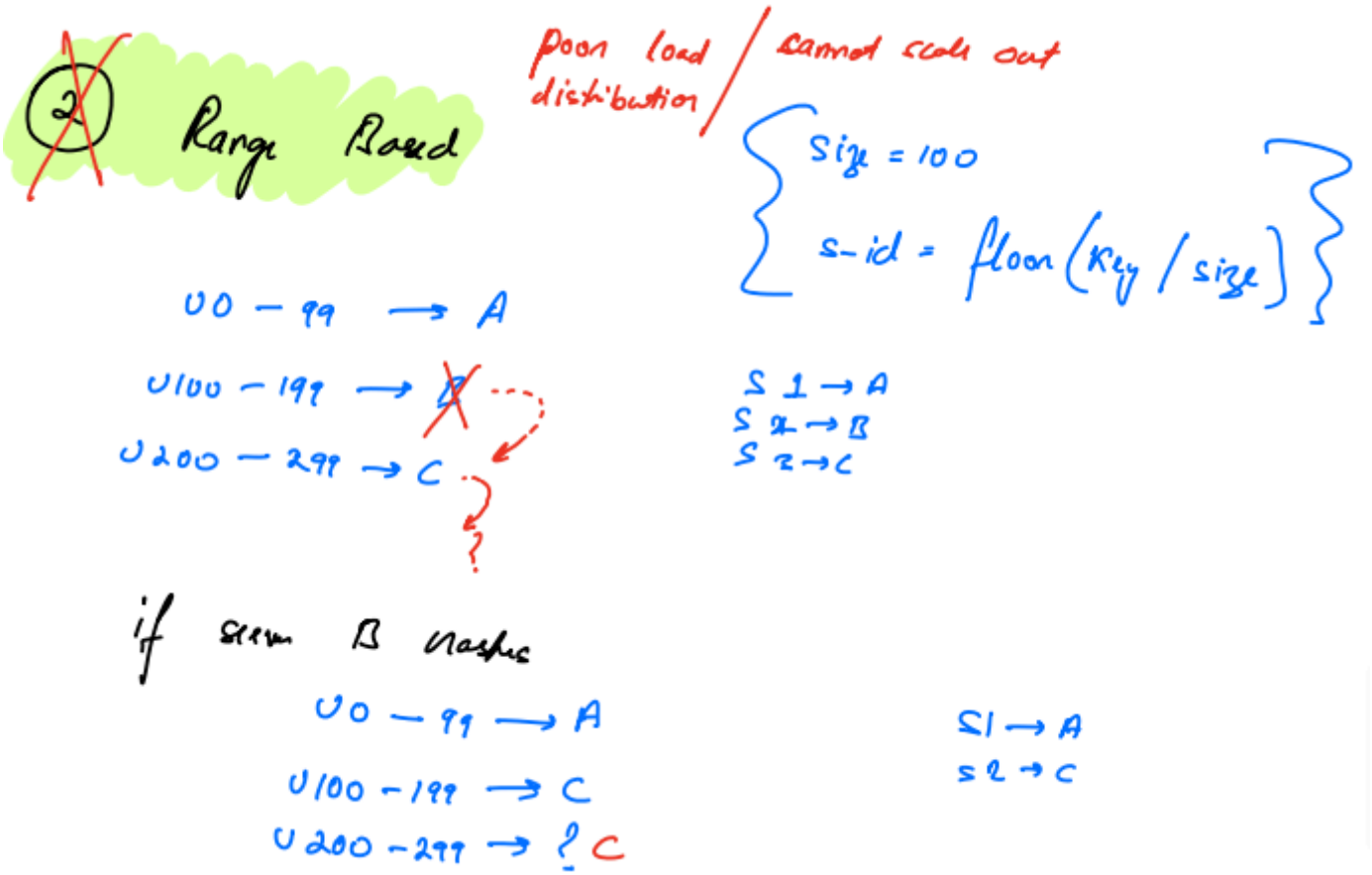
It seems to be working perfectly fine, but if the number of servers changes—due to a server crash or the addition of a new server—the modulus value will change accordingly. Consequently, the routing won't occur as expected, potentially redirecting the user to a different server where their data might not be available.



Range-based Approach:

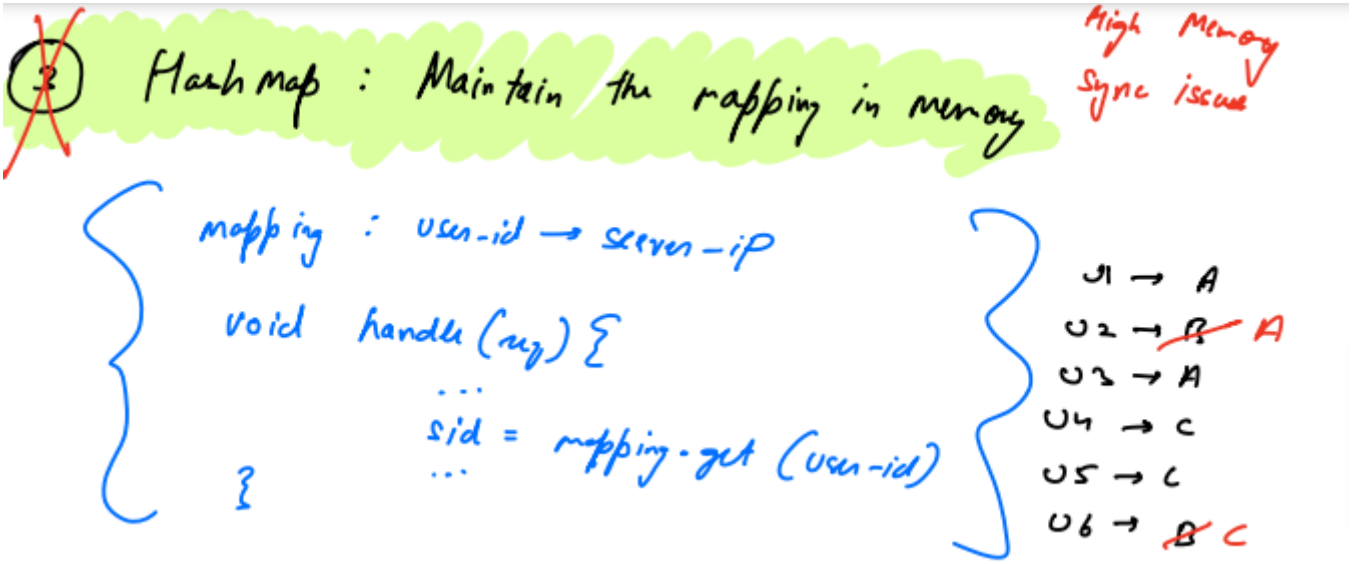
In a range-based approach, let's consider having 3 servers: A, B, and C, and a user set size of 100. Users 0 to 99 will be stored on server A, users 100 to 199 on server B, and users 200 to 299 on server C.

However, if server B fails, users originally directed to it will be redirected to server C, causing a cascading failure. Server C may become overloaded due to the additional traffic, leading to its failure as well. Moreover, when a new server is added, it won't share the load of existing users because it falls outside their assigned ranges. It must wait for new user signups to distribute its load. This poor load distribution hampers scalability for large user sets since adding new servers doesn't alleviate the load on existing users.



Hashmap Approach:

Maintain the mapping of users to servers in memory. If a server crashes, iterate through the map, find all users mapped to the crashed server, and assign each user to a new random server. When a new server is added, update the mapping for some users to direct them to the new server. **Due to these issues, this solution is not practical.**



However, this approach has several issues:

1. **High Memory Requirement:** For example, with an 8-byte user ID and a 4-byte server ID for each record, each entry would require 12 bytes. For 5 billion users, this would amount to around 60 GB of data, which is impractical to store in memory. Storing this data on an HDD isn't viable either, as HDDs are millions of times slower than RAM, which would severely slow down routing.
2. **Synchronization Across Multiple Load Balancers:** If multiple load balancers are used, each needs to maintain and synchronize its mapping. Keeping these mappings in sync is extremely difficult and almost impossible to achieve reliably.

Consistent Hashing:

Consistent hashing is an excellent algorithm that provides the following benefits:

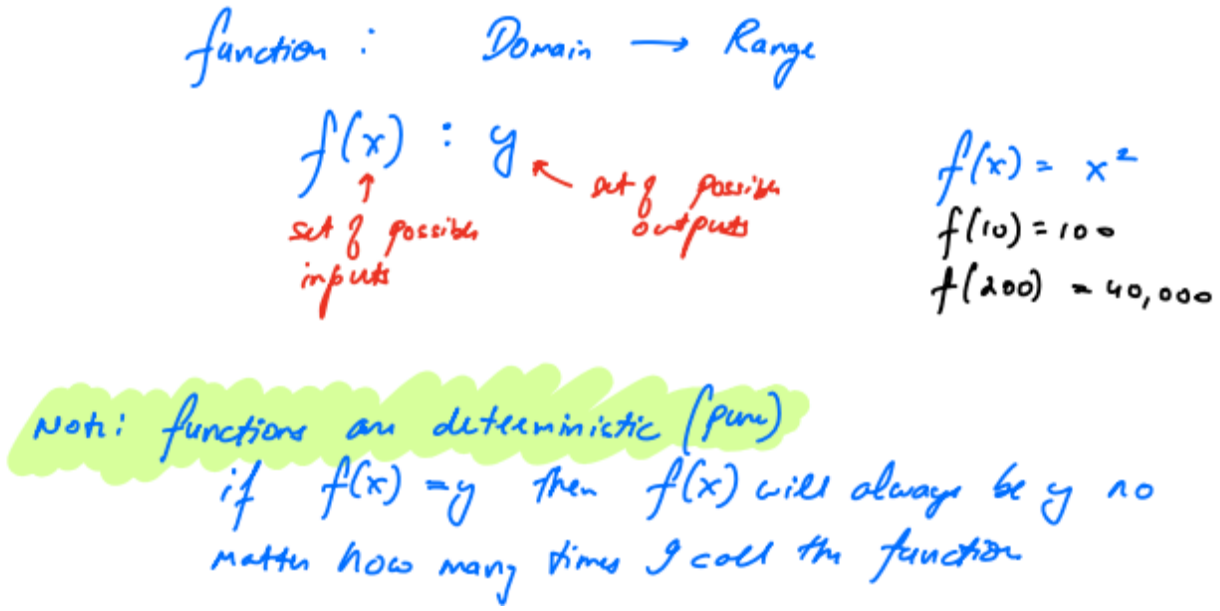
- Servers can be added or removed freely.
- Load distribution is always even.
- If a server crashes, its load is distributed equally across the remaining servers, preventing cascading failures.
- When a new server is added, it takes an equal share of the load from each existing server.
- No high memory requirement.
- No need for synchronization across load balancers.
- The algorithm is very fast, with almost O(log N) complexity.

Hash function:

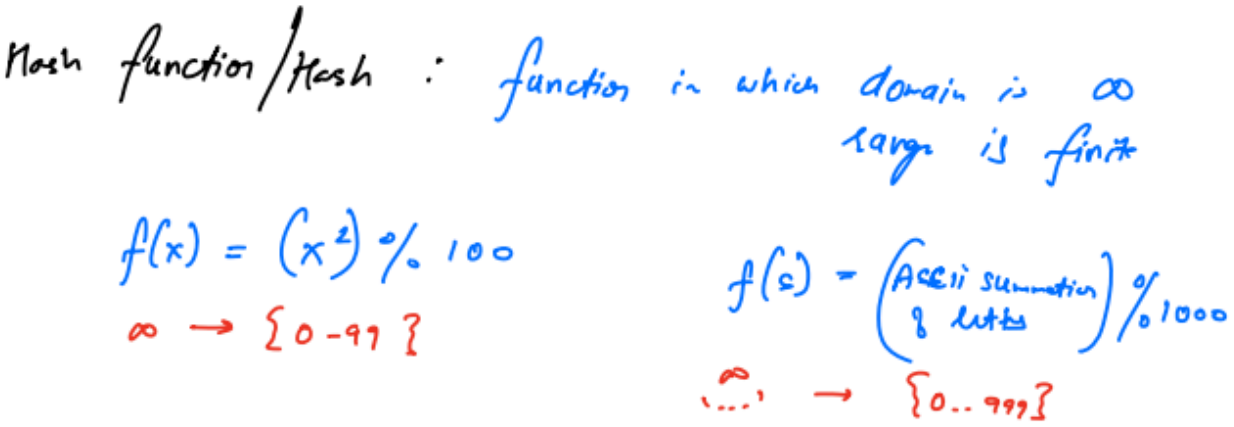
A hash function is a special kind of computer function that takes some data (like a word, a file, or a number) and turns it into a fixed-size string of characters, usually a mix of letters and numbers. This output is called a hash value or hash code.

For example, if you put a word like "apple" into a hash function, it might turn it into a string like "1f3870be274f6c49b3e31a0c6728957f".

Hash functions are used in many areas of computer science, like storing passwords securely, quickly finding data in large databases, and ensuring data has not been tampered with.



Note: Functions are always pure in programming, meaning they do not change anything outside themselves. If they do change something, then they are called procedures.



Family of hash functions

↳ existing algo for generating multiple 'good' hash functions distribute inputs randomly

Consistent Hashing

(16, 32, 64)

- K hash functions for server mapping
  $h_1(s\text{-}id) \to v_1$ $h_2(s\text{-}id) \to v_2$ ... $h_K(\cdot)$
- 1 hash function for user
  $h(u\text{-}id) \to v$

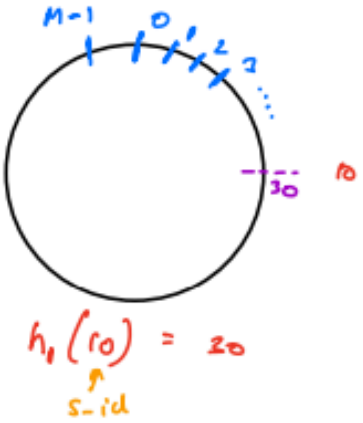
} pre-config in each LB in code

- each of these (K+1) hash functions will have the same output range (0 ... M−1)

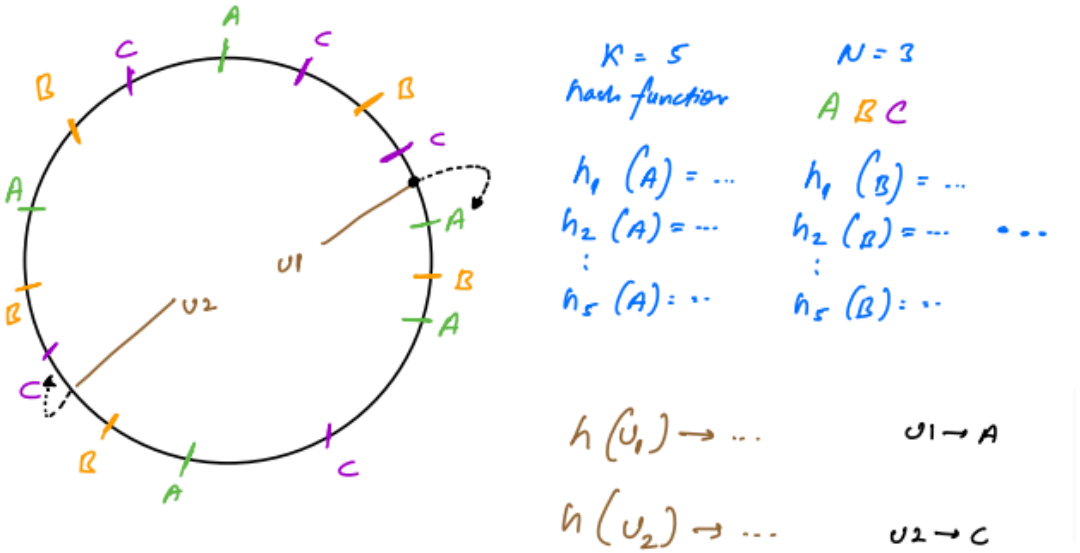
  $h_i(\ldots) \to (\ldots) \% M$

  M is typically $(2^{64}-1)$ & 10^{19}

Visualize this range as a ring



- each server is assigned K virtual spots on the ring (one for each hash)
- each user is assigned 1 spot on the ring (using the user's hash func)
- routing / shading → going clockwise on the ring till you hit the first server.



let's break it down step by step using consistent hashing for load balancing with 3 servers, 5 virtual nodes per server (k = 5), and 1 hash function for users.

Step-by-Step Explanation

1. **Set Up the Ring with Virtual Nodes**:
   - We have 3 physical servers (let's call them Server A, Server B, and Server C).
   - Each server is represented by 5 virtual nodes to ensure an even distribution of data. These virtual nodes are placed on the ring based on their hash values.
   - Let's assume the hash function maps the virtual nodes to positions like this:
     - Server A: Virtual Nodes at 10, 20, 30, 40, 50
     - Server B: Virtual Nodes at 60, 70, 80, 90, 100
     - Server C: Virtual Nodes at 110, 120, 130, 140, 150
2. **Hashing Users**:
   - Each user is hashed using the hash function, which places them somewhere on the ring. For example:
     - User 1: 25
     - User 2: 65
     - User 3: 115
3. **Assigning Users to Virtual Nodes**:
   - To find out which virtual node a user should go to, start from the user's position on the ring and move clockwise until you find the first virtual node.
   - Based on our example:
     - User 1 (25) will be assigned to the virtual node at 30 (Server A).
     - User 2 (65) will be assigned to the virtual node at 70 (Server B).
     - User 3 (115) will be assigned to the virtual node at 120 (Server C).
4. **Mapping Virtual Nodes to Physical Servers**:
   - After assigning users to virtual nodes, we map these virtual nodes back to their corresponding physical servers.
   - The resulting assignments are:
     - User 1 goes to Server A.

    - User 2 goes to Server B.
    - User 3 goes to Server C.

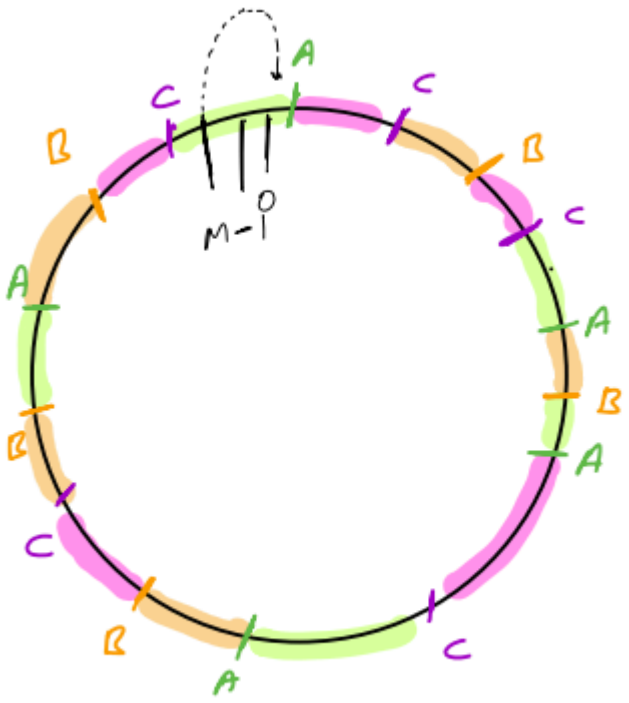
Adding a New Server

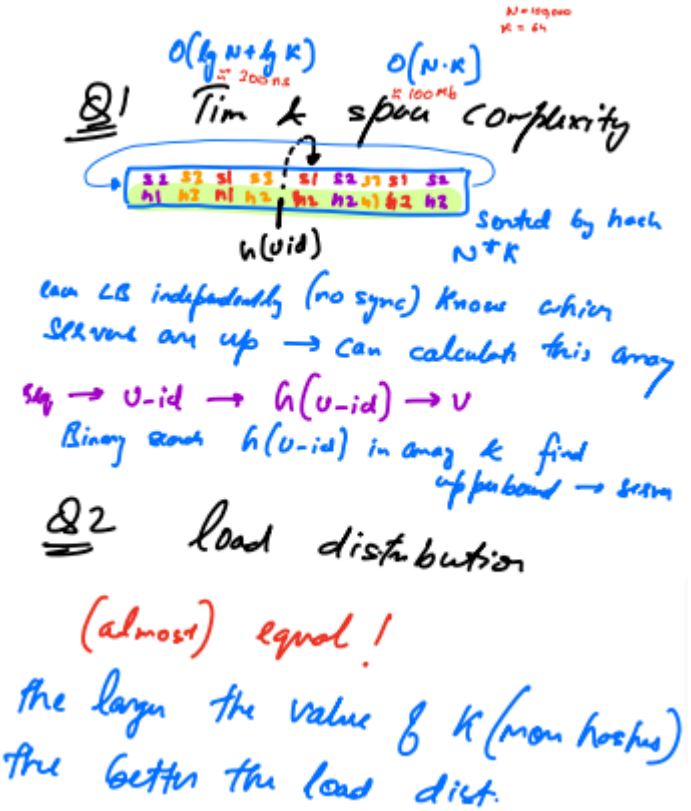
- When a new server (Server D) is added with its virtual nodes, it gets placed on the ring based on their hash values.
- Suppose Server D has virtual nodes at positions 35, 75, 115, 155, 175.
- Users originally assigned to virtual nodes that now fall under Server D's positions are reassigned to Server D.
    - For example, User 3 (115) now maps to Server D instead of Server C.

Removing a Server

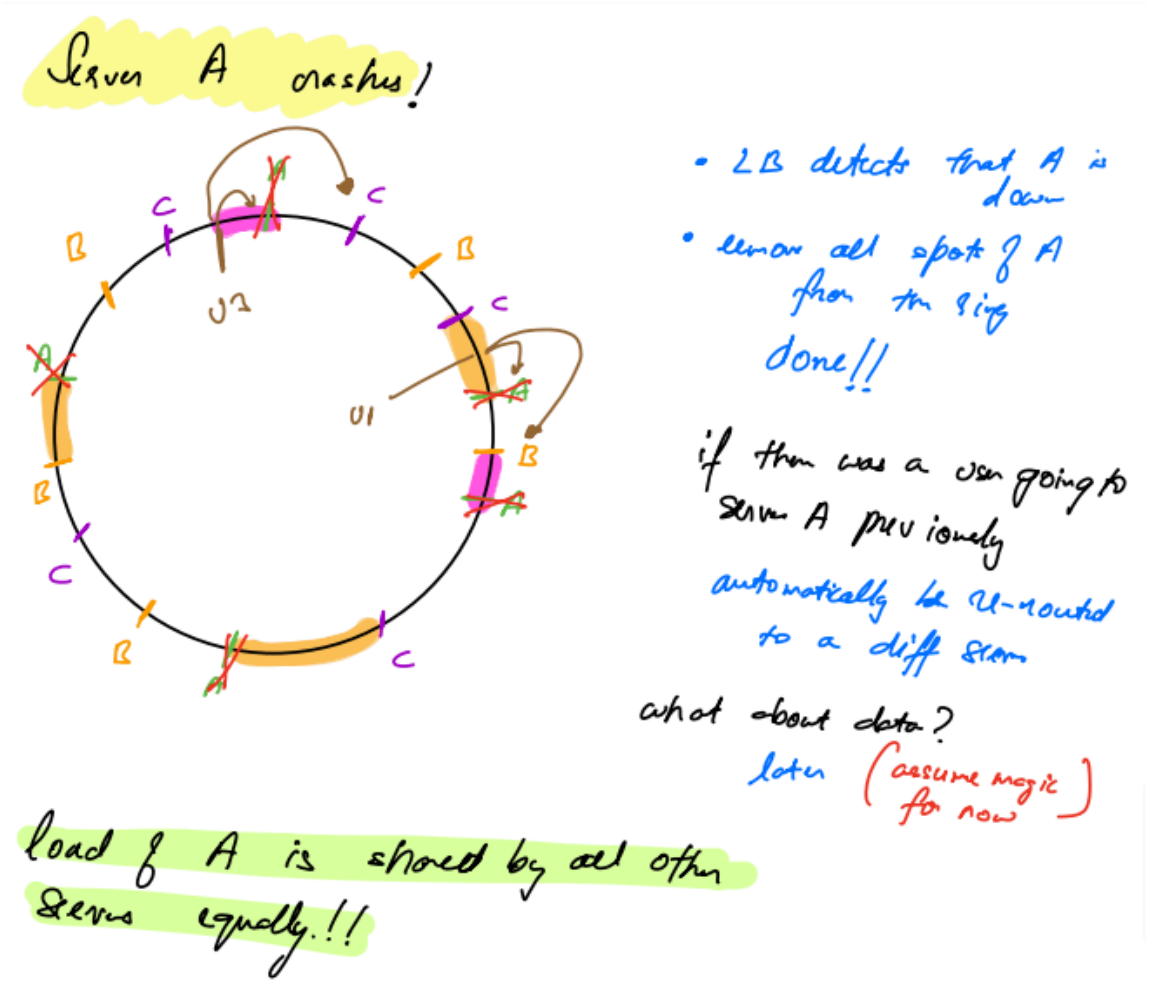
- When a server (e.g., Server B) is removed, its virtual nodes are taken off the ring.
- Users assigned to those virtual nodes are reassigned to the next available virtual node in the clockwise direction.
    - For example, User 2 (65) would move to the next available virtual node at 80 (Server B's other virtual node).

Constance hashing algorithm -> The consistent hashing algorithm does not use a real ring. Instead, it saves the hash values in a hashmap. When a value needs to be found, the algorithm uses the hash of the given value to look it up in the hashmap and retrieve the corresponding value.



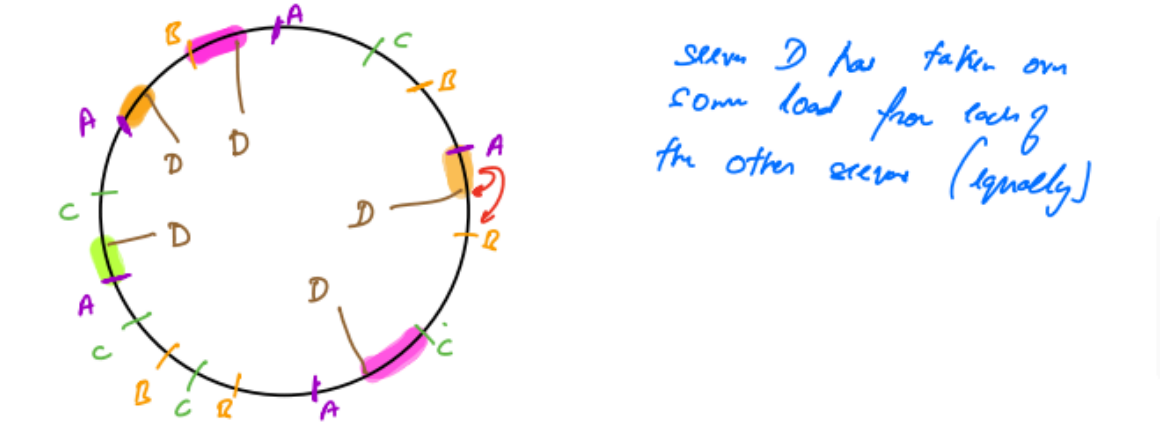


If a server crashes, the consistent hashing algorithm maintains load balancing by redistributing the data from the crashed server's virtual nodes to the next available virtual nodes in a clockwise direction. This ensures that only a small portion of the data is moved, minimizing disruption and maintaining an even distribution of the load across the remaining servers. Illustrated below.

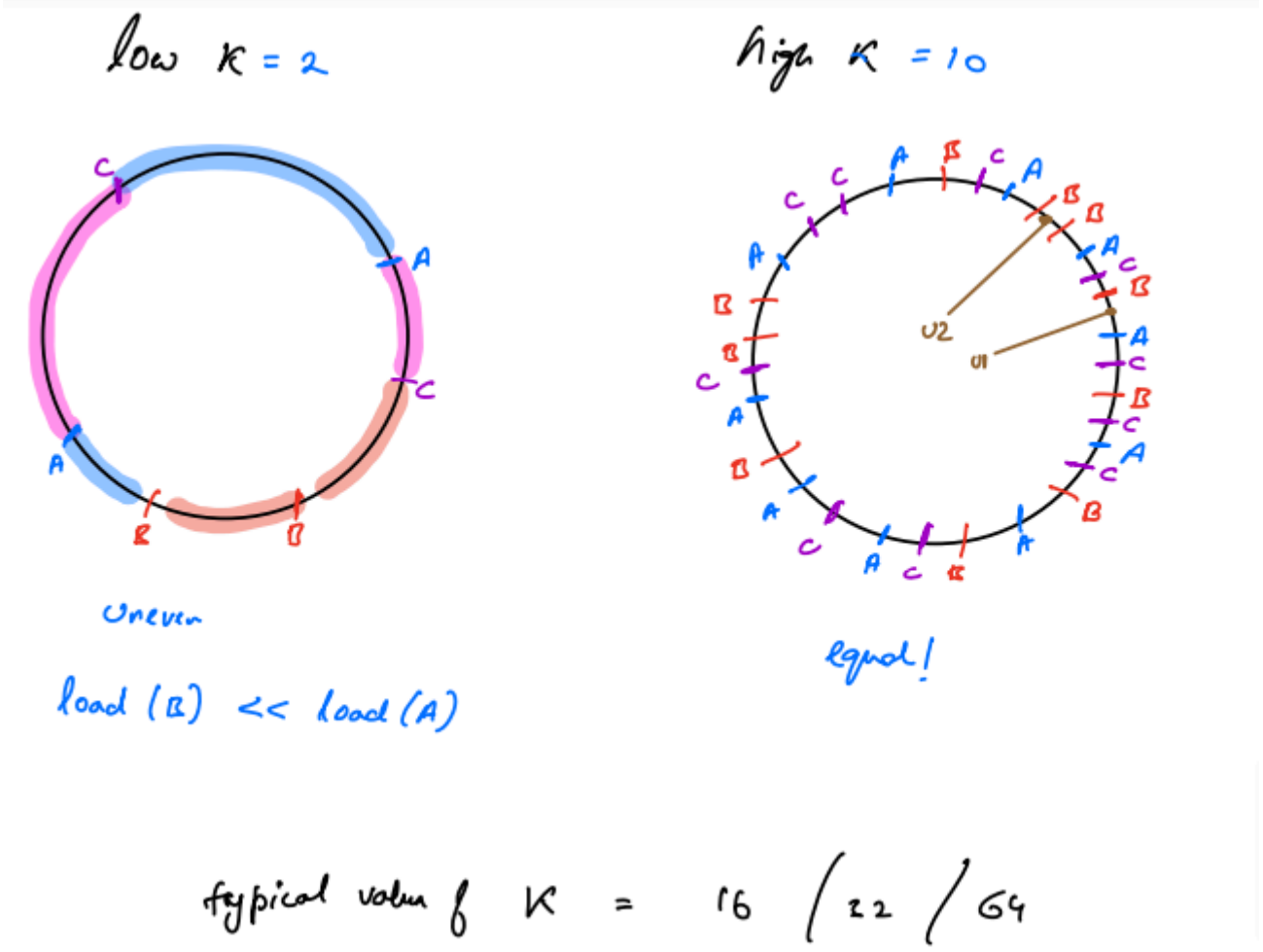


New Server D has been added

When a new server is added, the consistent hashing algorithm balances the load by placing the new server's virtual nodes on the hash ring. This causes some data that was originally assigned to the virtual nodes of existing servers to be reassigned to the new server's virtual nodes. Only the data corresponding to these virtual nodes is moved, ensuring minimal disruption and maintaining a balanced distribution of the load across all servers.



A higher number of virtual nodes (high k) leads to a more evenly distributed load compared to a lower number of virtual nodes (low k). This is because having more virtual nodes allows for finer granularity in distributing data, reducing the chances of any single server becoming a bottleneck.



Caching- June 10

- **Separating Application & Database**
- **Caching**
- **Caching Layers for Backend Infrastructure**
- **Content Delivery Network (CDN)**
- **What is Anycast and Why Anycast**
- **Backend Caching**
- **- Local Caching vs Global Caching**
- **- Single vs. Distributed Cache**
- **Issues with Cache**
- **Eviction Policies**
- **Immediate Consistency / Strong Consistency**
- **Eventual Consistency**

Separating Application & Database

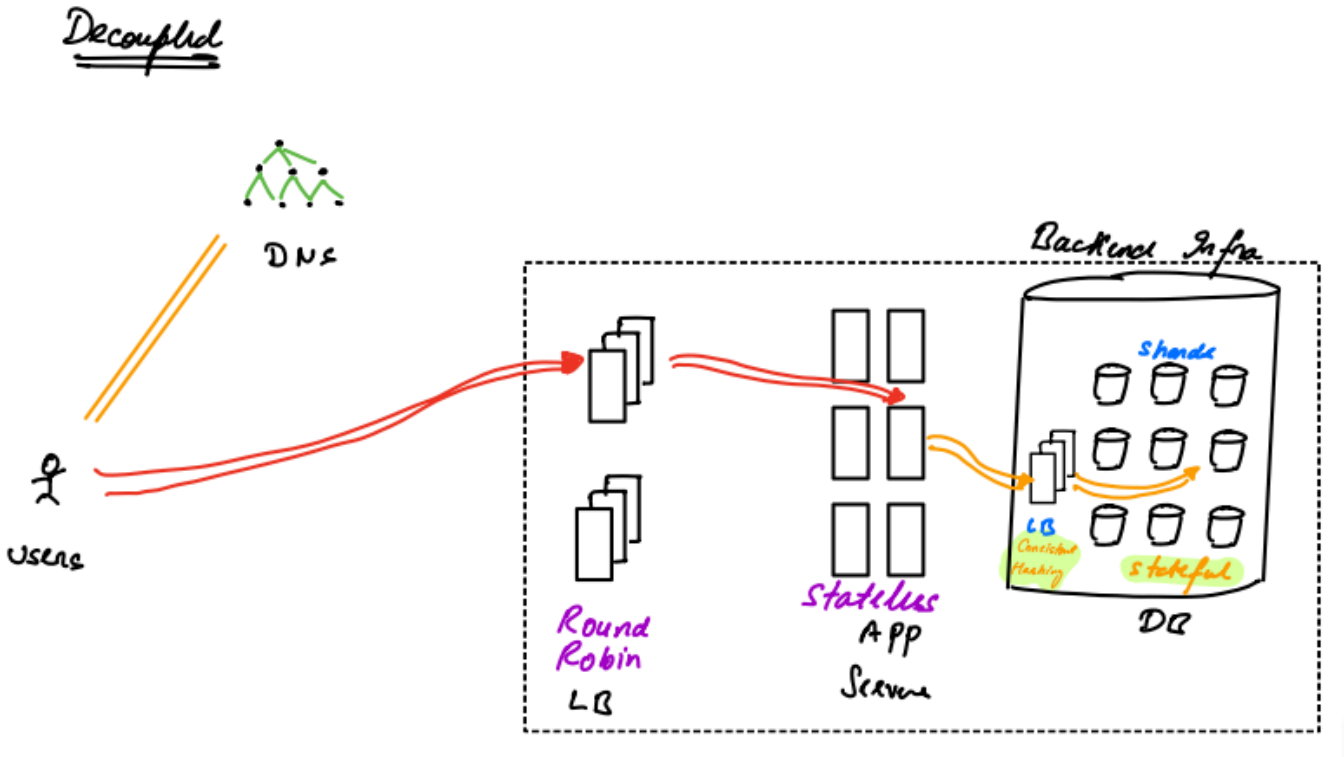
Right now, the app and the database are closely linked, causing several problems when we try to scale the system.

Cannot Scale Independently: If our database grows, it's very hard to scale it separately. We have to scale the entire server with the application as well, which requires many extra resources. Even if the app doesn't need that many resources, they still get used, leading to wastage.

App and DB Compete for Resources: The app and the database fight for resources like CPU, HDD, RAM, and network. When the database gets many requests, the app slows down. If the app gets many requests, the database slows down.

In the illustration below, we show that previously, the app and the database were together. To solve the issues, we separated them and created another layer for the database. We extracted all database data from the servers and used one load balancer. This load balancer uses a Consistent Hashing algorithm to distribute requests to the database.

We also changed the Consistent Hashing algorithm from the application server load balancer to Round Robin. Before, each server had its own database, making them stateful. state can be anything like DB, Cecheing, data. Now that we've extracted that logic, we use Round Robin for load balancing.



All abstractions are leaky - -> If we need massive scale or very high performance, we will eventually need to break the abstraction. This means that for large-scale operations, we might have to create our own system instead of using existing abstracted systems like EBS.

Caching

==========

Latency numbers that every programmer should know - It is vary old but still useful
https://gist.github.com/jboner/2841832

Need for Caching: Memory is always hierarchical, similar to how our brain works. Our brain has:

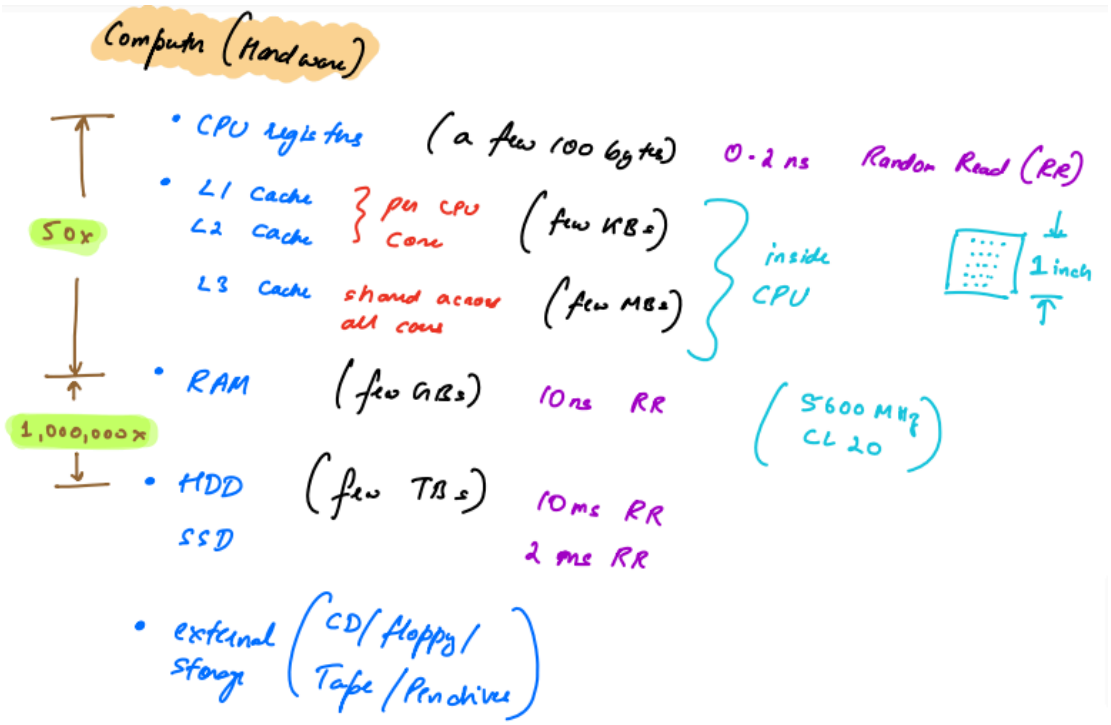
- **Working Memory**: Can store 4-7 items, very fast.
- **Short-term Memory**: Can store a few thousand items.
- **Long-term Memory**: Can store an infinite number of items, very slow.

When we work, we use our fast working memory. Data passes from long-term memory to short-term memory, then to working memory.

In Computers (Hardware): Memory also follows a hierarchy:

- **Registers**: Very fast, random read speed of 0.2 nanoseconds, capacity of a few hundred bytes.
- **L1, L2 Cache**: Each CPU core has its own L1 and L2 cache, slightly slower, capacity of a few KB.
- **L3 Cache**: Shared across all cores, capacity of a few MBs, located inside the CPU processor.
- **RAM**: Capacity of a few GBs, random read speed of about 10 nanoseconds, 50 times slower than registers.

- **HDD**: Capacity of a few terabytes, random read speed of about 10ms, much slower than RAM.
- **SSD**: Capacity of a few terabytes, random read speed of about 2ms, five times faster than HDD.
- **External Storage**: Includes CD, floppy, tape, and pen drives, generally slower than SSD.



As we observe, each level of memory serves as a cache for the lower level. This is necessary because the lower level is too small to store all the data. For instance, if we could store all data directly in registers, higher-level caching wouldn't be necessary. However, producing registers is very difficult and expensive compared to producing HDDs, which are relatively cheap. Therefore, we can conclude that the higher level is too small to store all the data, and the lower level is too slow to be used directly.

Now that we've explored hardware caching, let's delve into browser caching.

Caching Layers for Backend Infrastructure ->

User's Browser ->

Browsers have various caches:

1. DNS cache
2. File cache (for multimedia content like images, audio, video, PDFs, CSS, and HTML)

Storage options in the browser include:

1. **Cookies** (session storage) with a few kilobytes of storage capacity
2. **Local storage** with a capacity of up to 10MB, commonly used for user settings and theme preferences

3. **IndexedDB**, which can hold up to 10GB of data (although the actual capacity depends on browser allocation). It is IndexedDB is often used for storing large datasets like Google Maps data

If data isn't found in the browser cache, it requests it from the backend system.

There are two ways to fetch data from the browser:

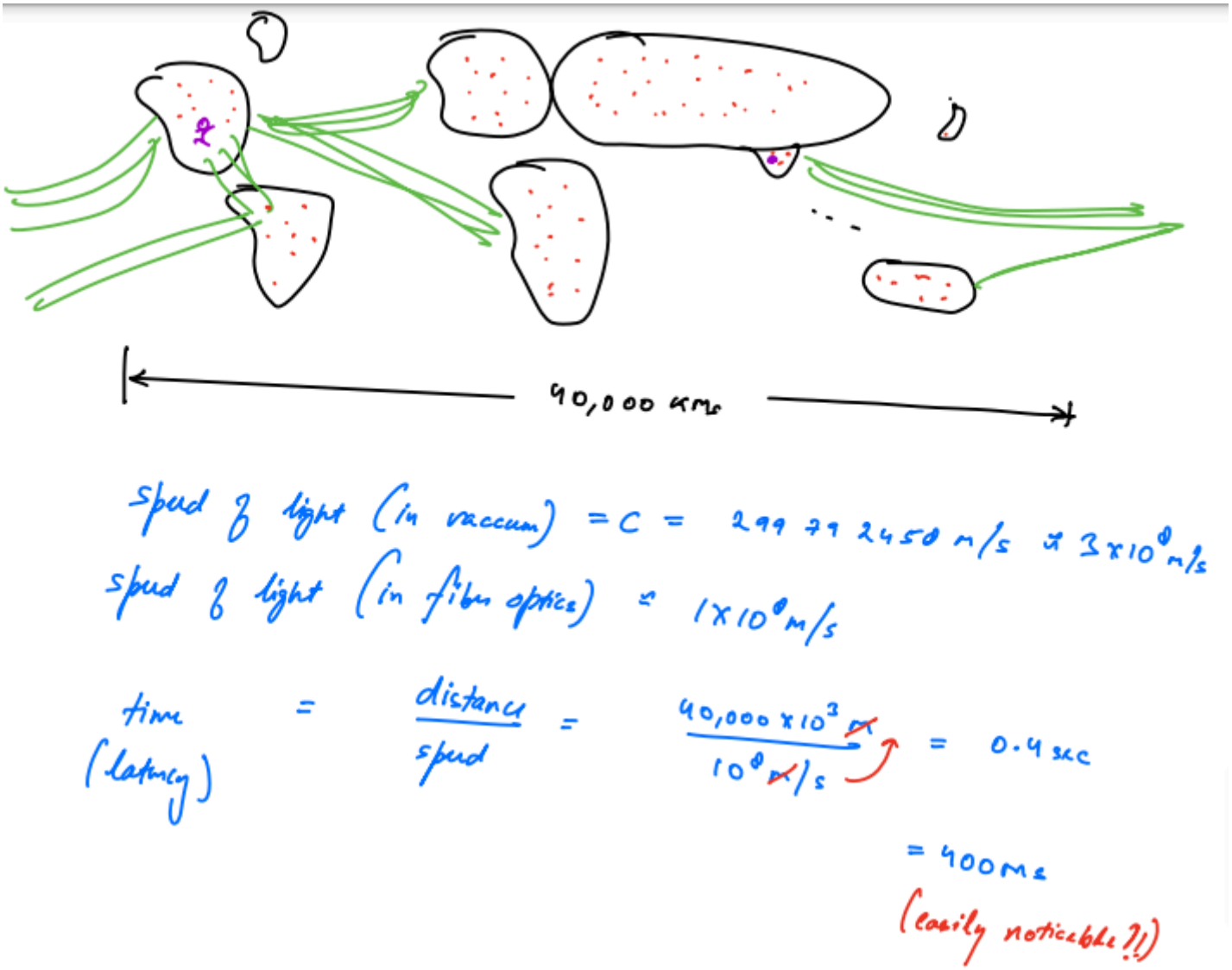
1. **CDN (Content Delivery Network)**
2. **Backend Infrastructure**

Primarily, the browser fetches data from the CDN. If the data is not available on the CDN, it fetches the data from the backend. The CDN also retrieves the data from the backend when needed. Within the backend, there are multiple caching layers:

- **Local Cache**: Inside the app server
- **Global Cache**
- **Database**:
    - In-memory cache
    - Data on disk

Content Delivery Network (CDN):

- **Scenario**: Consider a server and database located in Mumbai, South East Asia, hosted on AWS, with users worldwide, including the US and India. The distance between the US and India is significant. Using the speed of light as a reference, it takes approximately 400ms to send a signal and receive a response via fiber optics. This is the fundamental speed limit of the universe. Despite advanced technology, this delay is inevitable. For someone watching a video in the US from an Indian server, this 400ms delay is noticeable and problematic.



Problem: Minimize latency for users that are far from your database. We cannot change the speed of the universe.

Solution: Move data closer to users. However, moving data for just 10% of users is impractical. A better solution is to copy the data, and for this, we use a CDN.

Content Delivery Network (CDN): =>

A CDN consists of hundreds of thousands of servers distributed across the globe, forming a highly distributed and hierarchical caching network. As a small company without the funds to set up servers worldwide, we rely on companies like Google, Amazon, and Cloudflare, which have the revenue to maintain these extensive networks. These servers act as caches for our data. The server closest to the user caches the data, minimizing latency by serving the user from the nearby cache server.

These are major CDN providers that help distribute content globally through their extensive networks of servers:

Cloudflare | Akamai | CloudFront | Fastly

- **CDN’s is not a database it is just a caching.**
- **CDN's handles aprox 70% of all internet data!**
- **IT is the backbone of internet,**

Any data delivered via CDN is cached in "edge nodes," which are servers located close to the user.

- CDNs take time to cache data across edge nodes, so they are not suitable for frequently changing data.
- CDNs are primarily used to deliver mostly static data such as multimedia (audio, video, images, PDFs, ZIP files), HTML/CSS, JavaScript, and large data files (e.g., JSON, but not dynamic API response data).

Let's assume a scenario where we have a user, a database server, and a main CDN server that is very far away from the user. There might be a CDN edge node close to the user. Here’s how the data is handled:

1. The user calls the API and sends a request to the backend.
2. The backend returns a URL, which might be embedded in some HTML.
3. When the browser sees this URL, it makes a request to it. This URL is a CDN URL.
4. The browser first makes a request to the DNS to get the IP address.
5. The DNS returns the IP address of the main CDN server because it is not feasible to map hundreds of thousands of IPs for the CDN. It only contains the main server details.
6. After getting the main server IP, the browser requests the main CDN server.
7. The main CDN server returns the IP address of the closest CDN edge node. This process is called Anycast.
8. Finally, the browser sends the request to the CDN edge node and gets the data from the nearest node.
9. If the nearest node does not have the data, it fetches the data from the main database, caches it locally, and then serves this data.
10. If the user requests the same URL again, the data will be served directly from the nearby edge node until the cache is purged.

GEO DNS: A geographic DNS routing technique that directs user requests to the nearest server based on the user's location. This helps reduce latency and improves load times by serving content from a server geographically closer to the user.

Anycast: A network addressing and routing method in which a single IP address is assigned to multiple servers. When a user requests content, the request is routed to the nearest or best-performing server based on network conditions. This approach enhances load balancing and minimizes latency by ensuring users are served by the closest or most efficient server.

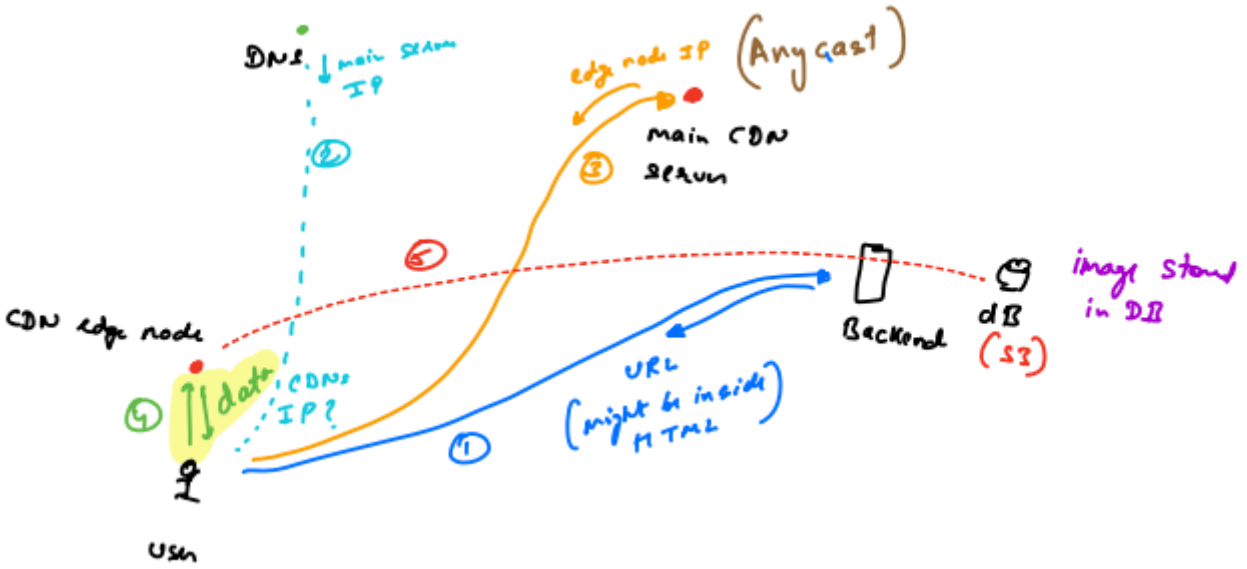
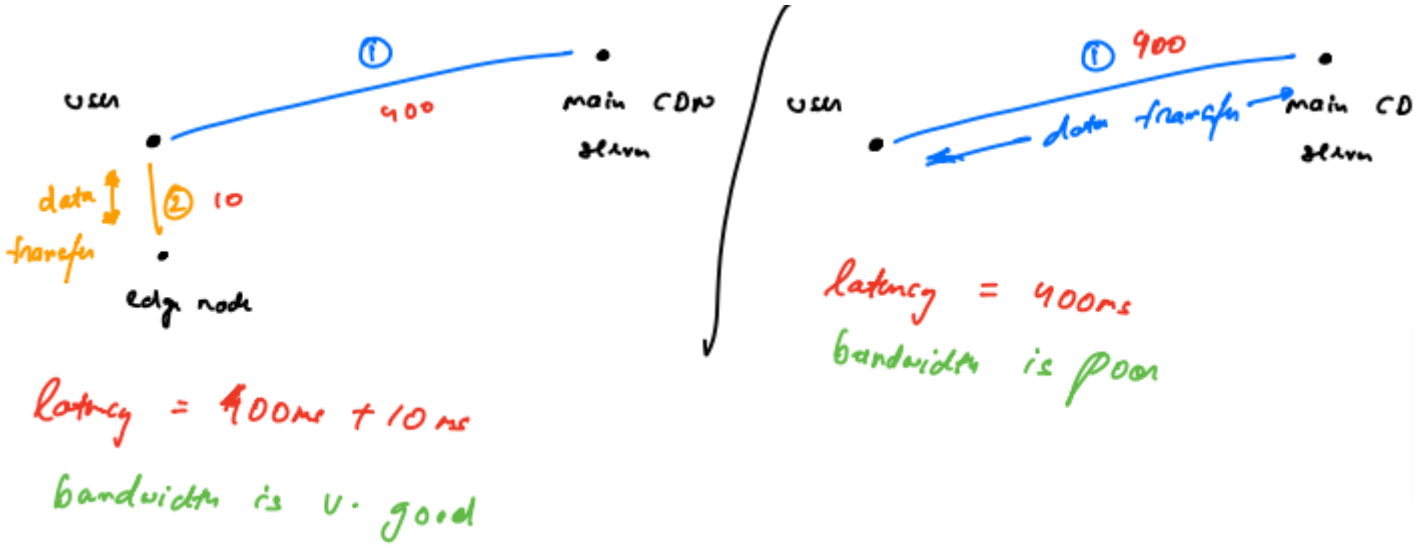


Illustration of Anycast

Note: CDNs cache data on the second request. Studies show that approximately 70% of all URLs are "one-hit wonders," meaning they receive only one request in their entire lifetime. Because of this, many developers send a fake request before the actual request to ensure the data is cached in the CDN.

Why Anycast -> Anycast all req first go to main server and then to edge nodes!
As illustrated, we use Anycast for its advantages despite the extra initial latency. When a request is made using Anycast, it first goes to the main server, which then redirects it to the edge node. This process takes around 400ms + 10ms. In contrast, a direct server request takes only 400ms.

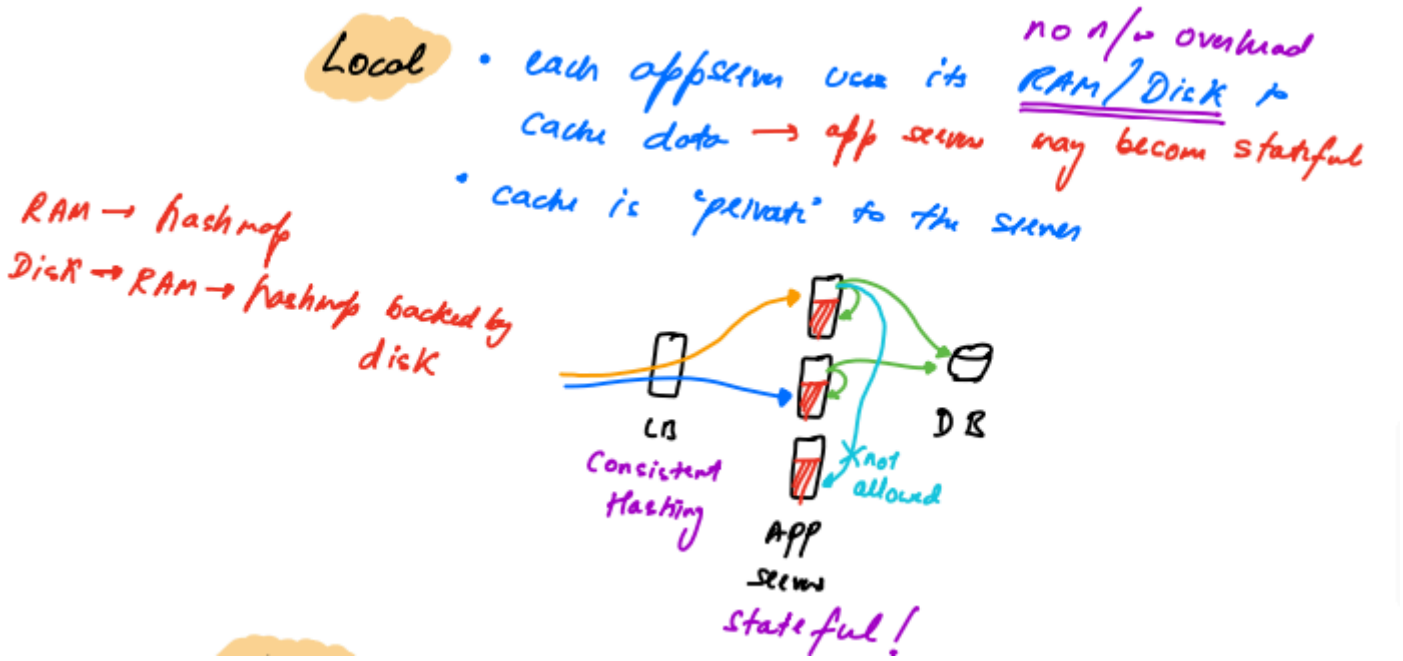
So, why do we use Anycast despite the extra time? The reason is that while initial latency is higher, bandwidth is improved because data is accessed from the nearest server. Bandwidth is more important than latency in this context, as it ensures faster and more reliable data transfer after the initial request.



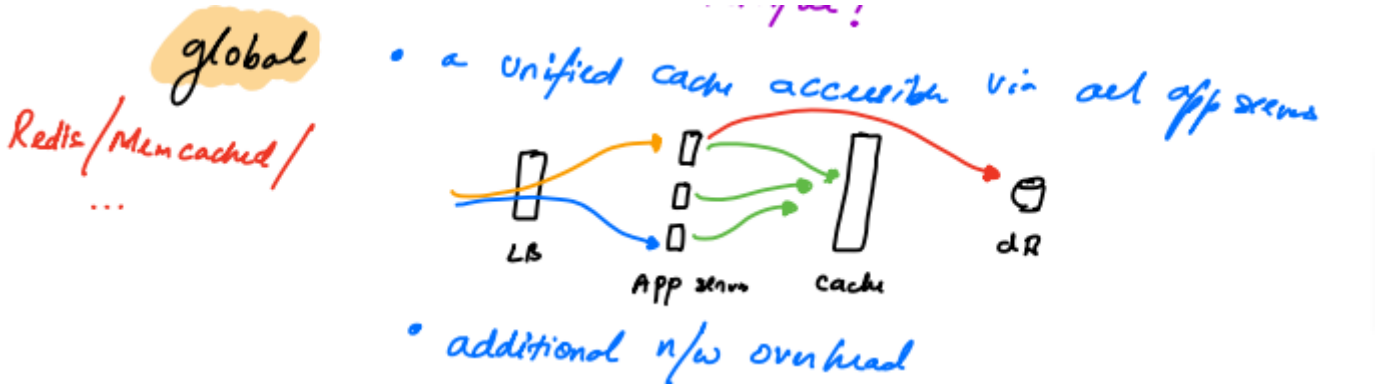
Backend Caching ->

Local Caching vs Global Caching ->

Local caching is performed within the app server itself, storing frequently accessed data specific to that server or instance. This approach is useful for items like user session data or configuration settings that do not need to be shared across multiple servers. By caching this data locally, the server can quickly access it without needing to reach out to a central database or external resource, thereby improving performance and reducing latency for server-specific operations.

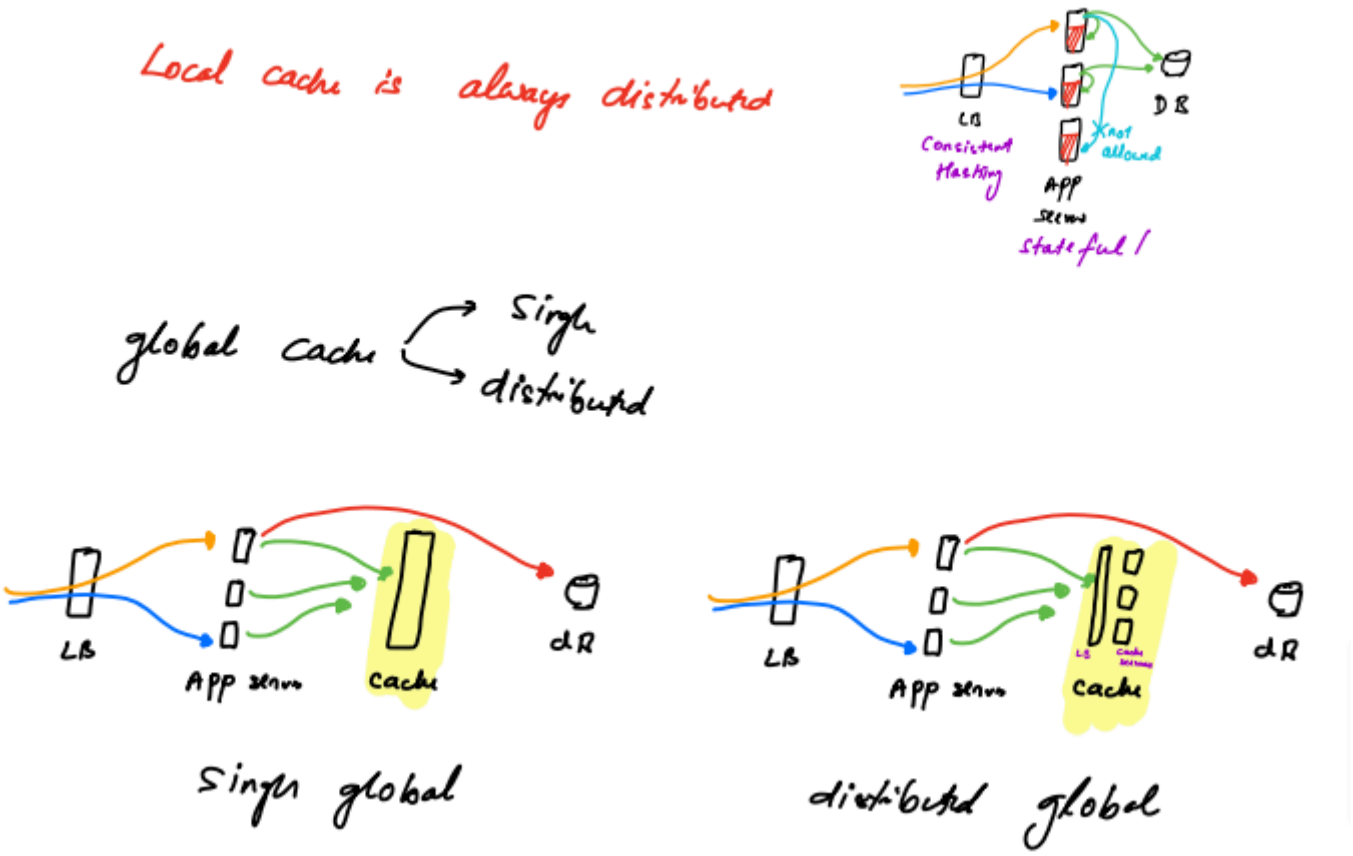


Global caching involves caching data across multiple servers or instances, making the cached data accessible globally. This type of caching is beneficial for data that needs to be shared across different instances or servers, ensuring performance improvements and consistency. Examples of global caching include using distributed cache systems like Redis or Memcached to store session data, user profiles, or other frequently accessed data needed by multiple instances. By sharing this data globally, global caching helps maintain consistency and provides quick access to data across the entire infrastructure, enhancing overall system performance.



Single vs. Distributed Cache

A single cache system may be adequate if its capacity is large enough to meet caching needs. However, if the cache's capacity is insufficient, a distributed cache becomes necessary. It's worth noting that a local cache inherently operates as a distributed cache.



Issues with Cache:

- **Limited Capacity**: Caches are typically smaller in size compared to the main database, which means they can become full. To accommodate new data in the cache, old data must be removed, a process known as eviction. Various eviction policies determine which data is evicted first.
- **Not the Source of Truth**: Cached data can become outdated or stale over time. When this happens, the cached data is no longer accurate or valid. To address this issue, a strategy known as invalidation is used to mark the data as invalid or outdated, prompting the cache to refresh its contents.

Note: Both eviction and invalidation are necessary simultaneously. They are independent processes, and different policies can be chosen for each to manage the cache effectively.

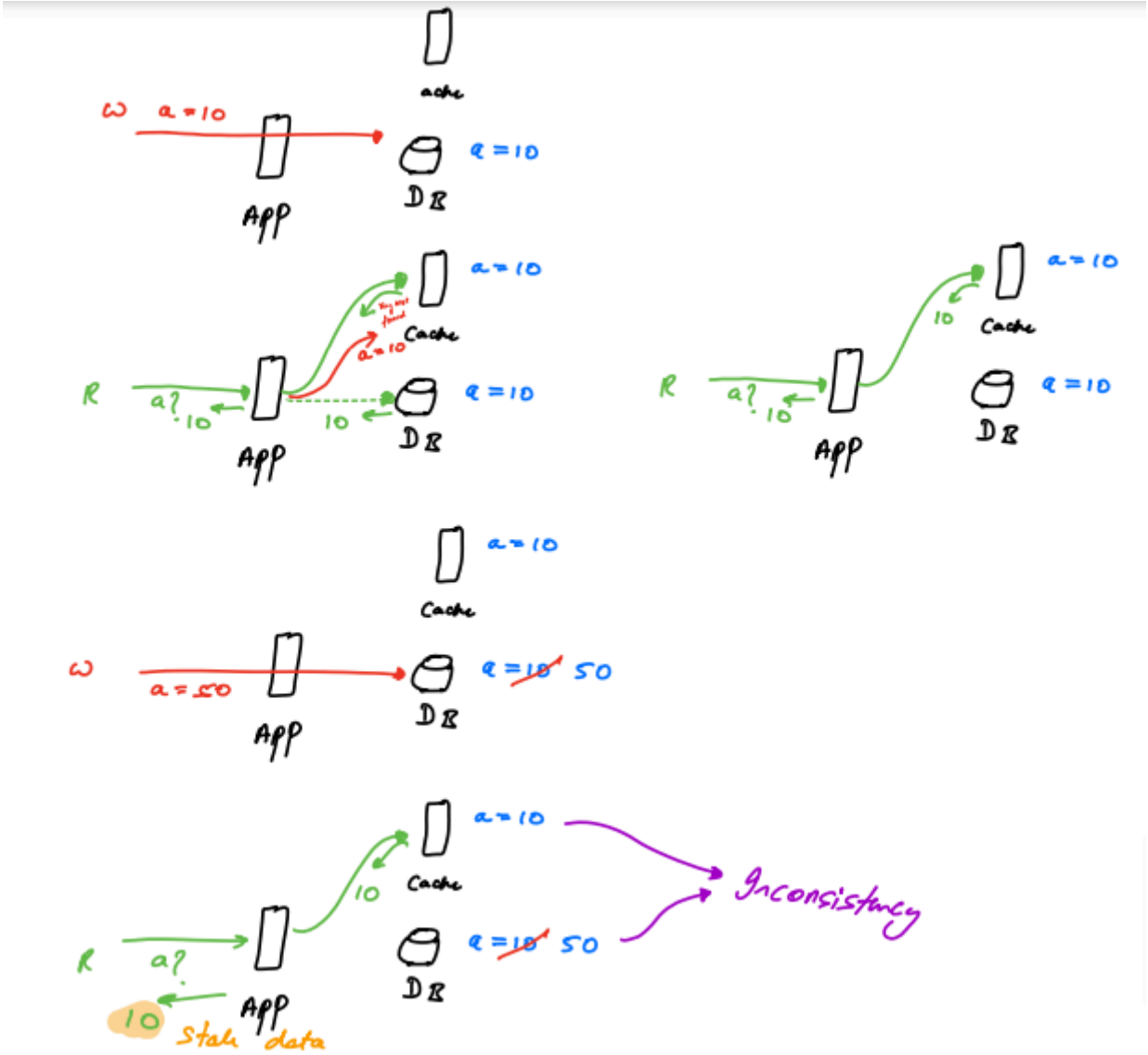
Eviction Policies -> Eviction Policies are rules or strategies used by cache systems to determine which data should be removed from the cache when the cache becomes full and new data needs to be stored. There are several eviction policies commonly used:

1. **Least Recently Used (LRU)**: This policy removes the least (less used) recently accessed data from the cache. If a cache entry has not been accessed for the longest time, it is evicted first.
2. **Least Frequently Used (LFU)**: LFU eviction policy removes the least frequently accessed data from the cache. If a cache entry has been accessed the fewest number of times, it is evicted first.
3. **First-In-First-Out (FIFO)**: With this policy, the data that was added to the cache earliest is evicted first. It operates like a queue, where the oldest entry is removed to make room for new data.
4. **Most Recently Used (MRU)**: Similar to LRU, this policy evicts the most recently accessed data from the cache. The entry that was accessed most recently is removed to make space for new data.
5. **Random Replacement (RR)**: In this policy, a random cache entry is chosen for eviction. While simple to implement, it may not be the most efficient strategy in all cases.

Consistency with Respect to High-Level Design (HLD)

Can we have stale reads? This issue arises when we have multiple copies of data (due to replication or caching). Let's consider the following illustration with one app server, one database, and one cache server:

1. **First Write Request (a=10)**: The request is sent to the database, which stores a = 10.
2. **First Read Request**: The request first hits the cache server. Finding no data, it then hits the database, which returns 10. This response is also saved to the cache.
3. **Second Write Request (a=50)**: The request is sent directly to the database, which updates the value to a = 50.
4. **Second Read Request**: This request hits the cache server, which still has the stale value of 10, resulting in an incorrect response. This is called inconsistency.



Immediate Consistency / Strong Consistency: All copies of the data are always exactly the same (consistent). Any writes (inserts/updates) are immediately reflected in all copies.

Eventual Consistency: The copies might not be consistent right now, but if you wait long enough, they will eventually become consistent through synchronization. Any write will eventually be reflected in all copies.

Caching-2 - June 14

- **Invalidation Policies**
  - **Time-to-Live (TTL)**
  - **Write-Around Caching Policy**
  - **Write-Through Caching Policy**
  - **Write-Back Caching Policy**

Useful resource links

Two Phase Commit (2PC)

This channel by Martin is called the god of a Distributed system

https://www.youtube.com/watch?v=-_rdWB9hN1c

This channel by Hussain Nasir
which principal architect provides important information about technology changes, such as how big companies like Amazon have shifted from microservices to monolithic architectures.
https://www.youtube.com/watch?v=eltn4x788UM)

This channel by Google engineer Ararpi Bhayani
https://www.youtube.com/watch?v=7FgU1D4EnpQ (Google engineer)

Redis

Quick start: https://redis.io/learn/howtos/quick-start

Online playground: https://try.redis.io/

Eviction policies & Cluster mode:
https://docs.google.com/document/d/1k4nzubvtX_yLctUT4VWK8ZJt4KCcOEdRJdxQgWCaiU8/

Caching - class summary

Basics & Invalidation policies:
https://docs.google.com/document/d/1lKkfPKfnFNEN63lkGhMab2Ghc0d5MQm3-dWU2bu39QA/

Case Studies:
https://docs.google.com/document/d/1taRE46pRaW5HMO1QHLQCug5oEguPHd5p/

**Invalidation Policies -> ** Caching Invalidation is about keeping stored data (cache) up-to-date. When you use caching, data is saved temporarily to make things load faster. But sometimes, this data can get old or change. Invalidation policies help manage this by deciding when to update or remove cached data.

Time-to-Live (TTL) -> Caching Policies are rules that set how long data stays in the cache before it is considered outdated and removed or refreshed.

Here's how it works:

1. **Setting the TTL:**
    - When data is added to the cache, a timer is set.
    - The TTL value determines how long the data will stay in the cache. This could be a few seconds, minutes, hours, or even days.
2. **Expiration:**

    - Once the TTL time runs out, the data is considered **expired/Stale**.
    - Expired data is either removed from the cache or updated with fresh data from the original source.
3. **Eventual Consistency:** Eventual Consistency is a concept used in distributed systems, like databases spread across many servers, to ensure that data will become consistent across all locations, but not necessarily right away. In these systems, data is stored in multiple locations, and when you update data in one place, it changes there immediately. However, this change might take some time to appear in other locations. Over time, all copies of the data will eventually sync up and become the same, although this process might take seconds, minutes, or longer depending on the system. For example, when you post a comment on a social media site, you might see your comment immediately, but friends in other parts of the world might see it after a few seconds or minutes. Eventually, everyone will see the same comment. The benefit of eventual consistency is that it allows systems to be faster and more efficient because they don't need to sync data immediately. This approach is useful for systems where it's acceptable if data isn't instantly up-to-date everywhere. However, a drawback is that there might be a short period where different users see different data, making it unsuitable for systems that require immediate consistency, such as bank transactions. In simple terms, eventual consistency means that while data changes might not show up everywhere right away, all locations will eventually have the same data, helping distributed systems work efficiently without always needing real-time updates.

Example ->

Imagine a news website caches articles for 10 minutes. This means:

- When you visit the site, it loads articles from the cache if they are less than 10 minutes old.
- If the articles are older than 10 minutes, they are fetched again from the server, and the cache is updated.

Writes:

- Only to the database.

Reads:

1. First, check the cache.
    - If the data is present in the cache:
        - Check if it has expired.
        - If valid, return the data from the cache.
        - If expired, delete it from the cache and return a 404 error.
    - If the data is not present in the cache:
        - Fetch it from the database.
        - Return the data to the user.
        - Insert the data into the cache asynchronously (non-blocking) and add a TTL/expiry time.

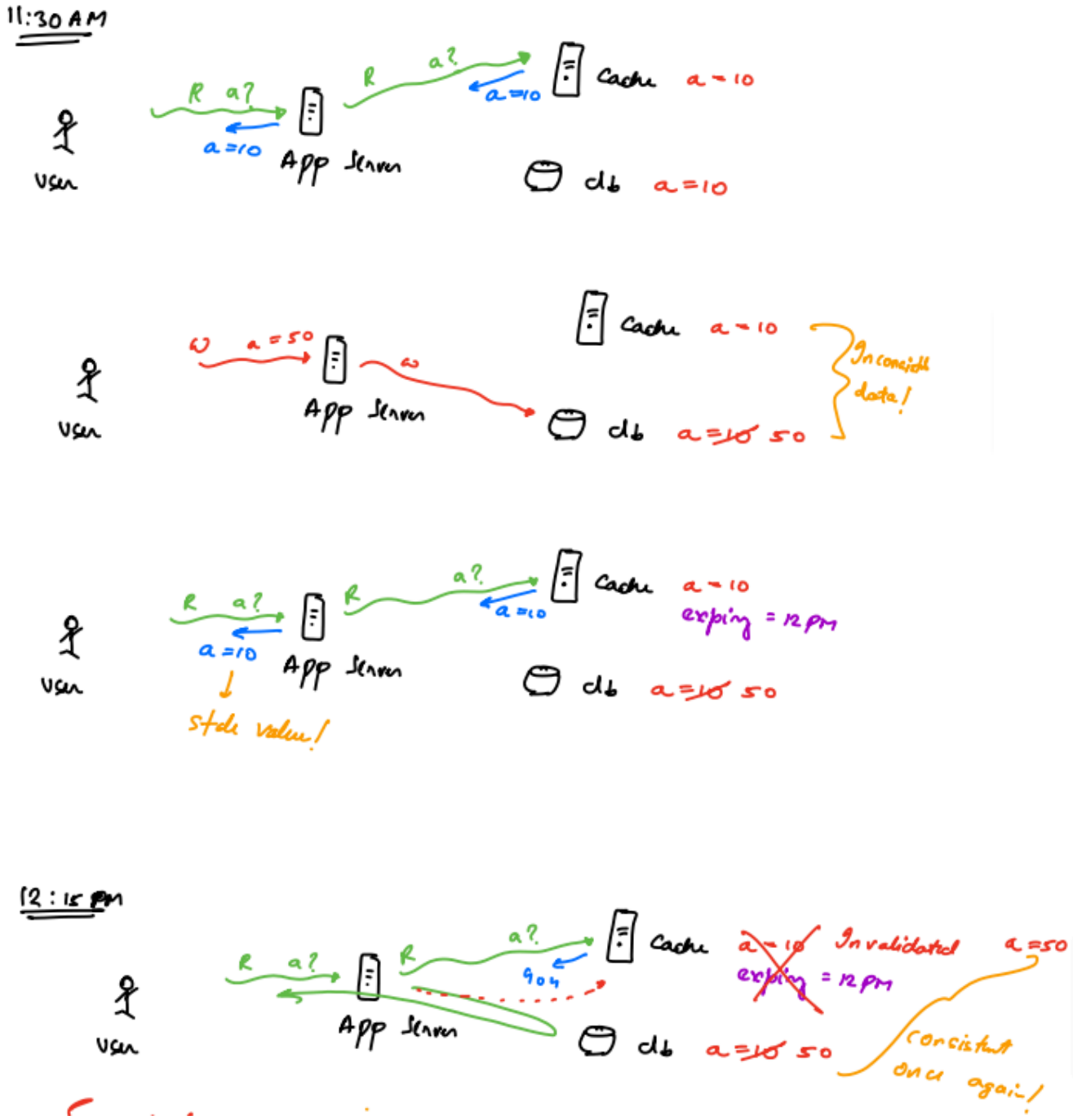
Let's assume we write `a=10` in the database at 10 AM. The first read happens at 11 AM. It will check the cache, but since the cache has no data, it will return a 404 error. Then it will fetch `a=10` from the database, return it to the user, and asynchronously write `a=10` to the cache with a TTL/expiry set for 12 PM.

The second read happens at 11:30 AM. The cache will return `a=10` because the data is still valid.

At 11:45 AM, a second write happens, updating the value in the database to `a=50`.

The third read happens at 12:15 PM. The request first goes to the cache, which finds the data present but checks the expiry time and sees it expired at 12 PM. The cache then removes the data based on eviction policy this will done by LRU/LFU and returns a 404 error. The read request then goes to the database, which returns `a=50`, and this value is inserted into the cache asynchronously without blocking the user response.

This is called eventual consistency. It means data is updated but not immediately; it is updated after the data expires in the cache.
Note: Eviction works in parallel with invalidation. Eviction and invalidation do not interfere with each other.



Eventual Consistency

Correct Design: The update in cache logic should reside in the application server. This is because all business logic is centralized on one server. In a correct design, the cache should be dumb, and the application server should be smart.

Weird Design (still sometimes used): The cache automatically refreshes from the database if the value has expired. This design is problematic because it splits the business logic between the application server and the cache server.

Note: When we invalidate, we should invalidate reads lazily. There are two types of invalidation:

Eager Invalidation: If we set an expiry of 1 minute, a timer checks every minute and removes expired keys. This approach is costly in terms of resources, even though there is no space wastage.

Lazy Invalidation: Here, we don't proactively remove expired values. Instead, we check if a value has expired only when a read request comes in. This method uses more space due to potentially retaining invalid entries but is significantly less resource-intensive. It's preferred because managing space is the responsibility of eviction policies, not invalidation.

Write-Around Caching Policy

The write-around caching policy is similar to TTL (Time-To-Live) but is used when computing or fetching data is extremely expensive, typically involving heavy joins and processing. This approach also provides eventual consistency. Typically data is small.

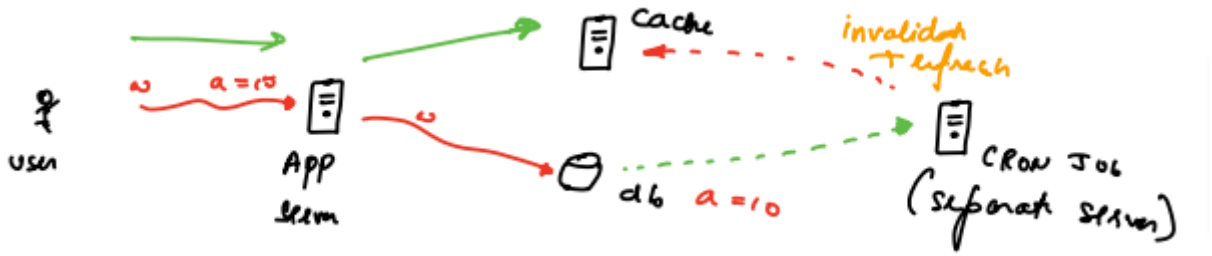
Writes: Writes only happen in the database.

Reads:

- **First, check the cache.**
  - If the data is present in the cache:
    - Check if it has expired.
    - If valid, return the data from the cache.
    - If expired, delete it from the cache and return a 404 error.
  - Read operations always happen from the cache, not from the database.
- **If the data is not present in the cache**, it will not fetch from the database. This is the main difference from TTL caching, as the database operations are very expensive.

Cache Update Mechanism:

- Since reads do not update the cache asynchronously due to the high cost of database operations, the cache is updated using a cron job.
- A cron job is a utility on the server used to schedule tasks periodically.
- Periodically, the cron job fetches the latest data from the database and updates the cache.

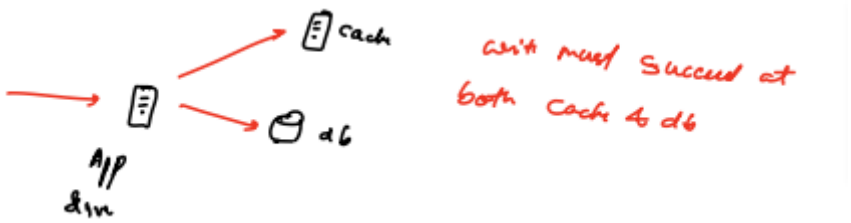


Write-Through Caching Policy

The write-through caching policy ensures that data is updated in both the cache and the database simultaneously, providing a high level of consistency and reliability.

Writes:

- When data is written, it is updated in both the cache and the database at the same time. This operation must be atomic, meaning either both updates succeed, or both fail, similar to a transaction. If a write occurs, it must succeed in both the database and the application server; if not, both operations are rolled back.



Potential Issues:

1. **Partial Failure**:
    - If one update succeeds but the other fails, the system should return a failure and roll back the changes at both locations.
2. **Acknowledgment Failure**:
    - If both updates succeed but the acknowledgment is only received from one, this could lead to inconsistencies. Failures can occur for various reasons, such as the request never reaching the server, the server failing after receiving the request, or the server successfully processing the request but failing to send an acknowledgment.
3. **Rollback Failure**:
    - If a rollback operation fails at one location, it can create a recursion problem.

To address these issues, the **Two-Phase Commit (2PC)** algorithm is often used:

- **Phase 1 (Prepare)**: The system prepares to commit the transaction by ensuring all parties (cache and database) are ready. Each party votes to commit or abort.
- **Phase 2 (Commit)**: If all parties vote to commit, the transaction is committed. If any party votes to abort, the transaction is aborted and rolled back.

Using this method ensures atomicity, reducing the risk of inconsistencies between the cache and the database.

Write-Back Caching Policy

The write-back caching policy, also known as write-behind, involves updating the cache immediately and deferring the write to the database until later. This approach can improve performance but introduces complexity in ensuring data consistency. This typically happens for **Analytics**

Writes and Reads: Both operations occur only at the cache, not the database. This is because the cache is extremely fast, resulting in ultra-high read and write throughput.

- Typical SQL databases handle around 100 to 1000 reads per second and 10 to 100 writes per second.
- In contrast, typical in-memory cache servers can handle 10,000 to 1 million reads or writes per second.

Data Consistency:

- There is no immediate consistency between the cache and the database.
- The database is updated periodically by dumping data from the cache in batches. This deferred batch write ensures eventual consistency but introduces the risk of data loss if the system crashes before the batch update occurs.

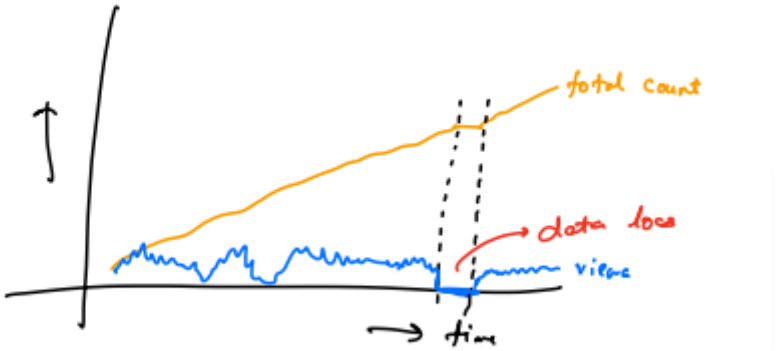
Handling Cache Server Failures ->

Potential Data Loss:

- If the cache server crashes before syncing to the database, any data stored only in the cache memory is permanently lost. While data loss is generally undesirable, it can be acceptable in certain scenarios.

Acceptable Data Loss Scenarios:

- **Example**: YouTube likes and view counts.
  - If you like a video or view it, and the data is lost before syncing to the database, it isn't critical.
  - For analytics purposes, such as tracking view counts and likes, the overall trend matters more than precise data.
  - Even if some views and likes are lost, the trend chart will show a slight dip but will continue to reflect the general trend accurately.



Caching - Case Studies - June 19

Agenda:

- **Invalidation Policies**
- Caching Architectures

Caching Architectures (Backend)

1. **Steps to Decide Caching**
    - **Choose Cache Type: Local vs. Global**
        - If we want less overhead and lower latency, we choose a local cache.
        - If we can handle a small cache that doesn't affect the network much, we choose a global cache.
        - The default choice is usually a global cache.
    - **Management: Single Server vs. Distributed**
        - This step is only needed for a global cache because a local cache is always distributed.
2. **Eviction Policy**
    - Decide how to remove cache entries when the cache is full. Options include:
        - **LRU (Least Recently Used):** Removes the cache entry that hasn't been used for the longest time.
        - **LFU (Least Frequently Used):** Removes the cache entry that has been used the least often.
        - **FIFO (First In, First Out):** Removes the oldest cache entry first.
        - **MRU (Most Recently Used):** Removes the most recently used cache entry first.
3. **Invalidation Policy**
    - Decide how to invalidate and mark cache entries as stale to make space for new cache entries. Options include:

- **TTL (Time to Live):** Each cache entry has an expiration time. Once it expires, it's invalidated.
- **Write-Around:** Writes data directly to the storage, bypassing the cache.
- **Write-Through:** Writes data to both the cache and the storage at the same time.
- **Write-Back:** Writes data to the cache first, then to the storage later.
- **Cache-Aside:** The application directly manages the cache. It fetches from storage and updates the cache when needed.
- **Write-Behind:** Similar to write-back, but data is written to storage asynchronously, meaning it doesn't wait for the write operation to complete.

Invalidation Policy for Scaler Code Judge

Why is an invalidation policy required?

It's necessary when we have two copies of data, leading to consistency issues. For example, we have one copy in the database and another in the cache. Invalidation is crucial when test cases are updated, typically during a contest, and stored in S3.

The Question: Is it necessary to update test cases if it happens rarely? Can we ignore invalidation?

Answer: No! Our infrastructure must handle updates effectively. Otherwise, it will create a bad experience for students. If a contest is ongoing and we receive feedback that a test case isn't working, updating the test case should be immediately reflected for all students. We cannot cancel the contest due to test case failures.

What is the right invalidation policy for Scaler Code Judge?

1. **Write-Back:**
   - With this policy, test cases are uploaded directly to the cache server (e.g., app server) and later to S3. However, if the app server crashes, we risk data loss and losing some test cases. Therefore, this is not suitable for our current scenario.
2. **TTL:**
   - This policy sets a Time to Live (TTL) for cache entries. There are issues with setting the right TTL:
     - **Low TTL (e.g., 1 minute):** Every minute, each app server must fetch 1GB of data from S3. With 1,000 app servers, this means fetching 1,000GB of data every minute, making the cache useless due to frequent fetching.
     - **High TTL (e.g., 1 hour):** If a contest lasts 3 hours and we update test cases quickly, students might have to wait an hour to see the updates, leading to stale data issues. Hence, this is also not a good solution.
3. **Write-Around:**
   - This policy involves setting a fixed schedule for updates using a cron job. However, finding the optimal timing is challenging, making this policy unsuitable as well.

Requirement: New test cases should take effect immediately, and any new submissions should use the updated test cases, ensuring immediate consistency. For this, we can consider the Write-Through policy.

4. **Write-Through:**
   - This policy seems perfect for ensuring consistency as new data is written to both the cache and the storage immediately. However, we must consider other factors before deciding if this is the best choice.

CAP Theorem, PACELC Theorem & Master-Slave - 21 Jun

Agenda:

1. **CAP Theorem**
2. **PACELC Theorem**
3. **Master Slave Theorem**
   - **3.1 Types of Master-Slave Systems**
   - **3.2 Drawbacks of Master-Slave Systems**

Usefull links
MySQL: Master Slave Replication - extra reference
https://docs.google.com/document/d/1w0dCLJgqfnC8ylWCLCj3yCVkLwfBTvx2QuVTlUKPtAI/edit

CAP Theorem: class summary
https://docs.google.com/document/d/1e3cKPLd8ajBEJ17HK6lAn7T1qP4QWSJ9cN3DHn3X_x0/edit

Consistency Overview

Consistency can be understood differently in various contexts, such as traditional SQL databases and high-level design (HLD) in distributed systems.

SQL Databases (ACID Properties)

In SQL databases, consistency is part of the ACID properties:

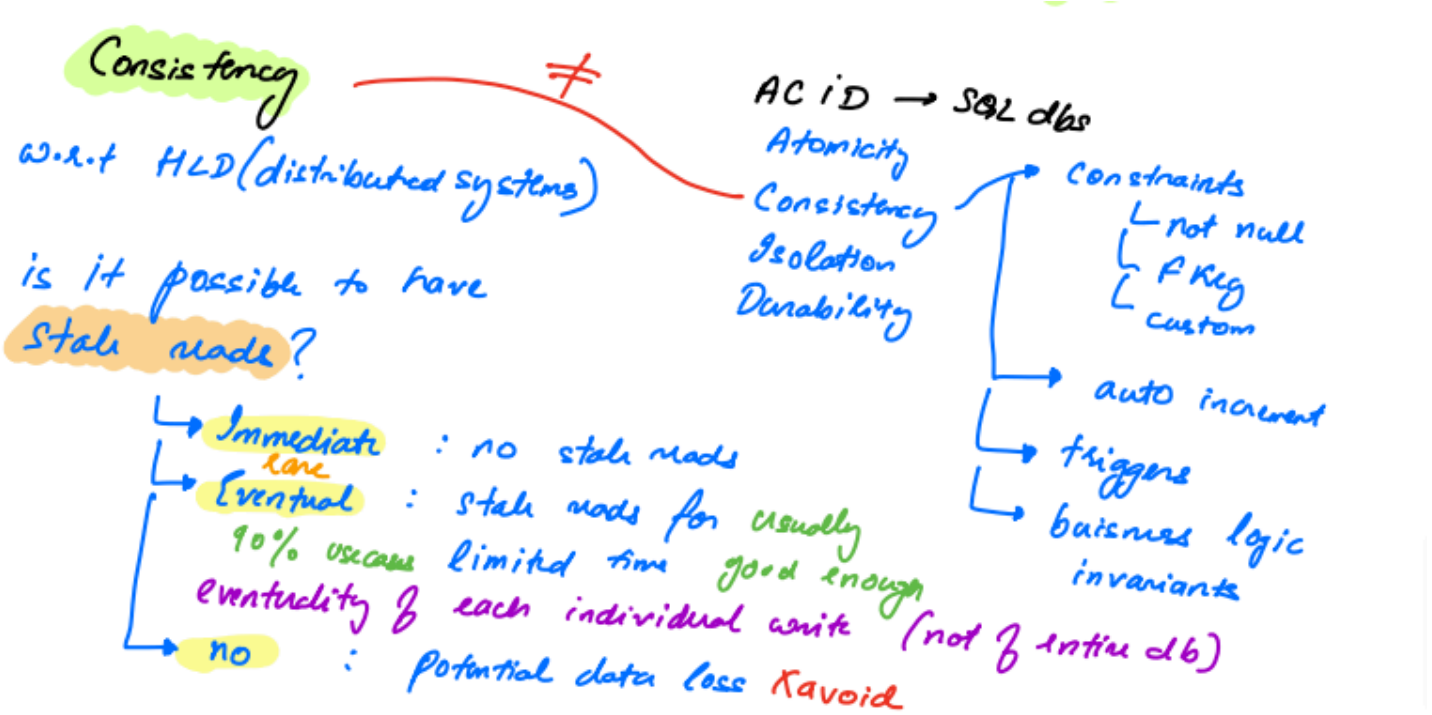
- **Consistency**: Ensures that a transaction takes the database from one valid state to another, maintaining the integrity of the database according to predefined rules. This means that any transaction will leave the database in a consistent state, adhering to all integrity constraints.

High-Level Design (HLD)

Consistency comes into the picture when we have replication, meaning multiple copies of data. In high-level design, particularly in distributed systems, consistency has different nuances. There are three types of consistency:

1. **Immediate Consistency**: Ensures no stale (cached) reads. Any read after a write will return the latest value.
2. **Eventual Consistency**: Allows stale reads for a limited time. Eventually, all reads will reflect the most recent write, but there may be a delay. 90% system design we will follow eventual consistency.
3. **No Consistency**: There is potential for data loss. Reads might not reflect the most recent writes, and data might be inconsistent.

In summary, while SQL databases focus on maintaining integrity through immediate consistency as part of the ACID properties, distributed systems in HLD can offer different consistency models depending on the requirements for latency and availability.

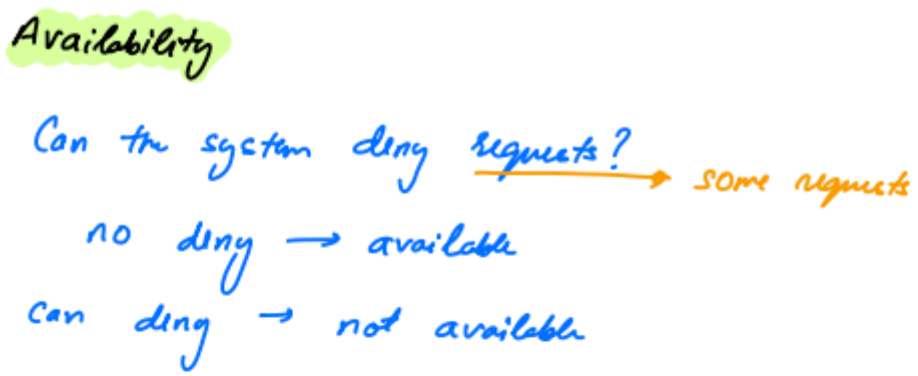


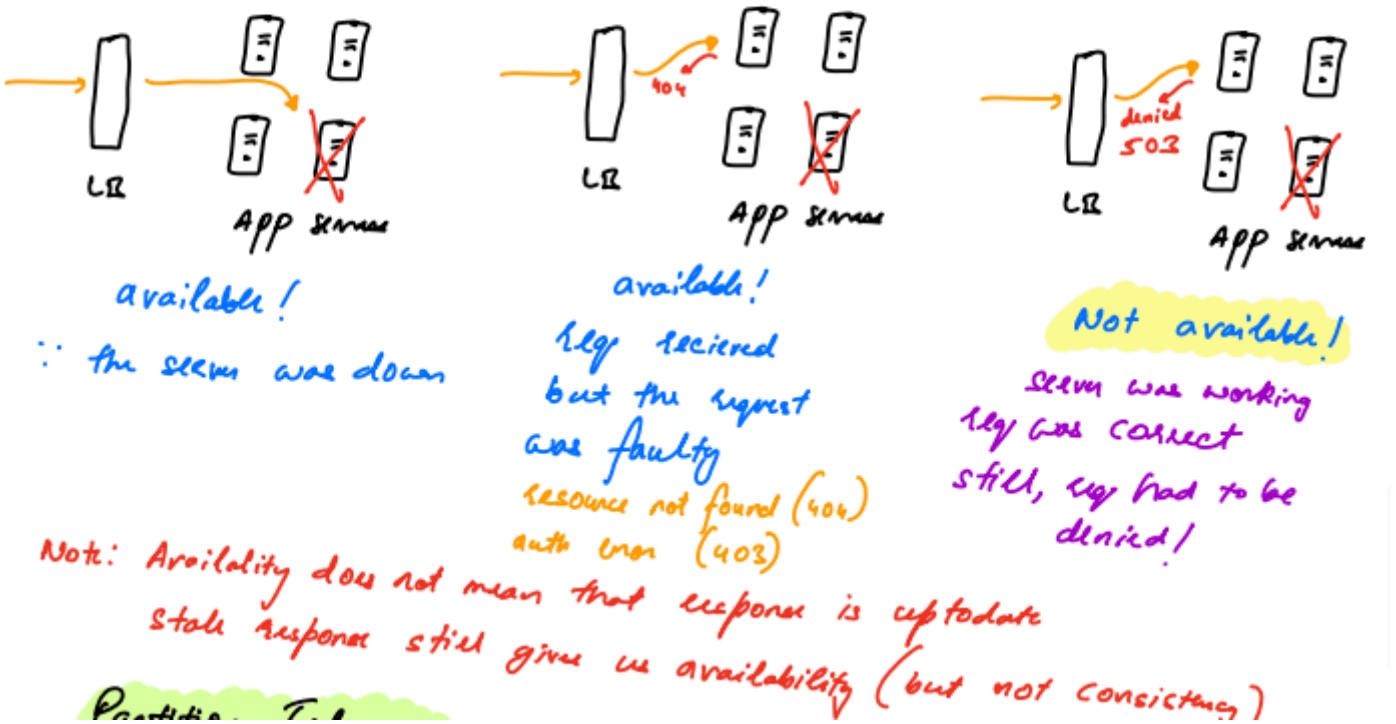
Availability Overview

Availability is about the system's ability to respond to requests. Here are different scenarios to understand availability better:

- **System Denying Requests**: If the system is healthy but denies some requests, it means there is no availability due to some communication failure.
- **System Crash or Downtime**: If the system is not working or has crashed, it is not considered a denial of service because the server is down and unavailable.
- **404**: If the system is accepting requests but returns errors like 404 (resource not found) the system is still considered available. These errors indicate that the server is reachable, but there are issues with specific resources or services.
- **Server Denying Valid Requests with 503**: 503 (service unavailable), If the server is healthy and working fine, and our request is valid, but the server denies the request, it means the server is not available.

In summary, a system's availability is judged by its ability to respond to requests. Errors and communication failures affect availability differently based on the underlying issues.



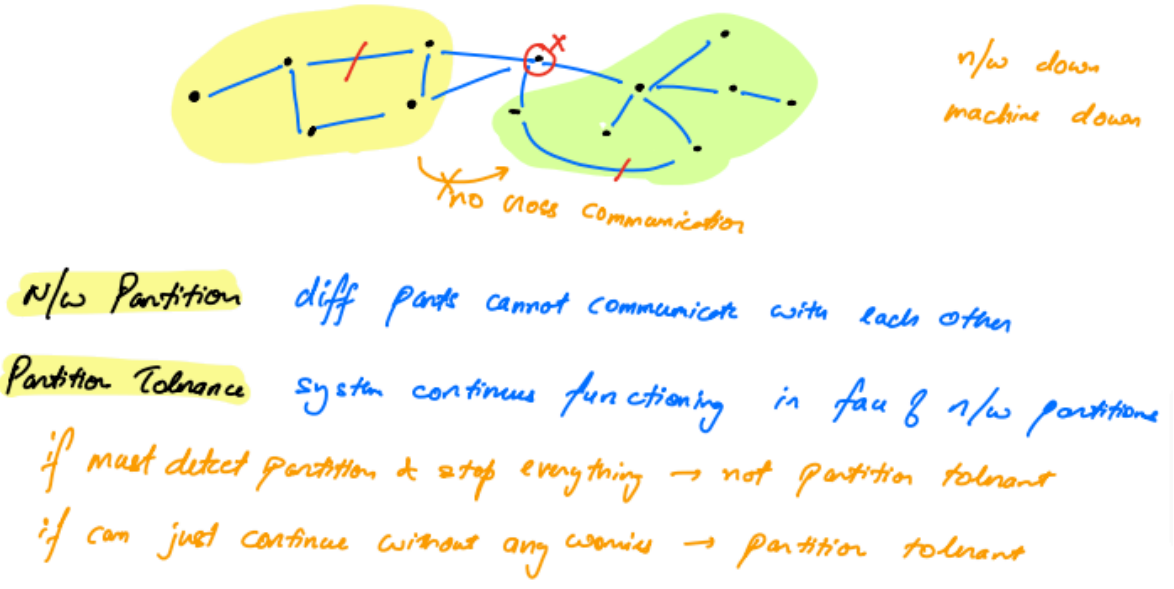


Partition Tolerance

Partition tolerance in the context of distributed systems refers to the system's ability to continue operating correctly even when network partitions occur. A network partition happens when there is a loss of communication between different parts of the system, causing nodes to be unable to communicate with each other.

Key Points About Partition Tolerance:

1. **Network Partitions**: Network partitions can occur due to various reasons such as network failures, configuration errors, or hardware issues. During a partition, some parts of the network can become isolated from others.
2. **System Resilience**: A partition-tolerant system continues to operate and provide its services even when communication between nodes is disrupted. This means that the system can still process requests and maintain some level of functionality despite the partition.



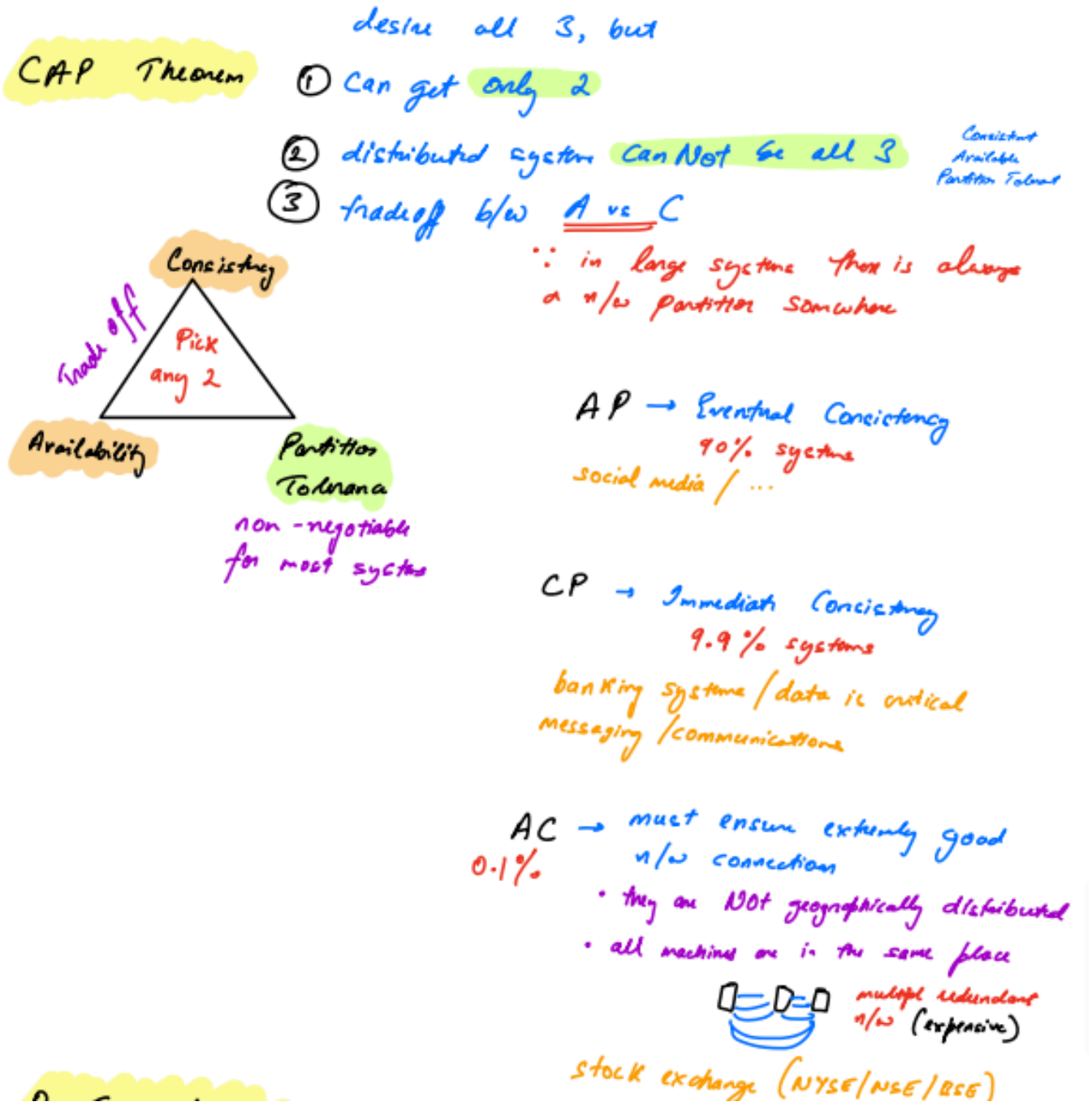
Trade-Offs: Consistency vs. Availability: To achieve partition tolerance, systems often have to balance between consistency and availability. For instance, during a partition, a system might choose to remain available (serve requests with possibly stale data) or to remain consistent (refuse to serve requests to prevent data inconsistency).

CAP Theorem:

According to the CAP theorem, a distributed system can only achieve two out of the following three guarantees simultaneously:

- **Consistency**: All nodes see the same data at the same time.
- **Availability**: Every request receives a response, either successful or failure.
- **Partition Tolerance**: The system continues to operate despite network partitions.

Given that network partitions can happen, any practical distributed system must be partition-tolerant. Therefore, the real trade-off is between consistency and availability in the presence of partitions.



PACELC Theorem

The PACELC theorem is an extension of the CAP theorem that provides a more comprehensive framework for understanding the trade-offs in distributed systems. While the CAP theorem addresses the trade-offs during network partitions, PACELC addresses both the scenarios when partitions occur and when they do not.

Breakdown of PACELC: (पेसल थयोरम)	**Mnemonic:**
● **P**: Partition Tolerance (as in CAP) ● **A**: Availability (as in CAP) ● **C**: Consistency (as in CAP) ● **E**: Else (when there is no partition) ● **L**: Latency ● **C**: Consistency	You can use the mnemonic **"Parrots Always Choose Elegant Little Cages"** to remember PACELC. Each word corresponds to the letters in PACELC: ● **Parrot** - Partition ● **Always** - Availability ● **Choose** - Consistency ● **Elegant** - Else ● **Little** - Latency ● **Cages** - Consistency

PACELC Trade-offs:

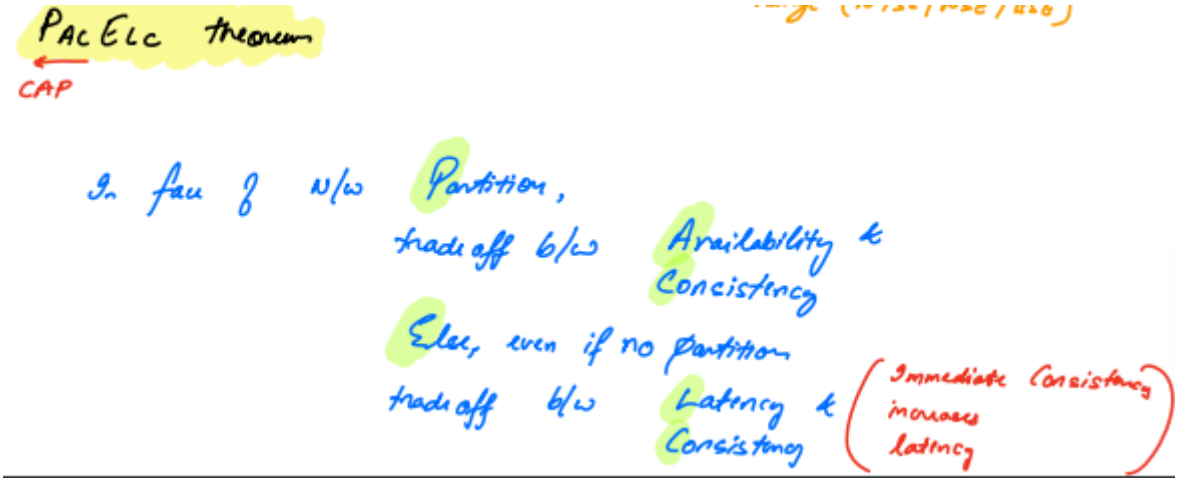
1. **Partitioned Scenario (PAC)**:
    - Similar to the CAP theorem, PACELC states that in the presence of a network **Partition**, a system must choose between **Availability** (A) and **Consistency** (C). This part is identical to the CAP theorem.
2. **Non-Partitioned Scenario (ELC)**:
    - When there is no partition (Else), PACELC states that the system must choose between **Latency** (L) and **Consistency** (C).

Summary of PACELC Choices:

1. **Partition (PAC)**:
    - **PA**: Prioritize Availability over Consistency during network partitions.
    - **PC**: Prioritize Consistency over Availability during network partitions.
2. **Else (ELC)**:
    - **EL**: Prioritize Low Latency over Consistency when there is no partition.
    - **EC**: Prioritize Consistency over Low Latency when there is no partition.

Examples:

1. **Amazon Dynamo (PA/EL)**:
    - During partitions, Dynamo prioritizes Availability (PA).
    - When there is no partition, Dynamo prioritizes Low Latency (EL) over strong Consistency.
2. **Google Spanner (PC/EC)**:
    - During partitions, Spanner prioritizes Consistency (PC).
    - When there is no partition, Spanner also prioritizes Consistency (EC) over Low Latency.



Note: Whenever designing a system we must always first apply PACELE Ethereum

---

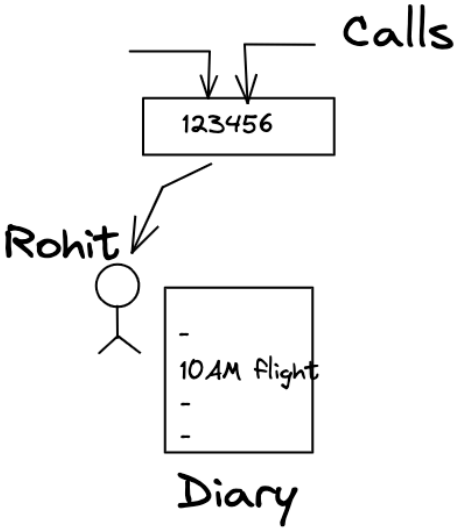
Let's take a real-life example

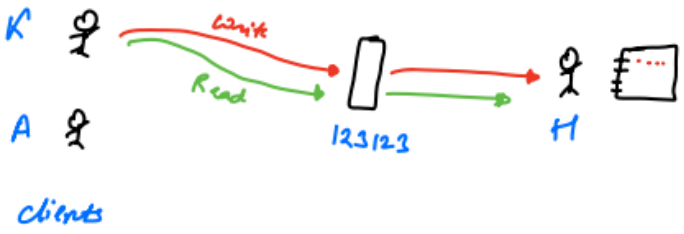
Hard-wavy proof for PACELC theorem

---

say a person named Rohit decides to start a company, "reminder", where people can call him and ask him to put the reminder, and whenever they call him back to get the reminder, he will tell them their reminders.

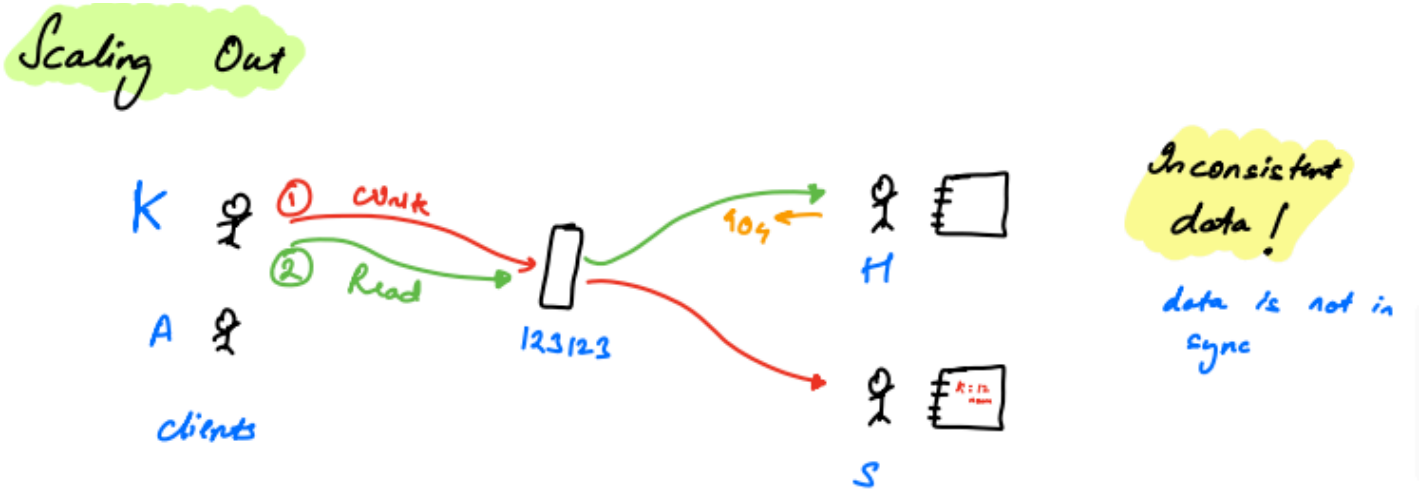
For this, Rohit has taken an easy phone number as well, 123456.
Now his business has started flourishing, and he gets a lot of requests, and he notes reminders in the diary.



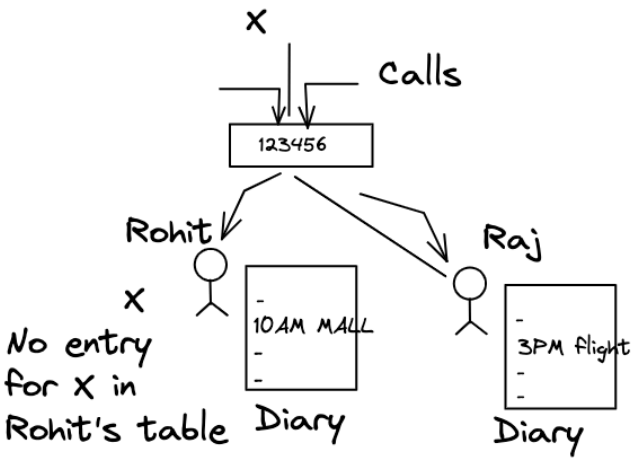


After a while, this process becomes hectic for Rohit alone, because he can only take one call at a time, and there are multiple calls waiting. Now, Rohit hires someone called "Raj", and both manage the business.

Q Can both Himanshu & Sandeep share the diary?
No!
Separate diaries
∵ if shared the diary will become a bottleneck & SPOF
(H & S will fight for the diary)

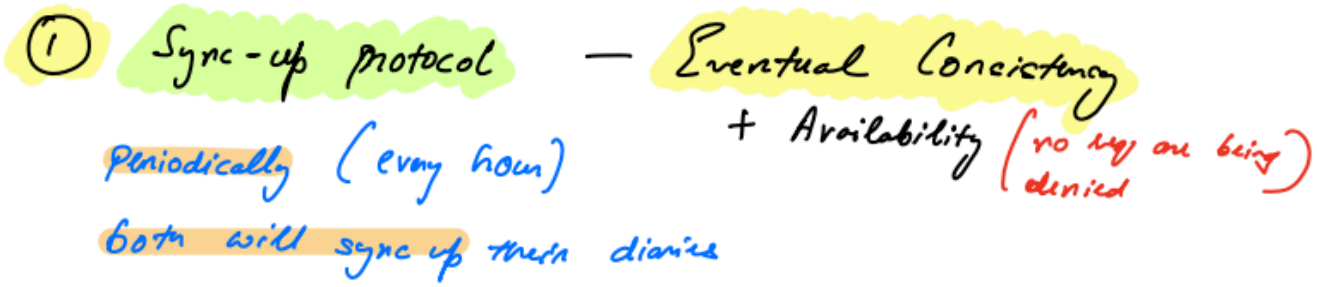


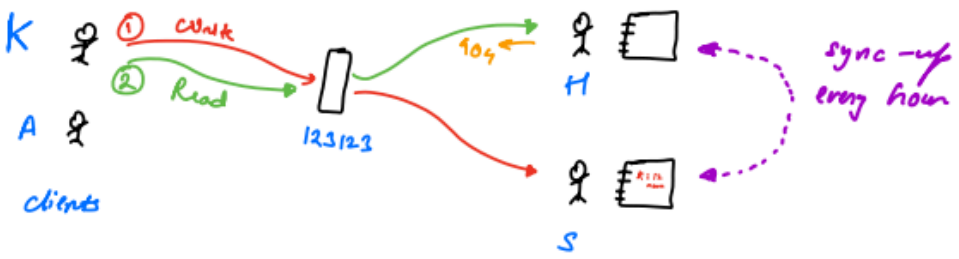
One day, Rohit gets a call from a person named “X” asking him the time of his flight, But Rohit was not able to get any entry for X. So, he says that he doesn't have a flight, but unfortunately, that person has that flight, and he missed it because of Rohit.
The problem is when person “X” called for the first time, the call went to “Raj”, so Raj had the entry, but Rohit didn’t. They have two different stores, and they are not in sync.



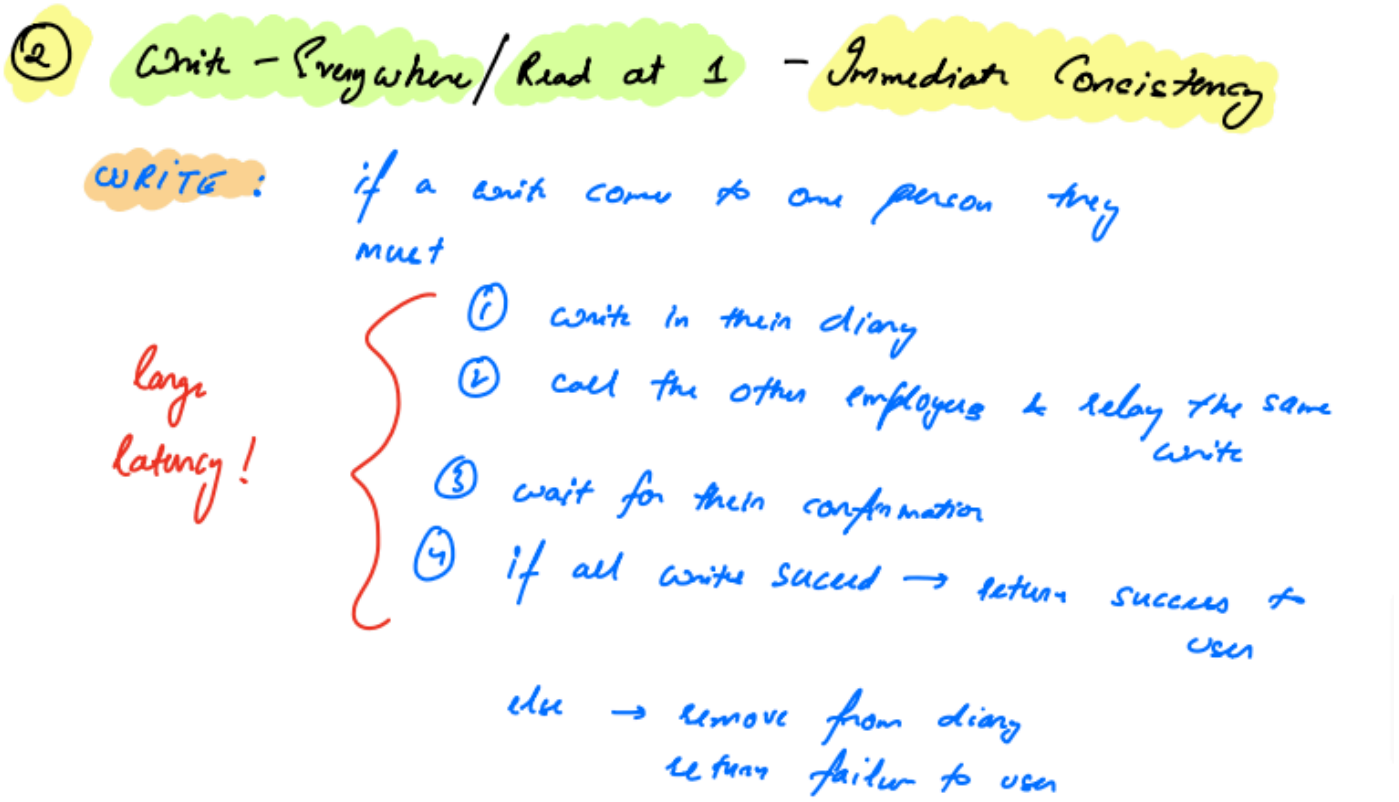
Problem 1: Inconsistency
It’s a situation where different data is present at two different machines.

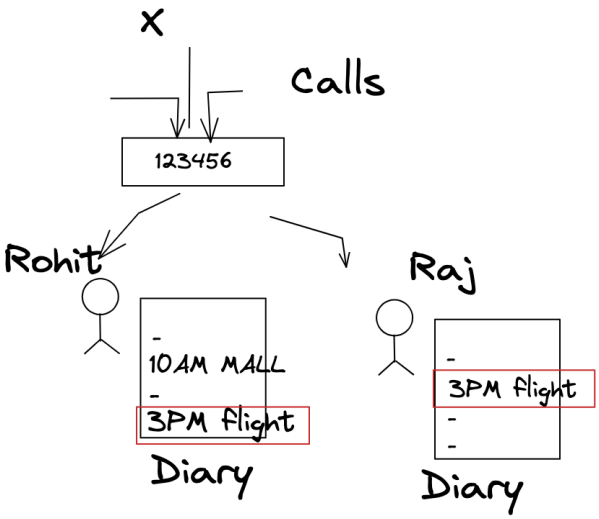
Solution 1:



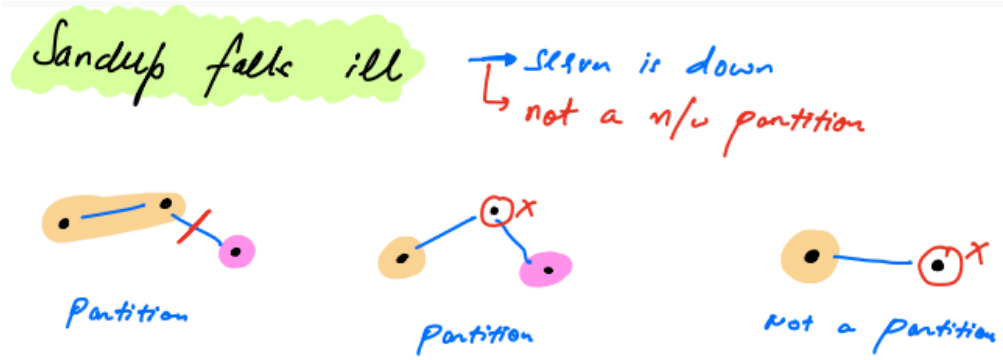


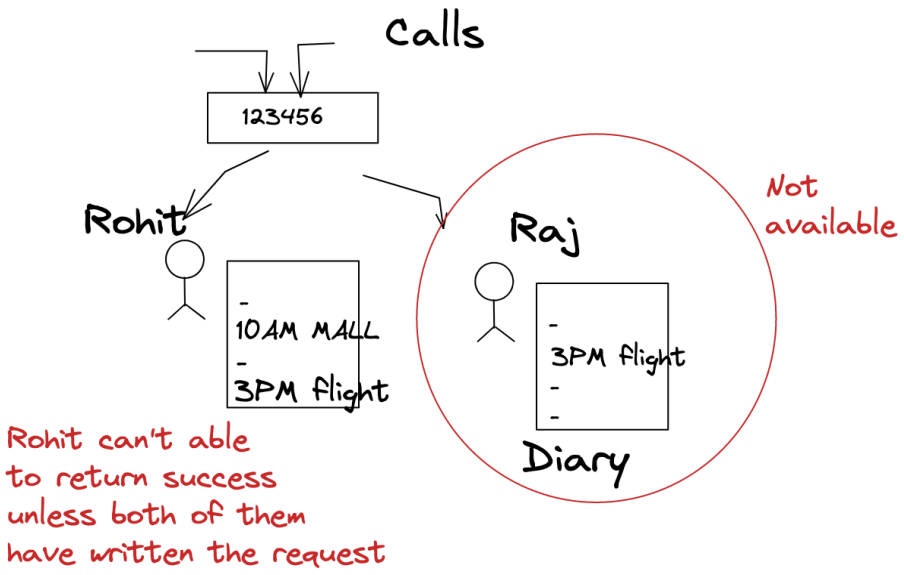
Solution 2:
Whenever a write request comes, both of them write the entry and then return success. In this case, both are consistent.



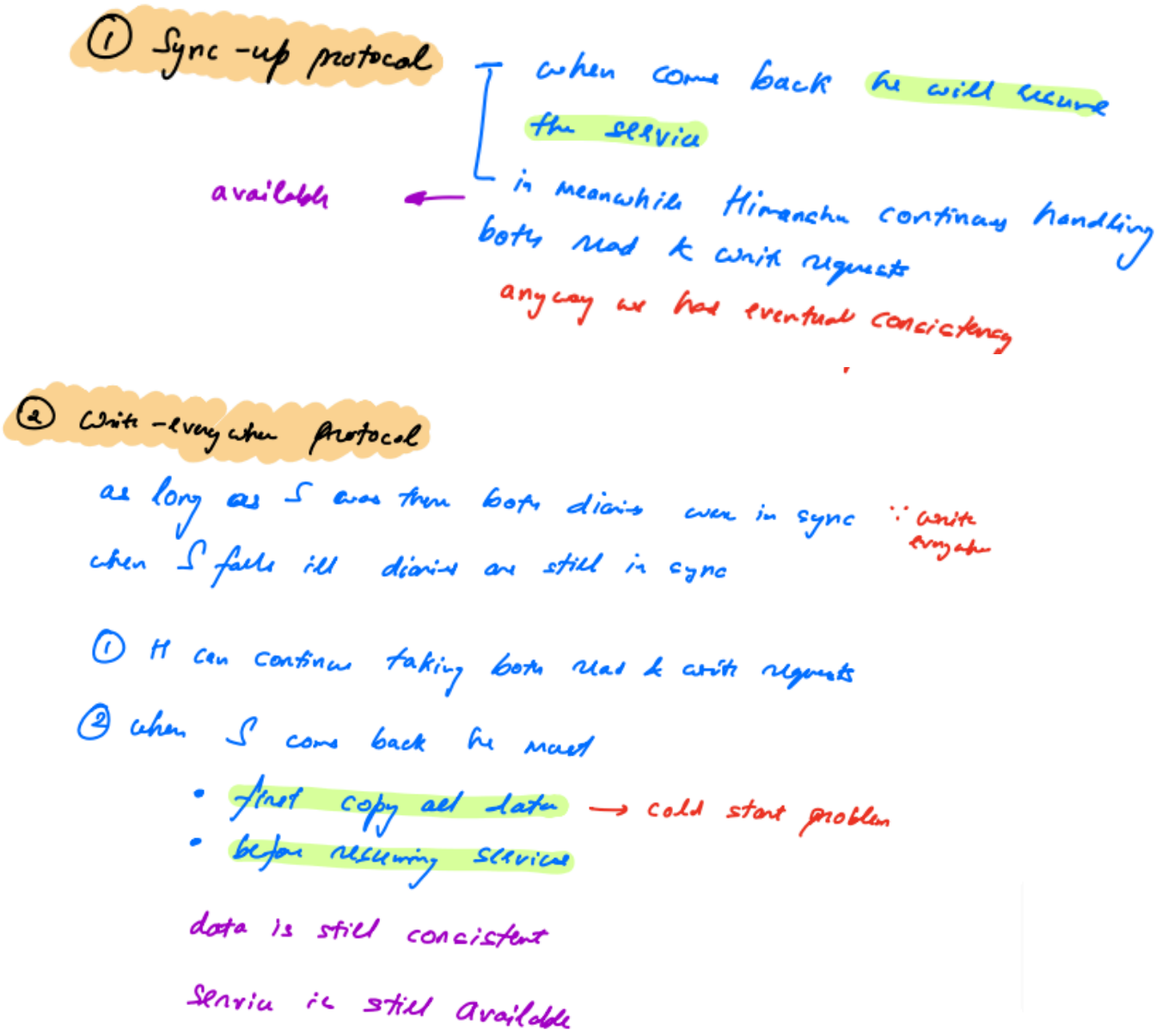


Problem 2: Availability problem.
Now one day, Raj is not there in the office, and a request comes. So because of the previous rule, only when both of them write the entry then only they will return success. Therefore the question is How to return the success now.

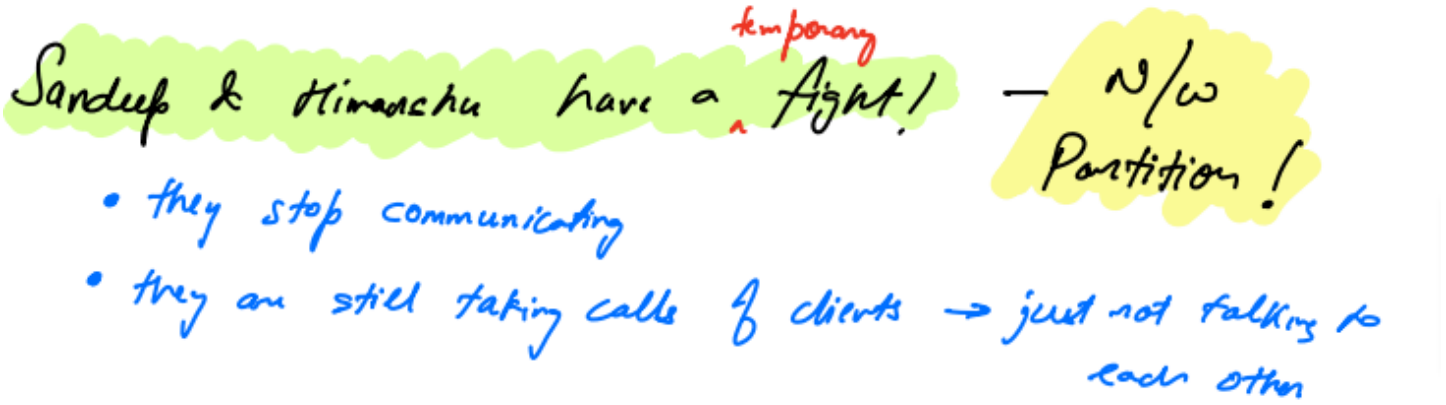




Solution 2:
When the other person is not there to take the entry, then also we will take the entries, but the next day, before resuming, the other person has to ensure they catch up on all entries before marking themselves as active (before starting to take calls).

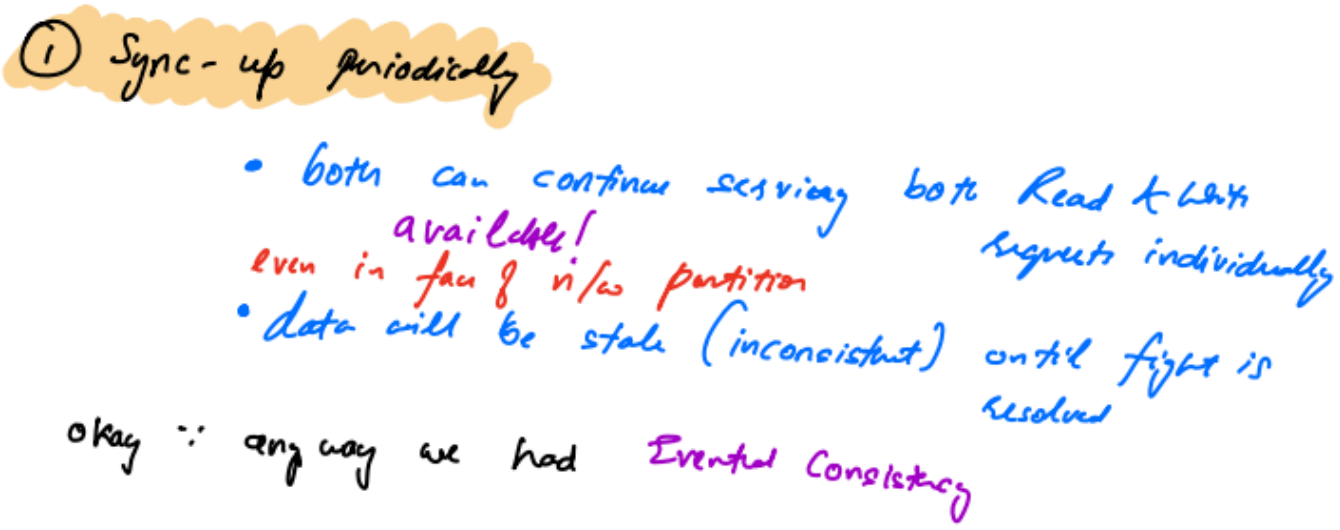


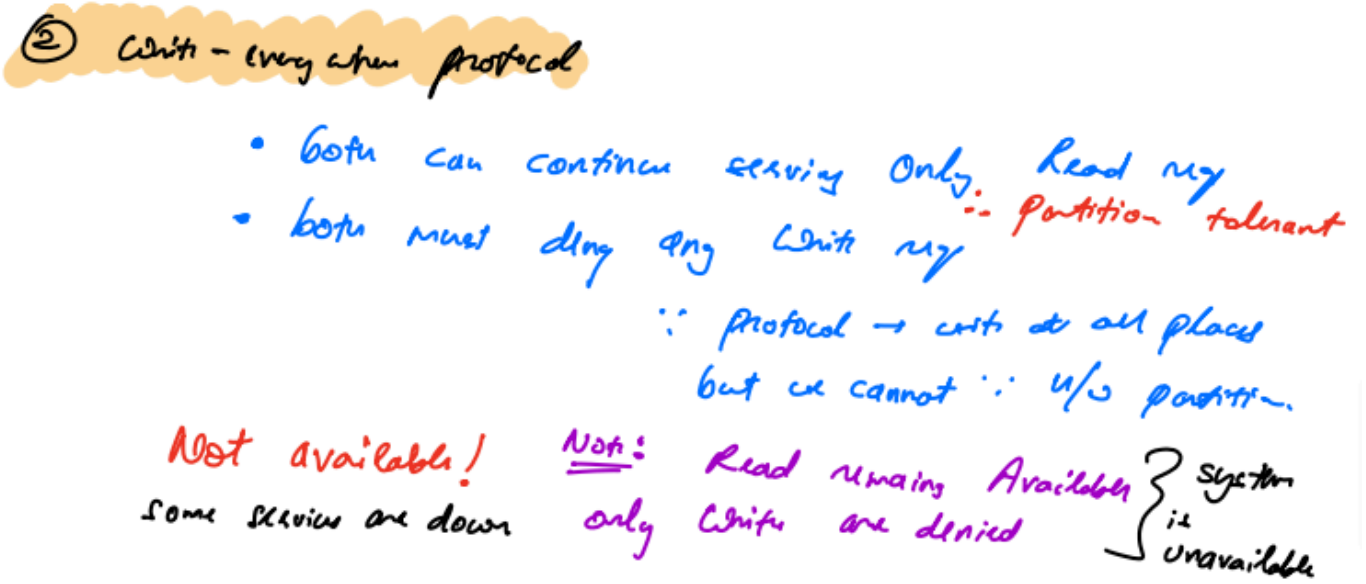
Problem 3: Network Partition.
Imagine someday both Raj and Rohit have a fight and stop talking to each other. Now, if a person X calls Raj to take down a reminder, what should Raj do? Raj cannot tell Rohit to also note down the entry, because they are not talking to each other.



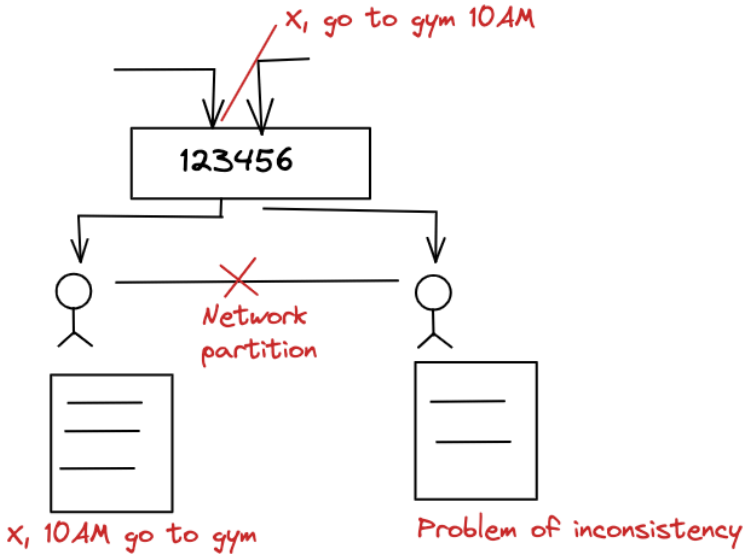
If Raj notes the reminder and returns success to X, then there is an inconsistency issue. [X calls back and the call goes to Rohit who does not have the entry].

If Raj refuses to note the reminder and returns failure to stay consistent, then it's an availability issue. Till Raj and Rohit are not talking to each other, all new reminder requests will fail.

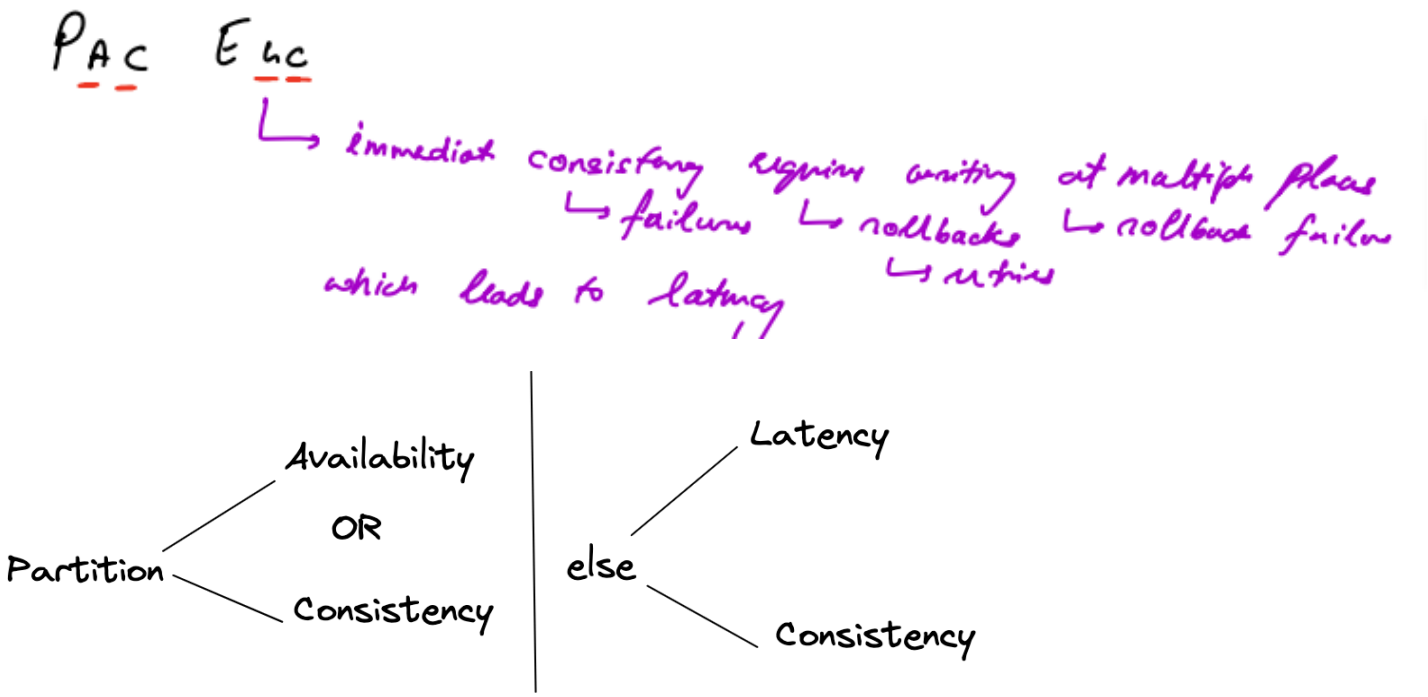




Hence, if there are 2 machines storing the same information but if a network partition happens between them then there is no choice but to choose between Consistency and Availability.



PACELC Theorem:



In the case of network partitioning (P) in a distributed computer system, one has to choose between availability (A) and consistency (C).

But else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C).

Latency is the time taken to process the request and return a response.

So If there is no network partition, we have to choose between extremely low Latency or High consistency. They both compete with each other.

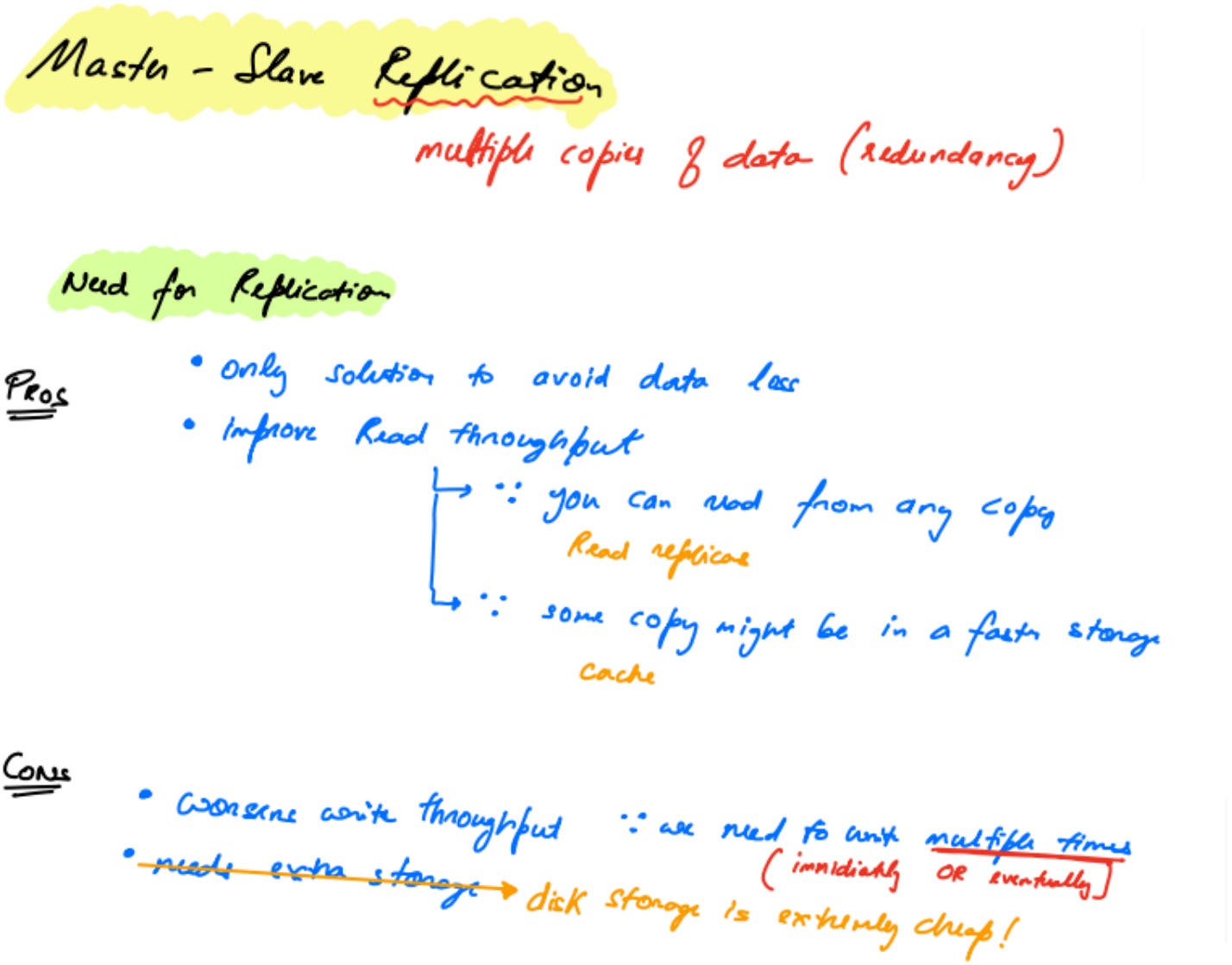
Some Examples of When to choose between Consistency and Availability.

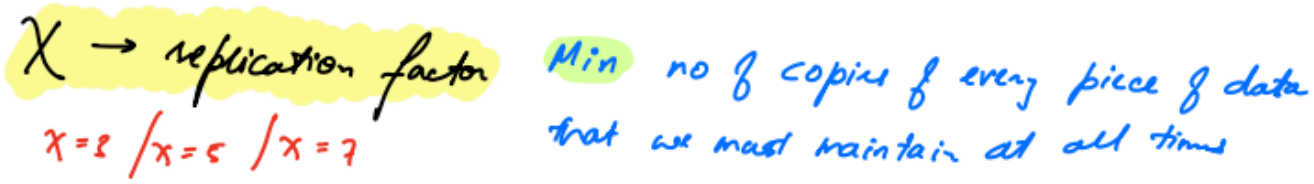
1. In a banking system, Consistency is important.
    a. so we want immediate consistency
    b. but in reality, ATM transactions (and a lot of other banking systems) use eventual consistency
2. In a Facebook news feed-like system, availability is more important than the consistency.
3. For Quora, Availability is more important.
4. For Facebook Messenger, Consistency is even more important than availability because miscommunication can lead to disturbance in human relations.

Extra Readings:

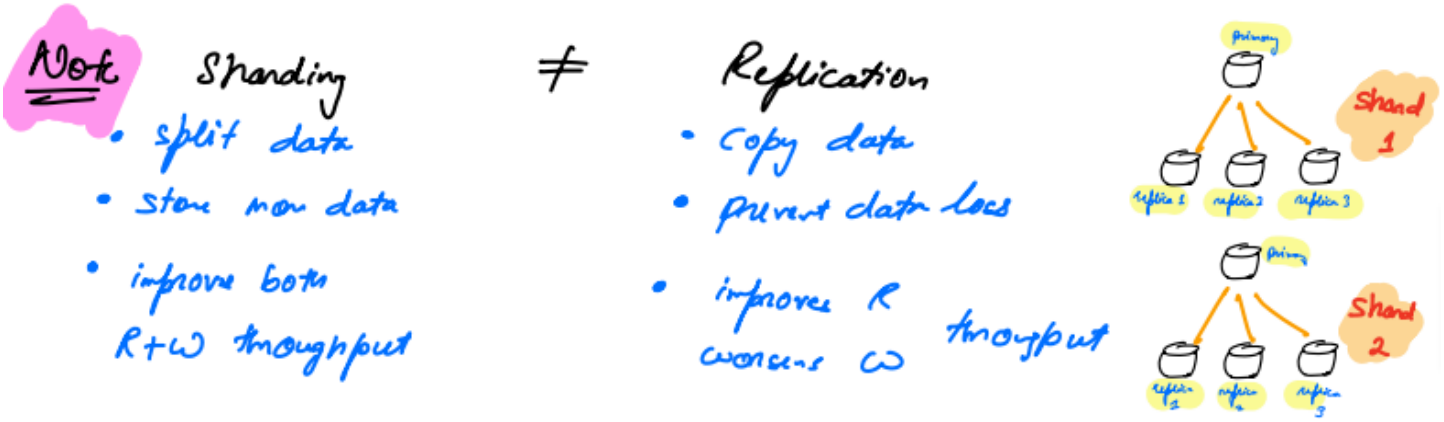
1. https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html
2. Does Google Spanner violate CAP theorem?
    a. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45855.pdf
    b. https://www.youtube.com/watch?v=oeycOVX70aE
    c. https://www.youtube.com/watch?v=LaLT6EC7Trc

Master Slave System:

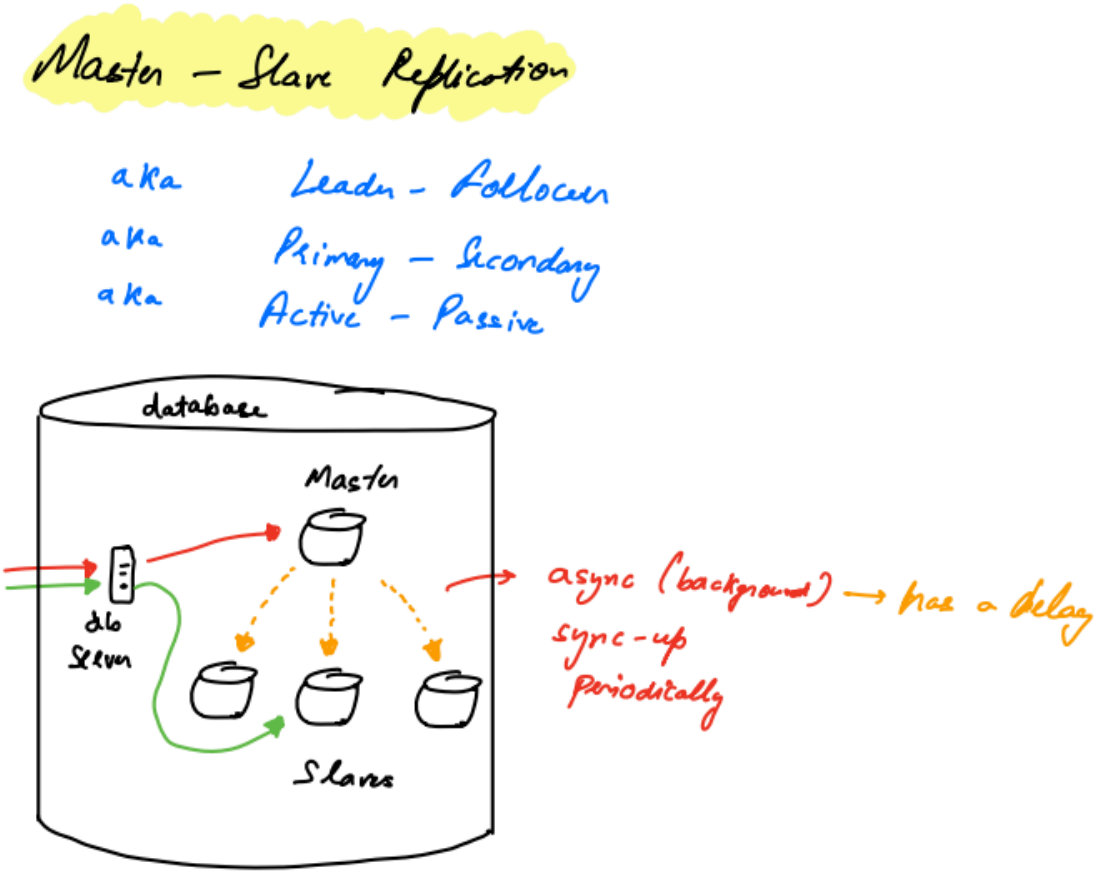




Sharding is not the same as replication

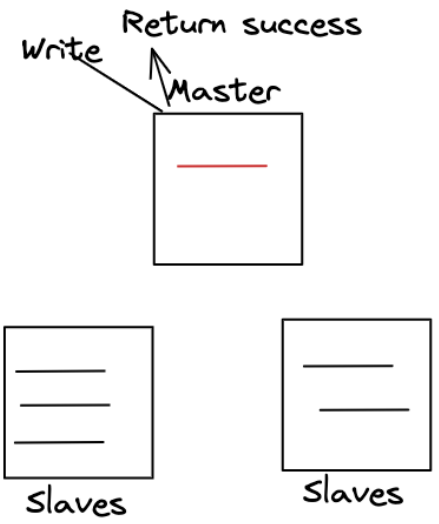


In Master-Slave systems, Exactly one machine is marked as Master, and the rest are called Slaves. Different terms used to describe the master-slave architecture include leader-follower, primary-secondary, and active-passive.



- Data flow is always from master to slave.
- If the master goes down, a slave must be elected as the new master. Once the old master comes back, it will rejoin as a slave.
- If a slave goes down, it does not cause issues.

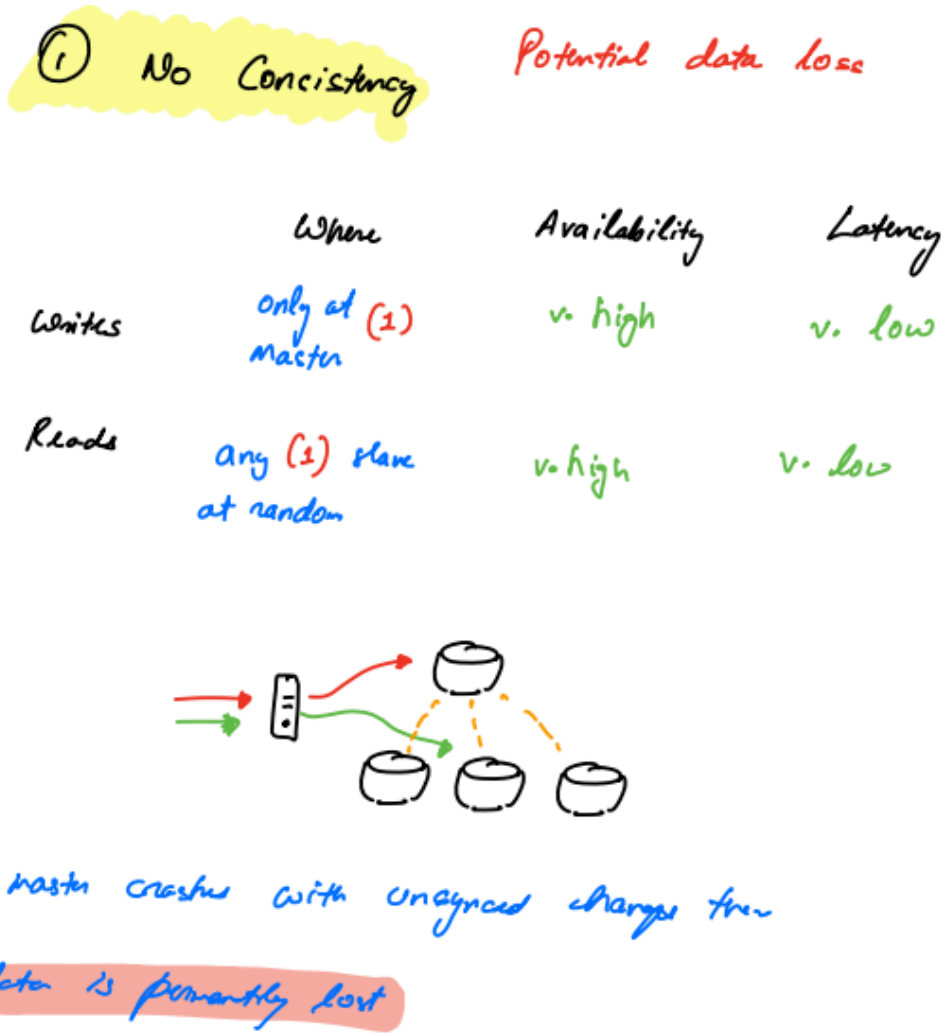
1. **Master Slave systems that are Highly Available and not eventually consistent.**



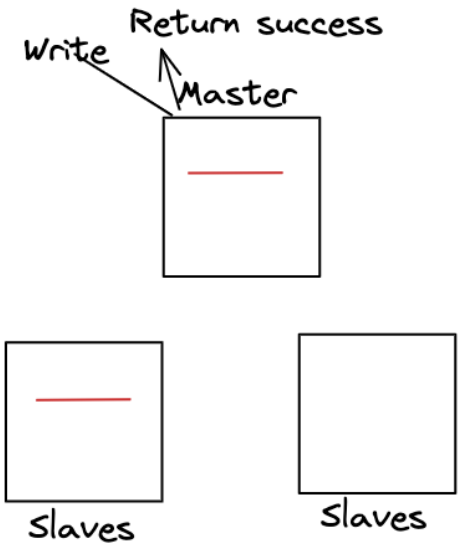
Steps:

- Master system takes the write. If the write is successful, return success.
- Try to sync between slave1 and slave2.

Example: Splunk, where we have a lot of logs statements now, there is so much throughput coming in, we just want to process the logs even if we miss some logs, it's ok.



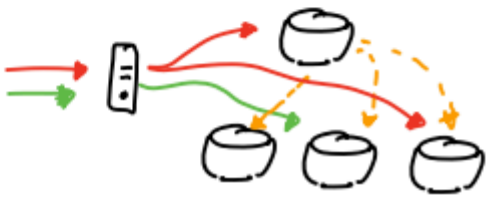
2. Master-Slave systems that are Highly Available and eventually consistent:



Steps:

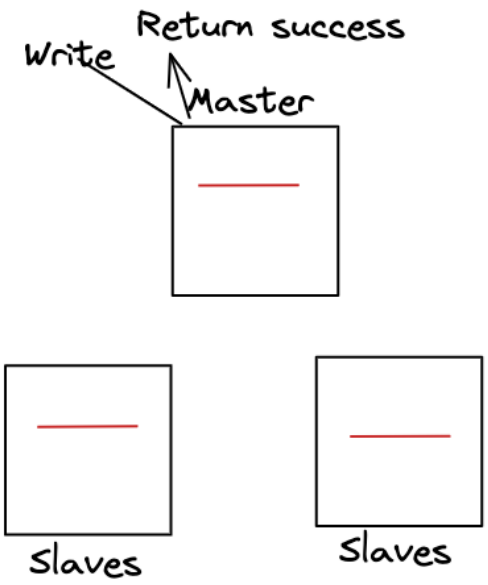
1. The Master system takes the write, and if one slave writes, then success is returned.
2. All slaves sync.

Example: Computing news feed and storing posts, there we don't want the post to be lost; they could be delayed but eventually sync up.

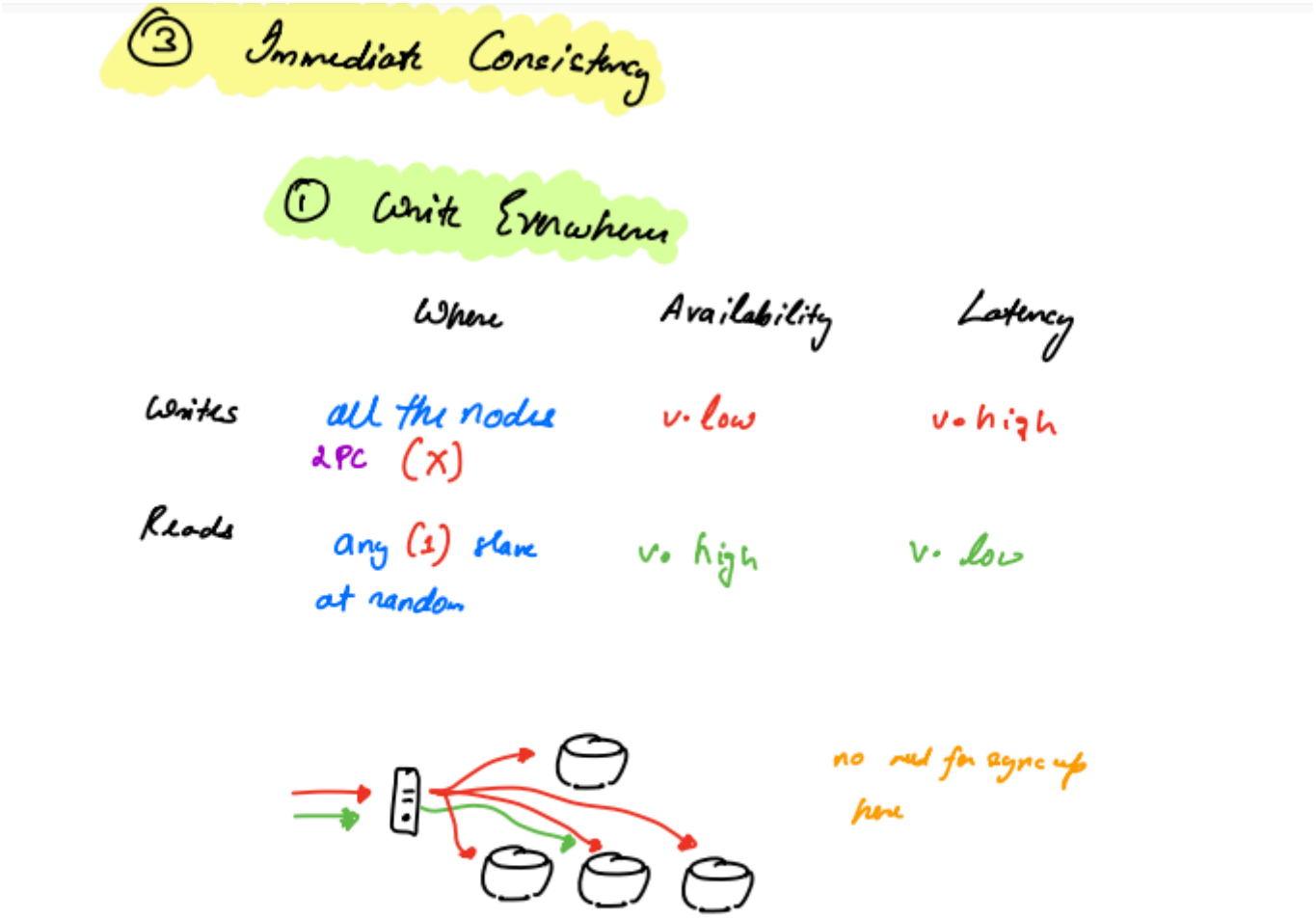


data reads can be stale if slave has not synced up with master recently

3. Master Slave that are Highly Consistent:



Steps:
Master and all slave take the writes, if all have written, then only return success.
Example: The banking system.



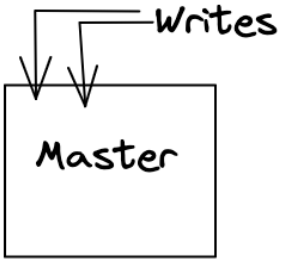
Note: For a write to be successful, all nodes must acknowledge it. If even one replica is down, the write operation must be denied.

In Master-Slave systems,

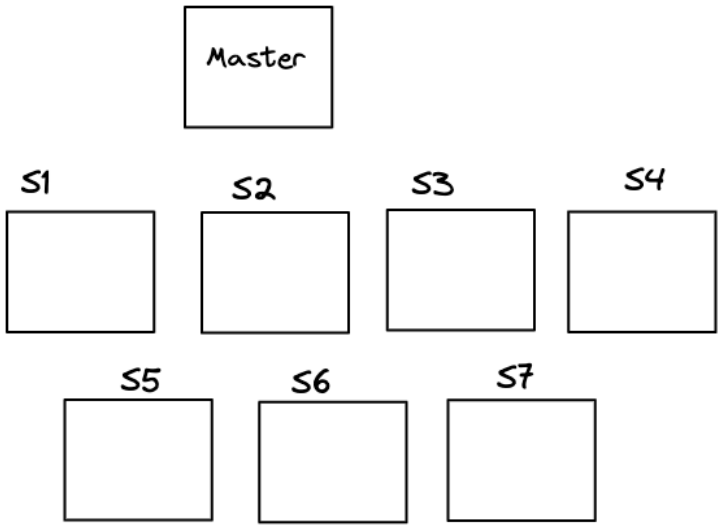
- All writes first come to Master only.
- Reads can go to any of the machines.
- Whenever the Master system dies, a new election of the master will take place based on a different elections algorithm.

Drawbacks of Master-Slave System:

1. A single master can become the bottleneck when there are too many writes.



2. In highly consistent systems, slaves increase which increases the rate of failure and latency also increases.



For example, For highly consistent systems, if there are 1000 slaves, the Master-slave system will not work. We have to do more sharding.

What is a Quorum?

Imagine you and a group of friends are making a decision together, like picking a movie to watch. To make sure the decision is fair and everyone agrees, you decide that at least more than half of the group needs to agree on the movie. This is similar to what a quorum does in a distributed system.

Quorum in Distributed Systems

1. **Group of Computers (Nodes)**:
    - In a distributed system, data or tasks are spread across multiple computers, called nodes.
2. **Making Decisions Together**:
    - When the system needs to make a decision (like updating data), it asks a group of these nodes for their agreement.

3. **Majority Agreement**:
    - A quorum is the minimum number of nodes that must agree to proceed with the decision. This is usually a majority, more than half of the nodes.
    - For example, if you have 5 nodes, a quorum might be 3 nodes. This means at least 3 nodes need to agree for the action to be successful.
4. **Ensuring Consistency**:
    - Quorum helps ensure that the data remains consistent across all nodes.
    - If fewer nodes than the quorum respond, the action (like a write operation) is denied to prevent inconsistencies.

Why is Quorum Important?

- **Fault Tolerance**: If some nodes fail or are unreachable, as long as a quorum of nodes agrees, the system can still function correctly.
- **Data Consistency**: By requiring a majority to agree, it ensures that data is consistent and up-to-date across the system.

Example

Let's say we have a distributed database with 5 nodes, and we set the quorum to 3.

- **Writing Data**:
    - When we want to write data, at least 3 nodes must confirm they received and stored the data. If fewer than 3 nodes confirm, the write operation is denied to prevent data inconsistencies.
- **Reading Data**:
    - When we read data, we might also need confirmation from at least 3 nodes to ensure we're getting the most recent and correct data.

By using a quorum, the system ensures that even if some nodes fail, the remaining nodes can still make reliable decisions and keep the data consistent.

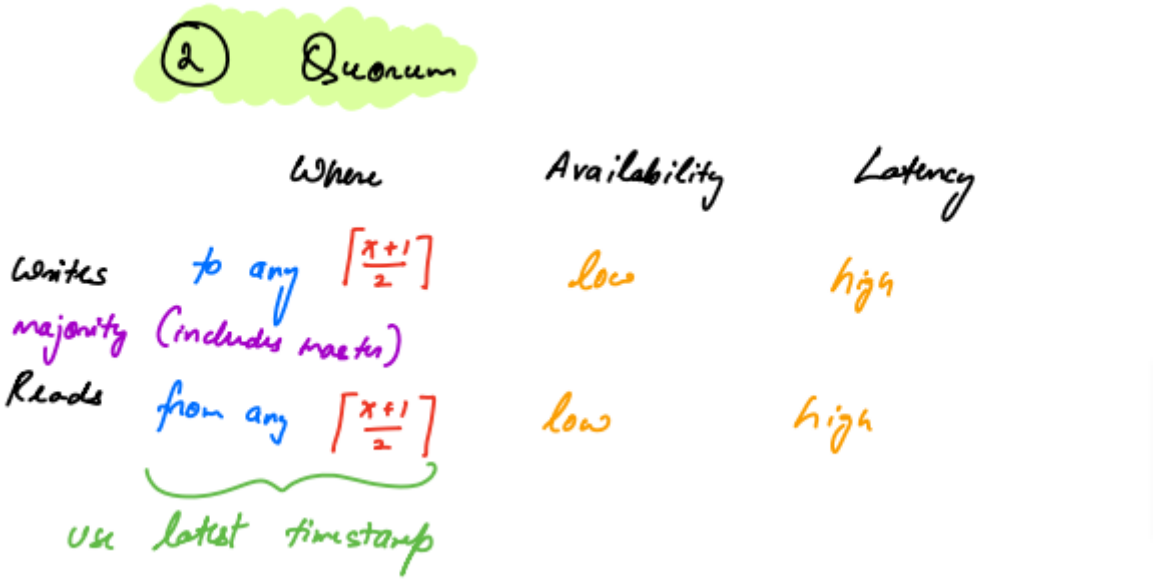
Quorum in Distributed Systems

In distributed systems, especially in databases and consensus algorithms, quorum mechanisms are used to maintain data consistency and ensure that operations like reads and writes are performed correctly despite failures. Here are some key points:

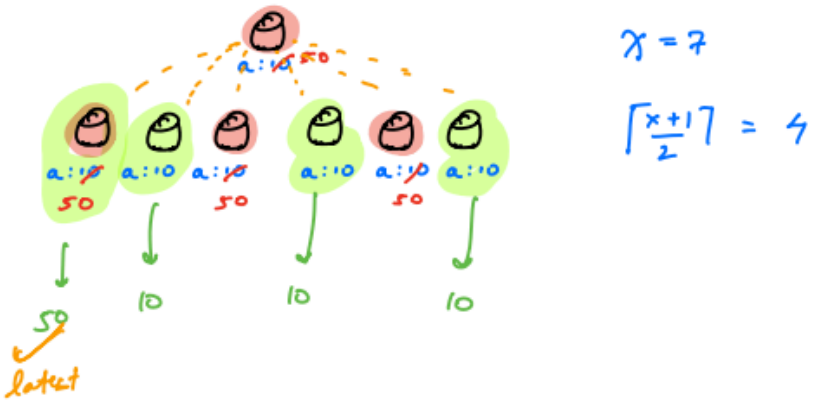
1. **Quorum-Based Replication**:
    - In a distributed database, data is often replicated across multiple nodes to ensure reliability and availability.
    - A quorum system requires that for any read or write operation to be successful, a certain number of nodes (a quorum) must participate.
    - Typical quorum configurations include:
        - **Write quorum (W)**: The minimum number of nodes that must acknowledge a write operation.
        - **Read quorum (R)**: The minimum number of nodes that must respond to a read operation.
        - **Total nodes (N)**: The total number of nodes in the system.

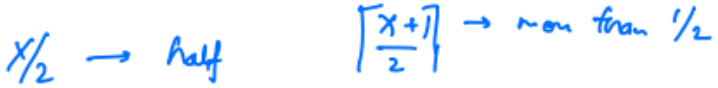
- The relationship between these is often described by the formula W+R>NW + R > NW+R>N, ensuring that there is an overlap between the sets of nodes involved in read and write operations, which helps in maintaining consistency.

2. **Consensus Algorithms**:
   - Consensus algorithms like Paxos and Raft use quorum mechanisms to agree on the state of the system.
   - In these algorithms, nodes propose values and must receive acknowledgments from a majority (quorum) of nodes to reach consensus.
   - This ensures that even if some nodes fail, the system can still function correctly as long as a quorum is available.
3. **Fault Tolerance**:
   - Quorum systems are designed to tolerate failures. For example, in a system with 5 nodes (N=5), setting a write quorum of 3 (W=3) and a read quorum of 3 (R=3) ensures that the system can tolerate up to 2 node failures while still maintaining consistency.



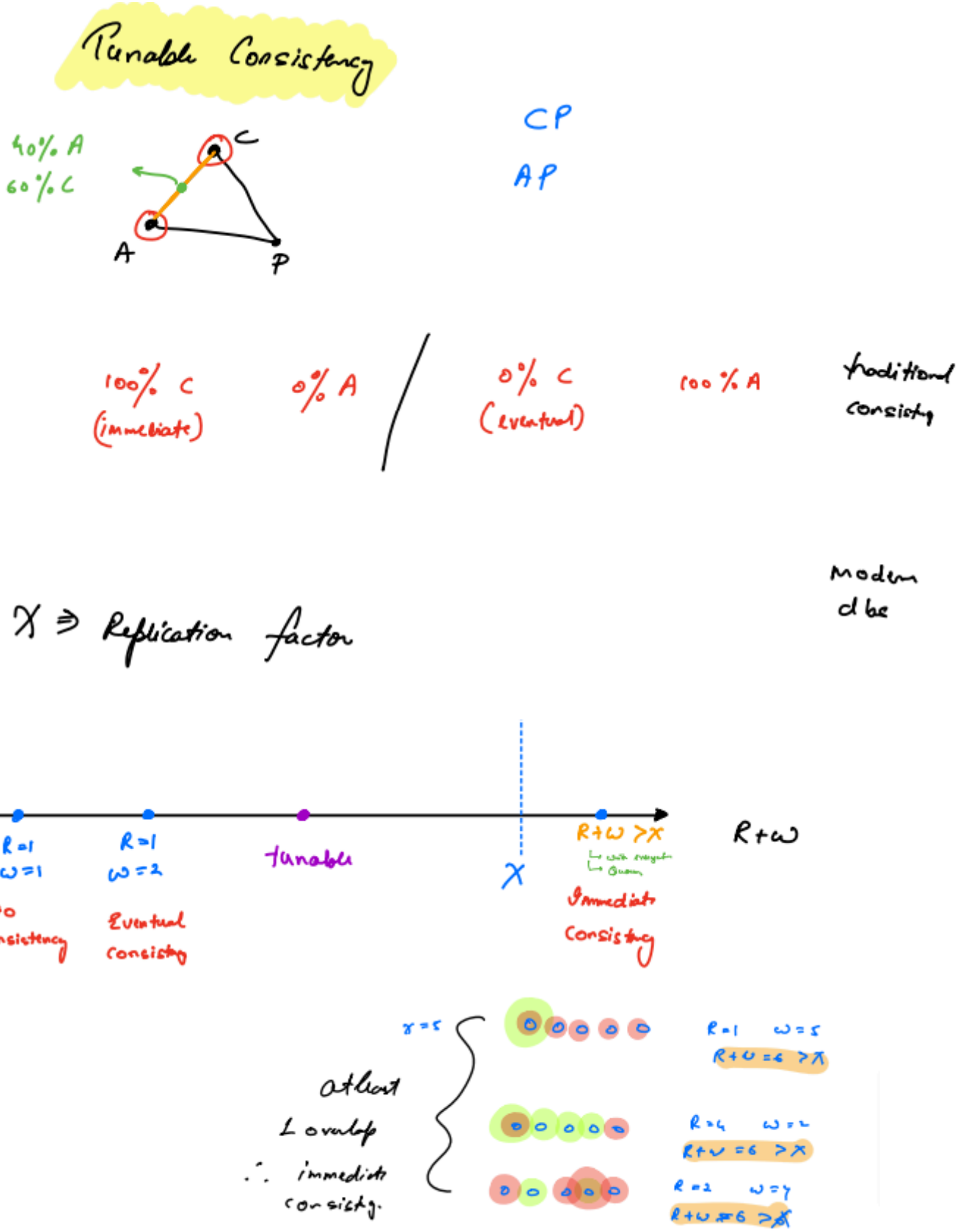
If you perform both read and write operations on more than half of the servers, there is guaranteed to be an overlap. At least one server will be involved in both reading and writing, ensuring that this server can provide the latest data.





Tunable consistency

It refers to the ability of a distributed database system to adjust the level of consistency based on the specific needs of an application. This flexibility allows developers to balance between consistency, availability, and performance by configuring the system to behave in a way that best suits their requirements.



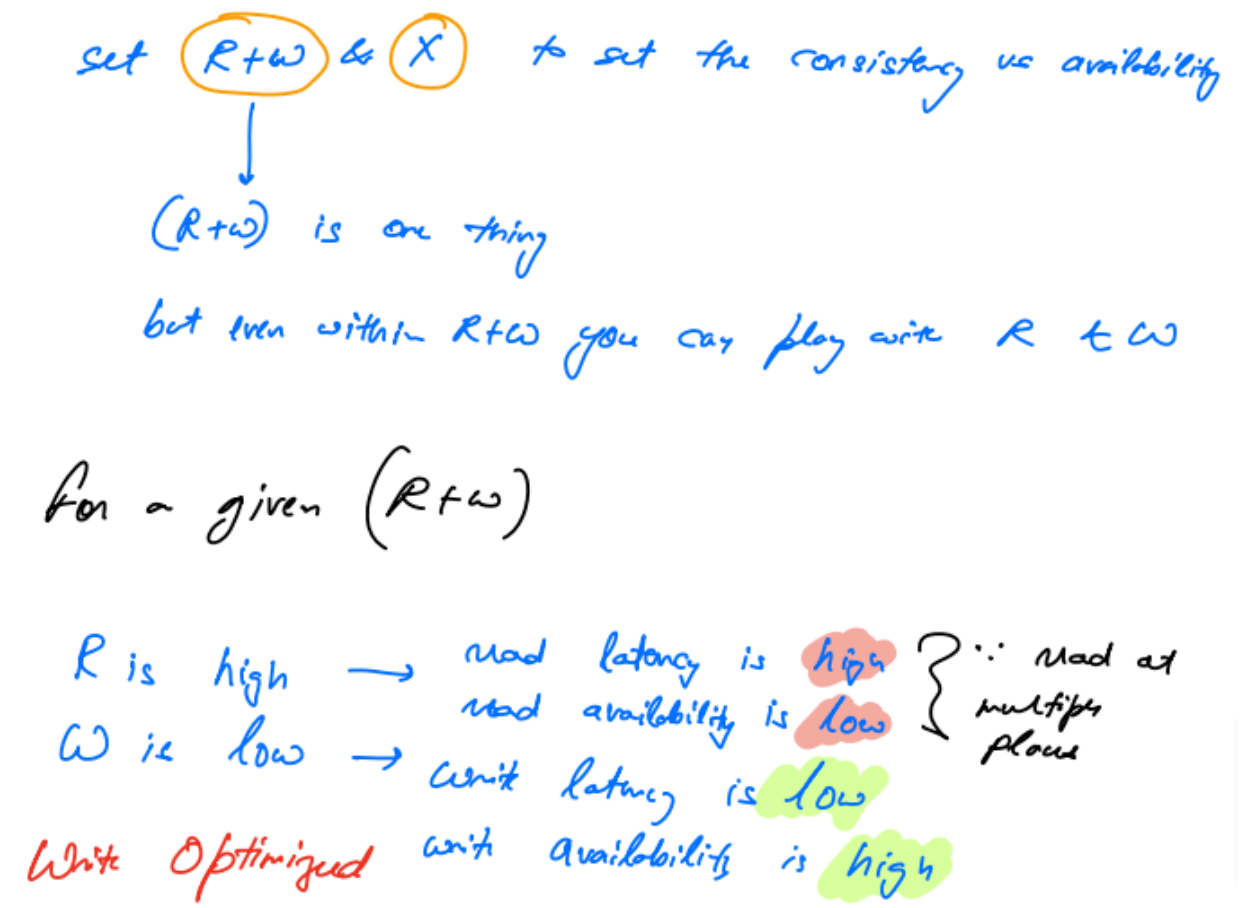
Key Concepts of Tunable Consistency

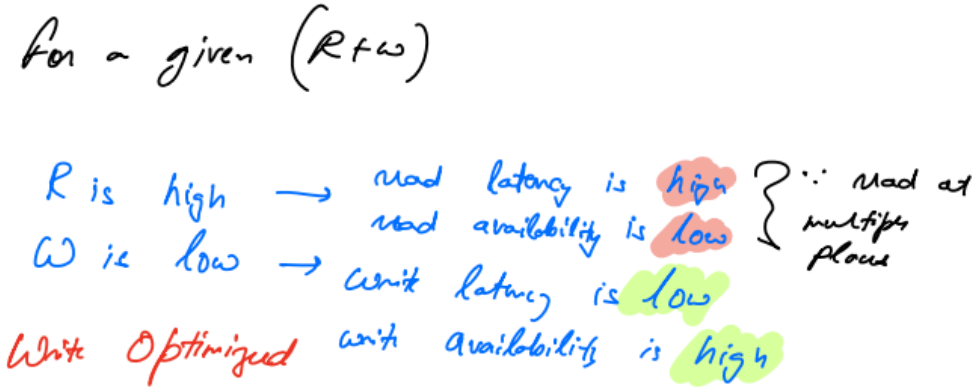
1. **Consistency Levels**:
    - **Strong Consistency**: Ensures that all read operations return the most recent write. This is like always having the latest version of a document.
    - **Eventual Consistency**: Guarantees that if no new updates are made, all reads will eventually return the same value, though it might take some time.
    - **Causal Consistency**: Ensures that reads and writes that are causally related are seen by all processes in the same order.
    - **Read-Your-Writes Consistency**: Guarantees that a read operation will always see the result of previous write operations by the same client.
2. **Configurable Parameters**:
    - **R (Read Quorum)**: The number of nodes that must respond to a read request.
    - **W (Write Quorum)**: The number of nodes that must acknowledge a write request.
    - **N (Total Nodes)**: The total number of nodes that hold a replica of the data.

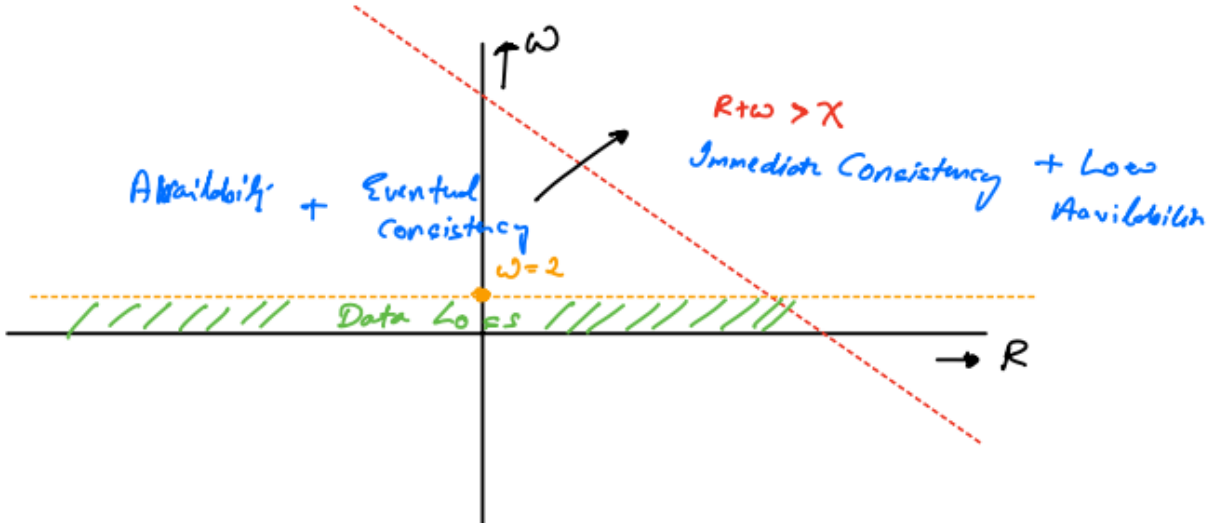
How Tunable Consistency Works

By adjusting the values of R, W, and N, you can control the consistency level:

- **Strong Consistency**: Set high values for both R and W (e.g., R + W > N). This ensures that there is always an overlap between read and write operations, guaranteeing the latest data is read.
- **Eventual Consistency**: Set lower values for R and W, which allows for faster reads and writes at the cost of not always having the most recent data immediately available.
- **Balanced Approach**: Adjust R and W to meet specific use cases where a balance between consistency and performance is needed.







Example Scenario

Suppose you have a distributed database with 5 nodes (N = 5):

- **Strong Consistency**:
  - Set R = 3 and W = 3. This ensures that at least 3 nodes are involved in every read and write, guaranteeing overlap and up-to-date data.
- **Eventual Consistency**:
  - Set R = 1 and W = 1. This allows operations to be faster, but there is no guarantee that the data read is the latest immediately after a write.
- **Balanced Consistency**:
  - Set R = 2 and W = 2. This provides a middle ground, offering a reasonable balance between consistency and performance.

Simple Example: Pizza Delivery Orders

Imagine you run a pizza delivery service with five stores spread across a city. Each store has a copy of the customer orders database. This is similar to how data is replicated across multiple nodes in a distributed system.

Scenario: Updating Order Status

When a customer places an order or updates their order (like changing the delivery address), this information needs to be updated in the database. Later, when the customer checks the status of their order, they should see the most recent information.

Tunable Consistency Settings

You can control how many stores need to acknowledge the update and how many stores need to respond to a status check. Let's define the terms:

- **N (Total Nodes)**: The total number of stores (in this case, 5).
- **W (Write Quorum)**: The number of stores that must acknowledge an update for it to be considered successful.
- **R (Read Quorum)**: The number of stores that must respond to a status check.

Strong Consistency

- **W = 3, R = 3**
- **Explanation**: For an order update to be successful, at least 3 stores must confirm they've received the update. When checking the status, at least 3 stores must respond.
- **Result**: Ensures that any order status check will always see the most recent update. If a customer changes their address, and then checks the status, they will see the new address because there's enough overlap between the stores that acknowledged the update and the stores responding to the status check.

Eventual Consistency

- **W = 1, R = 1**
- **Explanation**: For an order update to be successful, only 1 store needs to confirm the update. When checking the status, only 1 store needs to respond.
- **Result**: Updates are fast and require less communication between stores, but the customer might not see the most recent update immediately. If the address change hasn't propagated to the store that responds to the status check, the customer might see the old address.

Balanced Consistency

- **W = 2, R = 2**
- **Explanation**: For an order update to be successful, at least 2 stores must confirm the update. When checking the status, at least 2 stores must respond.
- **Result**: This setup offers a compromise. It provides better consistency than eventual consistency and is faster than strong consistency. The customer is likely to see the most recent update relatively quickly, though there may be a slight delay.

SQL vs NoSQL - 25 Jun

Agenda:

Introduction to SQL Database and Normalisation
ACID transactions

Introduction to SQL Databases and Normalization

There are many types of Relational Databases (SQL DBs) such as PostgreSQL, SQLite, MySQL, SQL Server, OracleDB, and IBM DB2. SQL, which stands for Structured Query Language, is used to query these databases. There are other query languages for relational databases too, like GraphQL.

Note: When you design a service, your default choice should be an SQL database if it is for a small-scale application, which means it deals with a smaller amount of data and fewer requests per second. Specifically, if one server can handle the data, which means it has 10 terabytes (TB) or less of storage, and handles 100-1000 read requests and 10-100 write requests per second, then you should use SQL.

For high-scale applications, start with SQL and see if it meets your needs. If it doesn't, figure out why and then decide what else to use. When you choose SQL, no explanation is needed. However, if you choose NoSQL, you need to explain why.

Pros of SQL Databases

- **Extremely Mature**: SQL databases have been around since the 1980s, with their roots tracing back to the 1960s. This long history means they are well-developed and reliable.
- **Highly Feature-Rich**: SQL databases are like a "Jack of all trades," meaning they offer a wide range of functionalities.

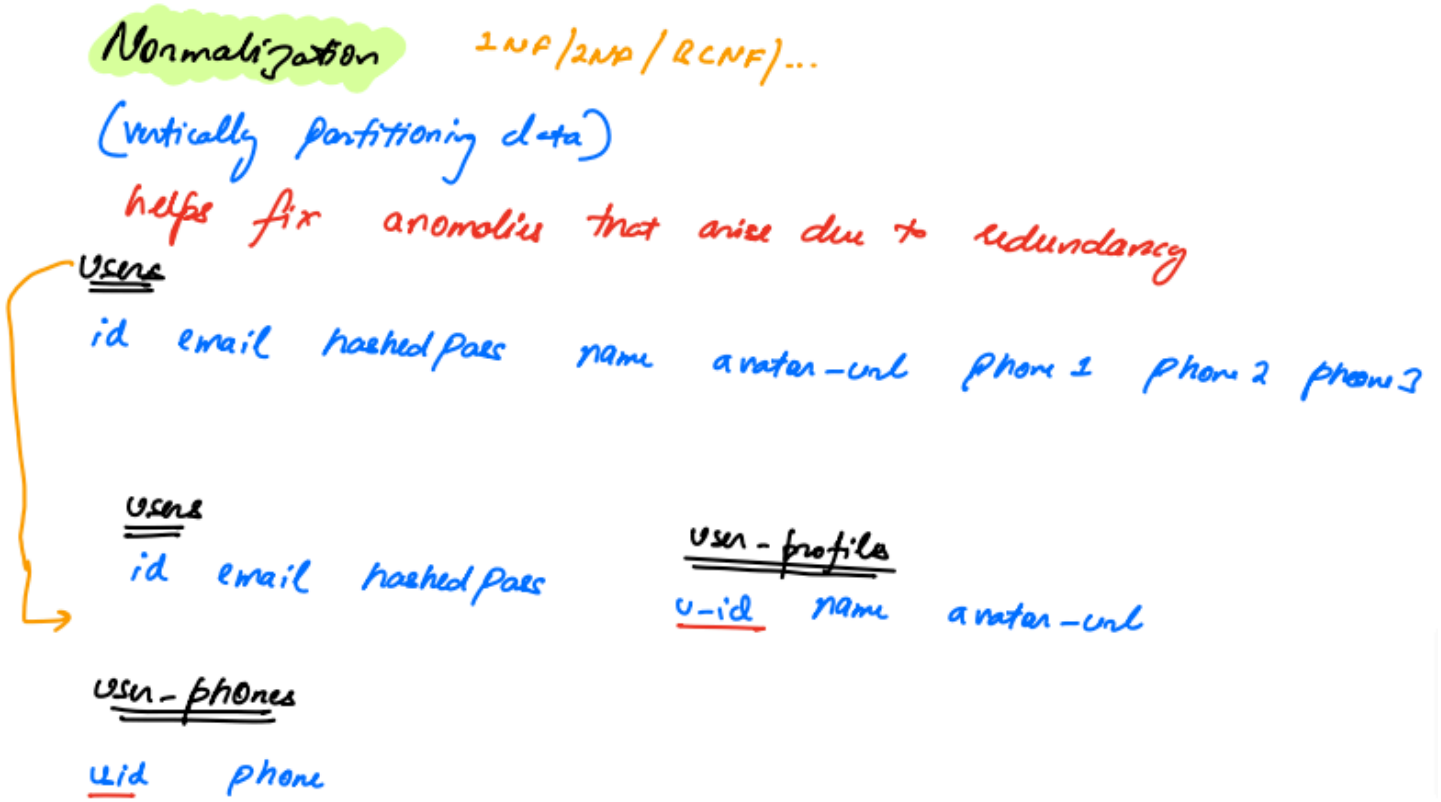
Features like

Normalization: Features like normalization (1NF, 2NF, BCNF) allow you to organize your data efficiently. Normalization involves structuring the database to reduce redundancy and improve data integrity by dividing the data into different tables and defining relationships between them.

1. Understanding Normalization in SQL

To understand normalization, let's look at an example. Imagine we have a table with the following columns: `id`, `email`, `hashedPass`, `name`, `avatar_url`, `phone1`, `phone2`, and `phone3`. This table is not normalized. To normalize it, we can divide it into three tables: `user`, `user_profile`, and `user_phone`.

- **User Table**: Contains `id`, `email`, and `hashedPass`.
- **User Profile Table**: Contains `uid`, `name`, and `avatar_url`.
- **User Phone Table**: Contains `uid`, `phone1`, `phone2`, and `phone3`.



Normalization helps to avoid various anomalies in SQL databases:

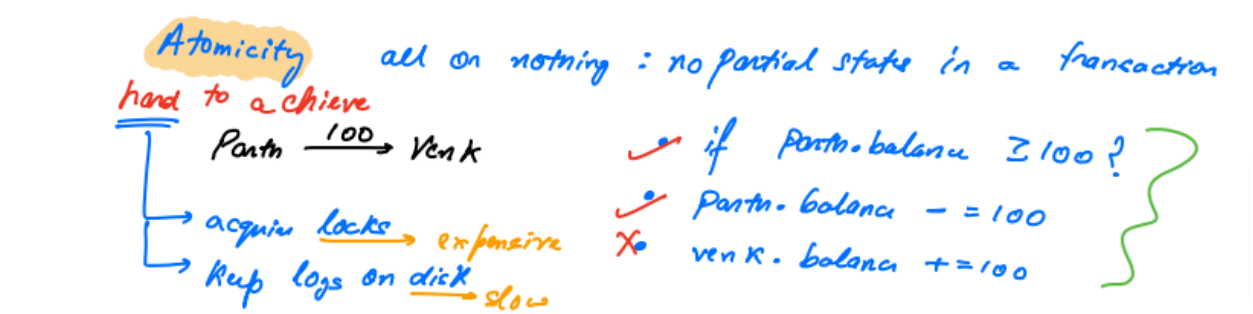
1. **Insertion Anomaly**: Without normalization, you might be forced to insert incomplete data. For instance, if a user doesn't have a phone number, you still have to insert a null value or a placeholder.
2. **Update Anomaly**: If a user's email changes, you would need to update it in multiple places if the data is not normalized, which can lead to inconsistencies.
3. **Deletion Anomaly**: If you delete a user who has multiple phone numbers, you might lose all related phone numbers as well, even if you wanted to keep that information.

By normalizing the table into `user`, `user_profile`, and `user_phone`, you can avoid these anomalies and ensure your data is more consistent and easier to manage.

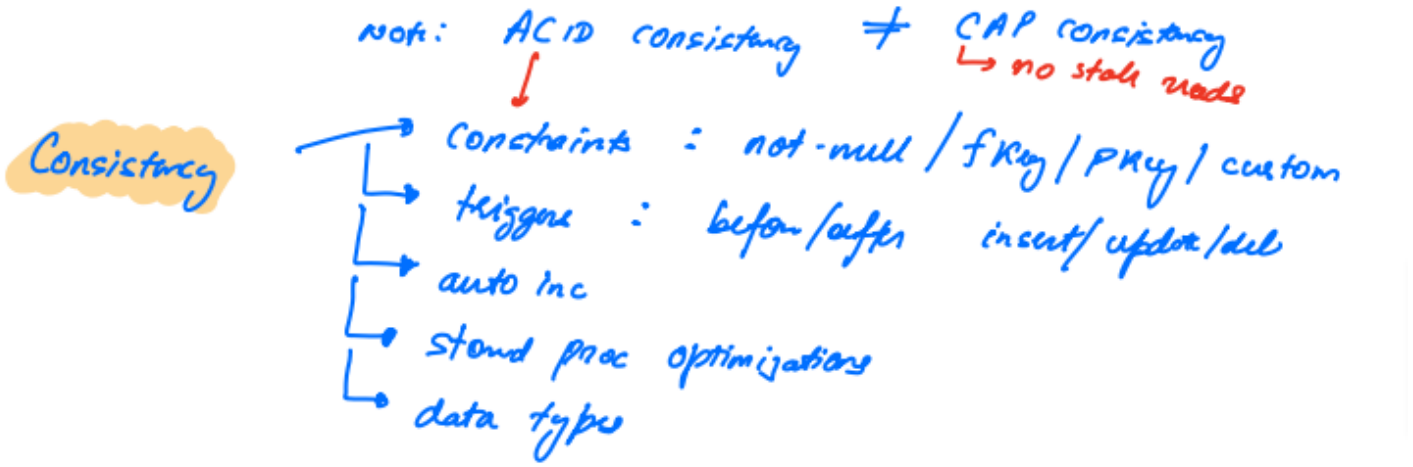
2. ACID Transactions in SQL Databases

ACID stands for Atomicity, Consistency, Isolation, and Durability. These are the key properties that ensure reliable transactions in a database:

1. **Atomicity**: This ensures that all parts of a transaction are completed successfully. If any part of the transaction fails, the entire transaction is rolled back, and the database remains unchanged. For example, if you are transferring money between two accounts, both the debit and credit operations must succeed, or neither will happen.



2. **Consistency**: This ensures that a transaction brings the database from one valid state to another. Data integrity constraints must be maintained before and after the transaction. For example, if a transaction violates a constraint (like a unique key), it will not be committed.



3. **Isolation**: This ensures that transactions are executed independently of one another. The intermediate state of a transaction is invisible to other transactions. This prevents "dirty reads," where one transaction reads data written by another uncommitted transaction.
4. **Durability**: This ensures that once a transaction has been committed, it will remain so, even in the event of a system failure. Committed transactions are saved to a permanent storage medium, ensuring they are not lost.

Together, these properties help maintain the integrity, reliability, and correctness of data in SQL databases, making them suitable for critical applications where data consistency and reliability are paramount.

3. Defined Schema

Any data inside an SQL database has a well-defined datatype. This is important because it provides validation, tells us the size of the data, and shows how to store it. We have a well-defined structure with the following parts:

- **Table:** A collection of data.
- **Columns:** Different fields in each table.
- **Datatype for each column:** Defines the kind of data each column can hold.
- **Other constraints:** Rules like primary key and foreign key.

Notes:

1. **Varchar in PostgreSQL:** When you define a varchar in PostgreSQL, it treats it as a text type, so the exact value inside varchar doesn't matter.
2. **Schema Rigidity in SQL Databases:** The schema is rigid, meaning it is difficult to change. You can change it using `ALTER TABLE`, but this is costly. Adding or removing a column requires rebuilding the whole schema, which can be very expensive if you have millions of records.

Shortcomings (Cons) of SQL Databases at High Scale (More Than One Server)

1. **Normalization:**
    - Normalization involves splitting data into multiple tables to reduce redundancy. This requires joining tables to get all the data, which is very hard to achieve across multiple servers.

    - Some databases, like PostgreSQL, provide extensions to join tables across servers, but SQL databases do not have a default feature for this. We have to use third-party libraries, and even then, joins across servers perform very poorly.
    - Note: Client UI (almost always) needs de-normalized data.
2. **ACID:**
    - Ensuring locks and maintaining atomicity across multiple servers is extremely hard and very slow. This can be achieved using Two-Phase Commit, but it is complex and inefficient.
3. **Difficult to Change (Rigid) Schema:**
    - SQL schemas are rigid, which is a problem when dealing with inherently unstructured data.

Example: Amazon

Imagine Amazon, which has thousands of product categories and 100 million products.

1. we used one giant table with all columns like id, name, price, vendor, ram, etc., and 1000 attributes for 1000 product categories, most columns in any row would remain null. This is problematic because:

- **Nulls are bad:**
    - **Ambiguous:** Nulls can indicate an error value or a missing value, which is unclear.
    - **Weird Comparisons:** Comparing nulls is tricky because null is not equal to another null value.
    - **Waste of Space:** Even if a value is null, space is still allocated according to the datatype of the column for each row.

Distributed Tables vs. Large Table

If we distribute the table instead of having one large table, we would need to create a different table for each product category. This approach has several problems:

1. **Too Many Tables:**
    - We would end up with thousands of tables, and SQL is not optimized for managing a large schema with many tables. It is optimized for handling large amounts of data within a reasonable number of tables.
2. **Too Many Joins:**
    - To perform queries like finding all products with a price greater than 100 rupees, we would need to join thousands of tables. This is almost impossible because having 1000 joins is like having 1000 for-loops, which is extremely inefficient.

NoSQL Databases

NoSQL databases became popular in the 1980s when very large companies like Google and Amazon emerged. These "web-scale" companies have billions of users, millions of requests per second, and petabytes of data. They realized that SQL databases are not suitable for large-scale applications, leading Google to develop the first NoSQL database, Bigtable.

Key Features of NoSQL Databases:

1. **Denormalized Data Storage:**
    - NoSQL databases can store unstructured and semi-structured data, unlike SQL databases that require structured data.
2. **Built-in Sharding Support:**

    - NoSQL databases support sharding out of the box, making it easier to distribute data across multiple servers.
    - In SQL databases, sharding is not natively supported, so developers have to:
        - Choose a sharding key.
        - Manually write code to shard the data.
        - Manually write code to distribute the queries.

Note:

- Nowadays, there are many managed database services like AWS RDS for SQL databases, but these services cost money.
- NoSQL databases are designed for horizontal scaling, distributing data across multiple servers. In contrast, SQL databases typically scale vertically, adding more power to a single server.

SQL vs NoSQL - 27 Jun

Agenda:

Types of NoSQL databases

Important Links

Google Spanner vs CAP theorem

1. https://www.youtube.com/watch?v=oeycOVX70aE
2. https://www.youtube.com/watch?v=LaLT6EC7Trc
3. https://www.youtube.com/watch?v=iKQhPwbzzxU
4. https://www.youtube.com/watch?v=IFbydfGV2lQ

Choosing the correct NoSQL DB

https://www.ml4devs.com/articles/datastore-choices-sql-vs-nosql-database/

Cassandra

1. **Try online:** https://jbcodeforce.github.io/db-play/
2. **Introduction:** https://cassandra.apache.org/_/cassandra-basics.html
3. **Case Studies:** https://cassandra.apache.org/_/case-studies.html
4. **Architecture - Overview:** https://cassandra.apache.org/doc/stable/cassandra/architecture/overview.html
5. **Architecture - Guarantees:** https://cassandra.apache.org/doc/stable/cassandra/architecture/guarantees.html
6. **Data Modeling:** https://cassandra.apache.org/doc/stable/cassandra/data_modeling/intro.html

MongoDB

1. **Try online:** https://mongoplayground.net/
2. **Tutorial:** https://www.mongodb.com/docs/manual/tutorial/getting-started/
3. **Docs:** https://www.mongodb.com/docs/manual/

Redis

1. **Try online:** https://onecompiler.com/redis
2. **Tutorial**: https://redis.io/university/
3. **Docs**: https://redis.io/docs/latest/

NoSQL types - class summary
https://docs.google.com/document/d/1CfwsMMw8NTn2zavVokiYG08EH648Gsd1zJ7aAM7NvS4/edit?tab=t.0#heading=h.t1dkjqxokvsp

Types of NoSQL Databases

- **Key-Value db**: Stores data as a collection of key-value pairs.
- **Document db**: Stores data as documents, often using JSON or XML.
- **Wide-Column db (column-family)**: Stores data in tables, rows, and columns, but columns can vary by row.
- **File Storage db**: Stores data as files, similar to how data is stored on a file system.
- **Graph db**: Stores data as nodes, edges, and properties to represent and query relationships.
- **Vector db**: Stores data in a format optimized for handling vector calculations and operations.
- **Special db**: Designed for specific use cases or applications.
- **In-Memory db**: Stores data in RAM for fast access.
- **On-Disk db**: Stores data on disk drives for persistence.
- **Time-Series db**: Optimized for storing and querying time-stamped data.
- **Object-Oriented**: Stores data as objects, similar to how data is represented in object-oriented programming.
- **Hybrid**: Combines features from multiple types of databases.

Note:

1. All modern databases are hybrid. For example, PostgreSQL is primarily a relational database but also supports document storage with JSON. It also includes features of special databases and object-oriented databases.
2. SQL databases are versatile, offering a wide range of features, but they don't excel in any one area. In contrast, NoSQL databases are designed to be highly efficient in specific tasks. They achieve their speed by focusing on one particular function and avoiding the complexity of additional features. For example, Key-Value databases do not support joins, transactions, searches, or normalization, which contributes to their fast performance.

Key-Value DB

Key-Value databases, such as Redis, Memcached, and DynamoDB (AWS), are designed for fast, efficient storage and retrieval of data. They work like in-memory hashmaps, storing keys and values as strings. Redis and Memcached are both open source, while DynamoDB is a managed service provided by AWS. These databases excel in scenarios where rapid data access is crucial, such as caching, session management, and real-time analytics.

Basic Operations In a Key-Value database:

get(key): It returns the value associated with the key or an error if the key doesn't exist

set(key, value): It stores the key-value pair and returns success or an error;

delete(key): It delete and return Acknowledge or error.

contains(key): I tchecks if a key exists and returns true or false.

Redis ->

Redis, a popular Key-Value database, is known for its exceptional speed. It's coded in C and operates as a single-threaded process. Despite the common belief that multi-threading is always better, Redis uses a single-threaded model to avoid the complexities and overhead associated with thread management and locking. It leverages asynchronous I/O operations, similar to async-await and promises, allowing it to perform context switching efficiently and handle numerous tasks simultaneously without the need for multiple threads.

Redis is an in-memory database, meaning it stores data primarily in RAM, which provides incredibly fast access times. However, this also means it requires substantial memory resources. To ensure data durability, Redis can be configured to persist data to disk using a Write-Ahead Log (WAL) file, allowing it to save changes and recover from failures. Redis's performance is impressive, capable of handling up to 100,000 requests per second per server, making it a powerful tool for applications requiring high-speed data operations. It can also handle complex data structure for values.

Normally, key-value databases don't handle complex data structures, but Redis is an exception. It supports advanced data types like arrays, sorted sets, bloom filters, and JSON.

On a multi-core system, Redis can utilize all cores by running multiple Redis instances, with each instance running on a separate CPU core.

For example, if we want to run Redis on 100 servers, instead of buying 100 single-core systems, we can buy 25 systems with 4 cores each. We can then run four Redis instances on each system, with each instance using one core. This way, each system effectively operates as if it has four Redis servers.

Level of Operations (Addressability)

- **Key-Addressability**: All read and write operations involve the entire key and value. It is not possible to update just part of the value; the entire value must be rewritten, even if only a small change is needed.

Note:

For complex data types that Redis supports, it is possible to address data more granularly. For example, if a value contains an array and you want to read or update a specific element of that array, you can do so in Redis. This is a unique feature of Redis but does not apply to all key-value databases.

Recommended Limitations

- The recommended size for keys and values is <= 10KB per entry. However, Redis has a constraint where the maximum size for a key or value can be up to 500MB. Although Redis allows for this larger size, it is generally advised to follow the recommended limitations to ensure optimal performance.

Use Cases

- **Caching**: Ideal for situations where data needs to be retrieved quickly from a cache.
- **Simple Data**: Suitable when the data is straightforward. Key-value databases are the simplest type of NoSQL database.

Sharding

- Sharding is handled automatically based on the key.

Document DB

Document databases, such as MongoDB, CockroachDB, Couchbase, and Cosmos DB, are designed to store, retrieve, and manage document-oriented information. These databases use a flexible, schema-less structure, typically using formats like JSON or BSON, which makes them ideal for handling semi-structured and unstructured data. This flexibility allows developers to quickly adapt to changing data requirements without needing to redefine the schema.

- **MongoDB**: The most popular document database, known for its ease of use, scalability, and rich querying capabilities.
- **CockroachDB**: A distributed SQL database with strong consistency and resilience, supporting document storage alongside relational features.
- **Couchbase**: Combines the strengths of a document database with the power of SQL-like queries and real-time analytics.
- **Cosmos DB**: A globally distributed database service by Microsoft Azure, supporting multiple data models including document storage.

MongoDB is the most popular among these document databases due to its robust features and wide adoption in the developer community.

It saves documents like below

```
{
       --doc_id:uuid v4,
      attribute1: "hello",
      attribute2: ['h', 'e', 'l'],
      attribute3: {
            a:"k",
            b: "bb"
      }
}
```

Operations

- **get(id)**: Returns the entire document or an error.
- **set({ document_id: uuid, json_doc })**: Stores the document and returns acknowledgment or an error.
- **search**: If searching using a partial document (e.g., `{ shardKey: userId, attr1: 'hi', attr2: '< 20' }`), it returns a list of matching documents.

Document databases support indexes on top-level attributes inside the document. For example, you can index `attr1` but not nested attributes. These indexes only work within the same shard (server) to prevent fan-out (searching across multiple servers). Therefore, a partial search must include a sharding key. There are no global indexes; each server has its own indexing.
Note:
- if sharding key is not part of search query it is called Broadcast Query which will search globly like fan-out.
- Never trust the front-page claim made by NoSQL dbs always read the docs and if possible code.

Addressability

- The entire document is accessible; you cannot retrieve just a single attribute without reading the complete document.

Limitations

- Typically, the size of a document should be <= 10MB, but in MongoDB, it can be up to 16MB.

Sharding

- **Primary key**: `doc_id`
- **Sharding key**: If none is provided, the default sharding key is `doc_id`, but it can be configured to any top-level attribute.

Use Cases

- Storing documents like profile information, post information (social media), and user-generated content (e.g., Stack Overflow, Reddit).
- Managing unstructured or semi-structured data, such as Amazon product information.
- Search databases, like Elasticsearch.

Column Family or Wide Column Database

Column family databases, also known as wide column databases, are among the most complex types of NoSQL databases due to their wide range of use cases and variety. The most common databases in this family are HBase, Cassandra, and Bigtable. While Bigtable is not the most widely used, it was the first NoSQL database created by Google. Each of these databases serves different purposes despite solving similar problems. Cassandra is particularly popular and modern, offering many architectural principles. It provides tunable consistency out of the box and supports multi-master replication. Recently, ScyllaDB has also gained popularity, with companies like Discord shifting to it.

Idea (Why to Use)

Wide-row databases are suboptimal for aggregate queries.

Example

first row	**second**	**third**	**fourth**	
1	**2**	**3**	**4**	**5**
A	**B**	**sanjay**	**C**	**Vinod**
10	**20**	**25**	**30**	**25**

200000 125000 250000 -

How SQL Stores Data

SQL databases store data in a row-major order, meaning everything is saved row by row. For example, if you query the average salary of all employees (`SELECT AVG(salary) FROM employees`), the database retrieves all rows and then extracts the specific information. If each row is approximately 1KB but you only need 4 bytes of salary data, you are effectively using only 0.4% of the data and discarding 99.6%.

How Column Family Databases Store Data

Column family databases store data in a column-major format. For the same employee table, the data is stored column-wise:

id	**1**	**2**	**...**
name	A	B	...
age	20	-	...
salary	100000	-	...
bonus	20000	-	...
sales-done	20	-	...

When performing the same query in a column family database, you only need to read the salary column to get the result, resulting in 100% data utilization. This leads to significant speed improvements, potentially up to 250 times faster, simply by changing the data storage method.

Additional Benefits

- **Compression**: In SQL, each row contains multiple data types, requiring multiple compression algorithms. In column family databases, each column has the same data type, allowing for more efficient compression with different algorithms for each column.

Use Cases

- **Aggregate queries and analytics**
- **High write throughput**: Instead of inserting complete rows, data can be appended to the end.
- **Pagination based on timestamp**

Case study 2 (Typeahead) - Jul 1

Agenda:

Case Study: TypeAhead

5-step approach to High-Level Design (HLD)

- **Problem Statement**
- **Requirement Gathering (Functional Requirements)**
- **Non-functional Requirements (Design Goals)**
- **Scale Estimation**
- **System Design**

Design Interview / Problem -> When we try any Data Structures and Algorithms (DSA) problem or interview, the requirements and constraints are clear. But in a design interview, it is different because we do not know the inputs, expectations, or scale.

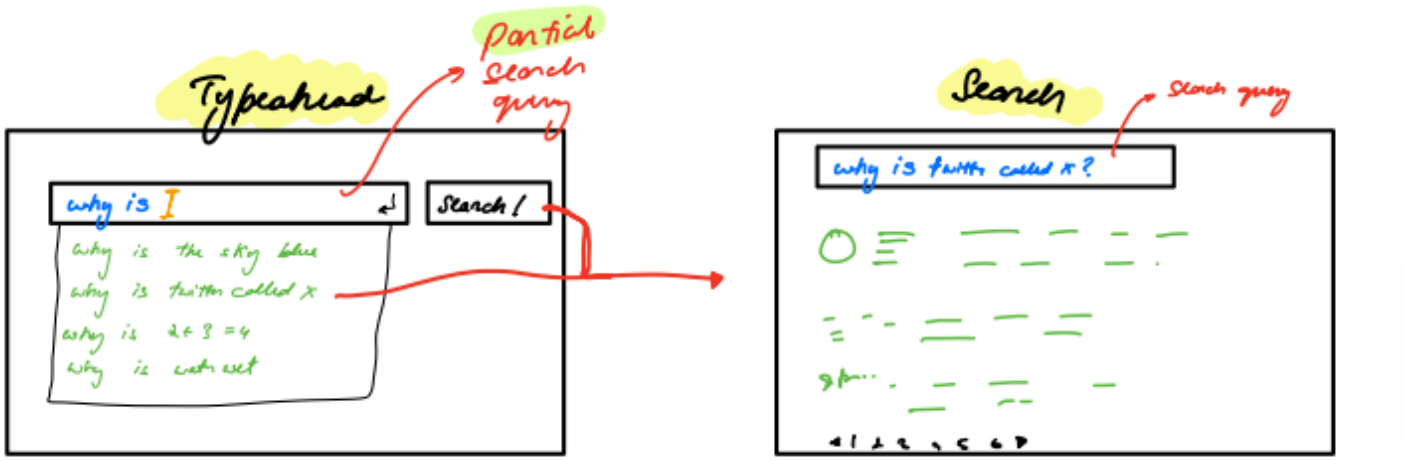
To solve this issue, we need to lead the conversation, gather requirements, think out loud, and explain the steps we are taking to the interviewer.

Case Study: TypeAhead

5-step approach to High-Level Design (HLD)

Step 1: Problem Statement - Understand the requirement. Clarify what we are building with examples. First, understand the problem. What type of problem is it? For example, TypeAhead is like an auto-suggestion. There are many types of auto-suggestions. When we type something on a mobile phone, we get some suggestions. When we type something in a code editor, we get suggestions. But these are not TypeAhead. We are talking about Google search auto-complete and Amazon search auto-complete. When we type something, it gives us suggestions based on what we are typing. These suggestions come from popular past search histories searched by many people all over the world.

As we can see in the first screen, it is TypeAhead showing suggestions from popular searches. It is not the actual search. After clicking on a suggestion, the actual search API will be called, and it will show results, just like in the second screen.



Step 2: Requirement Gathering (Functional Requirements) -> These are the requirements for which we will write code or expose APIs. They are related to business logic and functional requirements, also called MVP (Minimum Viable Product).

We can divide our features into three parts: **MVP, Advanced**, and **Bad**. Always focus on MVP. Discuss advanced features at the end of the interview. If we discuss advanced features at the beginning, we will get more questions, and this can prevent us from completing the task on time. Always avoid bad features because they can be a reason for rejection.

MVP: Whenever we talk about MVP: Discuss the features. Explain how the features work, like API (Input/output), and where the data is coming from, and how it is collected.

1. MVP Features (V0):

- Limit to 5 suggestions.
- Show suggestions only after the user has typed at least 3 letters.
- Which suggestions (data)?
    - All past search queries made by all users.
    - Filter by prefix match.
    - Rank search:
        - By popularity

2. Advanced Features (V1+):
 	
• Ranking:
  ○ By trends.
  ○ By geography.
  ○ User personalization.
• Spell check (V2+).
	3. Bad Features:
	
	• Multi-language (i18n).
	• Voice search.
	• Themes.
	• Image suggestions.
	• UI design.
	• Device support.

How TypeAhead Works:

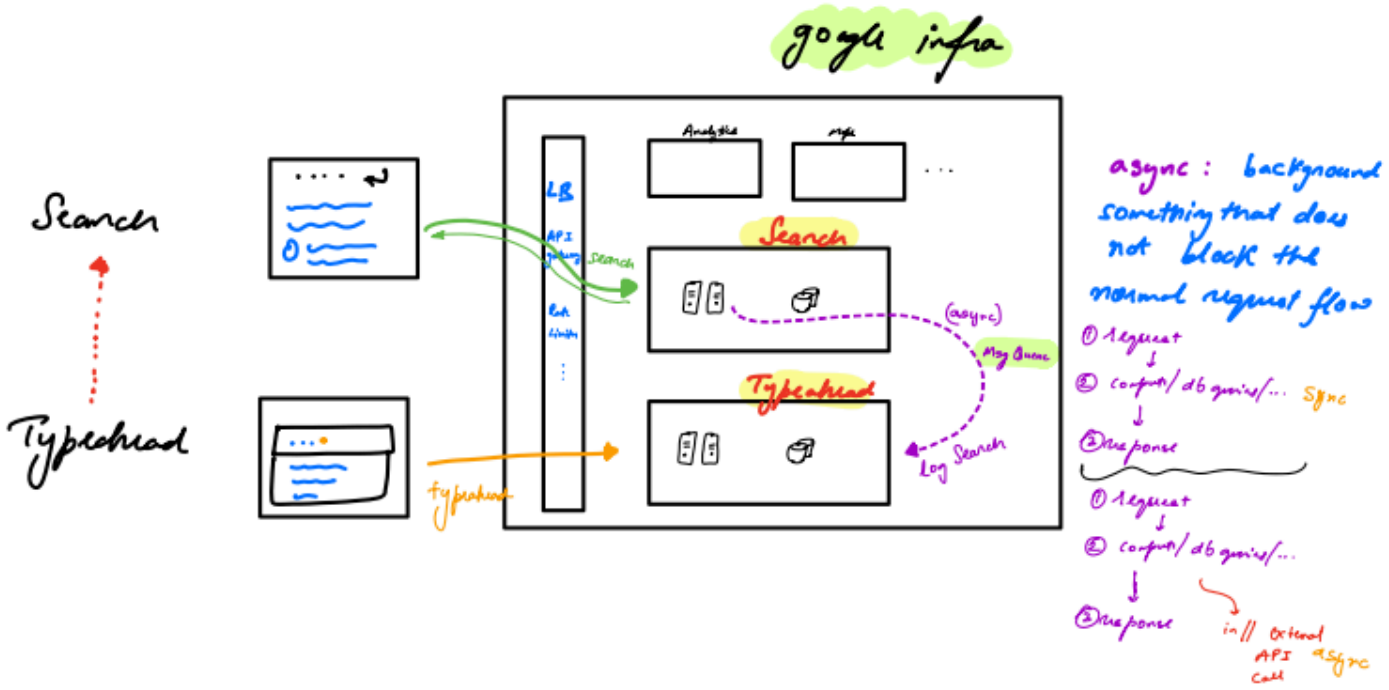
Our app has two parts:

1. **Partial Query**: When the user types, the app sends a partial query to the server after a short delay. This query goes to the TypeAhead database, where we have stored popular searches as a cache. The app then shows these popular searches to the user.
2. **Search Action**: When the user clicks on a suggestion from the TypeAhead database, it sends a query to the actual search database to get the search results.

Additionally, we need the search team to call our API whenever a search happens because we need to log the latest searches. We will expose another API called "log search." This API will log the search result but will not return anything because it is used internally and not exposed to the user. It is called async and it will not block to search result. It will works in background.

APIs:

- **api/typeahead (user-facing)** -> This is for the partial query. It will return a list of suggestions.
- **log-search (internal)** -> This is for the search query. It will return the search result.



Step 3: Non-functional Requirements (Design Goals)

-> To achieve this, we will follow the PACELC theorem. If a partition happens, we need to choose between availability and consistency or latency and consistency. For example, if we want availability (system always up), we get eventual consistency, meaning data updates after some time, not immediately. Until then, we might get stale data. If we want immediate consistency (always updated data), we will face latency, which means a delay in response.

Let's figure out whether to prioritize consistency or availability in our system.

In this context, eventual consistency means that when an actual search happens, there is a delay in updating the count/logging/typeahead results.

Eventual Consistency is Sufficient:

- Users do not need to know the exact count immediately.
- Perfect ranking is not crucial.
- It's okay to miss a few good suggestions as long as the shown ones are highly relevant.

Latency Requirements is ultra-low:

- Competing against the user's typing speed means it should have ultra-low latency. The suggestion list should appear quickly, ideally in less than 50ms.

Step 4: Scale Estimation -> We will do approximate calculations, also known as back-of-the-envelope calculations.

- Perfect accuracy is not important.
- It should be approximate.
- It's okay to be off by a factor of three - if the expected or actual value differs by up to three times more or less, it is still acceptable.

Goals:

- Estimate requests per second for each individual API.
- Do these estimations for both peak load and average load.
- Decide if our system is read-heavy or write-heavy.
- Estimate the amount of data.

Users Estimation ->

It's good to start with some estimation. For example, if someone asked how many requests Google gets in a day, it would be very hard to figure out directly. So, we estimate the number of daily active users.

First, we figure out the world population and other relevant numbers:

- World population: approx **8 billion**
- Number of internet users: approx **5 billion**
- Number of internet devices: approx **100 billion to 1 trillion**
- Population of India: approx **1.5 billion**
- Population of the USA: approx **350 million**

We can follow the Pareto principle (80/20 rule) or even the 80/20/1 principle for more detailed estimation. This means that a small percentage of users or devices account for the majority of activity or traffic.

Pareto Principal ->The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a statistical distribution that represents a situation where a small number of causes or factors account for a large proportion of the effect. This is often summarized by the 80/20 rule, which suggests that 80% of the effects come from 20% of the causes.

In more detail:

- **In economics**: A small number of people often control a large portion of wealth.
- **In business**: A small number of products or customers often generate most of the revenue.
- **In computing**: A small number of bugs can cause the majority of system errors.

The Pareto distribution is useful in many fields, including economics, business, engineering, and computer science, for identifying the most important factors to focus on for improvement or optimization.

Estimate the number of requests:
Note: Units in power kb = 3, mb = 6, b=9, tb=12, quad= 15, quint = 18, etc ...

Search:

- Total Google searches approximately 5 billion.
- Daily active users (DAU) are about 20% of 5 billion, approximately 1 billion.
- Average searches per user per day approximately 10.

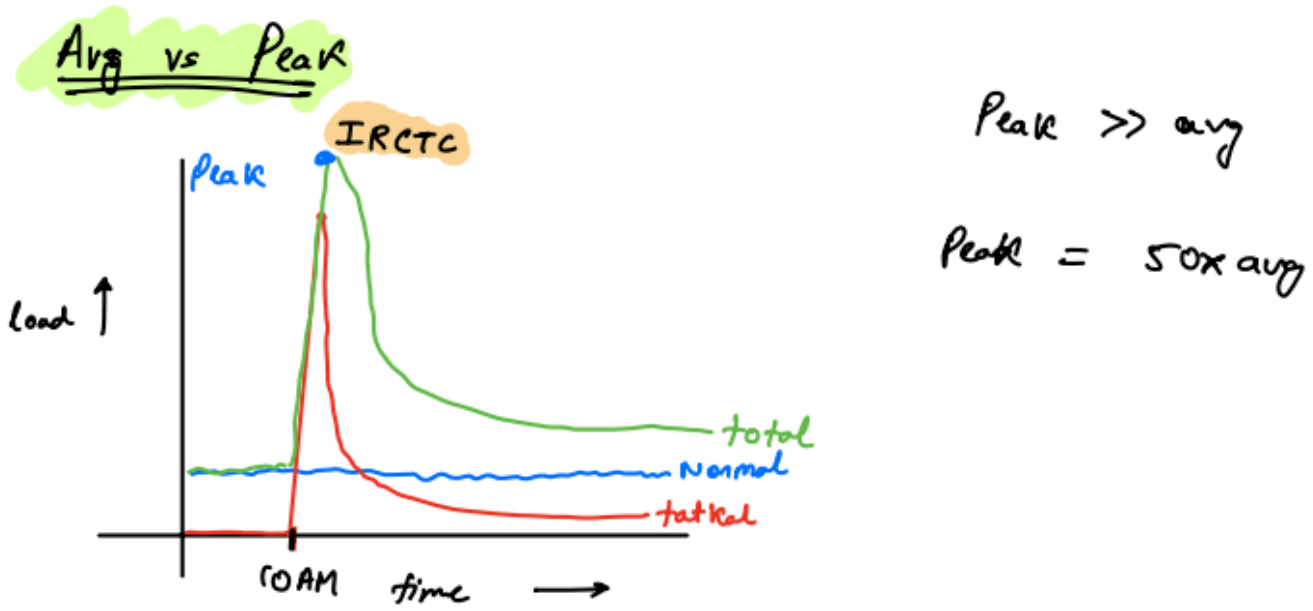
Total searches per day = 1 billion × 10 = 10 billion. Searches per second = 10 × 10^9 / 86400 (24 × 60 × 60) ≈ 100,000 searches per second.

Assuming the same rate for log-searches:

Let's calculate the number of requests for type-ahead: To estimate, we consider the behavior of type-ahead requests, triggered with every character typed. On average, a search query is about 20 characters long. Number of type-ahead queries per search query ≈ (20 - 3) because type-ahead starts after 3 characters, but approximately it's still about 20 characters.

20 type-ahead queries per search × 100,000 searches per second ≈ approximately 2 million type-ahead requests per second."

Average vs Peak: We assume peak average is 2x to 5x during specific events, like IRCTC experiencing higher traffic around 10:30 am during Tatkal ticket bookings compared to normal times.



Amount of Data:

Search Query | Count Why is sky blue | 100,000 Why is kite fly | 512,213

Let's assume each search query is about 20 bytes (varchar) and each count is about 8 bytes (bigint). Therefore, each record is approximately 30 bytes in size.

Total Number of Entries: With 10 billion searches per day, as estimated earlier, the number of unique entries is lower due to repeated queries. According to Google, about 10% of daily searches are entirely new.

Estimated Total Data for 10 Years: 1 billion new entries per day x 10 x 365 days x 30 bytes/entry.

To calculate $10^9 \times 10 \times 4 \times 10^2 \times 3 \times 10^1$ bytes:

1. Compute $10^9 \times 10$:

$$10^9 \times 10 = 10^{10}$$

2. Multiply by 4×10^2:

$$10^{10} \times 4 \times 10^2 = 4 \times 10^{12}$$

3. Multiply by 3×10^1:

$$4 \times 10^{12} \times 3 \times 10^1 = 12 \times 10^{13}$$

Therefore, $10^9 \times 10 \times 4 \times 10^2 \times 3 \times 10^1$ bytes equals 12×10^{13} bytes.

Terabyte (TB): 1 TB = 10^{12} bytes **aprox data size is 120TB**

SI prefixes, also known as International System of Units prefixes, are a set of prefixes used in the metric system to denote multiples or fractions of base units. These prefixes make it easier to express very large or very small quantities of a unit without using cumbersome numbers. Here are some common SI prefixes:

1. **Kilo- (k)**: 1 kilo- is equal to 10^3 (1,000 times).
2. **Mega- (M)**: 1 mega- is equal to 10^6 (1,000,000 times).
3. **Giga- (G)**: 1 giga- is equal to 10^9 (1,000,000,000 times).
4. **Tera- (T)**: 1 tera- is equal to 10^{12} (1,000,000,000,000 times).
5. **Peta- (P)**: 1 peta- is equal to 10^{15} (1,000,000,000,000,000 times).
6. **Exa- (E)**: 1 exa- is equal to 10^{18} (1,000,000,000,000,000,000 times).
7. **Deci- (d)**: 1 deci- is equal to 10^{-1} (0.1 times).
8. **Centi- (c)**: 1 centi- is equal to 10^{-2} (0.01 times).
9. **Milli- (m)**: 1 milli- is equal to 10^{-3} (0.001 times).
10. **Micro- (μ)**: 1 micro- is equal to 10^{-6} (0.000001 times).
11. **Nano- (n)**: 1 nano- is equal to 10^{-9} (0.000000001 times).
12. **Pico- (p)**: 1 pico- is equal to 10^{-12} (0.000000000001 times).

These prefixes are widely used in science, engineering, and everyday measurements to indicate orders of magnitude difference in quantities.

System Design:

Is Sharding Required? We have 120TB of data. Can it fit in one server? Yes, modern servers can handle that much data. However, one server cannot handle 2 million requests per second, and it definitely cannot handle 4 million requests per second during peak times. So, we need sharding.

Note: Do not mention specific tech stack names unless the interviewer asks. If we name a specific technology, they might start asking detailed questions about it, which can take the discussion off track. Instead, describe the tech requirements generally. For example:

- Instead of saying Redis, say "in-memory key-value database."
- Instead of Memcache, say "persistent, distributed message queue."
- Instead of Kafka, say "file storage system."

Handling Typeahead Query:

1. Find past search queries
2. Filter by prefix match
3. Rank by count (popularity)

Finding past searches is easy, but filtering by prefix match can be tricky. When we think about prefixes, we often think of Tries because they are efficient in storage and have O(L) complexity for insertions and lookups (L means length here). However, this is only true for checking if an entry exists, not for finding all matching entries.

Why We Can't Use Tries:

1. After prefix matching, we get subtrees, and we need to go through every leaf in the subtries to rank them, which can be billions of entries.
2. Sharding: Sharding by characters can be problematic because we have only 26 characters, leading to low cardinality. Sharding by 2 or 3 characters might help with cardinality but results in uneven distribution.
3. No popular database supports Tries out of the box.

What to Use Instead: Ask some questions:

- Is the data complex? Not really (each entry is small).
- Are the queries complex? Yes (prefix + rank).
- Latency requirement? Ultra low latency, so SQL and file storage databases won't work.
- Is the system read-heavy or write-heavy? It's read-heavy with 2 million reads per second and 100k writes per second.

Ideal Solution: For a read-heavy system with ultra-low latency, an in-memory key-value database like Redis is ideal.

Handling Typeahead Queries:

We need ultra-low latency, which isn't possible if we search and rank results in real-time. To solve this, we precompute the results and store them in a database. Now we have two databases:

1. **Primary Database**
2. **Derived Cache Database** with precomputed results

In the derived cache database, we store the top 5 search results for every prefix, sorted by count.

How to Handle Typeahead Queries: When we receive a query, we return the results like this:

```
typeahead(partialQuery) {

    return derivedDB.get(partialQuery)

}
```

Redis is ideal here because it can return search results in less than 1 millisecond.

Handling LogSearch (How to populate data?):

1. Update count in Primary Database: O(1)
2. Update any affected prefix in the secondary database. For a query like "what is the color of God?", the affected prefixes are:
    - "w..."
    - "wh..."
    - "wha..."
    - "what..."
    - "what i..."

We need to update around 20 prefixes for each query.

System Load Before and After:

- **Before:**
    - Read load: 4M/s
    - Write load: 100k/s
- **After:**
    - 100k logs/s × 21 writes/log (since each entry needs 21 writes)
    - Total write load: 2M writes/sec

Now, our system is both read-heavy (4M/s) and write-heavy (2M/s).

Solution: No database can be optimized for both high reads and high writes simultaneously, so we need to figure out a strategy:

1. **Use Cache:** This reduces read load, but won't help much because our database is already cached.
2. **Reduce Write Load:** Use batching or sampling, but this could lead to potential data loss. We could use a write-back cache to handle this.

In summary, we need to carefully manage both read and write loads to ensure system efficiency and data integrity.

NoSQL Internals - Jul 3

Agenda:

Typeahed

- Batching / Sampling

- Recency Factor

LSM Trees - covered in Overflow topics - LSM Trees & Kafka - Jul 13

- Write Ahead Log (WAL file)

- Mem Table

- Sorted String Table (SS Tables)

- Sparse Index

- Bloom Filter

Optimizing Write for Typeahead

Design Goal - PACELC:

- **Availability vs Consistency:** We can either achieve availability or consistency, not both.

Eventual Consistency:

- If a search happens, the count changes, but it's okay for the typeahead result to lag behind a bit. Users do not see the exact search count, so if some data is lost, it's acceptable as long as suggestions remain highly relevant. Missing some important suggestions due to inconsistent counts is also fine. Therefore, we choose eventual consistency for typeahead results.

In this case, trends are more important than individual data points. To manage this, we can use batching or data sampling to reduce write load.

Batching:

- Instead of updating immediately, we wait to collect a batch of queries and then update them all at once. Batching gives us eventual consistency without data loss.

LogSearch Scenario (Search Query API):

- Updating the count in the primary database is O(1).
- Updating suggestions in the secondary database requires updating approximately 20 matched prefixes.
- We update the primary database immediately but batch the updates to the secondary database.
- We update the secondary database only when the count has increased by 1000. This reduces the write requests by a factor of 1000. The batching threshold can be adjusted based on requirements, such as 100, 200, 500, or 1000.

Sampling: Randomly sample a subset of data, similar to exit polls in elections. Instead of talking to every voter, only a small subset is surveyed. This works because mathematics proves that any trends in the entire population will also exist in the sample, provided the sample is uniformly random and large enough.

In sampling, we instruct the search team not to call the log search on every entry. Instead, use a random function that generates a value between 1 and 1000, and call the API once for every 1 out of 1000 requests.

If a query is searched very few times, it might not appear in the database due to sampling. That's okay. This is actually a good thing because we want to show the most popular suggestions, not the rarely searched ones.

How to Handle some advanced requirements such as:

- Trends/news
- User personalization
- Geography

Geographically: We can shard by location but also need global results. To achieve this, we maintain a global database alongside the location-specific shards.

User Personalization: For example, Google first looks at your browser history to determine your personal preferences. It shows suggestions from your browser history first, then adds suggestions from the typeahead results.

Trends/News: Recently popular items should appear higher in the suggestions.

Recency Factor in Typeahead:

For example, during the COVID-19 pandemic's first and second waves, everyone searched for related information, making its search count extremely high. Comparatively, a recent event like an India vs. South Africa T20 World Cup match might be highly searched in India and South Africa but not globally. Thus, older but highly searched topics like COVID-19 would appear first even if they are no longer relevant. This is where the recency factor comes in.

To address this, we should weigh new searches more and reduce the prominence of older searches over time.

Solution: To solve this issue, we implement a decay mechanism for search counts. We decrease the count over time, for example, halving it every week. This decay affects both the primary and secondary databases, ensuring that newer searches have more weight in the typeahead results.

No SQL Internals - How to store data

Important link about disk storage

1. Magnetic:
2. https://www.youtube.com/watch?v=wtdnatmVdIg
3. Solid State: https://www.youtube.com/watch?v=5Mh3o886qpg
4. RAM: https://www.youtube.com/watch?v=7J7X7aZvMXQ

Key-Value Store: A key-value store uses a MemTable to store data, which functions like an in-memory hashmap and acts as a cache. It employs a write-through policy for consistency, invalidation policy, and LRU (Least Recently Used) eviction. All reads and writes happen directly in the MemTable.

1. MemTable: Issues with Mem Table

1. **Data Loss on Server Restart**: To prevent data loss, the data must be persisted to disk.
2. **Memory Overflow**: If the memory is full, eviction occurs, and reads may fail.

Persisting Data: To avoid these issues, data must be saved on disk. However, we cannot store a hashmap directly on disk because it is inherently a random and fragmented data structure.

Hashmap in Memory vs. Disk:

- In memory, a hashmap appears as an array where each pointer points to a linked list distributed across various locations.
- To retrieve values, it requires jumping to multiple random points.
- Disks are very slow for random reads, taking approximately 10ms to 100ms, whereas RAM is much faster for random reads, taking about 10ns to 100ns—a difference of nearly one million times.

Sequential vs. Random Reads on Disk:

- **Sequential Reads**: Fast, as data is saved in a continuous strip and read by a disk head spinning at high speeds (e.g., 7200 RPM).
- **Random Reads**: Slow and time-consuming because changing tracks on a spinning disk is difficult.

Modern SSDs:

- Faster than hard disks for random reads (1ms to 5ms).
- Extremely fast for sequential reads (7.4GB/s to 14GB/s in Gen 4).

Conclusion: Due to the slow nature of random reads on disks, data should be stored and accessed sequentially on disks for optimal performance.

Write Ahead Log (WAL):

The Write Ahead Log (WAL) is a technique used in database management systems to ensure data integrity. The primary purpose of WAL is to keep a log of changes before they are applied to the main database. This log can be used to recover the database in case of a crash or failure. It is immutable, meaning only inserts are allowed and no updates. To update, you must reinsert the data, appending it to the end.

How it works:

1. **Logging Changes:**
    - Before any changes are made to the main database, they are first recorded in the WAL. This includes inserts, updates, and deletes.
2. **Applying Changes:**
    - Once the changes are safely logged, they are then applied to the main database.
3. **Recovery:**
    - If the system crashes before the changes are written to the main database, the WAL can be used to replay the changes and bring the database back to a consistent state.

Benefits:

- **Data Integrity:** Ensures that all changes are recorded and can be recovered in case of a failure.
- **Consistency:** Helps maintain a consistent state of the database by ensuring that no changes are lost.
- **Performance:** Allows changes to be batched and written to the main database in an optimized manner, improving performance.

Use Cases:

- **Database Systems:** Commonly used in relational databases like PostgreSQL, MySQL, and SQLite.
- **Distributed Systems:** Ensures consistency and reliability in distributed databases and systems.

SQL Databases Use a WAL File:

- Improve Write Throughput
- Perform Master-Slave Replication

In SQL, the column size is fixed based on the assigned type. Space is allocated for every column regardless of the data stored in it. For example, for a column `name: varchar(50)`, if we save `name="sanjay"`, it will store "sanjay" followed by null values to fill the allocated space. This ensures the row size remains constant, even if the data is updated.

Write in SQL (without WAL file):

- Inserting or updating the table is fast.
- Maintaining the index: if data changes, the index also changes.
    - Finding data in the index: O(log N)
    - Updating the index: O(log N)

Since the index is saved randomly, random reads are very slow on disk. Balancing the tree and changing multiple nodes is very expensive, approximately O(k log N).

Write in SQL (with WAL file):

- Uses batch writes: only appends to the WAL file.
- Periodically (after every 100MB of writes), it updates the table and index.
- This approach is fast because it allows planning the writes.

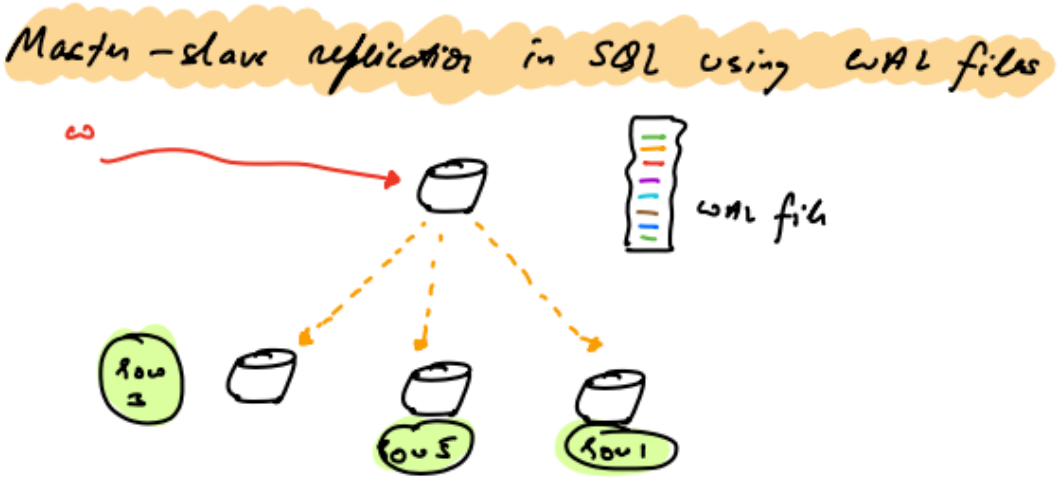
Now, reads must happen from both the table and the WAL file. This makes writes 1000% faster but reads 10% slower.

Master-Slave Replication in SQL Using WAL Files

In master-slave replication, every write happens on the master, and the changes are then synced to the slave. The master maintains a WAL file, making it easy to sync with the slave. Each slave keeps a pointer in the WAL file to track the last updated data. The slave then requests the master to provide any unsynced changes after this point, and the master supplies the necessary updates.

Master-Slave Replication in SQL Using WAL Files

In master-slave replication, all writes occur on the master, and the changes are then synced to the slave. The master maintains a WAL file, which simplifies syncing with the slave. Each slave keeps a pointer in the WAL file to track the last update it received. The slave then requests the master to provide any unsynced changes from that point onwards, and the master supplies the necessary updates.



NoSQL Databases and WAL Files

In NoSQL databases, entries do not have a fixed size, which means records cannot be updated in place because it would overwrite other data. Instead, new records are appended to the end. This behavior is similar to how WAL files work, so WAL files are also used in NoSQL databases.

2. MemTable + WAL

To solve the MemTable issue, we will store the MemTable entries in the WAL file as follows:

Writes:

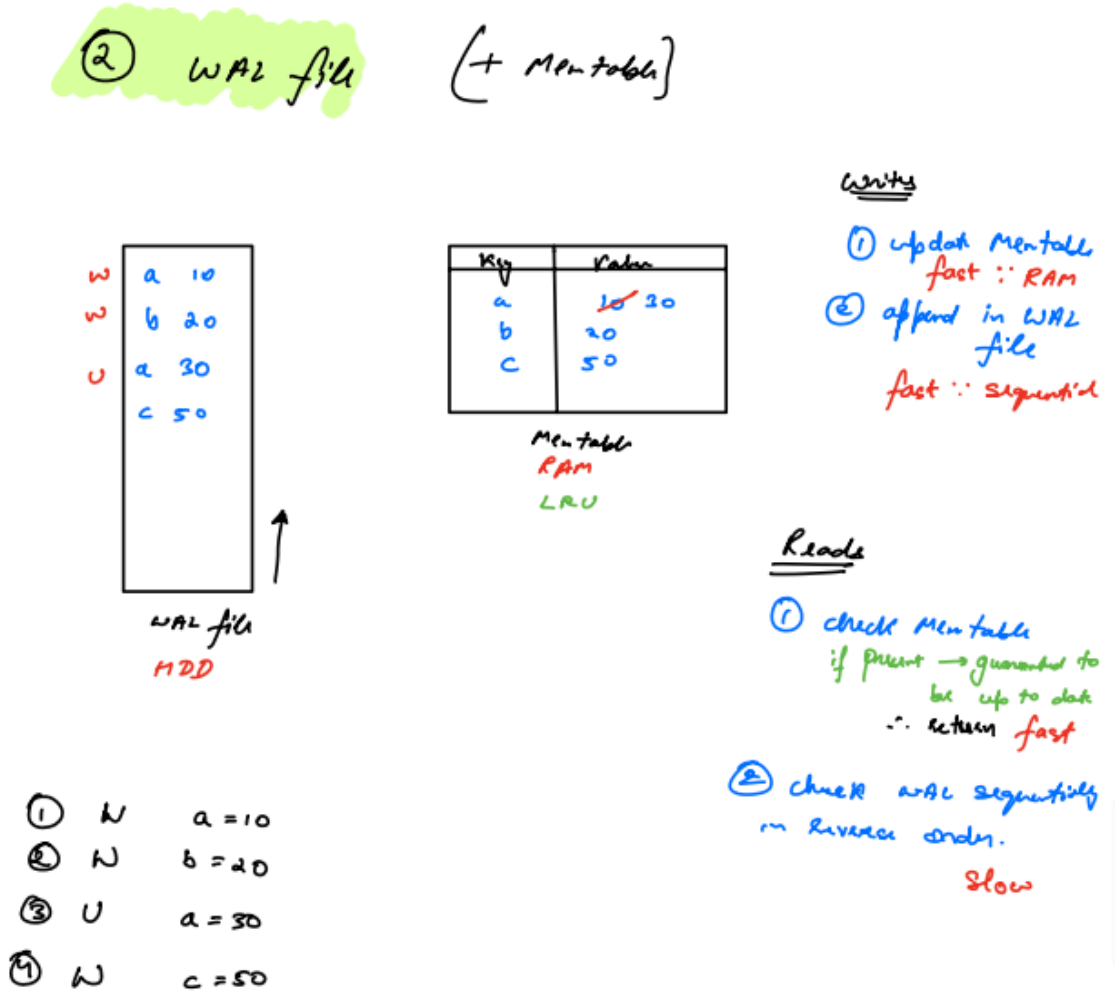
- When writing to the table, an entry is made in the WAL file.
- The MemTable is fast because it is stored in RAM.
- The WAL file is also fast because data is stored sequentially, making sequential writes and reads very fast on disk.

Reads:

- Check the MemTable first. If the entry is present, it is guaranteed to be up-to-date, and the data can be returned immediately.
- If the entry is not in the MemTable, we must check the WAL file from bottom to top, which is very slow.

By introducing the WAL file, we solve two issues:

1. Data loss after a system restart, as we can recover data from the WAL file.
2. Data eviction when memory is full, as we can read the data from the WAL file.



Another Issue with Using WAL Files:

- **Reads are very slow:**
  - WAL files can become very large (up to 1 TB).
  - Sequentially reading a 1 TB WAL file is very slow.

Solution: Keep the WAL file shorter.

3. Sorted String Table (SSTable):

- Periodically dump the content of the WAL file into an SSTable.
- SSTables are immutable; no data can be inserted once created.
- Once created, only reads are allowed.
- SSTables are sorted by key, allowing binary searches.
- SSTables are stored on the hard disk.
- There are multiple SS Tables

Note: size (Main table) > size (WAL file)

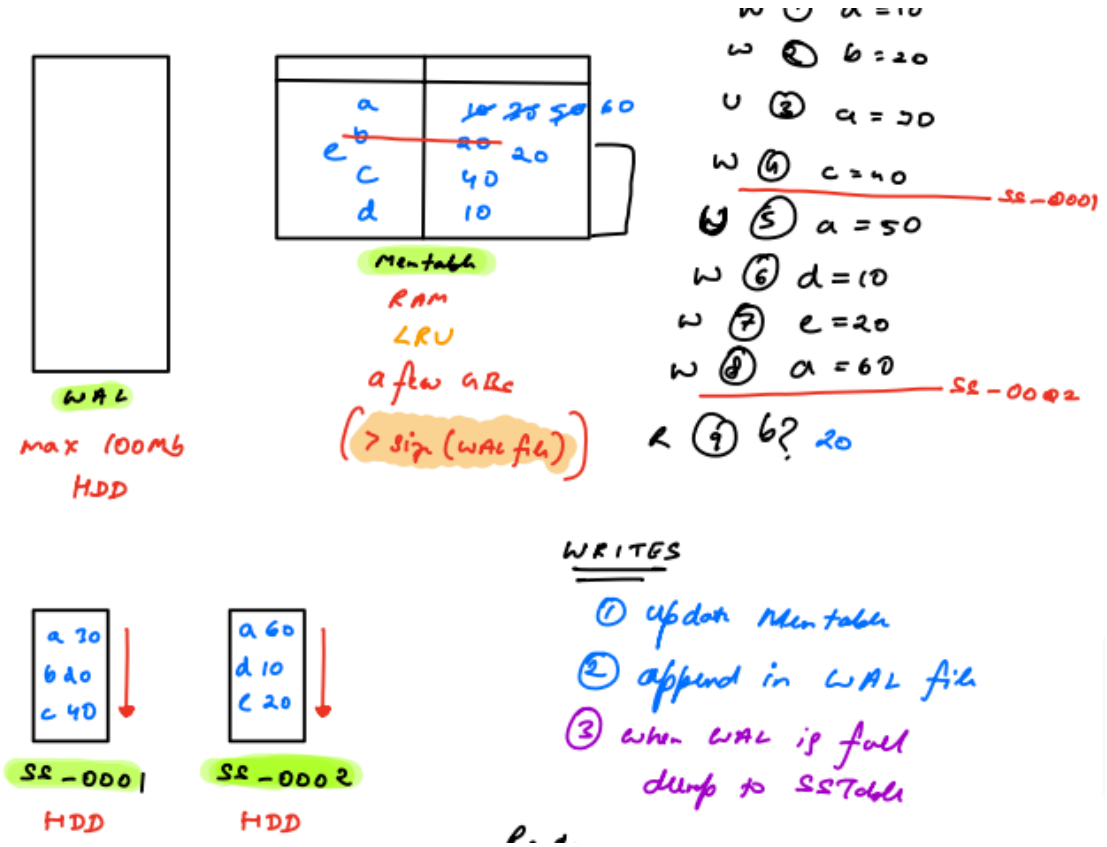
We will fix the size of the WAL file to 100MB. Whenever it exceeds this size, we will dump the data into new SSTable files, which are sorted, and then purge the WAL file.

Writes:

- Update the main table.
- Append the entry to the WAL file.
- When the WAL file is full, dump its content into SSTables and purge the WAL file.

Reads:

- Check the main table first. If the entry is present, return it because it is guaranteed to be up-to-date.
- If not found in the main table, scan the SSTables. Read the SSTables in reverse order, and within each SSTable, use binary search to find the key.



Case study 3 - Messaging - Jul 6

Agenda:
Case Study on Messaging Apps

How to approach a High-Level Design (HLD) problem. We have to follow 5 steps:

1. **Problem Statement**: Think of a similar system that you know well.
2. **Functional Requirements (MVP)**: Minimal requirements. For each requirement, think of:
    - API (input/output)
    - Who is the client? (User/Internet)
    - How is data created?
3. **Non-functional Requirements (Design Goals)**:
    - Think about PACELC Theorem, which means we have to think about:
        - Availability (+ Eventual Consistency) vs Consistency (Immediate)
        - Latency vs Consistency
    - Other things to consider to meet our design goals, like Rate Limiting, Security, Reliability, Idempotency, Order, etc.
4. **Scale Estimations**: Here we will try to find sharding is required or not
    - Start from daily active users
    - Pareto Principle (80/20/1): For social media, this means 80% of users log in, but only 20% are active (comment, like), and only 1% create posts (actual content creators).
    - When thinking about daily users, set goals for:
        - Size of data for each database
        - Number of requests per second for each API, considering both average and peak times.
    - Find out if the system is more read-heavy or write-heavy.
5. **System Design**: Now, we can start developing the system design because we have all the needed information. We need to discuss aspects like sharding, choice of database, cache, load balancer, microservices, logic, and flow while creating a system design.

Problem Statement: Think of an application you are familiar with and list it down. Only include applications that provide a unique feature not found in other applications. Do not give random application names. Here are some examples:

- Facebook Messenger: 1-to-1 chat
- WhatsApp: Real-time messaging with low latency
- Slack: Group chat with 100,000+ members per group, multi-tenancy (Multi-tenancy means a single software instance serves multiple customers, keeping their data isolated and invisible to each other.)
- Discord: Voice messaging discussions
- Snapchat: Privacy
- FaceTime: Video messaging

As we list these applications, we will use some unique features to create our application, such as 1-to-1 chat, group chat, and low latency.

Functional Features (MVP):

There are three types of features: basic, advanced, and bad features. We should first list basic features and start with those. At the end, we can discuss advanced features to show scalability and avoid bad features as they can spoil your interview and lead to rejection.

Note: Typically, we do not discuss user login/registration flow because there is nothing special about it unless we are building an OAuth service. We assume every service we are building already has user login.

MVP (V0)

- **Users can send messages to other users:**
  - **API**: Here is the API for sending a message with input and output. It will return an acknowledgment or errors:

```
sendMsg(
  sender_id: int/uuid,
  recipient_id: int/uuid,
  msg: json
) -> ack / error
```

Users can view messages they have sent or received:

- **API**: Here is the API for viewing messages. It will take the inputs below and use offset and limit for pagination, since we will see some messages, not all messages. It will return a list of messages.

```
viewHistory(
  user_1_id: (me),
  user_2_id: (other person),
  offset: int,
  limit: int
) -> List(msg)
```

Group messaging:

- This API is similar to the above. The difference is in the `viewHistory` API, where both `user_1` and `user_2` can be either conversation IDs or group IDs. When two users are talking, it is called a conversation ID. In the `sendMsg` API, the `sender_id` will be the same; only in the `recipient_id`, it can be either a user_id or a group_id.

Show latest conversations on top

- **API: It will take `user_id` and return a list of conversation IDs.**

```
getRecentConversations(
  user_id: int
) -> List<conversation_id> (user1 + user2) / group_id
```

Advanced (v1+)
 	
● **Message Receipts:**
  ○ **Sent: single tick**
  ○ **Received: double tick**
  ○ **Read: single blue tick**
● **Online Status:**
  ○ **Online**
  ○ **Typing**
  ○ **Last seen**
● **Broadcast**
	Bad (fail the interviews)
	
	● **Managing friends/contacts**
	● **Managing groups**
	● **Settings management**
	● **Profile management**
	● **Attachments**
	● **Download attachments**
	● **Invite links**
	● **Favorite/pin messages**
	● **Search**
	● **Disappearing messages**
	● **Pictures/videos**
	● **Deleting messages**
	● **Backup**

Non-functional Requirements (Design Goals) (PACELC)

Availability (+ Eventual Consistency) vs Consistency (Immediate)

What does eventual consistency mean?

- Eventual consistency means stale reads. In our case, there are no stale reads because once a message is sent, it's sent—we are not editing or deleting any messages.

What are stale reads here?

- When you send a message, the server receives it and sends you back an acknowledgment. This means the message is delivered.
- If the other person tries to read the message, but the server says there are no new messages, this is a stale read. This is what we call eventual consistency. Stale reads do not mean the message will reach the person after some time—that is called latency.

We think availability is important, but consistency is more important in communication because communication is the backbone of society. Poor communication causes many issues.

One reason for eventual consistency could be that a message is sent but not delivered to the user. Another reason could be out-of-order message delivery, where some messages are delivered, but some are not.

It is clear that between Availability and Consistency**, we will choose Consistency.**

Latency Requirement:

We would be okay with a message latency/delay of less than 5 seconds.

Idempotency:

This means that performing the same operation multiple times will have the same effect as doing it once. For example, if you send the same message multiple times, the result should be as if the message was sent only once.

API:

POST APIs are typically not idempotent because repeated requests create duplicate resource entries in the database.

- **Mobile networks are unreliable**: We want the frontend to auto-retry sending messages. For example, if you send a "hi" from the frontend to the server, and the request fails on the backend, the frontend should automatically retry until it gets a successful response. This is the desired scenario.
- **Another scenario**: You send a request to the server, the server receives it but fails to send back an acknowledgment. The frontend retries sending the message, and the server accepts it again, saving the same message twice. This is an idempotency issue, as the server saves a duplicate message.

Note: Whenever the frontend can auto-retry POST requests, your backend should be idempotent to prevent duplicates.

Scale Estimations

Let's assume we have 2 billion users.

- Daily active users: Around 20% of 2 billion = 400 million
- Number of messages per user per day = 50
- Total messages sent per day: 50 messages * 400 million = 20 billion messages per day

Writes:

- Average number of requests for sendMsg(): 20 billion messages/day = 20 x 10^9 messages / ~10^5 seconds = 200,000 messages per second
- Peak: 2 times the average

Reads:

- Average number of times a message is read = 10 times per message (considering group message reads)
- Average number of requests for viewHistory(): 200,000 messages/second * 10 reads/message ≈ 2 million reads per second

Estimate the data size:

- Average size of a message = ~1 KB

```
{
  msg_id: 8 bytes,
  sender_id: 8 bytes,
  recipient_id/group_id: 8 bytes,
  text: 500 bytes (average),
  attachment-url: 200 bytes (average),
  timestamp: 8 bytes,
  location: 10 bytes,
  status: 1 byte
}
```

10 years storage requirement:

- 10 years x 365 (approximately 400) days x 20 billion messages/day x 1 KB/message
- = 10 x 400 x 20 x 10^9 x 10^3 bytes
- = 8 x 10^16 bytes
- = 80 petabytes of message data

System design ->

Idempotency:

- **Frontend can retry**: If the frontend retries, it should not create duplicates in the backend.
- **User can still create duplicate entries**: Users can send the same message multiple times, which is okay. However, if a user sends one message, the system should not generate duplicates.

Solution: Add a unique `msg_id`. This should be added on the frontend (client side). Typically, `msg_id` is generated on the backend, but if it is created on the backend, the backend cannot determine if it's a duplicate. Therefore, it should be created on the client side. If the user retries multiple times, the `msg_id` should not change. The backend will ensure this message ID is stored once. The frontend can generate this using UUID.

Ensure Message Order:

To solve this, we should create a message chain. The idea is simple: every message has the previous message ID. If messages appear out of order on the recipient's side, it will detect that intermediate messages are missing. Only display a message if its previous message has also been displayed, and show "wait for msg." if not.

HLD: Zookeeper + Kafka - Jul 8

Agenda:

- **Why Zookeeper- state tracking in a distributed envirment**
- **what is Zookeeper**
- **Master-Slave without Zookeeper**
- **Master-Slave with Zookeeper**
- **DB Master Dies**
- **Master Re-election Using Zookeeper**

Why Zookeeper - State Tracking in a Distributed Environment

In modern big applications, having thousands of servers is normal. The biggest issue is maintaining the state and configuration, which we manage using configuration files. **It is not a Database**. **It is configuration management system**

These files might contain:

1. **State / Configuration**
    - Database IP
    - Database dialect

    - API info (Schema)
    - Database internals
        - Shards
        - Replication factor
        - Which server is in which shard
        - Extra replica
        - Master-slave architecture
    - Log format
    - Timezone
    - Security / keys
2. **Servers can be added and removed.**
3. **Every server must always see a consistent picture of the state.** Immediate consistency about config across thousands of servers is needed.

Apache has many products like Zookeeper, Kafka, and Hadoop. One of them is Zookeeper.

Zookeeper

Zookeeper is a tool from Apache that helps manage the state and configuration in a distributed environment. It makes sure that all servers have a consistent view of the state. This is important when you have thousands of servers, and you need to keep everything in sync. Zookeeper can help with:

- Coordinating different parts of an application
- Keeping configuration data consistent
- Managing server nodes (adding and removing servers)

By using Zookeeper, you can solve the problem of state tracking and configuration management in big applications.

Zookeeper behaves like a filesystem over the network. Typically, a filesystem has files and folders. In Zookeeper, these are called Zookeeper nodes.

Persistence Node:

- A persistence node is a type of Zookeeper node that remains in the system even if the client that created it disconnects or shuts down.
- These nodes store data permanently until they are explicitly deleted.
- They are used for data that should survive client sessions, such as configuration settings or shared resources.

Ephemeral Node:

- An ephemeral node is a temporary node that exists only as long as the client that created it is connected to Zookeeper. It means every server own one Ephemeral node in ZK.
- If the client disconnects, the ephemeral node is automatically deleted.
- These nodes are used for short-lived data, such as temporary status information or locks.
- ZK maintains a heartbeat with evry server if heartbeat fails then ephemeral own by server automaticly deleted
- Zookeeper maintains a heartbeat with every server. If the heartbeat fails, the ephemeral nodes owned by that server are automatically deleted.
- An ephemeral node can only be written to by its owner. It can be read by anyone, but only the owner server can write to it.

Master-Slave without Zookeeper

DB App Server:

- Assume we have a master-slave application and one DB server that manages everything.
- The app server needs to know which DB server is the master and which is the slave.
- It tracks server IPs, but if any server dies, it is hard to maintain.

Config Management System:

- To solve this issue, we use a config management system.
- The app server continuously connects to this config system.
- When a request comes, the app server asks the config system which is the master and which is the slave.

Issues:

- There is an extra step for every request.
- There is a bottleneck, which can be a single point of failure.

Zookeeper Solution:

Prevent Extra Step (hop):

- Zookeeper sets a watch to prevent the extra step.
- It tracks ephemeral nodes owned by the server, which have the server state and IP. If a server dies, its nodes are auto-deleted.
- Setting a watch means whenever a Zookeeper node changes, Zookeeper informs all servers that set a watch on this node.
- When the app server starts, it asks Zookeeper for master-slave configuration and stores it locally. It then sets a watch on Zookeeper. Whenever changes happen, Zookeeper informs the app server.

Prevent Bottleneck:

- Zookeeper uses a push-back mechanism, so there is no need to ping Zookeeper for every request.
- Load on Zookeeper only occurs when config changes, like manual changes, system crashes, or when a server is added.

Prevent Single Point of Failure:

- Zookeeper itself is replicated using master-slave + quorum.

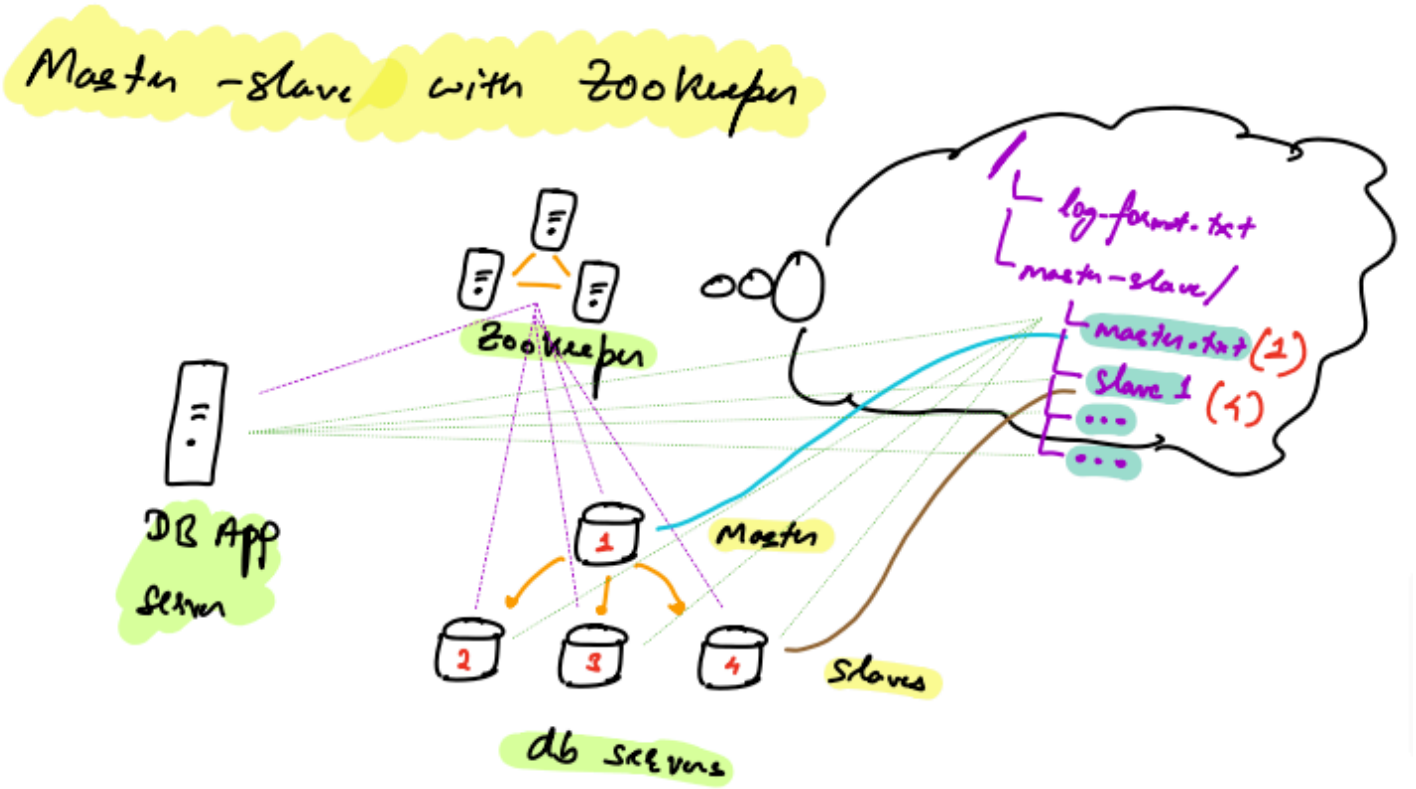
Master-Slave with Zookeeper:

- Zookeeper is a master-slave application and connects with other master-slave servers using a heartbeat mechanism.

How It Maintains:

- Zookeeper behaves like a filesystem, with a root, and inside the root, we have files like `logformat.txt`.

- There might be a folder called `master-slave`. Inside this folder, there is `master.txt`, an ephemeral node created by the master server. It holds the master IP.
- Similarly, there are files for each slave system with their IPs.
- All servers set a watch on Zookeeper. When they need info, like who the master is, Zookeeper provides the master IP to all applications that set a watch.
- Whenever the master dies, Zookeeper deletes the `master.txt` file and broadcasts the changes to all watchers.



DB Master Dies:

- Zookeeper detects a failed heartbeat and deletes the `master.txt` ephemeral node.
- It informs the app server and slaves (watchers).
- Until a new master is configured, the app server will deny writes.
- Slaves will try to re-elect a new master.

Master Re-election Using Zookeeper

Every slave follows this protocol:

- Write the master's IP to `master.txt`.
- When the master fails, Zookeeper notifies them.
- Immediately, each slave tries to write its own IP to `master.txt`.
- The first slave that succeeds becomes the master, and the others are rejected by Zookeeper.
- Whenever a new master is elected, Zookeeper informs all watchers.
- If the old master comes online again, it will become a slave.

Case Study 4 (Elastic Search) - 10 Jul

Agenda:

Types of NoSQL databases

Full Text Search:

When we need to search full text, it is very slow because it has to scan the entire table. This has a time complexity of O(n*m), where (n) is the number of rows and (m) is the average review length. For example, if we search using "SELECT * FROM productReviews WHERE review ILIKE %good quality% LIMIT 10", it will be slow. Here, (ILIKE) means case insensitive. Even if we have an index, it won't help with text search. Indexes are useful for exact matches or prefix words. SQL is not good for full text search. Even though PostgreSQL and MySQL support full text search, they are not suitable for distributed systems.

When we do a full text search, there are many problems. For example, we have a query "find all reviews about 'good quality'."

Problems are as follows:

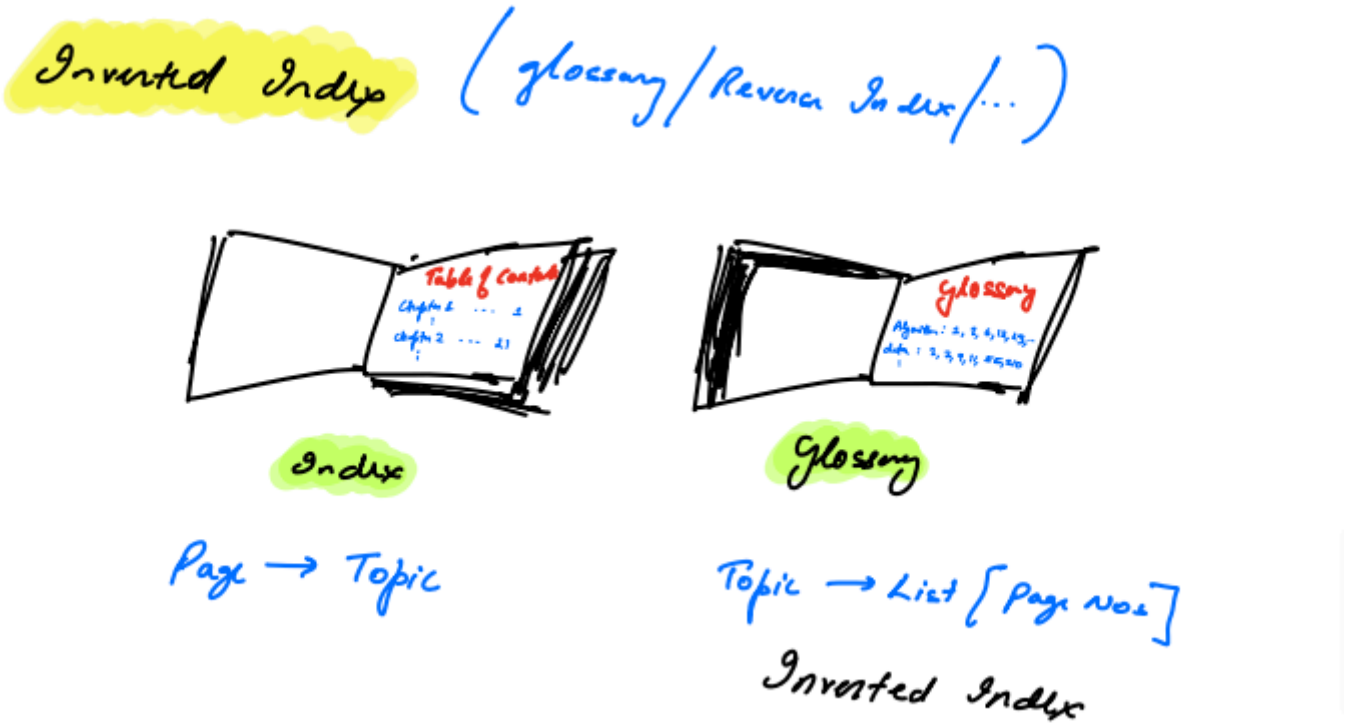
- **Partial Word Match**: The word "good" might match in some text, and the word "quality" might match in other text.
- **Negative Sentiment**: Even if there is a full text match, a review saying "not good quality" is negative.
- **Synonyms**: Sometimes the query might use different words with similar meanings, like "best" instead of "good."
- **Sentiment Analysis**: We need to understand the emotions or opinions in the reviews, not just the words.

Here is the process to full text search

1. **Extract Tokens**: When we do a full text search, we need to break down the text into smaller parts called tokens (words, parts of words, or units of text).
2. **Build an Inverted Index**: We create a special list that helps us find where each token appears.
3. **Search via the Inverted Index**: We use this list to quickly find the information we are looking for.

Inverted Index: Think of a book. At the front, there is an index listing topics and the pages where they are discussed. At the back, there's a glossary listing words and the pages where those words appear. The glossary is like an inverted index: instead of topics to pages, it maps words to pages.

For our reviews, we make a list that shows which words appear in which reviews. For example, if a word appears in several reviews, we list all the review IDs where it appears. This helps us search quickly.



Text Pre-processing

1. **Tokenization**: Split the text into words or tokens. For example, the sentence "I love cats & dogs" becomes the array ["i", "love", "cats", "&", "dogs"].
2. **Stop Word Removal**: Remove unhelpful or unwanted words, such as:
    - Bad language, like abusive words.
    - Sensitive topics, like murder or crime.
    - Copyrighted material.
    - Competitor words.
    - Useless terms that don't add much meaning or appear too often.
3. **Replacement**:
    - **Stemming**: Remove common word endings (like ing, er, s, tion). This process is fast but not always accurate. For example, "caring" might become "car," changing the meaning.
    - **Lemmatization**: A smarter version of stemming that uses advanced language models (LLMs) to understand the correct form of a word. It's more accurate but requires more processing power.
    - **Reducing to Root Word**: Use a dictionary to map words to their root form. This method is more accurate than simple stemming. Examples include:
        - Removing suffixes: For example, "caring" becomes "care" and "cars" becomes "car".
        - Synonym support: For example, "best" or "excellent" can be mapped to "good".
        - Fixing misspellings: For example, "scalar" becomes "scaler".
        - Adding extra products: When a specific search term is not found, similar alternatives are added. For example, if someone searches for "amul milk" and it's not available, suggest other milk brands.

We can also simplify the text while keeping its meaning. For example, "I love cats & dogs" can be shortened to ["love", "cats", "dogs"], and it still conveys the same message.

Building a Reversed Index means creating a special list that helps quickly find where each word appears in your data. Here's how it works:

1. **Tokenization**: Break down the text into individual words or tokens. For example, the sentence "I love cats & dogs" becomes ["love", "cats", "dogs"].
2. **Create the Reversed Index**: Make a list (or map) where each word points to the locations (like document IDs or page numbers) where it appears. For example, if "cats" appears in reviews with IDs 1, 3, and 5, the index will show "cats: [1, 3, 5]".

- "I love cats & dogs" -> ["love", "cat", "dog"]
- "Cats love salmon (fish) but hate bread" -> ["cat", "love", "fish", "hate", "bread"]
- "I love cats but hate dogs" -> ["love", "cat", "hate", "dog"]
- "I hate cats" -> ["hate", "cat"]
- "I love bread & tuna" -> ["love", "bread", "fish"]

Building the Inverted Index (Hashmap):

```
cat  -> [(1(doc), 1(count)), (2, 1), (3, 1), (4, 1)]
love -> [(1, 1), (2, 1), (3, 1), (5, 1)]
fish -> [(2, 1), (5, 1)]
hate -> [(2, 1), (3, 1), (4, 1)]
bread -> [(2, 1), (5, 1)]
dog -> [(1, 1), (3, 1)]
```

As we can see, we created an inverted index where we note which document each word appears in and how many times it appears.

What is the ideal DB type for storing an inverted index?

While it might look like a **key-value** store, it isn't. The best type of database for storing an inverted index is a search engine database like Elasticsearch or Apache Solr. These databases are designed specifically for efficient full-text search and can handle inverted indexes very well.

Perform the Query ->

Query: **"I adore kittens & pups"**

To perform the query, we need to preprocess it with steps like tokenization and stop word removal. In the sentence below, we removed common words and used synonyms: "adore" becomes "love," "kittens" becomes "cat," and "pups" becomes "dog."

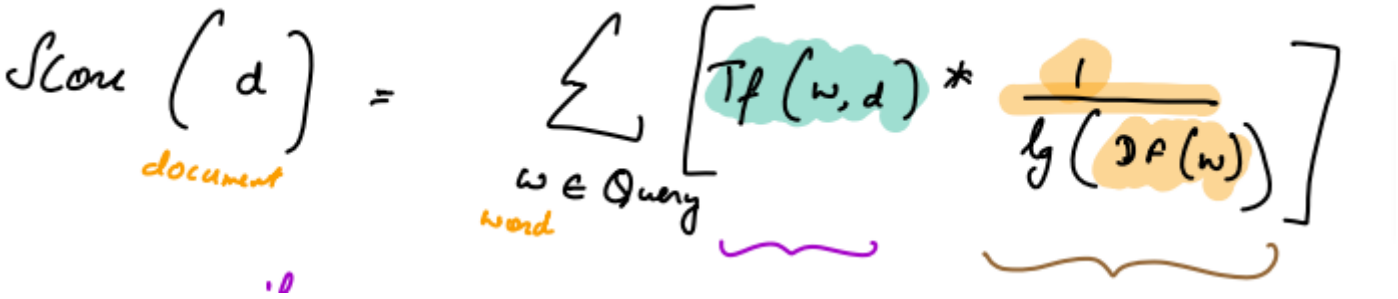
Step 1: **Preprocess the query**: "love cat dog"
Step 2: Find all docs (reviews) for each word and union them: Using our inverted index, find the document numbers for each word and combine them.

- **love** -> 1, 2, 3, 5, 6
- **cat** -> 1, 2, 3, 4, 6
- **dog** -> 1, 7, 6
- **Union** is 1, 2, 3, 4, 5, 6

Step 3: Rank using TF-IDF (Term Frequency-Inverse Document Frequency)

- **Term Frequency (TF)**: How many times does the word (w) appear in the document (d)?
- **Document Frequency (DF)**: How many different documents does the word (w) appear in?

Here is the formula



TF(w,d): We use TF(w,d) because if a query word appears more often in a document, that document is more relevant.

1/lg(df(w)): If a word is common, it has less importance because it contributes less meaning.

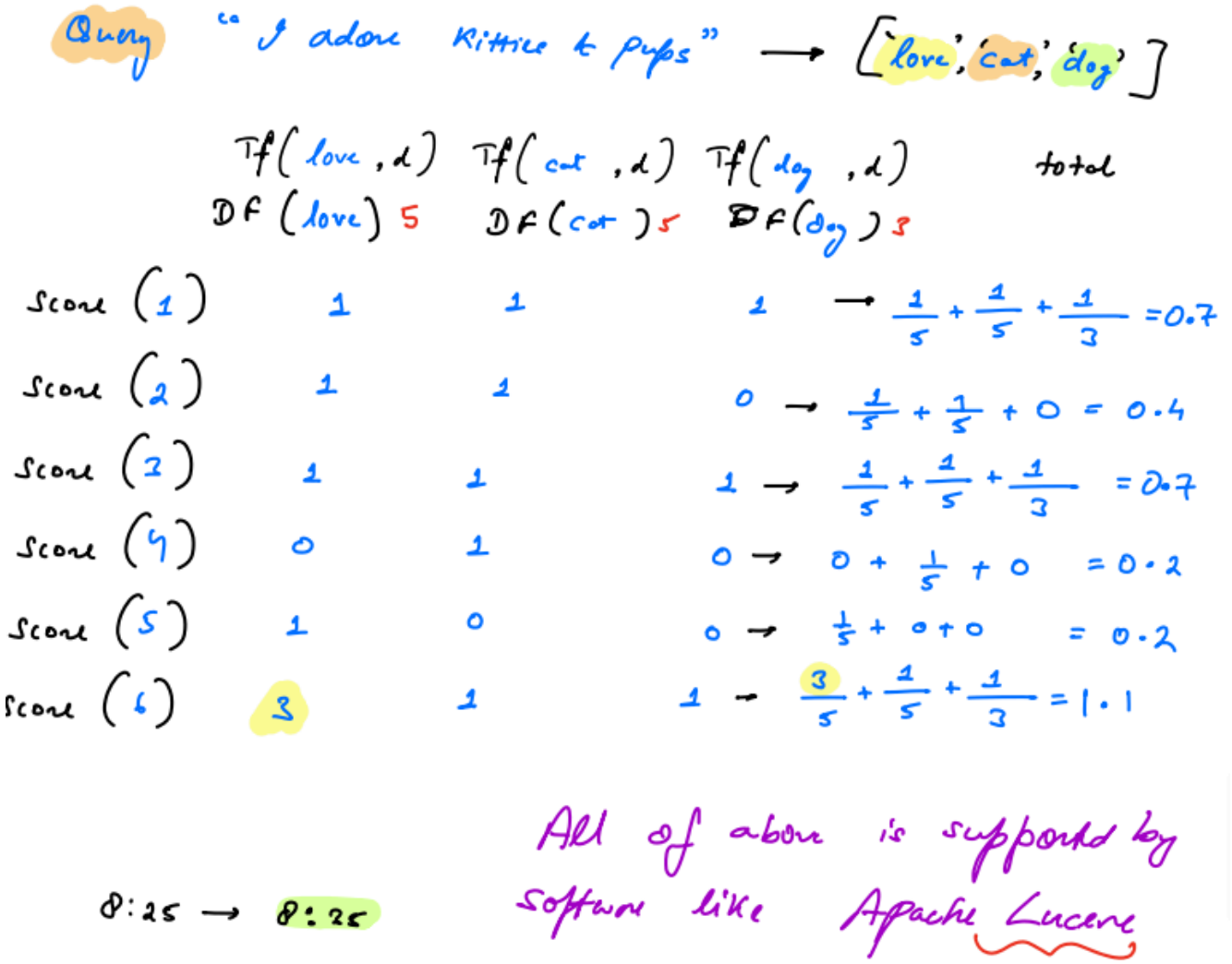
Logarithm: Using a log function helps normalize big values. This concept is similar to **Zipf's Law**.

Zipf's Law is a rule that describes how often words are used in language:

1. **Word Frequency**: The most common word is used the most. The second most common word is used about half as much as the first. The third most common word is used about one-third as much as the first, and so on.
2. **Example**: In a book, if the word "the" appears 1000 times, the next common word might appear 500 times, and the third common word might appear 333 times.
3. **Meaning**: A few words are used a lot, and most words are used rarely.

Zipf's Law helps understand and work with language and search algorithms.

As we seen in below left side we have all document in score() and write side we have a counts present in the documents and then we will get the total freq of its.



Scaling Full Text Search

We need to scale full text search for common use cases like:

- Product reviews (e.g., Amazon)
- Social media posts (e.g., Twitter, Facebook)
- Resume searches
- Indexing websites (e.g., Google Search, which does much more)
- Log processing, which can be in sizes up to petabytes

Sharding

Assume we have 2 types of data:

1. **Documents (Primary)**: Primary data for posts, reviews, logs. The ideal sharding key for this is `doc_id`.
2. **Inverted Index (Derived)**: This contains the inverted index created through preprocessing. The ideal sharding key for this is also `doc_id`, not individual words.

Note:

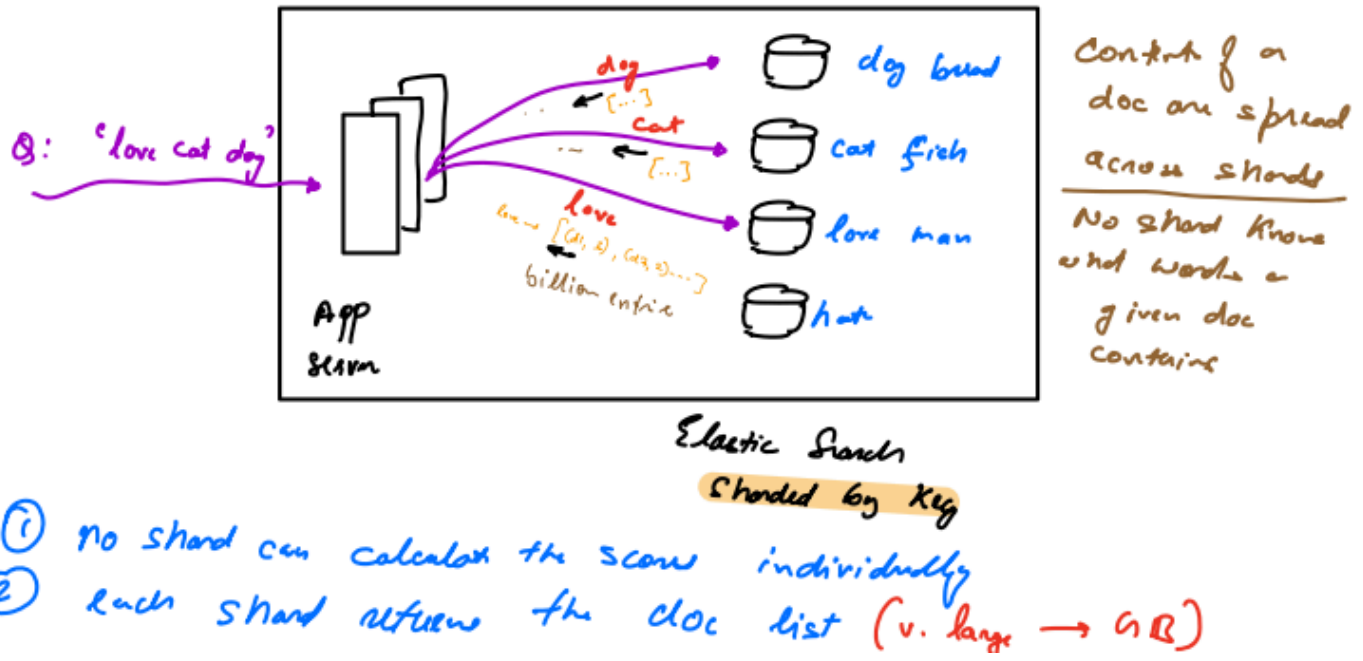
- Elasticsearch is sharded by `doc_id`.
- Elasticsearch is known as a document database.

If we shard the inverted index by words, different words go to different shards. When a query comes in, it is very hard to get the TF-IDF score because each shard only has information about one word, not all words in the query. This means we have to fan out the query, searching all shards,return list of documents from all shards. This process involves handling terabytes of data over the network, increasing network overhead.

We can't transfer gigabytes of data for every query. It is very expensive.

all app servers have to get all this data, process billions of documents, and calculate billions of TF-IDF scores, then sort and return the top 10 results, it becomes a very high computation load.

Sharding Inverted index by key (wrod)



Sharding Inverted Index by `doc_id`

We will shard by document, meaning each shard contains full documents. When a query comes in, all required words are in a single shard.

- Each shard builds an independent inverted index for the documents it contains.
- The app server simply forwards the query to all shards (fanout). This is manageable in this case.
- Each shard returns the top k results along with scores, and k is configurable. This reduces network overhead.
- The app server then merges these results and returns the top k. This reduces the computation load.

System Design - S3 + Quad trees (nearest neighbors)

Agenda:

How do we store large files?

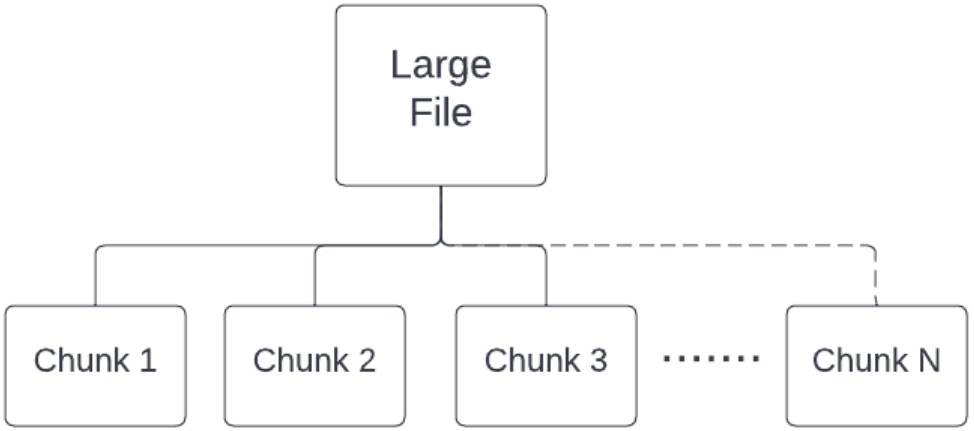
In earlier classes, we discussed several problems, including how we dealt with ***metadata***, facebook's newsfeed, and many other systems.

We discussed that for a post made by the user on Facebook with images(or some other media), we don't store the image in the database. We only store the metadata for the post (user_id, post_id, timestamp, etc.). The images/media are stored on different storage systems; from that particular storage system, we get a URL to access the media file. This URL is stored in the database file.

In this class, our main discussion is how to store these large files (not only images but very large files, say a 50 TB file). A large file can be a large video file or a log file containing the actions of the users (login, logout, and other interactions and responses), and it can keep increasing in size.

Conditions for building a large file system:
- Storage should be able to store large files
- Storage should be reliable and durable, and the files stored should not be lost.
- Downloading the uploaded file should be possible
- Analytics should be possible.

One way to store a large file is to divide it into chunks and store the chunks on different machines. So suppose a 50 TB file is divided into chunks. What will be the size of the chunks? If you divide a 50 TB file into chunks of 1 MB, the number of parts will be
50TB/1MB = $(50 * 10^6)$ MB / 1 MB = $5 * 10^7$ parts.



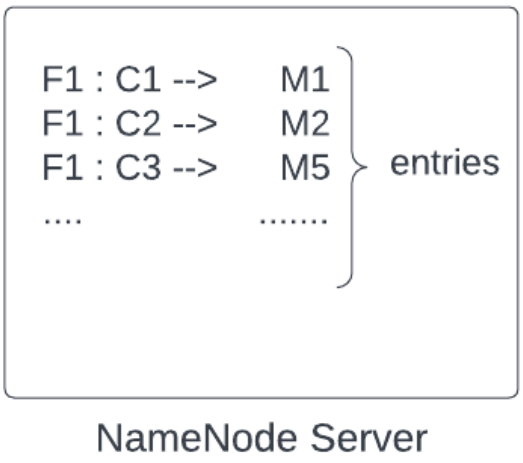
From this, we can conclude that if we keep the size of the chunk very small, then the number of parts of the file will be very high. It can result in issues like

1. **Collation of the parts:** concatenating too many files and returning them to the client will be overhead.
2. **Cost of entries:** We must keep metadata for the chunks,i.e., for a given chunk of a file, it is present on which machine. If we store metadata for every file, this is also an overhead.

HDFS

HDFS stands for Hadoop Distributed File System. Below are certain terminologies related to HDFS:

- The default chunk size is 128 MB in HDFS 2.0. However, in HDFS 1.0, it was 64 MB.
- The metadata table we maintain to store chunk information is known as the '**NameNode server**'. It keeps mapping that chunks are present on which machine(**data node**) for a certain file. Say, for File 1, chunk 1 is present on machine 3.
- In HDFS, there will be only one name node server, and it will be replicated.



You may wonder why the chunk size is 128 MB.
The reason is that large file systems are built for certain operations like storing, downloading large files, or doing some analytics. And based on the types of operations, benchmarking is done to choose a proper chunk size. It is like 'what is the normal file size for which most people are using the system' and keeping chunk size accordingly so that system's performance is best.

For example,

chunk size of X1 performance is P1
chunk size of X2 performance is P2,
Similarly, doing benchmarking for different chunk sizes.
And then choosing the chunk size that gives the best performance.

In a nutshell, we can say benchmarking is done for the most common operations which people will be doing while using their system, and HDFS comes up with a value of default chunk size.

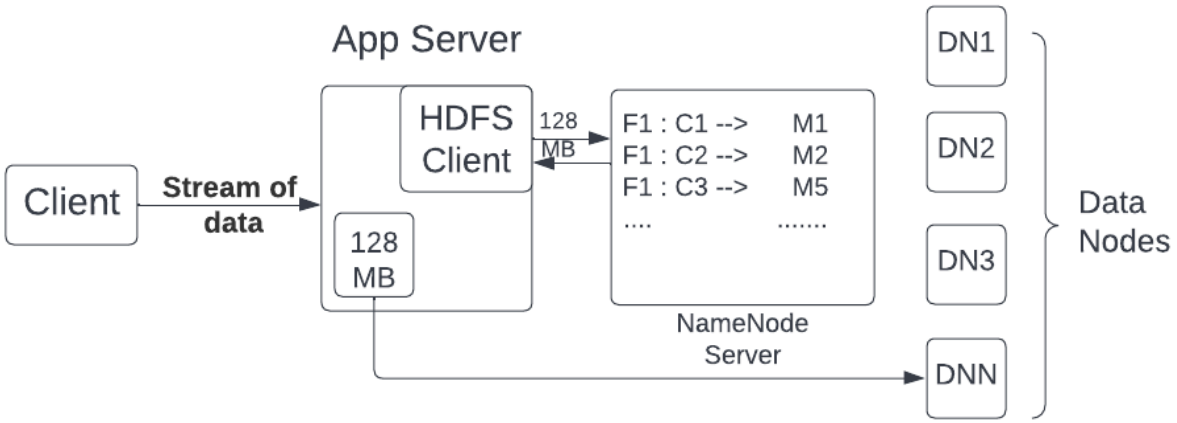
Making System reliable: We know that to make the distributed system reliable, we never store data on a single machine; we replicate it. Here also, a chunk cannot be stored on a single machine to make the system reliable. It needs to be saved on multiple machines. We will keep chunks on different data nodes and replicate them on other data nodes so that even if a machine goes down, we do not lose a particular chunk.

Rack Aware Algorithm: For more reliability, keep data on different racks so that we do not lose our data even if a rack goes down. We avoid replicating the chunks on the machines of the same rack. This is because if there comes an issue with the power supply, the rack will go down, and data won't be available anywhere else.

So this was about chunk divisions and storing them on HDD. Now comes the question of who does this division part.

The answer is it depends on the use case.

- Suppose there is a client who wants to upload a large file. The client requests the app server and starts sending the stream of data. The app server on the other side has a client (HDFS client) running on it.
- HDFS also has a NameNode server to store metadata and data nodes to keep the actual data.
- The app server will call the name node server to get the default chunk size, NameNode server will respond to it (say, the default chunk size is 128 MB).
- Now, the app server knows that it needs to make chunks of 128 MB. As soon as the app server collects 128 MB of data (equal to the chunk size) from the data stream, it sends the data to a data node after storing metadata about the chunk. Metadata about the chunk is stored in the name node server. For example, for a given file F1, nth chunk - Cn is stored in 3rd data node - D3.
- The client keeps on sending a stream of data, and again when the data received by the app server becomes equal to chunk size 128 MB (or the app server receives the end of the file), **metadata about the chunk is stored in the name node server first and then chunk it send to the data node.**



Briefly, the app server keeps receiving data; as soon as it reaches the threshold, it asks the name node server, 'where to persist it?', then it stores the data on the hard disk on a particular data node received from the name node server.

Few points to consider:

- For a file of 200MB, if the default chunk size is 128 MB, then it will be divided into two chunks, one of 128 MB and the other of 72 MB because it is the only data one will be receiving for the given file before the end of the data stream is reached.
- The chunks will not be saved on a single machine. We replicate the data, and we can have a master-slave architecture where the data saved on one node is replicated to two different nodes.
- We don't expect very good latency for storage systems with large files since there is only a single stream of data.

Downloading a file
Similar to upload, the client requests the app server to download a file.

- Suppose the app server receives a request for downloading file F1. It will ask the name node server about the related information of the file, how many chunks are present, and from which data nodes to get those chunks.
- The name node server returns the metadata, say for File 1, goto data node 2 for chunk 1, to data node 3 for chunk 2, and so on. The application server will go to the particular data nodes and will fetch the data.
- As soon as the app server receives the first chunk, it sends the data to the client in a data stream. It is similar to what happened during the upload. Next, we receive the subsequent chunks and do the same.

(More about data streaming will be discussed in the Hotstar case study)

Torrent example: Do you know how a file is downloaded very quickly from the torrent?

What is happening in the background is very similar to what we have discussed. The file is broken into multiple parts. If a movie of 1000MB is broken into 100 parts, we have 100 parts of 10 MB each.
If 100 people on torrent have this movie, then I can do 100 downloads in parallel. I can go to the first person and ask for part 1, the second person for part 2, and so forth. Whoever is done first, I can ask the person for the next part, which I haven't asked anybody yet. If a person is really fast and I have gotten a lot of parts, then I can even ask him for the remaining part, which I am receiving from someone, but the download rate is very slow.
https://www.explainthatstuff.com/howbittorrentworks.html

Nearest Neighbors

There are a lot of systems that are built on locations, and location-based systems are unique kinds of systems that require different kinds of approaches to design. Conventional database systems don't work for location-based systems.

We will start the discussion with a problem statement:
On google Maps, wherever you are, you can search for nearby businesses, like restaurants, hotels, etc. If you were to design this kind of feature, how would you design the feature that will find you nearest X number of neighbors(say ten nearby restaurants)?

Bruteforce

Well, the brute-force approach can simply get all restaurants along with their locations (latitude and longitude) and then find the distance from our current location (latitude and longitude). Simply euclidian distance between two points (x1, y1) and (x2, y2) in 2D space can be calculated with the formula $d = \sqrt{[(x2 – x1)^2 + (y2 – y1)^2}$. This will calculate the distance around all points around the globe to get our X (say 10) nearest neighbors. The approach will take a lot of time.

Finding Locations Inside a Square

We cannot use the circle to get all the locations around the current position because there is no way to mark the region; therefore, using a square to do the same.

Another approach is to draw a square from our current location and then consider all the points/restaurants lying inside it to calculate X nearest ones. We can use the query:

```
SELECT * FROM places WHERE lat < x + k AND lat > x -k  AND long < y + k AND long > y-k
```

Here **‘x’** and **‘y’** are the coordinates of our current location, **‘lat’** is latitude, **‘long’** is longitude, and **‘k’** is the distance from the point **(x,y)**.

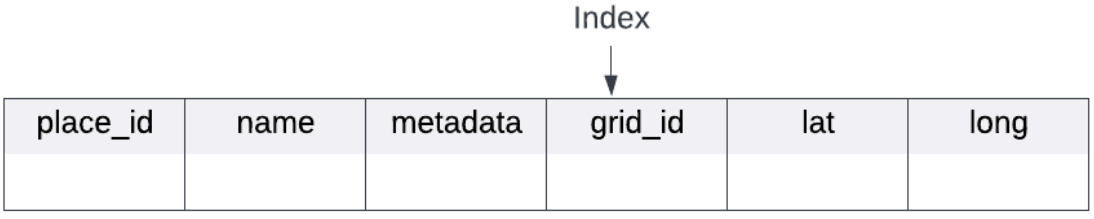
However, this approach has some issues:

1. Finding the right ‘**k**’ is difficult.
2. Query time will be high: Only one of lat or long index can be used in the above query and hence the query will end up spanning a lot of points. .

Grid Approach

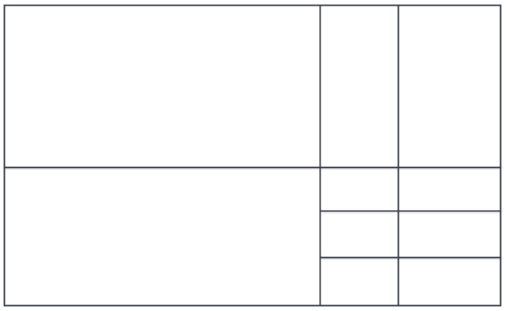
We can break the entire world into small grids (maybe 1 km sq. grids). Then to get all the points, we only need to consider the locations in the grid of our current location or the points in the adjacent grids. If there are enough points in these grids, then we can get all the nearest neighbors. The query and to get all the neighbors is depicted below:

```
SELECT * FROM places WHERE  grid_id IN (grid1, grid2, grid3……)
```



What should be the size of the grid?
It is not ideal to have a uniform grid size worldwide. The grid size should be small for dense areas and large for sparse areas. For example, the grid size needs to be very large for the ocean and very small for densely populated areas. The thumb rule is that size of the grid is to be decided based on the number of points it

contains. We need to design variable-size grids so that they have just enough points. Points are also dynamically evolving sets of places.



Varible Size Grids

Dividing the entire world into variable-size grids so that every grid has approximately 100 points
So our problem statement reduces to preprocess all the places in the world so that we can get variable-size grids containing 100 points. We also need to have some algorithm to add or delete a point (a location such as a restaurant).

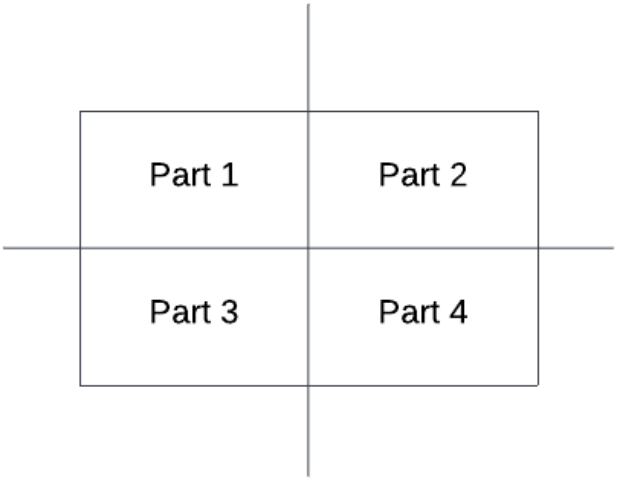
This can be achieved using **quadtrees**.

QuadTree

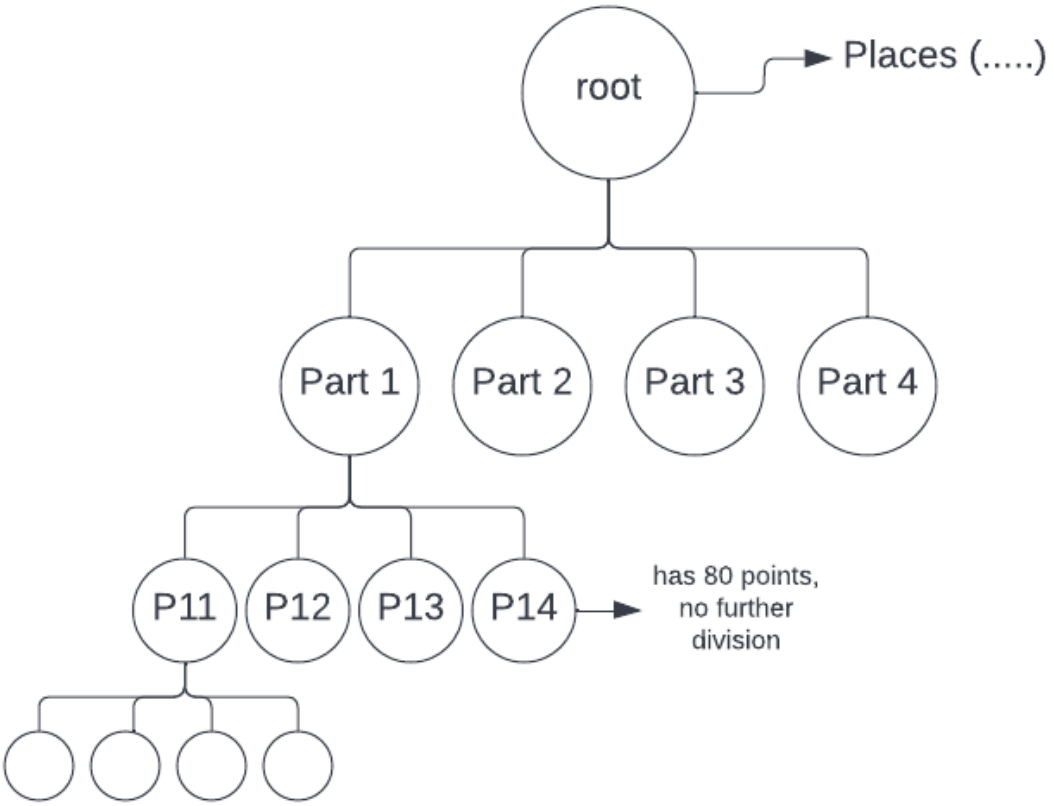
Creation

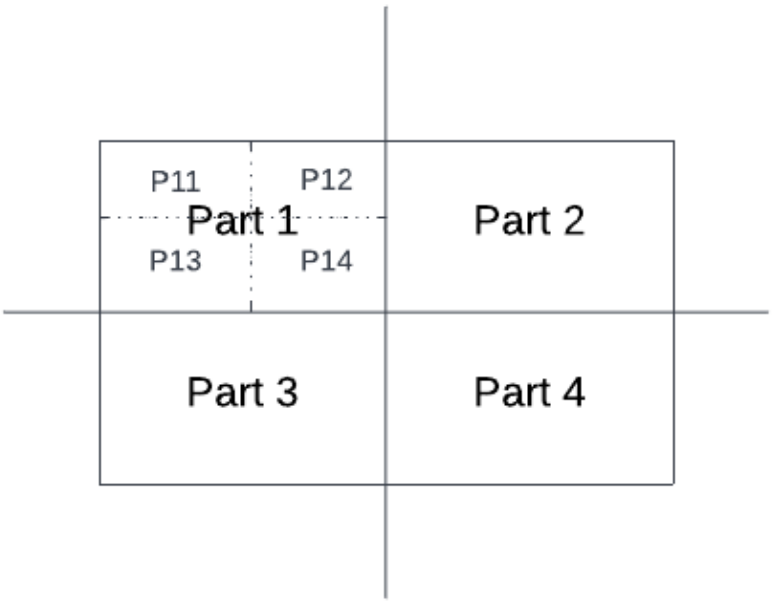
Imagine the entire world with billions of points (think of a world map, a rectangle with points all over).

- We can say that the entire world is a root of a tree and has all of the places of the world. We create a tree; if the current node has more than 100 points, then we need to create its four children (four because we need to create the same shape as a rectangle by splitting the bigger grid into four parts).



- We recursively repeat the process for the four parts as well. If any children have more than 100 points, it further divides itself into four children. Every child has the same shape as the parent, a rectangle.

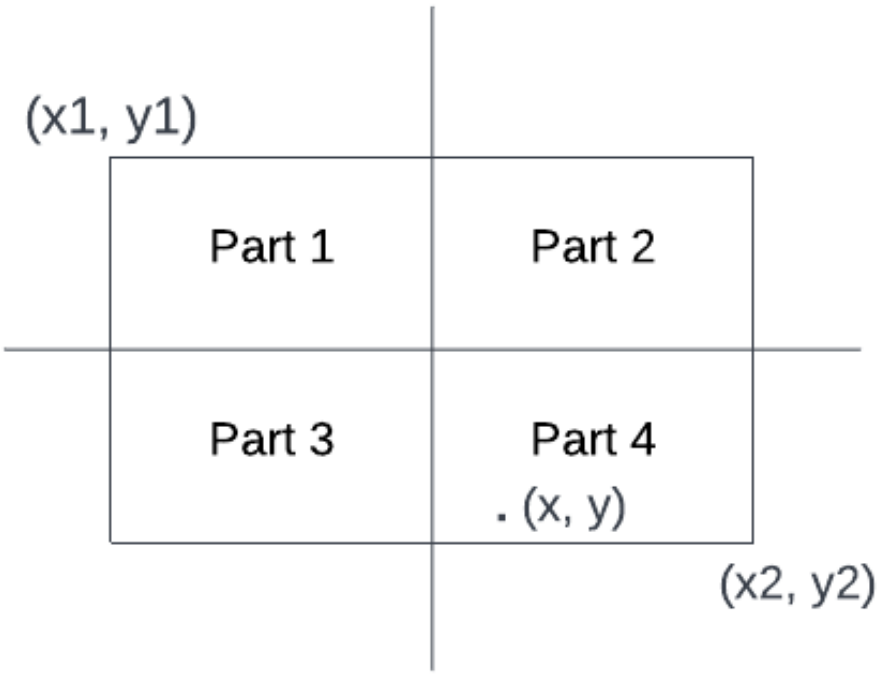




- All the leaf nodes in the tree will have less than 100 points/places. And the tree's height will be **~log(N),** **N** being the number of places in the world.

Finding Grid ID

Now, suppose I give you my location **(x,y)** and ask you which grid/leaf I belong to. How will you do that? You can assume the whole world extends between coordinates (x1, y1) and (x2, y2).



What I can do is calculate the middle point for the x and y coordinates, X_{mid}= (x1+x2) / 2, Y_{mid}= (y1 + y2) / 2. And then, I can check if the x is bigger than X_{mid}. If yes, then the point will be present in either part 2 or 4, and if smaller than X_{mid}, the point will be in part 1 or 3. After that, I can compare y with Y_{mid} to get the exact quadrant.

This process will be used to get the exact grid/leaf if I start from the root node, every time choosing one part out of 4 by the above-described process as I know exactly which child we need to go to. Writing the process recursively:

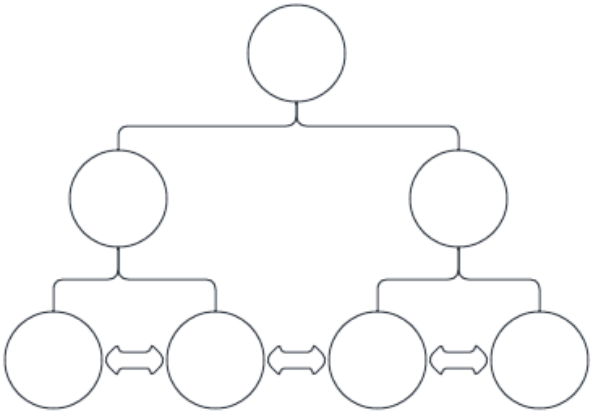
```
findgrid(x, y, root):
    X1, Y1 = root.left.corner
    X2, Y2 = root.right.corner
    If root.children.empty():      // root is already a leaf node
          Return root.gridno       // returning grid number
    If x > (X1 + X2) / 2:
          If y > (Y1 + Y2) / 2:
                findgrid(x, y, root.children[1])
          Else:
                findgrid(x, y, root.children[3])

     Else  y > (Y1 + Y2) / 2:
          If x > (X1 + X2) / 2:
                findgrid(x, y, root.children[0])
          Else:
                findgrid(x, y, root.children[2])
```

What is the time complexity of finding the grid to which I belong by above-mentioned method?
It will be equal to the height of the tree: log(N).

Once we find the grid, it becomes easy to calculate the nearby points. Every place in the world has been assigned a grid number and it is stored in MySQL DB. We can easily get all the required neighboring points. If neighbors are not enough, we also have to consider neighboring grids.

To find the neighboring grids:

- Next pointer Sibling: While creating the tree, if we also maintain the next pointer for the leaves, then we can easily get the neighbors. It becomes easy to find siblings. We can travel to the left or right of the leaf to get the siblings.



- Another way is by picking a point very close to the grid in all eight directions. For a point (X, Y) at the boundary, we can move X slightly, say X + 0.1, and check in which grid point (X+ 0.1, Y) lies. It will be a log(N) search for all 8 directions, and we will get all the grid ids.

Add a new Place

If I have to add a point (x, y), first, I will check which leaf node/ grid it belongs to. (Same process as finding a grid_id). And I will try to add one more place in the grid. If the total size of points in the grid remains less than the threshold (100), then I simply add. Otherwise, I will split the grid into four parts/children and redivide the points into four parts. It will be done by going to MySQL DB and updating the grid id for these 100 places.

Delete an existing place

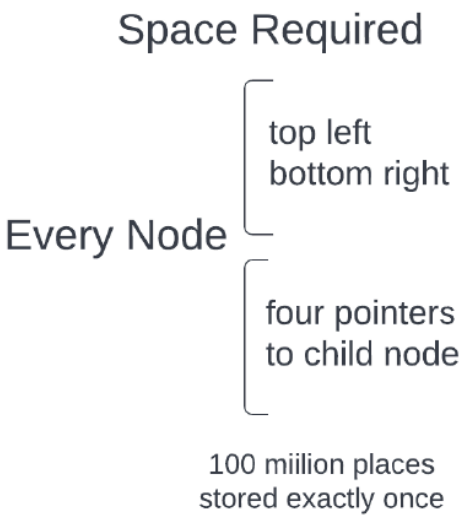
Deletion is exactly the opposite of addition. If I delete a point and the summation of all the points in the four children becomes less than 100, then I can delete the children and go back to the parent. However, the deletion part is not that common.

How to store a quad tree

For 100 million places, how can we store a quadtree in a machine?
What will we be storing in the machine, and how much space will it need?

So what do you think, whether it's possible to store data, i.e., 100 million places in a single machine or not?

Well, to store a quadtree, we have to store the **top-left** and **bottom-right** coordinates for each node. Apart from that, we will store **four pointers** to child nodes. The 100 million places will be stored in leaves only; every time a node contains more than X(say 100) places, we will split it into four children. In the end, all leaves will contain less than equal to 100 places, and every place will be present in exactly one leaf node.



Let's do some math for the number of nodes and space required.

Say every leaf stores one place (for 100 million places); there will be 100 million leaf nodes in the quadtree.

Number of parents of leaf nodes will be = (100 million) / 4
Parents of parents will be = (100 million) / 4
And so on.

So total number of nodes = $10^8 + 10^8/4 + 10^8/16 + 10^8/64$ ……...
= 10^8 (1 + 1/4 + 1/16 + 1/64 ……..)
The above series is an infinite G.P., and we can calculate the sum using formula 1/(1-r) { for series 1 + r + r^2 + r^4 + ……..}. In the above series, r = 1/4
*The sum will 10^8 * 1/(1-(1/4)) = 10^8 * (4/3) = 1.33 * 10^8*

*If we assume every leaf node has an average of 20 places, the number of nodes will be equal to (1.33 * 10^8) / (average number of places in a leaf) = (1.33 * 10^8) / 20*

*(1.33 * 10^8) / 20 = (1.33 * 5 * 10^6) =* ***6.5 million nodes***

So we have to store *100 million places + 6.5 million nodes*.

Now calculating space needed:
For every node, we need to store top-left and bottom-right coordinates and four pointers to children nodes. Top-left and bottom-right are location coordinates (latitude and longitude), and let's assume we need two doubles (16 bytes) to get the required amount of precision.

*For boundary, the space required will be 16 * 4 = 64 bytes.*

Every pointer is an integer since it points to a memory location,

*Storage required for 4 pointers = 4bytes * 4 = 16 bytes*

Every node requires 64 + 16 = 80 bytes

To store 100 million places, the storage required (say latitude and longitude each is 16 bytes)
*= 10^8 * 32*

Total space required = space required for nodes + space required for places
*= 6.5 million * 80 bytes + 100 million * 32 bytes*
*= 520 * 10^8 bytes + 3200 * 10^8 bytes*
= ~ 4000 million bytes = ***4GB***

So the total space required is 4GB to store a quadtree, and it can easily fit inside the main memory. All we need is to make sure that there are copies of this data is multiple machines so that even if a machine goes down, we have access to data.

A lot of production systems have 64GB RAM, and 4GB is not a problem to store.

Problem statements for the next class

1. How can we find the nearest taxi cabs and get matched to one of them? Note that taxis can move, and their location is not fixed :) {Uber case study}

Extra Reference Material

1. HDFS Architecture: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
2. HDFS Archival: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html
3. GeoHash
    a. https://www.youtube.com/watch?v=UaMzra18TD8
    b. https://www.youtube.com/watch?v=vGKs-c1nQYU
4. Comparison of Geohash, QuadTree, R-Tree https://tarunjain07.medium.com/geospatial-geohash-notes-15cbc50b329d

Overflow topics - LSM Trees & Kafka - Jul 13

Agenda:
LSM Trees

Important links

Quad Trees

1. Theory:
2. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/quadtrees.pdf
3. Practice: https://leetcode.com/problems/construct-quad-tree/description/
4. Visualization: https://jimkang.com/quadtreevis/
5. Proximity services: https://www.youtube.com/watch?v=M4lR_Va97cQ

Bloom Filters

1. https://brilliant.org/wiki/bloom-filter/
2. https://www.youtube.com/watch?v=V3pzxngeLqw
3. https://www.youtube.com/watch?v=gBygn3cVP80
4. https://www.youtube.com/watch?v=-jiOPKt7avE
5. https://www.youtube.com/watch?v=Z9_wrhdbSC4

Issues with SSTables

1. **Linear Scan**:
    - **Problem**: When there are too many SSTables, reading data takes a long time because it requires scanning each table one by one.
    - **Solution**: Reduce the number of SSTables through compaction.
2. **Binary Search on Disk**:
    - **Problem**: Binary search is slow on disk because it requires random reads, which involve jumping to different locations on the disk.
    - **Solution**: Move the binary search to RAM for faster access, and use linear reads on disk.
3. **Wasting Space**:
    - **Problem**: Older SSTables have values that are overwritten by newer SSTables, wasting space.
    - **Solution**: Compaction helps by merging tables and removing old or overwritten data.
4. **Deletion**:
    - **Problem**: Deleting data can leave tombstones (markers for deleted data) that still take up space.

Solving Issues with Compaction

- **First and Third Issues**: Compaction merges SSTables, reducing their number and eliminating outdated data, which saves space and improves read performance.

- **Background Process**: Compaction runs in the background, continuously organizing SSTables into fewer, larger, and more efficient tables.

LSM Tree

- **LSM Tree**: Compaction forms the structure of an LSM tree, which is designed to handle high write volumes and improve read performance through efficient management of SSTables.
- First shard the data then create LSM trees inside shard only it is not a global.

Compaction in SSTables

SSTables (Sorted String Tables) are used in databases that use LSM (Log-Structured Merge) trees. Compaction is a key process to manage SSTables efficiently. Here's how it works in simple terms:

What are SSTables?

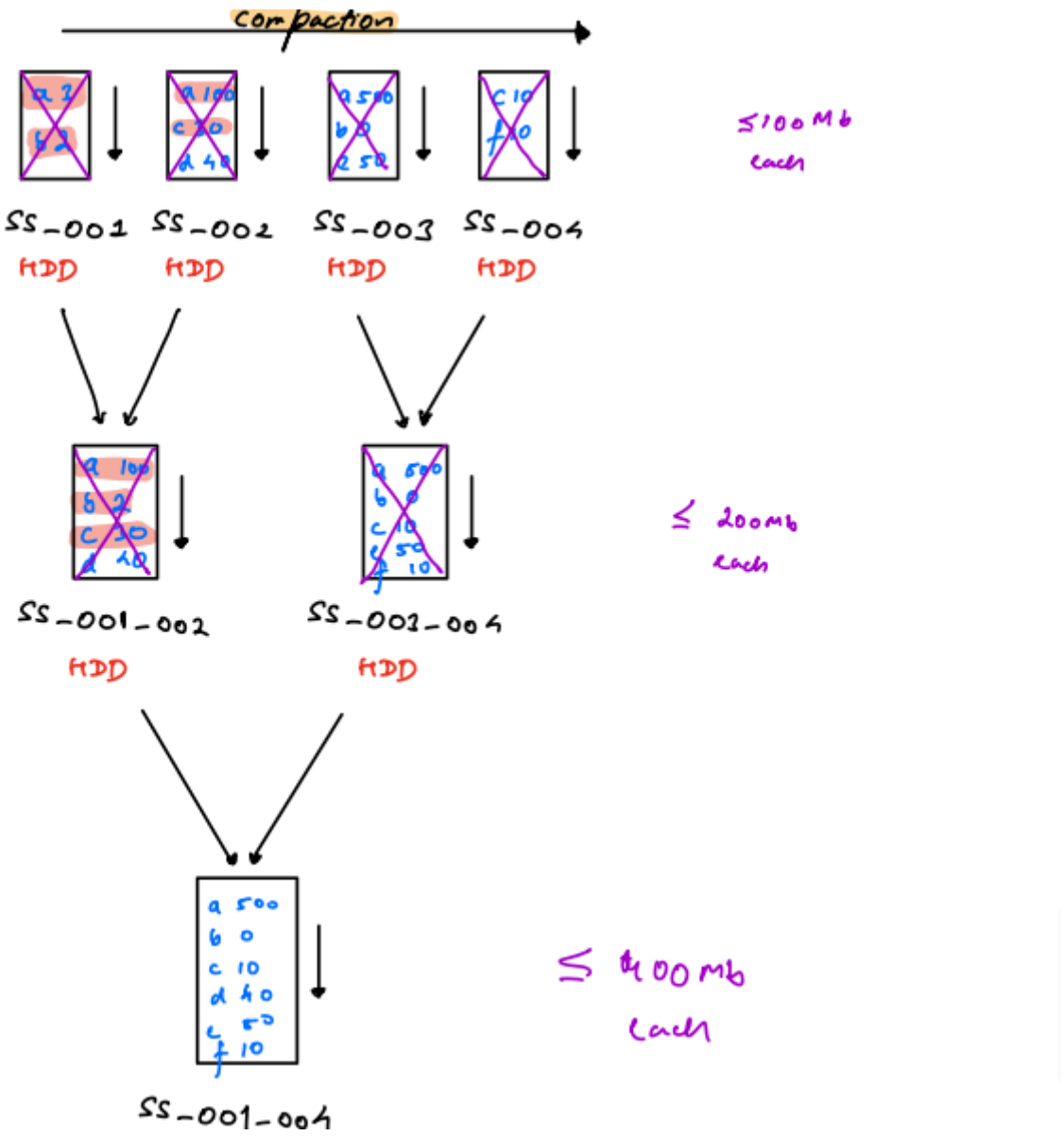
- **SSTables** are immutable (unchangeable) files that store sorted key-value pairs.
- They are created when the in-memory data structure (like a MemTable) is flushed to disk.

Why Compaction is Needed?

1. **Multiple SSTables**: Over time, many SSTables are created, leading to a lot of small files on disk.
2. **Overlapping Data**: Different SSTables can have overlapping data, meaning the same key might be present in multiple SSTables.
3. **Old Data**: Some data might be outdated or deleted, taking up unnecessary space.

Compaction Process

1. **Selection**: Choose several SSTables that need to be compacted.
2. **Merging**: Merge the selected SSTables into one or more new SSTables. It is happen in backgroun. first it will copy and then do the merging because of this read operation continue.
3. **Sorting**: Ensure the data is sorted during the merging process.
4. **Removing Duplicates**: Discard old versions of data and deleted entries.



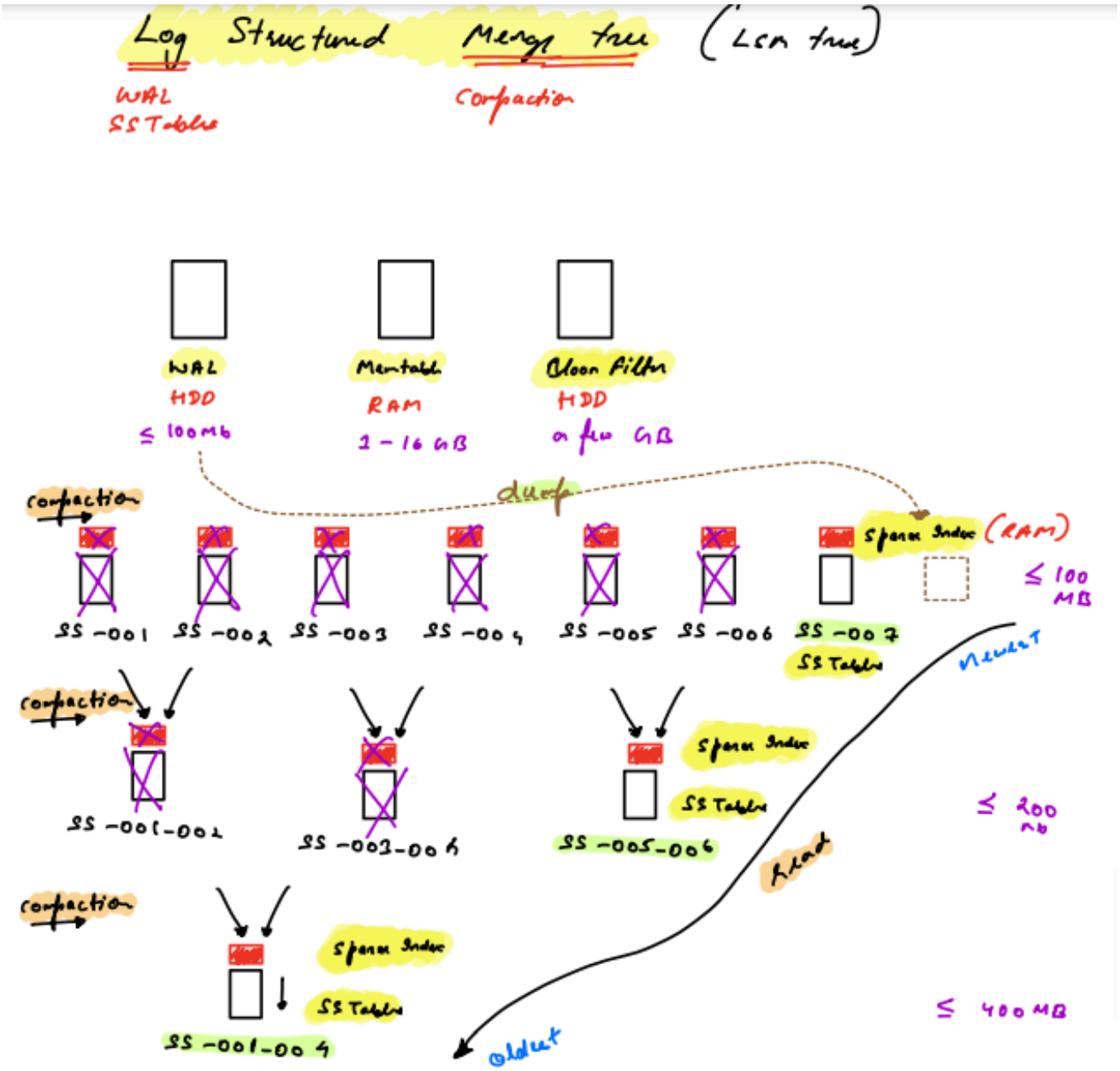
How Compaction Works
When compaction happens, it merges SSTables from the left and reads the most recent values from the rightmost merged SSTable.

This ensures that the latest data is always available and older, outdated data is removed. This process helps in reducing the number of SSTables and keeping the data more organized and efficient for reading.

Understanding how compaction works is important because you need to configure when it runs. Poorly managed compaction can severely affect database performance.

- **Compacting Too Often**:
  - **Issue**: If compaction happens too frequently, the database spends too much time compacting, which increases the read load since reads also occur during compaction.
  - **Result**: This slows down the database because it's busy with frequent compactions and reads.
- **Not Compacting Enough**:
  - **Issue**: If compaction doesn't happen often enough, the database ends up with many SSTables.
  - **Result**: This slows down both write and read operations because the database has to handle many SSTables.

Properly configuring compaction ensures the database remains efficient, with fewer SSTables and faster read and write operations.



Sparse Index

To solve the issue of slow binary search on disk, we use a sparse index. A sparse index is created when the table is created and deleted when the table is deleted. These sparse indexes are stored in RAM and are associated with every SSTable.

Disk Reads and Writes

Disk reads and writes always happen in blocks:

- **4 KB** for HDD
- **512 KB** for SSD

Example Calculation

Number of blocks in a 100 GB file:

$$\frac{100\,\text{GB}}{4\,\text{KB}} = \frac{100\times10^9\,\text{bytes}}{4\times10^3\,\text{bytes}} = 2.5\times10^7 = 25\,\text{million blocks}$$

Number of entries (key-values) in the 100 GB file (average entry size = 100 bytes):

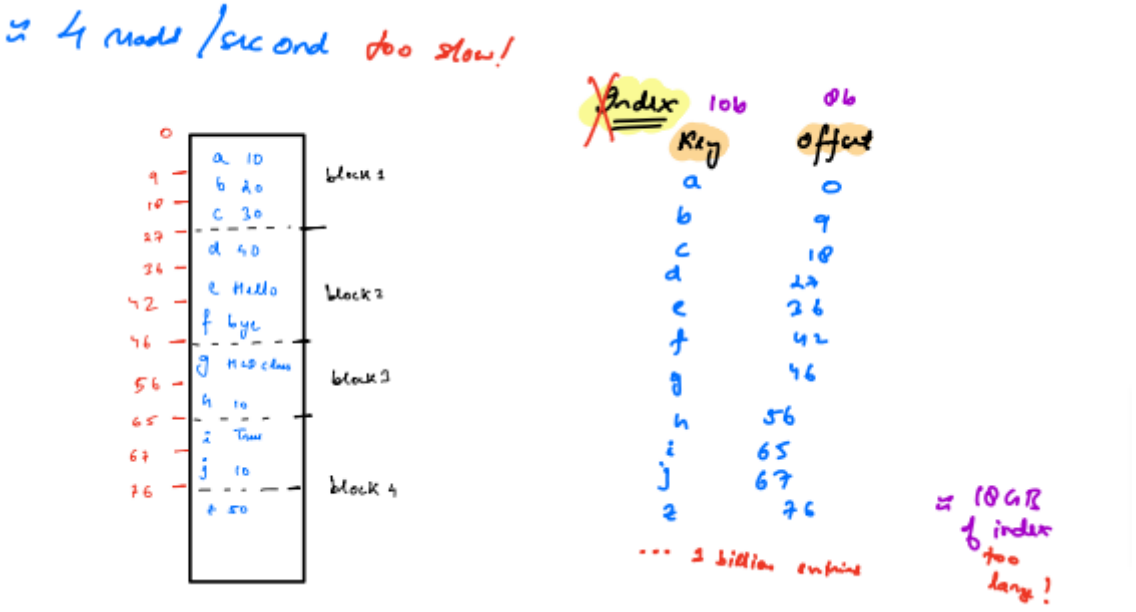
$$\frac{100\,\text{GB}}{100\,\text{bytes}} = 1\,\text{billion entries}$$

Read without Sparse Index

1. **Binary Search on Disk**:
   - Perform a binary search on the disk, which involves reading blocks (since disks read entire blocks, not specific entries).
   - Within each block, perform another binary search in RAM to read the block entries.
2. **Disk Reads**:
   - The number of reads depends on the number of blocks, not entries.
   - With 1 billion entries, there are 25 million blocks.
3. **Time Complexity**:
   - $$\log_2(\text{number of blocks}) = \log_2(25\times10^6) \approx 25$$
   - This means approximately 25 disk reads are needed.
   - Each read requires a disk seek, which involves the disk head moving to the correct position while spinning.

Disk Seek Time:

- Each disk seek takes about 10 ms.
  $$25\,\text{disk seeks}\times10\,\text{ms per seek} = 250\,\text{ms}$$
- This results in a read time of approximately 250 ms, which is very high.



Read with Index

If we have an index in RAM, we can read directly from the disk in a single access, reducing 25 disk accesses to just 1. However, with 1 billion entries and each entry being 18 bytes, this would require an 18 GB index, which is too large and impractical.

Read with Sparse Index

The idea behind a sparse index is to store only the starting keys of each block, rather than all the keys. This sparse index is simply a sorted array. Since there are 25 million blocks, the sparse index will have 25 million entries, with each key being about 10 bytes.

Size Calculation:

- Size of the sparse index:

  $$10\,\text{bytes} \times 25\,\text{million} = 250\,\text{MB}$$

  On HDD (4 KB blocks):

- Still manageable at 250 MB.
- On SSD (512 KB blocks):
  - Also manageable at 250 MB.

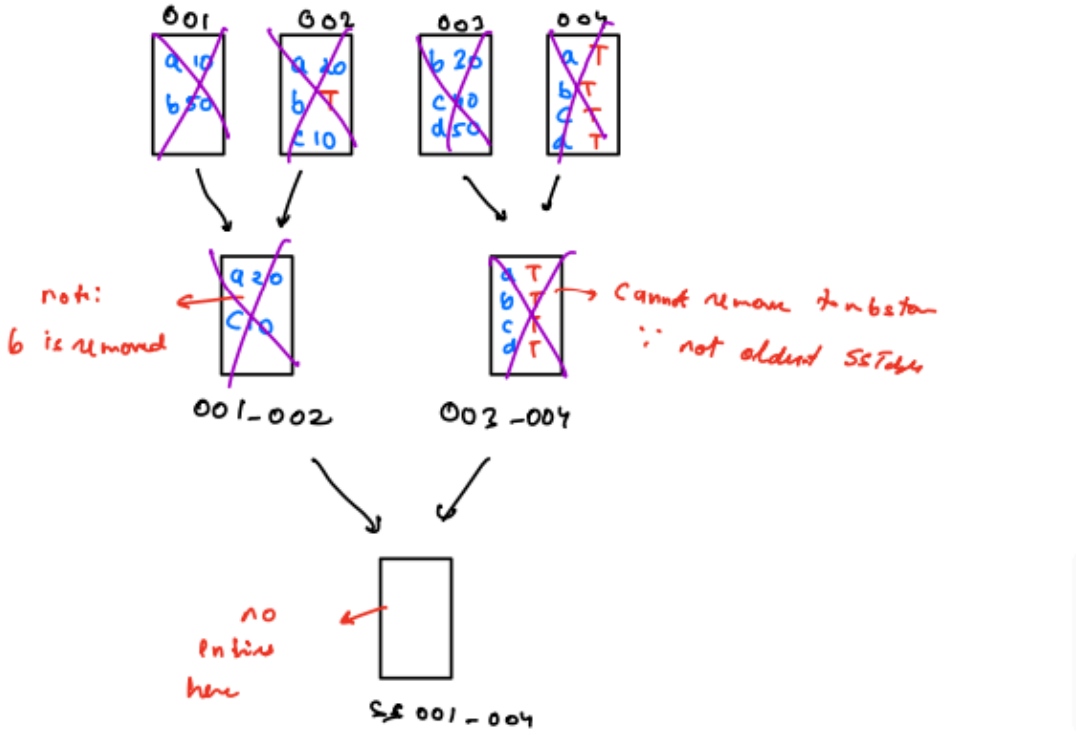
With the sparse index, we can perform a binary search for the given key and quickly locate the corresponding block on the disk. This allows us to read the entire block into RAM in a single disk read. Then, we can perform another binary search in RAM to find the specific entry within that block, effectively reducing 25 disk accesses to just 1, which is very fast.

Deleting an Entry from SSTable

When we delete an entry from the MemTable and do not insert the deleted key into the SSTable, the entry will not be found in the MemTable. If we try to read from the SSTable, it won't be found in the most recent table but may still exist in older SSTables during a linear scan.

To address this, we use a **sentinel value** (guard value) or **tombstone**. Instead of actually deleting the value, we update it and add a special flag that is guaranteed not to occur in the data, like a 128-bit random string. This means the deletion is treated as an update.

Note: Tombstones (the flag values) will eventually be removed during compaction if there are no other entries to the left. This means we keep the tombstone in the most recent table and remove it during the final merge process.



Reading a Value That Was Never Inserted

Reading a value that was never inserted may seem like a fast operation, but it can actually be very slow. First, we check the MemTable, and if the value isn't found, we then perform a linear search in the SSTables, which is a slow process. To solve this, we can use a Bloom filter.

Bloom Filter (probabilistic set)

A **Bloom filter** is a space-efficient data structure used to test whether an element is a member of a set. It can quickly tell you if an item is definitely **not** in the set or possibly **may be** in the set.

- It is a probabilistic data structure.
- It is very space-efficient.
- It can store infinite data in finite space, but as the amount of data increases, the probability of errors also increases.

Bloom filter is a **Probabilistic set**. This means it can tell you if an item is:

- **Definitely Not in the Set**: If the Bloom filter returns false, the item is definitely not present.
- **Possibly in the Set**: If it returns true, the item might be in the set, but there is a chance of a false positive.

This probabilistic nature allows Bloom filters to be very space-efficient while trading off some accuracy.

How operation works with bloom fitler

- Before inserting a key-value entry into the database, first insert the key into the Bloom filter.
- When a read request occurs, check if the key is in the Bloom filter:
- If yes, the key may have been inserted in the database, so proceed to read from the database.
- If no, the key was never inserted, and we can return a 404 response.

How It Works

1. **Bit Array**: A Bloom filter uses a fixed-size bit array (all bits start as 0).
2. **Hash Functions**: It uses multiple hash functions to map each item to several positions in the bit array.
3. **Adding Items**:
    - When you add an item, the hash functions calculate several positions in the bit array, and those bits are set to 1.
4. **Checking Membership**:
    - To check if an item is in the set, the hash functions again compute the positions in the bit array.
    - If all the bits at those positions are 1, the item might be in the set. If any bit is 0, the item is definitely not in the set.

Benefits

- **Space Efficiency**: Uses much less memory compared to storing actual items.
- **Fast Operations**: Both adding items and checking membership are very fast.

Drawbacks

- **False Positives**: It can mistakenly indicate that an item is in the set when it is not (false positive).
- **No Deletion**: You cannot remove items from a Bloom filter without affecting other entries.

Use Cases

- **Databases**: To quickly check if a key exists before querying the database.
- **Web Caches**: To avoid unnecessary data retrieval.
- **Network Systems**: To reduce the amount of data sent over the network.

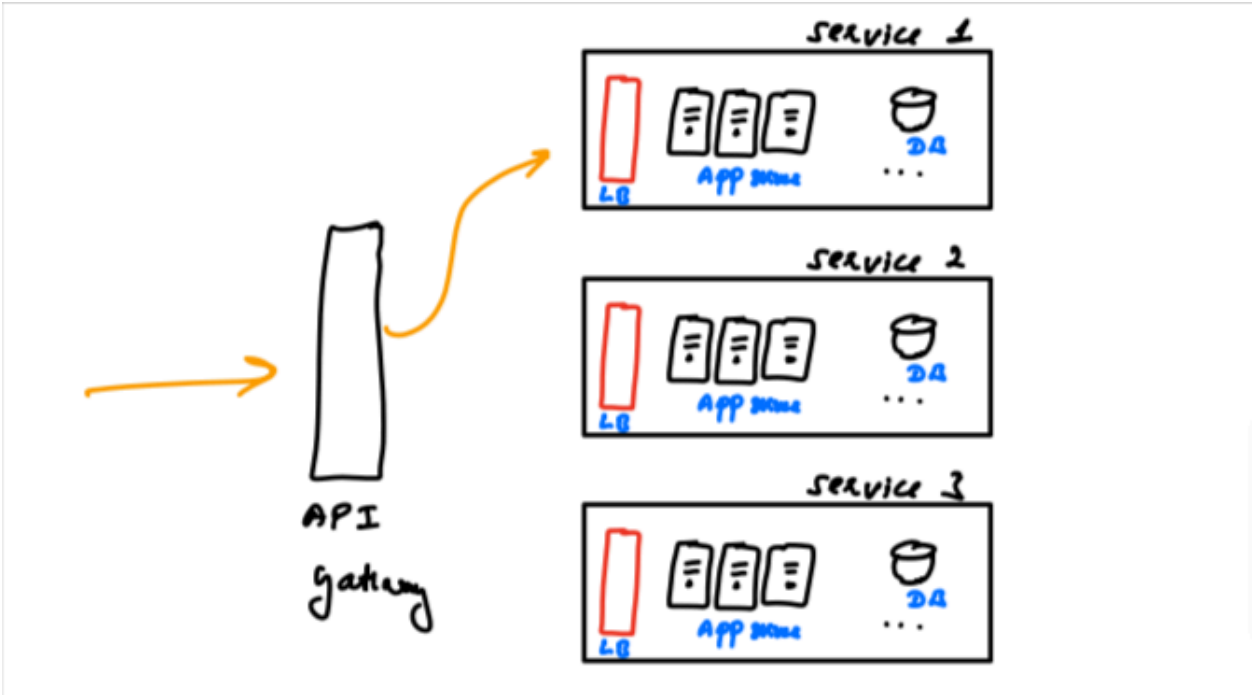
HLD: Microservices 2 - Jul 24

Agenda:

- **Untrusted network login**
- **Breaking a monolith into microservices**
- **Services Oriented Architecture (SOA) vs. Microservices Architecture**

https://microservices.io/

So far, we have seen that every microservice needs a load balancer with many servers and a separate database. This means we have many microservices, and one API service will manage all of them. But there is a problem: it increases infrastructure costs. For example, if we have 100 microservices, we need 100 load servers. To solve this, we will remove the load balancer from each microservice because it is set up the same way. Instead, we will create one load balancer for all microservices and solve it using Untrusted network login.



Untrusted network login:

To achieve this, we will separate the backend into two layers: one is the untrusted network and the other is the microserver, like a trusted VPC (Virtual Private Cloud). The trusted VPC is not directly connected to the internet; users must communicate through the untrusted layer. The untrusted network can have a local network but is not accessible from the internet.

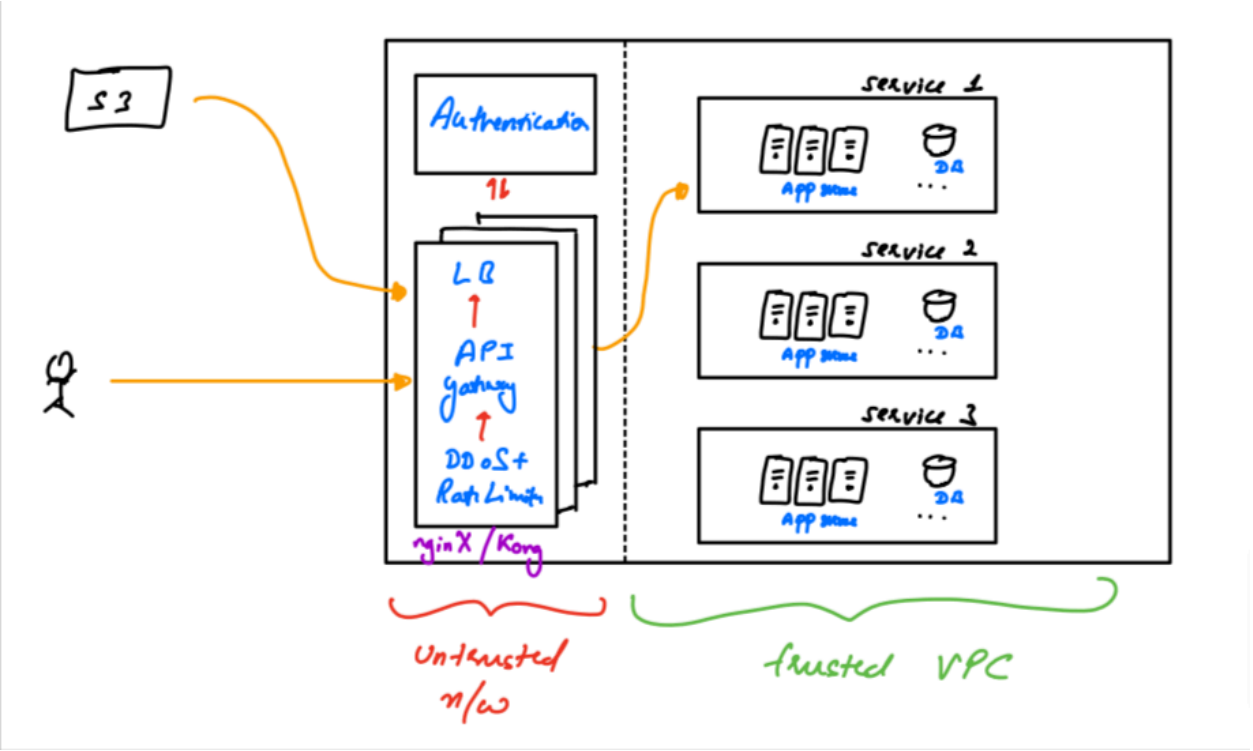
In the untrusted network, we will:

1. Keep the Authorization service on a separate server.
2. Have a load balancer, API gateway, rate limiter, and protection against DDoS (Distributed Denial of Service) attacks on another server.

Here's how it works:

- The first request goes to the rate limiter.
- Then it goes through the DDoS protection.
- Next, it reaches the API gateway.
- Finally, it hits the load balancer, which then communicates with the authentication service.
- Once authenticated, the request is sent to the microservice.

Many companies offer services for load balancing, rate limiting, DDoS protection, and API gateways, such as NGINX and Kong. After the untrusted network authenticates the request, authentication happens again at the application server level for each microservice.



Breaking a monolith into microservices:

1. **List all modules** in the monolith and create a graph showing their dependencies.
2. **Find clusters** of modules that are tightly coupled (modules that depend on each other).
3. **Focus on the easiest tasks first**:
    - Look for the easiest modules to migrate.
    - Consider the business impact of migrating each module. For example, a search service usually works separately and isn't tightly coupled with other modules, so it can be turned into a microservice to improve performance.
    - Plan the development: Ask management for two developers for two weeks to create the microservice, while other new features continue without much impact on the business.

Note: Many big companies, like Flipkart, still call themselves startups because they constantly add new features, similar to how Google creates and sometimes discontinues products (see Killed by Google).

Hybrid Stage: Any large company usually operates in a hybrid stage, meaning they have a large monolith with many modules at the core and some modules extracted as microservices. These microservices have their own servers and might have separate databases or share the monolith's database.

Services Oriented Architecture (SOA) vs. Microservices Architecture:

Services Oriented:

- Multiple services
- Okay to share databases across services
- Communicate synchronously via API calls
- Immediate consistency using distributed transactions via 2-phase commit

Microservices:

- Multiple services
- Databases should not be shared across services; use CQRS instead, which requires duplicating data

- Communication can be both synchronous and asynchronous using event-driven architecture like the pub/sub model
- Eventual consistency using distributed transactions via the saga pattern

Sync Communication:

- Sync communication happens via HTTP requests, web sockets, or RPC (gRPC). For HTTP communication, the service must know about the service it is sending the request to.
- One issue with sync communication is called cascading failure.
- Services are tightly coupled because if you change one service, it affects another service it depends on. For example, if you have two services, A and B, and A communicates with B, any code change in B might require changes in A as well. If A's code isn't updated, it may get errors or miss new features when calling B.

Observability: It tells you if something breaks and how you will detect it.

- Detect when something is significantly wrong (like services breaking) and alert the developers.
- Diagnose the issue so developers can fix it.
- Monitor and ensure that the fix is correct.

To achieve this, we will do the following:

1. **Aggregate the logs to a central location using ELK (Elasticsearch, Logstash, Kibana)**:
    - Each server generates logs locally.
    - A cron job pushes logs from the server to the central repository. Logstash provides a central location to store logs.
2. **Search on these logs**:
    - Perform full-text search using Elasticsearch.
    - Track requests using an ID (trace ID, correlation ID, transaction ID, request ID) to correlate log lines with events. This trace ID is generated at the load balancer level.
    - For every user request (external request), generate a unique trace ID (e.g., UUID V6).
    - Ensure every service passes this trace ID in every API call/message and includes it in every log line.
3. **Provide analytics on process logs and server health logs (like CPU, HDD, RAM)**:
    - Metrics handled by Kibana include:
        - Average CPU utilization
        - Average RAM usage
        - P95 latency
        - Number of exceptions

Alerts:

- Analyze metrics and generate alerts, such as:
    - If average CPU utilization > 95% for 30 minutes, send an alert to the scaling team of DevOps.
    - If pending messages in the task queue > 10k, alert the message queue team.

P95 Latency: P95 latency, also known as the 95th percentile latency, is a measure of performance that tells us that 95% of all requests or operations are completed within a certain amount of time. In simple words, it means that only 5% of the requests take longer than this time. It helps to understand how fast most of the requests are being handled, excluding the slowest 5%.

HLD: Microservices 3 - Jul 26

Agenda:

- **Important Resource**
- **Consistency in Microservices -> Distributed Transactions**
- **The two-phase commit**
- **Saga Pattern**

  - Orchestration

  - Choreography

Important Resources

Good HLD/LLD resources

1. https://www.youtube.com/@ByteByteGo
2. https://www.youtube.com/@AsliEngineering
3. https://www.youtube.com/@hnasr
4. https://www.youtube.com/@kleppmann

gRPC vs Rest vs WebSockets

1. **gRPC vs WebSockets:** https://ably.com/topic/grpc-vs-websocket
2. **gRPC vs Rest:** https://aws.amazon.com/compare/the-difference-between-grpc-and-rest/
3. **gRPC vs Rest:** https://blog.postman.com/grpc-vs-rest/
4. **learning gRPC:**
   - https://grpc.io/docs/what-is-grpc/introduction/
   - https://grpc.io/docs/what-is-grpc/core-concepts/
   - https://grpc.io/docs/languages/python/basics/

CQRS

1. https://microservices.io/patterns/data/cqrs.html
2. https://martinfowler.com/bliki/CQRS.html
3. https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs

Saga Pattern

1. **idea:** https://microservices.io/patterns/data/saga.html

2. **aws:** https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html
3. **azure:** https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga

Two Phase Commit (2PC)

1. **theory:** https://www.youtube.com/watch?v=-_rdWB9hN1c
2. **explanation 1:** https://www.youtube.com/watch?v=eltn4x788UM
3. **explanation 2:** https://www.youtube.com/watch?v=7FgU1D4EnpQ
4. **implementation:** https://www.youtube.com/watch?v=oMhESvU87jM

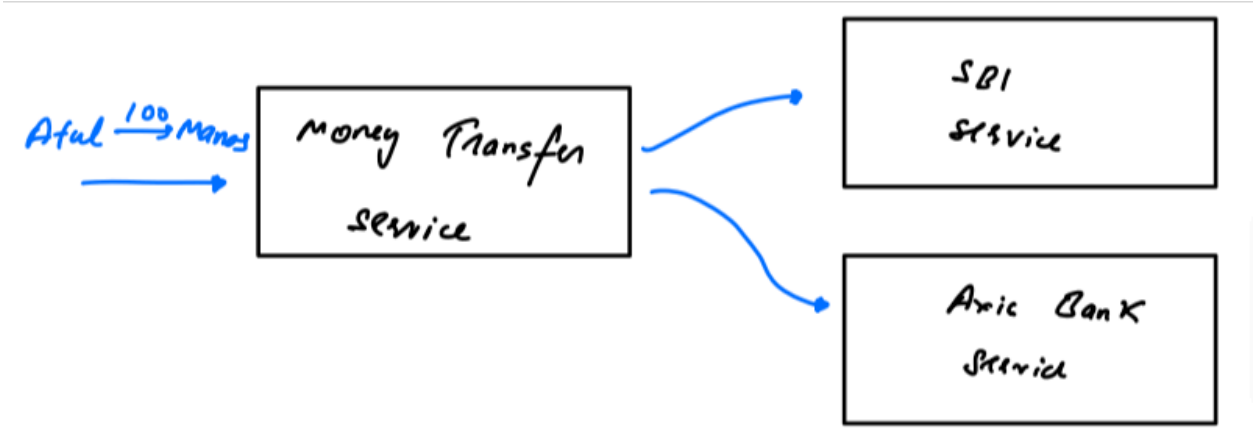
Consistency in Microservices -> Distributed Transactions

Microservices mean that actions for a request can span multiple microservices (Distributed).

For example, Sanjay wants to transfer 100 rupees to Vinod. In a monolithic system (a single server), this is easy because we can lock Sanjay's account so no other transaction can add or subtract money from it. We then subtract the money from Sanjay's account and add it to Vinod's account.

But in microservices, things are different. Let's say Sanjay has an account in SBI bank and Vinod has an account in AXIS bank.

Now, we have three services: "Money Transfer," "SBI Bank Service," and "AXIS Bank Service." These services are on different servers, so acquiring a lock is not easy. We cannot directly acquire a lock across these services.

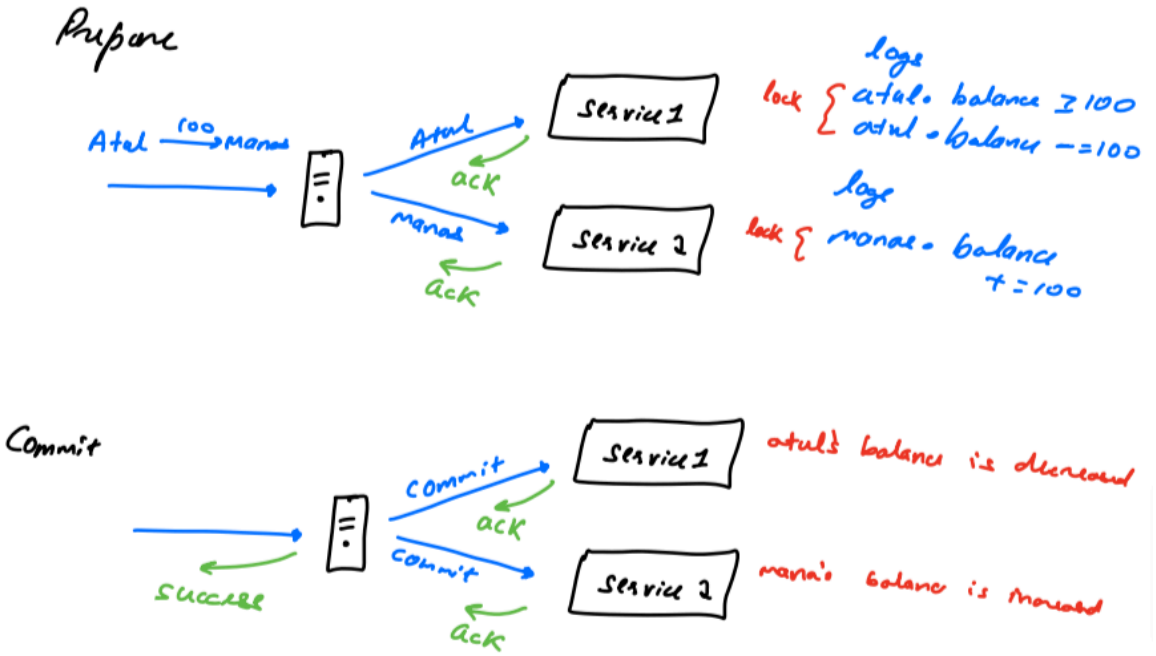


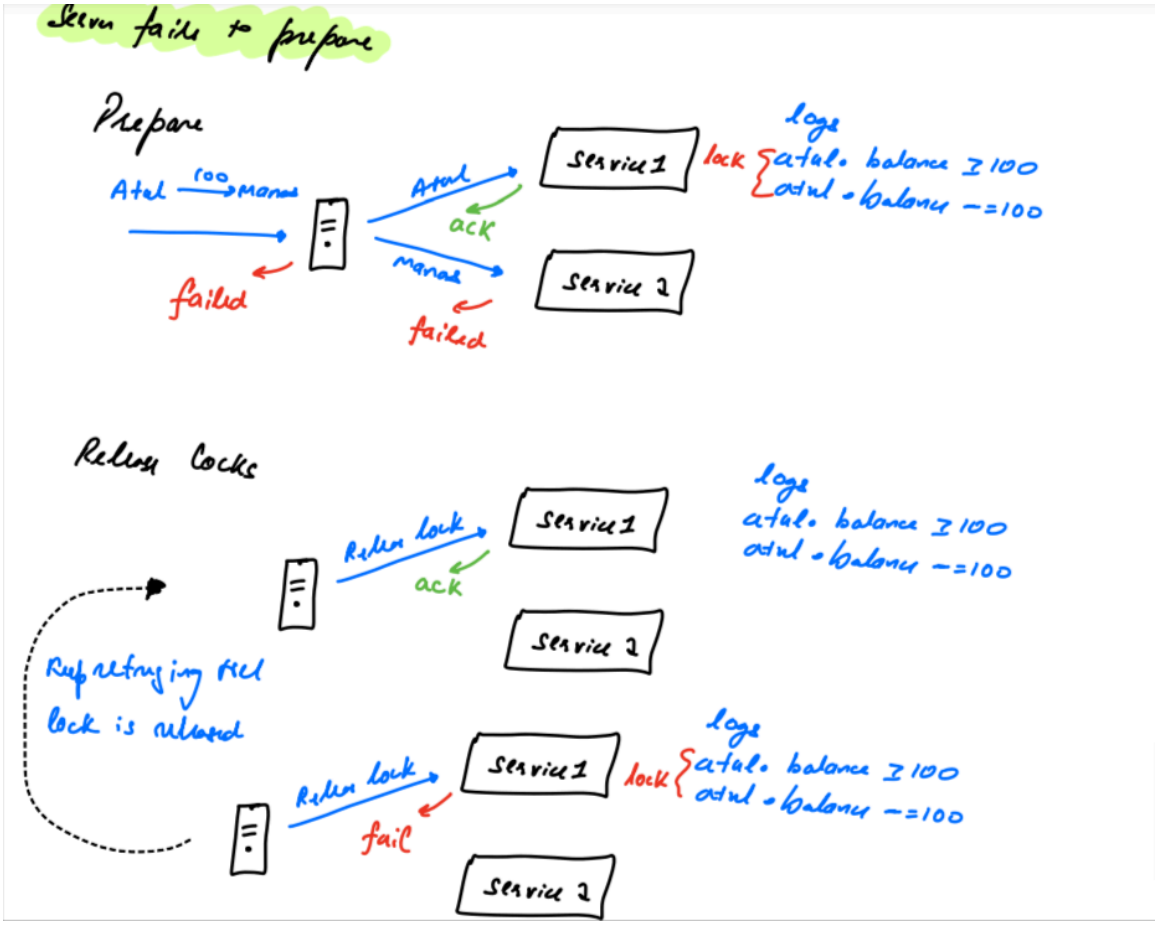
To solve this issue, we have two ways to handle these distributed transactions:

1. **Two-Phase Commit**: This ensures all parts of the transaction are done or none are done.
2. **Saga Pattern**: This breaks the transaction into smaller steps, each with a way to undo it if something goes wrong.

The two-phase commit -> is a way to make sure that a transaction (a set of actions) either completes fully or doesn't happen at all. It has two steps:

1. **Prepare Phase**: The coordinator asks all involved systems if they can commit the transaction. Each system responds with "yes" (if it can commit) or "no" (if it can't commit).
2. **Commit Phase**: If all systems respond "yes," the coordinator tells them to commit the transaction. If any system responds "no," the coordinator tells them to cancel the transaction.





Two-Phase Commit is Considered an Anti-Pattern for Microservices

1. **Acquire Locks**: This causes poor read and write performance.
2. **Very Complex to Implement**: It's hard to set up correctly.
3. **Requires Multiple Retries and Rollbacks**: This means higher latency (slower performance).
4. **Long Running Transactions**: It's not suitable for transactions involving multiple services.
5. **Requires Synchronous Communication**: All parts must communicate at the same time, which can be difficult.

Saga Pattern

The Saga Pattern is like a chapter in a book or part of a sequence.

- **Create a List of Steps**: Identify the steps that need to be executed.
- **Group Them by Microservice**:
    - **Step 1**: Requires service A
    - **Step 2**: Requires service A
    - **Step 3**: Requires service A
    - **Step 4**: Requires service B
    - **Step 5**: Requires service B

    - **Step 6**: Requires service A
    - **Step 7**: Requires service A
    - **Step 8**: Requires service C
    - **Step 9**: Requires service C

For example:

    - Steps 1-3 are Saga 1
    - Steps 4-5 are Saga 2
    - Steps 6-7 are Saga 3
    - Steps 8-9 are Saga 4

Here we have 9 steps but only 4 sagas.

- **Execute the Saga in a Consistent and Atomic Manner**: There are two main ways to execute sagas, each with its own pattern:
    - **Orchestration**
    - **Choreography**

Orchestration

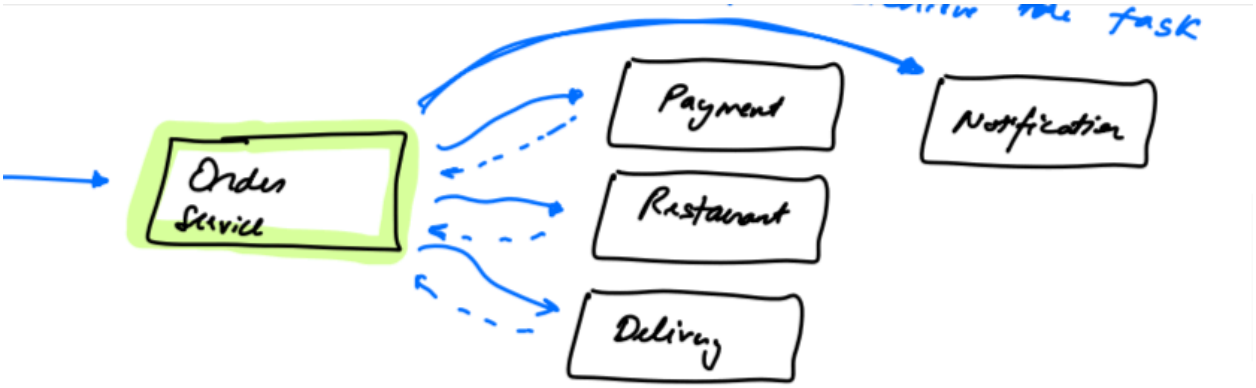
Orchestration is like a music orchestra where there is a conductor who guides all musicians on how to play, and all musicians follow the conductor's commands. In orchestration:

- One of the services acts as the brains of the operation.
- The orchestrator will communicate synchronously or asynchronously with the other services to achieve the task.

Example: Food Delivery Service

1. **Order Service**: Sends a request to the Payment Service.
2. **Payment Service**: Confirms the payment and sends a response back to the Order Service.
3. **Order Service**: Sends a request to the Restaurant Service.
4. **Restaurant Service**: Prepares the food and sends a response back to the Order Service.
5. **Order Service**: Sends a request to the Delivery Service.
6. **Delivery Service**: Delivers the food and sends a response back to the Order Service.
7. **Order Service**: Communicates with the Notification Service to notify the customer.

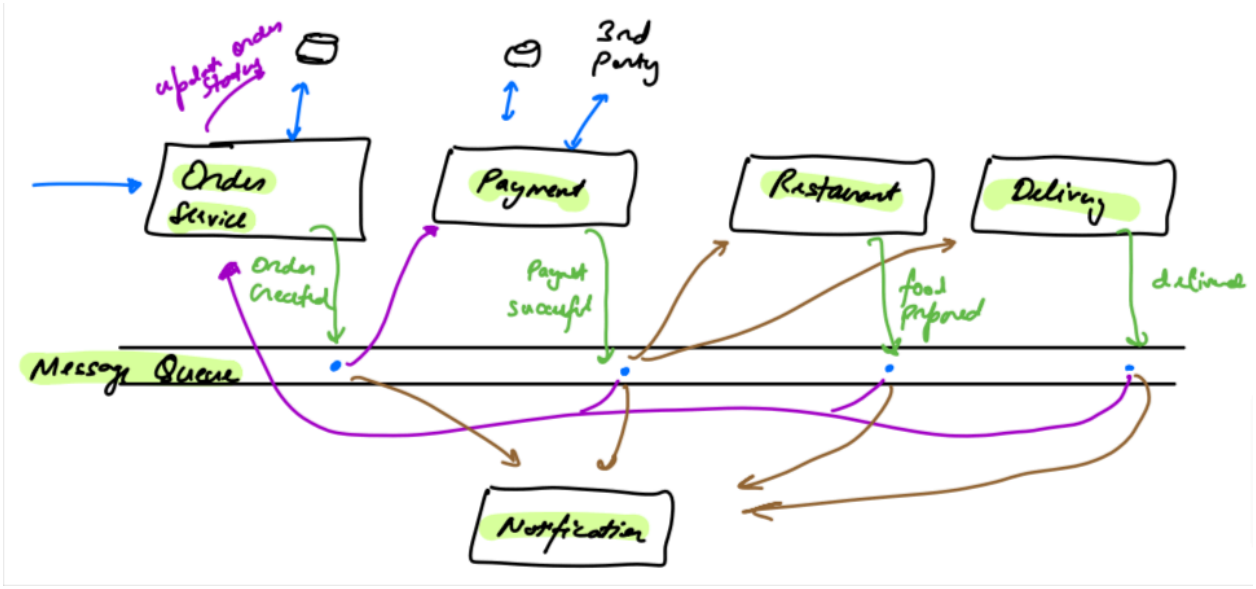
In this way, the orchestrator (Order Service) guides the entire process, ensuring each step is completed in sequence.



Choreography

Choreography is like a dance where each dancer performs their steps independently, as described by the choreographer. Everyone does their job as explained. In this context:

- **Communication Among Teams**: Developers of various teams must communicate to understand the state machines.
- **Independent Steps**: Each service is responsible only for executing its own steps.
- **Asynchronous Communication**: All communication happens asynchronously using some message queue.



Handling Failure in Choreography

- **Emit Compensating Events and Messages**: When a failure occurs, the service emits events or messages to compensate for the failure.
- **Rollback by Appropriate Services**: These failure events must be consumed by the appropriate services to perform a rollback.

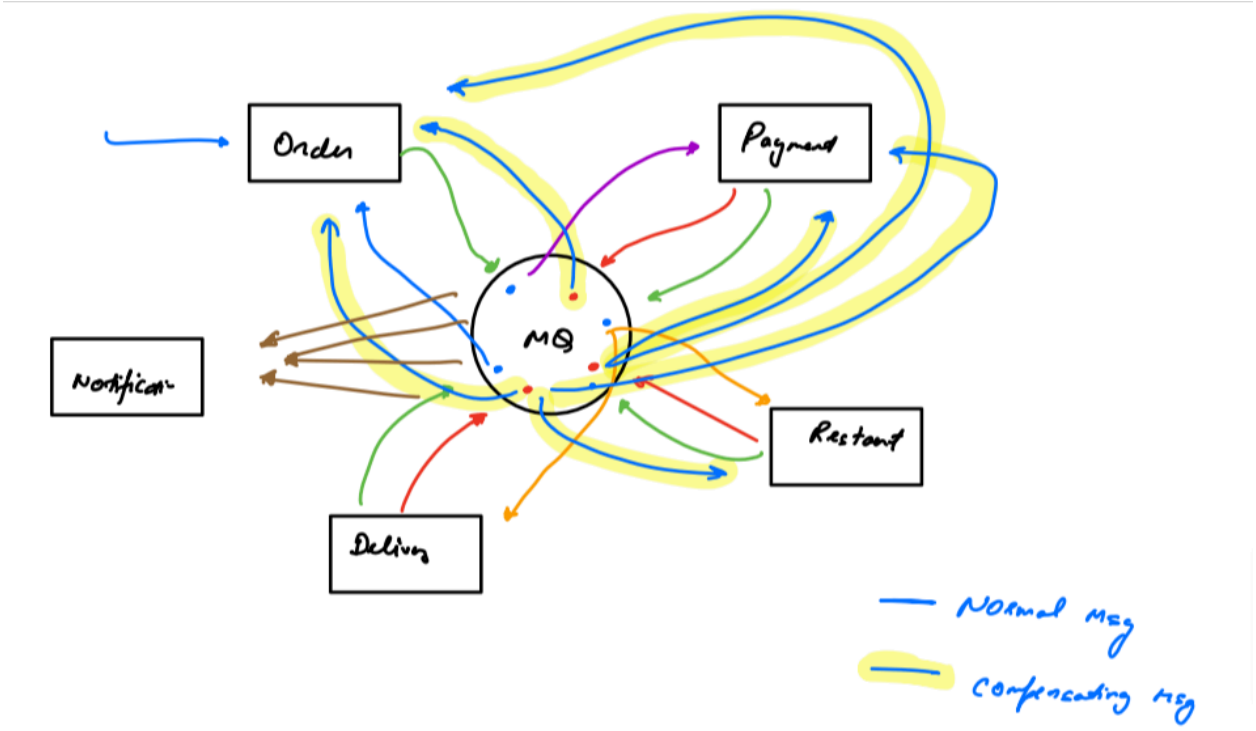
Using the same food delivery example with Order, Payment, Restaurant, Delivery, and Notification services, and one message queue:

1. **Order Creation**: When an order is created, it drops a message in the Message Queue (MQ).
2. **Payment Service**: The payment service reads the message. It then sends either a success or failure message.

- **If Payment Fails**: The failure message is consumed by the order service, which updates the order status to failed.
- **If Payment Succeeds**: The success message is consumed by the restaurant and delivery services.

3. **Restaurant and Delivery Services**: Both services drop messages in the MQ after processing.
   - **If Delivery Succeeds but Restaurant Fails**: The restaurant drops a failure message in the MQ. This message is read by the payment and order services.
     - The order service updates the order status to failed.
     - The payment service handles the refund.
   - **If Restaurant Succeeds but Delivery Fails**: The delivery service drops a failure message in the MQ. This message is read by the order service and the restaurant service.
     - The order service updates the order status to failed.
     - The restaurant service handles the refund.
4. **Notification Service**: All services send messages to the notification service to inform the customer about the order status.

This way, each service performs its role independently, and failure handling is managed through compensating events and messages.



Orchestration vs Choreography

Orchestration

- **Centralized Control**: Requires a central controller to manage the process.

- **Single Point of Failure**: To avoid bottlenecks and single points of failure, multiple orchestrator servers are needed. If an orchestrator server fails, remaining tasks may be incomplete.
- **State Management**: Requires additional tools (like Redis) to maintain transaction state if a failure occurs.
- **Task Queue**: Another orchestrator must pick up pending transactions and complete them, requiring a task queue.
- **Service Simplicity**: Services are simple to code, but the orchestrator itself is very complex.
- **Code Dependency**: Changes to any service may require changes in the orchestrator, making all teams dependent on the orchestrator team.
- **Use Case**: Ideal for faster transactions where communication is synchronous, like in banking transactions.

Choreography

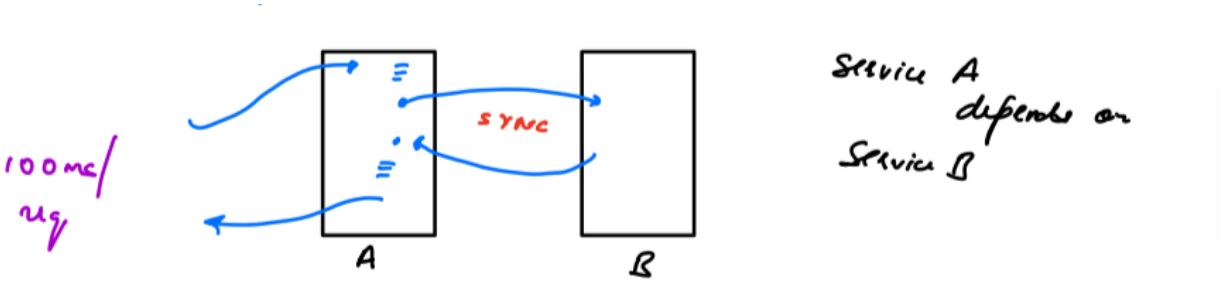
- **Decentralized Architecture**: Does not require a central controller, but teams need to coordinate.
- **Independent Execution**: Any server within a service can consume messages, perform tasks, and emit success or failure messages.
- **Complex Data Flow**: Information and data flow are complex, but each service is simple to code.
- **Independent Teams**: Each team can code independently without affecting others.
- **Asynchronous Communication**: Communication is always asynchronous. This is suitable for slower transactions, such as food delivery apps, where fast transactions are not required.

Cascading failure ->

Cascading Failure

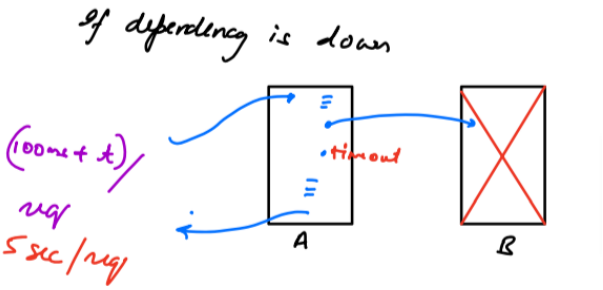
In synchronous communication, if Service A depends on Service B, here is what happens:

1. **Request Flow**: When a request comes to Service A, it forwards it to Service B, waits for confirmation, then returns a response or acknowledgment to the customer.

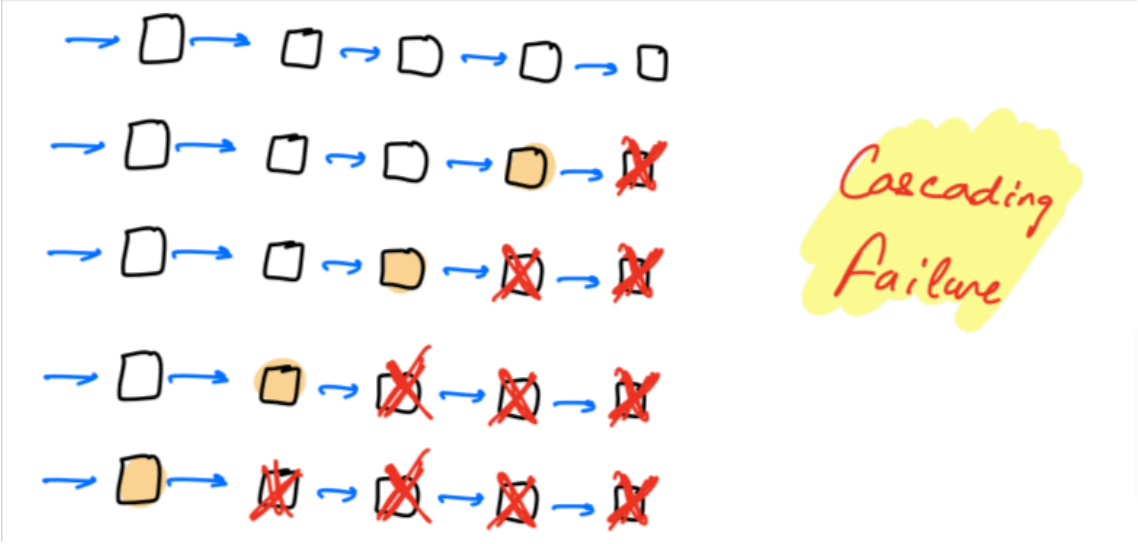


2. **Concurrent Requests**: Multiple requests come to Service A at the same time. Service A handles these using a thread pool (one thread per request) or an async task queue with an event loop.

If Service B Fails:



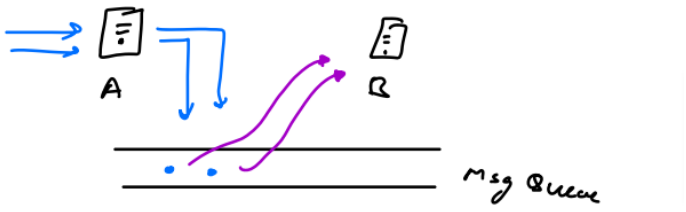
- Every request to Service A that depends on Service B times out.
- Service A starts to get slow and eventually cannot handle new requests.
- Even though Service A is not actually down, it appears to be down because it cannot fulfill requests.
- This situation is called cascading failure. If one service fails, it can cause other dependent services to fail, eventually leading to the failure of the entire system.



Thundering Herd

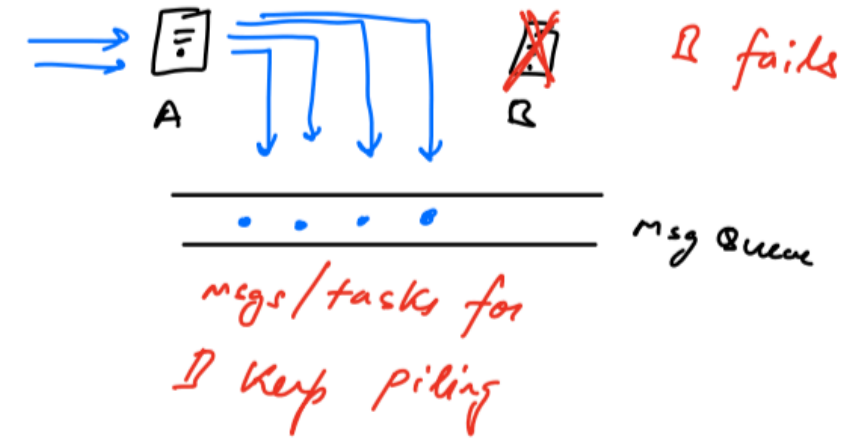
The Thundering Herd problem typically occurs with asynchronous communication, but it can also happen with synchronous communication. Here's how it works:

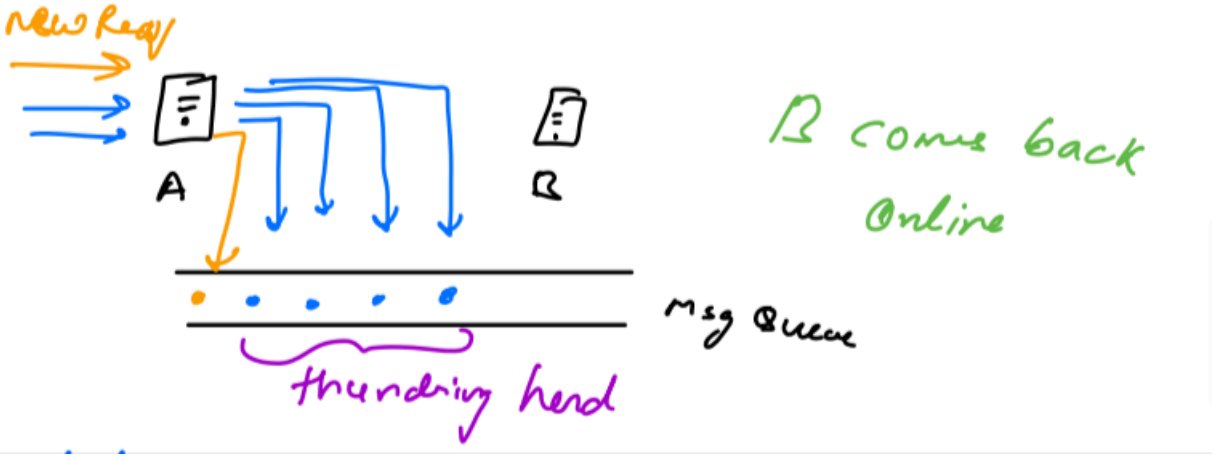
1. **Setup**: We have Service A, Service B, and a message queue.
2. **High Volume**: Service A receives lots of requests and drops many messages into the queue. These messages are consumed and processed by Service B.



If Service B Fails:

- Service A keeps adding messages to the queue.
- Messages for Service B keep piling up in the queue.
- When Service B comes back online, it first has to process all the old tasks in the queue before it can handle new requests.
- This situation is called the Thundering Herd. From the user's perspective, Service B seems slow, unable to handle tasks, or times out because it is busy processing the backlog of tasks.





Circuit Breaker

- **Detection**: It detects when a dependency (like a service) is down.
- **Prevention**: If the dependency is down, it stops sending requests to it continuously.
- **Re-checking**: It keeps checking if the dependency comes back online.

Failure Categories

1. **Regular Failures**: These are not due to infrastructure problems, meaning the server is fine but the request is faulty. Examples include:
    - User not found
    - Authentication error

    - Bad resource
    - Bad request (4xx errors)

These do not usually require tracking.

2. **Non-Transient Failures**: These indicate that the service is faulty, and tracking them is important. Examples include:
    - Timeout
    - Connection reset
    - Internal server error (5xx errors)

You need additional infrastructure to track these, such as a cache system or key-value database to store failure rates and statuses for each microservice.

Status Tracking

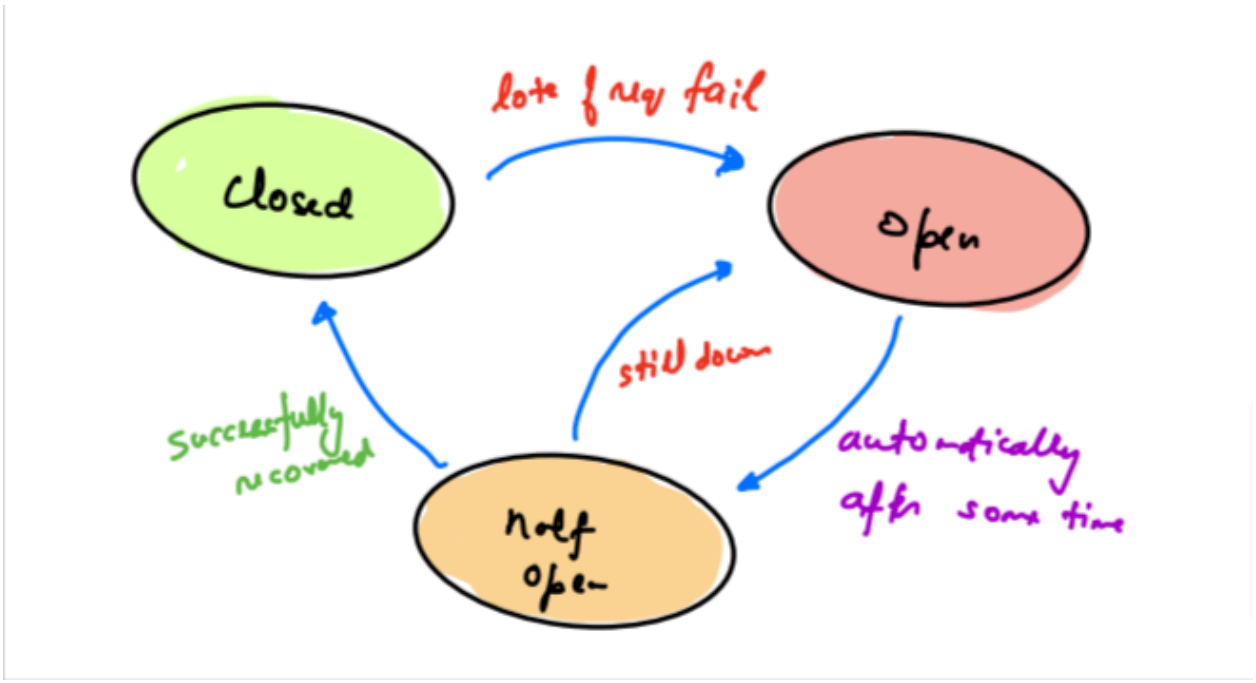
Statuses help understand the health of the service:

- **Closed**: The service is healthy.
- **Open**: The service is faulty.
- **Half Open**: The service is recovering.

Whenever Service A sends a request to Service B, it checks the status:

- **Healthy (Closed)**: Service A sends the request.
- **Faulty (Open)**: Service A denies the request and informs the client that it is not down; it is managing responses in time.
- **Recovering (Half Open)**: Service A sends a few requests to check if Service B is responding properly. If successful, it changes the status to Closed. If not, it changes the status back to Open.

To avoid checking the status on every request, you can cache the status with a short TTL (e.g., 5 seconds).


Closed
Open
Half Open
Successfully recovered
still down
automatically