System Design 101

What's system design?

Low level design covers the structure of code in a given component. However, a large scale system will have multiple components / services. High level design is about the optimal design of which components to have for a fast and efficient system. More on this later.

Why do we need distributed systems?

Let's take a real story of a website that started on a single laptop in a dorm room (Exactly how we write code today). Back in 2003, there was a website that went by the name of Del.icio.us (https://en.wikipedia.org/wiki/Delicious_(website)).
Browsing the internet was completely based on bookmarks and you would lose bookmarks the moment you changed browser / machine. So, delicious built a bookmarking website. You login, and then bookmark websites using delicious tool. That way, when you go to any other machine/browser, all you need to do is to login into delicious with your account to get access to all your bookmarks. Basically, largely delicious implemented following 2 functions:

- addBookmark(userId, site_url)
- getAllBookmarks(userId)

If you were to code those 2 functions on your laptop, would you be able to? Assume you store entries in MySQL database which is also on your laptop.
If yes, congratulations. Your version of delicious is almost ready.

Problem 1: How do I ensure that when people type del.icio.us in their browsers, they reach my laptop?

The internet world only understands IP Addresses. How do people know the IP address of my laptop when they type del.icio.us?

How do you setup your personal website today?

- You go to GoDaddy to buy a domain.

Ok, but how does GoDaddy know which domain name is available? People can buy domains from GoDaddy / NameCheap / domains.google and tons of other websites.
There must be a central place maintaining domain names and their owners. And yes, there is. It's called ICANN (The Internet Corporation for Assigned Names and Numbers). It's non profit and has a directory of all registered domain names along with their owner details and the date validity.

Alright. But that still does not solve my problem. If I go to GoDaddy and buy delicious domain name, is my issue solved? A random user's browser still does not know how to reach my laptop.

So, that means I should be able to associate my domain name to my laptop's IP address. That is exactly what happens. You can create "A" record in GoDaddy / Namecheap that is then registered centrally.

- Further reading:
    - 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/

Ok, so now ICANN knows IP address of my laptop tied to delicious domain name that I bought. Which means theoretically, when someone types delicious in their browser, they can get the IP address they need to go to from ICANN. But is that a good design?
Not really. ICANN becomes the single point of failure for the entire internet.

Ok, then what do we do? Imagine if there were thousands of machines all around the internet that had a copy of the information there on ICANN. Then my problem could have been solved. Because now people typing delicious on their browser, could find out the IP address from these machines.

Very simplistically, these machines are called DNS machines (Domain Name Servers). While the DNS architecture is decently complicated (You can read https://support.dnsimple.com/articles/differences-between-a-cname-alias-url/ if interested), in simple words, DNS machines maintain a copy of information present centrally and they keep pinging every few hours to get any recent updates from the central machines.

DNS Overview

DNS, or Domain Name System, is a critical component of the internet that translates human-readable domain names (like www.example.com) into IP addresses (like 192.0.2.1), which are used to locate and identify devices on a network. Here's a detailed step-by-step explanation of how DNS works:

1. User Input: You start by typing a URL (Uniform Resource Locator) into your web browser, such as "www.example.com".

2. Local DNS Cache Lookup: Before making a request to a DNS server, your device checks its local DNS cache to see if it has previously looked up the IP address for the given domain. If the IP address is found in the cache and is still valid (not expired), the device can use it directly without further queries. This cache helps to speed up the process and reduce the load on DNS servers.

3. Local OS-Level DNS Cache Lookup: If your device has not found the information in its local cache, it may also check the operating system's DNS cache. This cache is maintained by the operating system and acts as an additional layer of caching to speed up DNS resolution. If the IP address is found and still valid in the OS-level cache, your device can use it without further queries.

4. Recursive DNS Server: If the IP address is not found in the local cache or has expired, your device contacts a recursive DNS server, also known as a resolver. This resolver is usually provided by your Internet Service Provider (ISP) or a public DNS service like Google DNS (8.8.8.8) or Cloudflare (1.1.1.1).

5. Root DNS Servers: If the recursive DNS server doesn't have the IP address information, it starts the resolution process by querying the root DNS servers. These root servers maintain information about the top-level domains like ".com", ".org", ".net", etc. There are only a few hundred of these servers worldwide, operated by different organizations.

6. TLD (Top-Level Domain) DNS Servers: The root DNS server responds with a referral to the appropriate Top-Level Domain (TLD) DNS server for the requested domain. In our example, it would be the TLD server for ".com".

7. Authoritative DNS Servers: The TLD server then refers the recursive DNS server to the authoritative DNS server for the specific domain, in this case, "example.com". The authoritative DNS server is responsible for storing and providing the accurate IP address information for the domain.

8. DNS Query to Authoritative Server: The recursive DNS server sends a DNS query to the authoritative DNS server for "example.com", asking for the IP address associated with "www.example.com".

9. Response from Authoritative Server: The authoritative DNS server responds with the requested IP address for "www.example.com" to the recursive DNS server.

10. Response to User: The recursive DNS server now has the IP address and sends it back to your device.

11. Browser Initiates Connection: Armed with the IP address, your browser initiates a connection to the web server at that IP address.

12. Web Server Response: The web server receives the connection request and sends back the requested web page's data to your browser.

13. Rendering the Web Page: Your browser processes the data received from the web server and renders the web page, displaying it for you to see and interact with.

It's important to note that DNS is a distributed and hierarchical system, involving multiple types of DNS servers working together to resolve domain names into IP addresses. This process happens incredibly quickly, allowing you to access websites with human-readable domain names even though computers primarily communicate using IP addresses.

Anycast Routing

Anycast routing is commonly used in DNS resolution to improve performance, increase redundancy, and distribute the load across multiple DNS servers. Anycast is a network addressing and routing method where multiple servers share the same IP address, and the routing infrastructure directs the client's request to the nearest or best-performing server based on network conditions.

In DNS, anycast is often implemented at various levels of the DNS hierarchy:

1. Root DNS Servers: Some of the root DNS servers are deployed using anycast. This means that there are multiple instances of these root servers located at different geographical locations, all sharing the same IP address. When a DNS query is sent to a root DNS server, the routing infrastructure directs the query to the nearest instance of the root server, reducing latency and distributing the load.

2. TLD (Top-Level Domain) DNS Servers: Similarly, many TLD DNS servers are deployed using anycast. Each top-level domain has its own set of authoritative DNS servers, and these servers are often spread across multiple locations. Anycast ensures that when a recursive DNS server queries a TLD DNS server, the query is directed to the nearest instance of that TLD server.

3. Authoritative DNS Servers: Some authoritative DNS servers for specific domains also use anycast. This ensures that when a recursive DNS server queries an authoritative DNS server for a particular domain, the query is directed to the closest or best-performing instance of that authoritative server.

Anycast routing helps improve the speed and reliability of DNS resolution by automatically directing DNS queries to the most optimal server based on factors such as network latency, congestion, and server health. This approach enhances the overall performance of the DNS system and contributes to a more robust and resilient internet infrastructure.

Beyond this, Anycast routing is also used for DNS resolver servers. DNS resolver servers are the ones that users and client devices (like your computer or smartphone) connect to in order to resolve domain names into IP addresses. Anycast routing is implemented for DNS resolvers to improve their performance, increase availability, and distribute the load across multiple server instances.

Here's how anycast routing is used for DNS resolver servers:

1. Anycast Addresses for DNS Resolvers: Organizations that provide DNS resolver services, such as Internet Service Providers (ISPs) or public DNS providers like Google DNS (8.8.8.8) or Cloudflare (1.1.1.1), often deploy multiple instances of their resolver servers at different geographic locations. These resolver servers share the same anycast IP address.

2. Routing to Nearest Resolver: When a user's device sends a DNS query to the anycast IP address of a DNS resolver, the routing infrastructure directs the query to the nearest instance of the resolver server based on network conditions. This helps reduce the latency of DNS queries and improves the overall responsiveness of the DNS resolution process.

3. Load Distribution and Redundancy: Anycast routing also provides load distribution and redundancy. If one instance of a DNS resolver server becomes unavailable due to network issues or high traffic, the anycast routing system automatically directs queries to other available instances of the resolver server. This ensures that DNS resolution remains reliable and responsive even in the face of server failures or network disruptions.

4. Dynamic Routing: Anycast routing is dynamic, meaning that the routing infrastructure continually monitors network conditions and adjusts the routing paths to ensure that DNS queries are sent to the optimal resolver instance. This dynamic nature of anycast ensures that users are always directed to the fastest and most responsive resolver server.

By implementing anycast routing for DNS resolver servers, organizations can enhance the performance and reliability of their DNS resolution services, leading to faster web page loading times, improved user experiences, and better overall internet accessibility.

More information about how DNS works: https://howdns.works/

How do we scale?

Ok, so now we are live. Delicious is now serving users.
There is a small problem though. Everytime I want to add new features and re-deploy and restart my laptop with new code, delicious is unavailable for a few seconds. That's not good. So, what do I do?

Vertical and Horizontal Scaling

Vertical scaling and horizontal scaling are two approaches to increasing the capacity and performance of a system, typically in the context of IT infrastructure or application deployment. They involve adding resources to handle increased workloads, but they differ in how those resources are added.

Vertical Scaling (Scale Up):
Vertical scaling involves adding more resources to a single server or instance, typically by upgrading its hardware components. This approach focuses on increasing the capacity of an individual server to handle more load.

Key characteristics of vertical scaling:

1. Adding Resources: Resources such as CPU, memory, storage, and network bandwidth are increased by upgrading components within the existing server. For example, adding more RAM, replacing the CPU with a faster one, or expanding storage capacity.

2. Single Point of Failure: Since all the workload is handled by a single server, there is a higher risk of a single point of failure. If the server experiences a hardware failure, it can lead to downtime.

3. Simplicity: Vertical scaling is often simpler to implement because it involves making changes to a single server. There is no need to manage and distribute the load across multiple instances.

4. Limited Scalability: There is a limit to how much a single server can be scaled vertically. Eventually, hardware constraints may restrict further upgrades.

Horizontal Scaling (Scale Out):

Horizontal scaling involves adding more instances or servers to distribute the workload, rather than upgrading individual components. This approach focuses on increasing capacity by distributing the load across multiple machines.

Key characteristics of horizontal scaling:

1. Adding Instances: More servers or instances are added to the system to share the workload. Each instance is capable of handling a portion of the overall traffic.

2. Improved Redundancy and Availability: Horizontal scaling provides better redundancy and availability since the failure of one instance does not necessarily lead to downtime. Traffic can be redirected to healthy instances.

3. Load Balancing: Load balancers are used to distribute incoming requests across multiple instances, ensuring even distribution of the workload.

4. Better Scalability: Horizontal scaling can provide better scalability as new instances can be added as needed, allowing for more flexible capacity expansion (elasticity).

5. Complexity: Managing multiple instances, load balancers, and distributing the workload can introduce increased complexity compared to vertical scaling.

Examples:

Vertical Scaling: Upgrading a single server's RAM, CPU, or storage capacity to handle increased traffic to a database or web application.

Horizontal Scaling: Adding more web server instances to handle a sudden surge in website visitors, with a load balancer distributing traffic among the instances.

In summary, vertical scaling involves increasing the capacity of a single server by upgrading its hardware, while horizontal scaling involves adding more instances or servers to distribute the workload across multiple machines. The choice between vertical and horizontal scaling depends on factors such as the nature of the application, the scalability requirements, budget considerations, and the desired level of redundancy and availability. Communication between multiple machines over the network is another very important consideration that we will study more about in the following sections.

Stateful and Stateless Load Balancing

Stateful and stateless load balancing are two different approaches to distributing incoming network traffic across multiple servers or resources in order to improve performance, availability, and reliability. These approaches are commonly used in various network and application scenarios.

Stateful Load Balancing:
In stateful load balancing, the load balancer maintains awareness of the ongoing connections and sessions between clients and servers. It keeps track of the state of each client-server interaction, including the session information, request/response data, and other context. This allows the load balancer to make routing decisions based on factors like server load, response times, and existing session information.

Key characteristics of stateful load balancing:

1. Session Affinity (Sticky Sessions): Stateful load balancers often use session affinity to ensure that a client's requests are consistently directed to the same server throughout a session. This can be important for applications that require consistent state, such as online shopping carts.

2. Context-Aware Routing: Stateful load balancers consider factors beyond just server availability, such as server health, CPU usage, and network latency, to make routing decisions.

3. Complexity: Stateful load balancing can be more complex to implement and manage, as the load balancer needs to maintain and synchronize session data across multiple servers.

Examples of stateful load balancing:

1. An e-commerce website that ensures a user's shopping cart remains with the same server for the duration of their session.
2. A chat application where the next replies are based on the entire context of the conversation

Stateless Load Balancing:
In stateless load balancing, the load balancer does not keep track of the ongoing connections or sessions. Each incoming request is treated as an independent unit, and the load balancer makes routing decisions based on factors like server availability, response times, and overall load. The load balancer does not store any session-specific information and can freely distribute requests to any available server.

Key characteristics of stateless load balancing:

1. Scalability: Stateless load balancing is generally more scalable since there is no need to manage session state. New servers can be added to the pool without worrying about maintaining session data.

2. Simplicity: Stateless load balancing is simpler to implement and manage since the load balancer doesn't need to maintain session information or synchronize state between servers.

3. Uniform Distribution: Stateless load balancers evenly distribute incoming traffic across available servers without any preference for session continuity.

Examples of stateless load balancing:

1. A web application that serves static content, like images or CSS files, where each request is independent and can be directed to any available server.

2. A calculator application where context is not required

In summary, stateful load balancing maintains awareness of ongoing connections and sessions, while stateless load balancing treats each request as independent. The choice between these approaches depends on the specific requirements of the application and the desired trade-offs between complexity, scalability, and session continuity.

Now for del.icio.us, maybe instead of one laptop, I have multiple laptops with the same code and same information [Horizontal Scaling]. However, when my code is being deployed to a laptop X, how do I ensure no traffic is coming to X?
We need a Load Balancer which keeps track of laptops, which ones are running and is responsible to split the load equally.

How does Load balancer do that?

- Which machines are alive? - Heartbeat / Health Check
- Splitting load? - Round robin / Weighted Round Robin / Ip Hash

https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/ has example of a config setup of a load balancer.

Imagine, Del.icio.us becomes majorly popular. It starts getting massive traffic. It starts getting a million new bookmarks every day.

Now, remember this is 2004. Best machines had 40GB hard disk. If you were getting 1 Million new bookmarks every day, and every bookmark is 200 bytes roughly, then you are adding 200MB of new bookmarks every day. Which means you will run out of space in 6 months. What do you do?

You will have to consider splitting the information you have between machines. This is called sharding.
Step 1: Choose a sharding key. Basically what information should not get split between machines, and should reside in the same machine.
Show what happens if you choose site_url as the sharding key. getAllBookmarks has to go to all machines.
We choose user_id to be the sharding key, which means a user and all their bookmarks go to one shard.

Step 2: Build out an algo for userId -> shard mapping.
Following constraints:

- Finding shard given userID should be extremely lightweight. Can't add a lot of load to LB.
- Load should be somewhat equally distributed (no load skew)
- Addition of new shards should be easy and should not cause major downtime.
- Same for removal of shards.

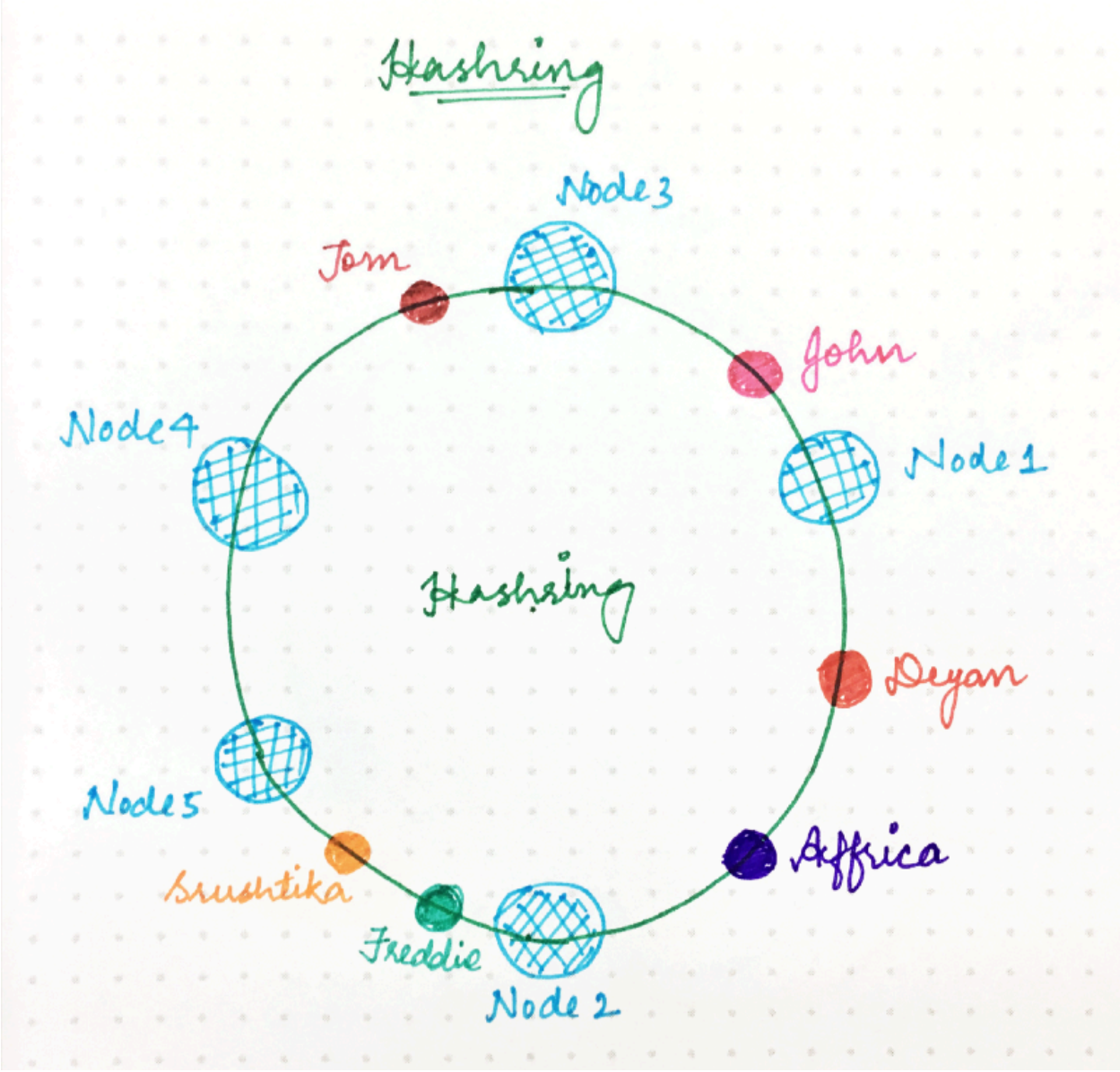
Let's check a certain approach for sharding.

- Approach 1: Assign userId to userId % number_of_shards. While this approach is great, it fails when the number of shards change, as it causes almost every user's data to be copied to another machine. Massive downtime when shard is added.
- Approach 2: Range based assignment. Load skew - first adopters are more likely to be busier users. Also, every range's total storage usage will only increase as they add more bookmarks. Addition of new shard does not help existing shards.

Let's look at the real approach used in most cases - Consistent Hashing.

Consistent Hashing

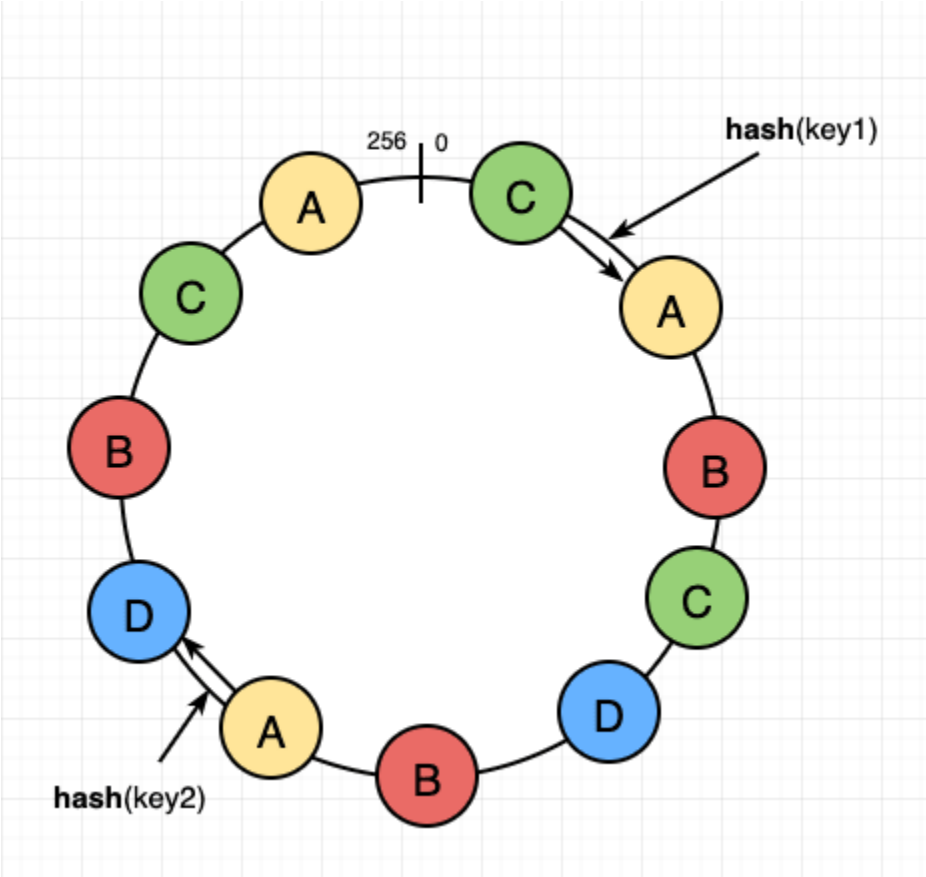
Imagine a circle with points from [0, 10^18]. Imagine there is a hash function H1, which maps every machineId to a number in [0, 10^18], which you then mark on the circle. Similarly, there is another hash function H which maps userId to [0, 10^18].
Let's assume we assign a user to be present on the first machine in the cyclic order from the hash of the user.



For example, in the diagram above, Deyan and Affrica are assigned to Node 2, Freddie and Srushtika on Node 5 and so on.
In implementation, if you have a sorted array with hashes of nodes, then for every user, you calculate the hash, and then binary search for the first number bigger than the given hash. That machine is what the user will be assigned to.

However, this design suffers from an issue. What happens when you remove a shard. Let's say Node 2 is down. All load of Node 2 (Deyan + Africa) get assigned to Node 5 and Node5's load basically doubles. At such high load, there is a good probability that Node 5 dies which will triple the load for Node 4. Node4 can also die and it will trigger cascading failure.

So, we modify the consistent hashing a little bit. Instead of one hash per machine, you use multiple hashing functions per machine (the more, the better). So, Node 1 is present at multiple places, Node 2 at multiple places and so forth.



In the above example, if node A dies, some range of users is assigned to B, some to D and some to C. That is the ideal behavior.

System Design - Caching

Agenda:

1. AppsServer Layer
2. Caching
    - Where?
    - Invalidation?
    - Eviction?
    - Local caching case study

In the last class, we learned that when we type something on the browser(or search for a website specifically), the first thing the browser has to do is talk to a DNS and figure out which IP address the browser should be talking to, and it communicates with the machine at that particular IP. And when you go from one machine to multiple machines, you need a load balancer to distribute traffic uniformly, but after a point when the amount of information cannot fit into a single machine, then the model needs to be shared. It can be done using consistent hashing.

However, the machines had both the code and storage in the previous model. **Do you think it is a good model?**

Tightly Coupled

It's not, the reasons being:

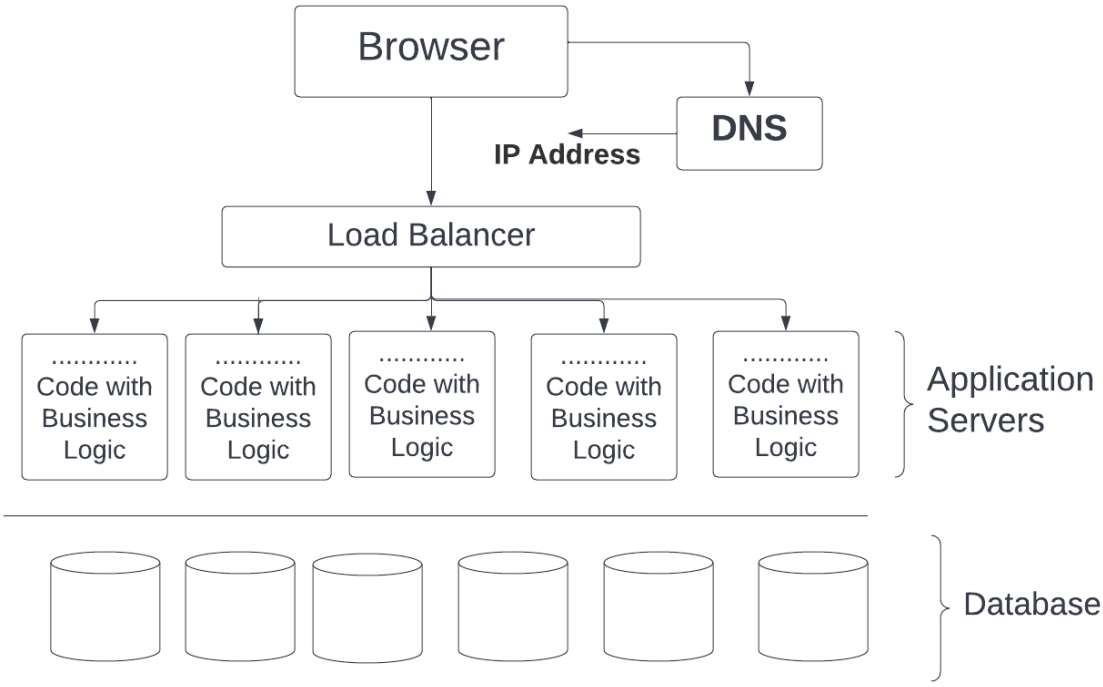
1. Code and database are tightly coupled, and code deployments cause unavailability.

2. Fewer resources are available for the code since the database will also use some of the resources.

So it is better to decouple code and storage. However, the only **downside of decoupling** is the additional latency of going from one machine to another (code to the database).

So it can be concluded that it is not ideal for storing code and database on the same machine. The approach is to separate the code and storage parts to increase efficiency.

Different machines storing the same code running simultaneously are called **Application Server** Machines or App Servers. Since they don't store data and only have code parts, they are stateless and easily scalable machines.



Caching

The process of storing things closer to you or in a faster storage system so that you can get them very fast can be termed caching.

Caching happens at different places. The first place we start caching is the browser. Browser stores/caches things that you need to access frequently. Now let's look at different levels of caching.

1: In-Browser caching

We can cache some IPs so that browser doesn't need to communicate with the DNS server every time to get the same IP address. This caching is done of smaller entries that are likely not to change very often and is called **in-browser hashing**. Browser caches DNS and static information like images, videos, and JavaScript files. This is why a website takes time to load for the first time but loads quickly because the browser caches the information.

2: CDN (Content Delivery Network)

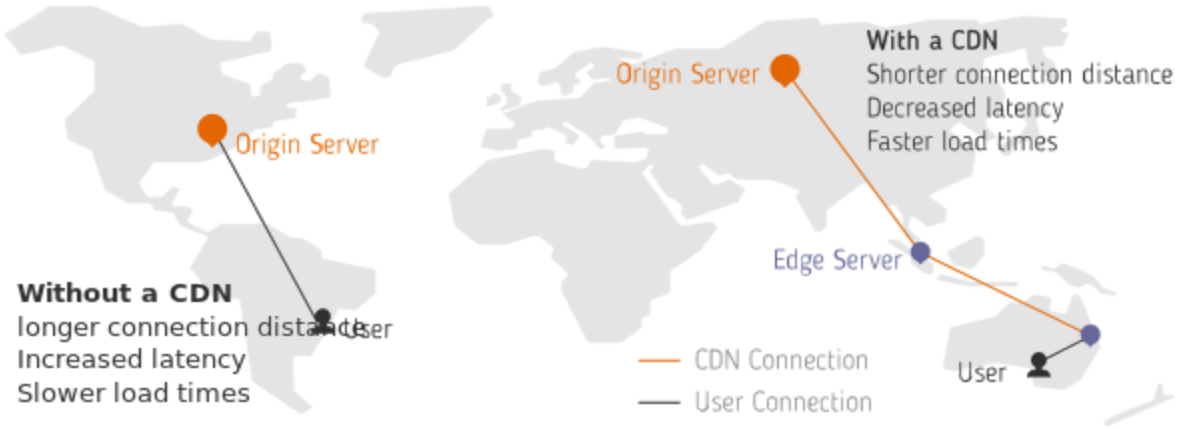
I will discuss fetching images/multimedia from the website (like **del.icio.us**). You have the browser in some region(say India), and you must fetch the files from the servers located in another region(say the US). When you try to access from your browser, a request is made to the load balancer, and then it goes to the application server and requests files from the file storage. You know that transferring files and other data will be fast for the machines in the same region. But it can take time for machines located on different continents.

From the website perspective, users worldwide should have a good experience, and these separate regions act as a hindrance. **So what's the solution?**

The solution for the problem is CDN, Continent Delivery Network. Examples of CDN are companies like

- Akamai
- Cloudflare
- CloudFront by Amazon
- Fastly

These companies' primary job is to have machines worldwide, in every region. They store your data, distribute it to all the regions, and provide different CDN links to access data in a particular region. Suppose you are requesting data from the US region. Obviously, you can receive the HTML part/ code part quickly since it is much smaller than the multimedia images. For multimedia, you will get CDN links to files of your nearest region. Accessing these files from the nearest region happens at a much larger pace. Also, you pay per use for using these CDN services.



One question you might think is **how your machine talks to the nearest region only**(gets its IP, not of some machine located in another region), when CDN has links for all the regions. Well, this happens in two ways:

1. A lot of ISP have CDN integrations. Tight coupling with them helps in giving access to the nearest IP address. For example, Netflix's CDN does that.
2. Anycast (https://www.cloudflare.com/en-gb/learning/cdn/glossary/anycast-network/)

This CDN process to get information from the nearest machine is also a form of caching.

3: Local Caching

It is caching done on the application server so that we don't have to hit the database repeatedly to access data.

4: Global Caching

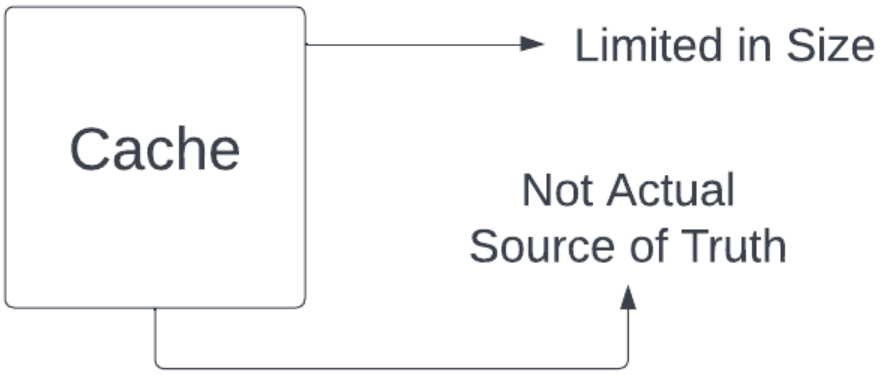
(This will be discussed in more detail in the next class)
This is also termed In-memory caching. In practice, systems like Redis and Memcache help to fetch actual or derived kinds of data quickly.

Problems related to caching

There are also two things related to the cache that you can derive from the discussions so far.

- Cache is limited in size

- It is not the actual source of truth; that is, the actual data is somewhere else. It stores a replica of data.



There are a few problems that you may face:

- Data can become stale and inconsistent with time (Data in Database - actual source of truth - changes. But is not reflected in cache)
- The cache can become full due to its small storage capacity.

So what do you think will be the solution to these two problems:

1. What do you have that doesn't become inconsistent?
2. How can you add entries if the cache is full?

We will be discussing ways to prevent these problems.

Case Invalidation Strategy

One solution that is proposed so that cache doesn't become stale is

TTL (Time to Live)

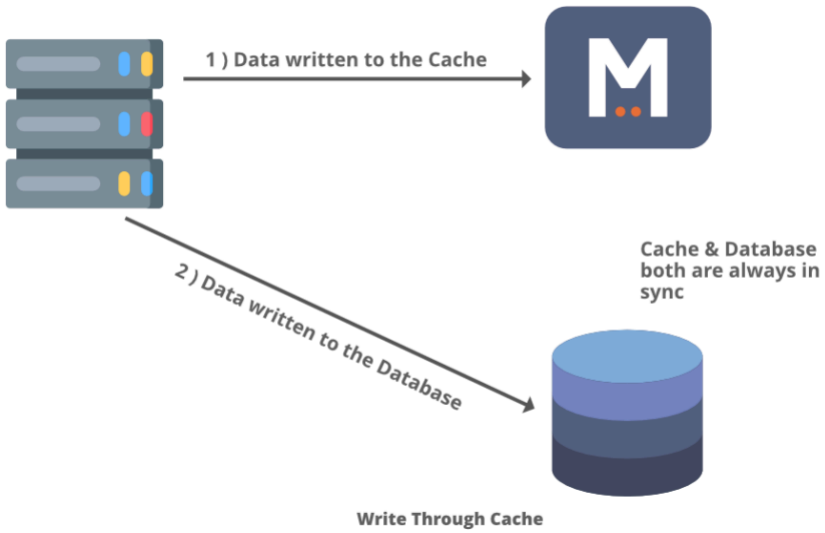
This strategy can be used if there is no problem with the cache being invalid for a very short time, so you can have a periodic refresh. Entries in the cache will be valid for only a period. And after that, to again get the entries, you need to fetch them again.
So, for example, if you cache an entry X at timestamp T with TTL of 60 seconds, then for all requests asking for entry X within 60 seconds of T, you read directly from cache. When you go asking for entry X at timestamp T+61, the entry X is gone and you need to fetch again.

We will look at more cache invalidation strategies in this doc through case studies.

Keeping cache and DB in sync

This can be done by the strategies like Write through cache, Write back cache, or Write around the cache.

Write through cache: Anything to be written is database passes from cache first(there can be multiple cache machines), storing it (updating cache), and then updating it to the database and returning success. If failed, changes will be reverted in the cache.
It makes the writing slower but reads much faster. For a read-heavy system, this could be a great approach.



There are other methodologies as well, like

Write back cache: First, the write is written in the cache. The moment write in the cache succeeds, you return success to the client. Data is then synced to the database asynchronously (without blocking current ongoing request).. The method is preferred where you don't care about the data loss immediately, like in an analytic system where exact data in the DB doesn't matter, and analytical trends analysis won't be affected if we lose data or two. It is inconsistent, but it will give very high throughput and very low latency.

Write around cache: Here, the writes are done directly in the database, and the cache might be out of sync with the database. Hence we can use TTL or any similar mechanism to fetch the data from the database to cache to sync with it.

Now let's talk about the second question: *How can you add entries if the cache is full?*

Well, for this, you be using an eviction strategy.

Cache eviction

There are various eviction strategies to remove data from the cache to make space for new writes. Some of them are:

- FIFO (First In, First Out)
- LRU (Least Recently Used)
- LIFO (Last In, First Out)
- MRU (Most Recently Used)

The eviction strategy must be chosen based on the data that is more likely to be accessed. The caching strategy should be designed in such a way that you have a lot of cache hits than a cache miss.

SUMMARY

1. It is not a good idea to have application code and database on the same machine; hence we separated them.
2. Next, you learned caching and why it is necessary to get back to the user as soon as possible. Caching can happen at multiple levels. It can happen within the browser, on the CDN layer, or inside our systems, where you can cache data locally in app servers or have a global cache like central in-memory storage.
3. However, you still need to look at how you will invalidate the cache so that data should not be stale and the cache doesn't run out of space. For this TTL, write back, write through and write around can be used.
4. Lastly, you learned how to evict data so that you can store the most relevant information in the cache.
5. For the next class: Assignment to invalidate the cache when the test data changes.

System Design - Caching Continued

Agenda:

- Local Cache Case Study
- Global Cache → why?
- Global Cache Case Study

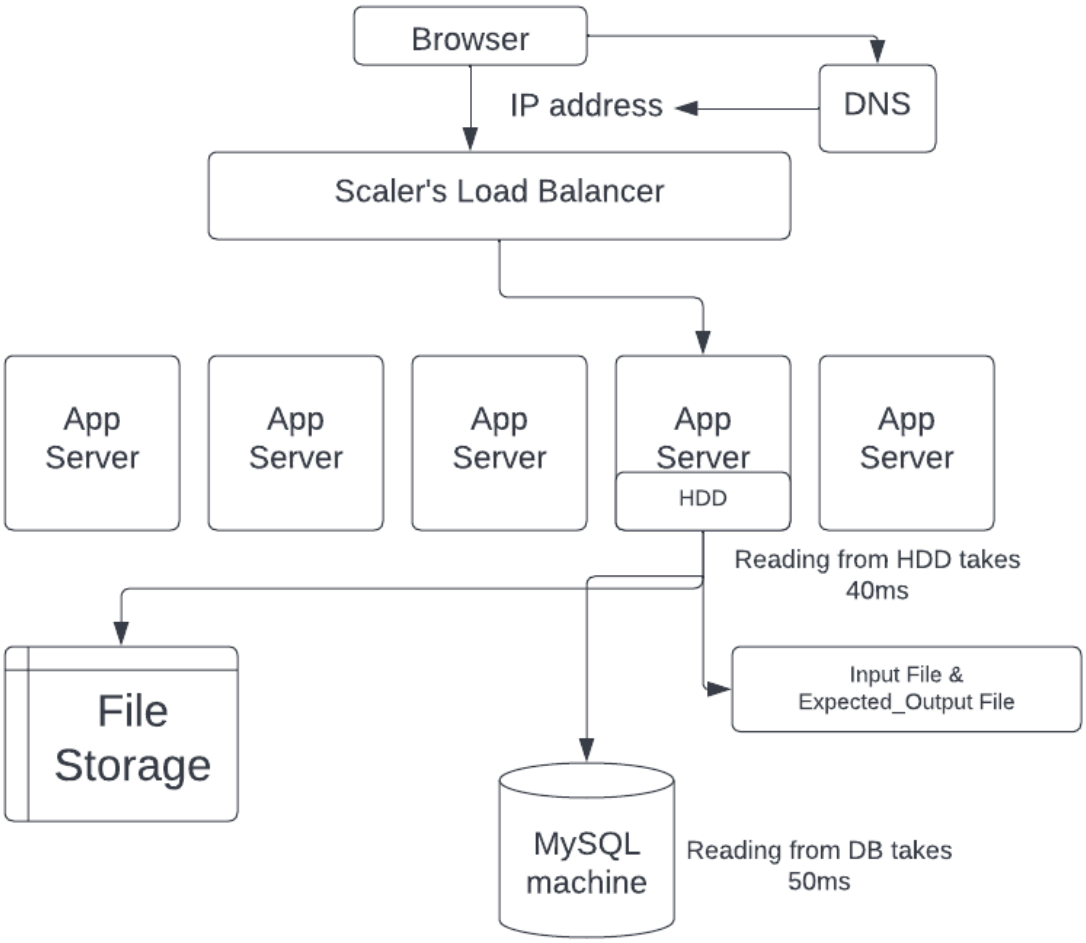
In the last class, we discussed how caching could be done at multiple layers: in-browser, using CDN for larger resources, in the application layer, or in the database layer. We initially started with the local caches and ended the class on the case study. The problem statement was:

Consider the case of submitting DSA problems on Scaler; when you submit a problem on Scaler, the browser talks to Scaler's load balancer. And the submission goes to one of the app servers out of many. The app server machine gets the user id, problem id, code, and language. To execute the code, the machine will need the input file and the expected output file for the given problem. The files can be large in size, and it takes time (assumption: around 2 seconds) to fetch files from the **file storage**. This makes code submissions slow.

So, how can you make the process fast?

Assumptions:

1. If the file is present on the machine itself, on the hard disk, then reading the file from the hard disk takes 40ms.
2. Reading from a DB machine (MySQL machine), reading a table or a column (not a file) takes around 50ms.



It can be noted that input files and the expected output file can be changed. The modified changes should be immediately reflected in the code submissions.

Solution

Different approaches to solve the problem can be

TTL

TTL Low: If TTL is very low (say for 1 min), then cache files become invalid on the app server machines after every minute. Hence most of the time, test data won't be available on the machine and is to be fetched from file storage. The number of cache misses will be high for TTL very low.

TTL High: If TTL is very high, then case invalidation happens too late. Say you keep TTL 60 min, and in between the time you change the input & expected files, the changes will not be reflected instantly.

So TTL can be one of the approaches, but it is not a good one. You can choose TTL based on the cache miss rate or cache invalidation rate.

Global Cache

Storing the data in a single machine can also be an option, but there are two problems with this:

1. If storing in memory, the remote machine has limited space and can run out of space quickly because the size of the input-output files is very large.
2. The eviction rate will be very high, and the number of cache misses will be more.

If instead you store it in the hard disk, then there is the issue of transferring huge amount of data on network.

File Metadata

The best approach would be we identify whether the file has changed or not using the metadata for the files.

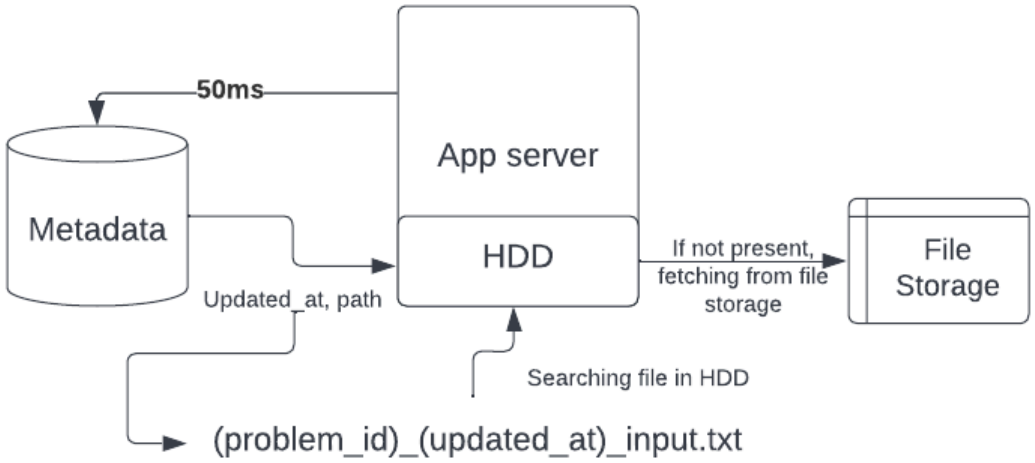
Let's assume in the MySQL database, there exists table problems_test_data. It contains details problem_id, input_filepath, input_file_updated_at, input_file_created_at for input files, and similar details for the output files as well. If a file is updated on the file storage, its metadata will also be updated in the SQL database.

problems_test_data

problem_id	input_filepath	input_file_updated_at	input_file_created_at

Updating meta data of a file in DB when uploading/modifying to file storage.

Now all the files can be cached in the app server with a better approach to constructing file names. The file can be **(problem_id)_(updated_at)_input.txt**



When a submission comes for a problem, we can go to the database (MySQL dB) and get the file path and its last updated time. If a file with problem_id_updated_at_input.txt exists in a machine cache, it is guaranteed that the existing file is the correct one. If the file doesn't exist, then the path can be used to fetch it from the file storage (along with now storing it locally on the machine for the future). Similar things can be done for the output files as well. Here the metadata about the file is used to check whether the file has been changed/updated or not, and this gives us a very clean cache invalidation.

Updating a file,
All cache servers have some files stored if an update is to be done for a file stored in S3. The process looks like this:
For a problem (say for problem_id 200) if an update request comes to modify an input file to a newly uploaded file.

- Upload new input file to file storage (S3). It will give a path (say new_path) for the file stored location.
- Next, MySQL DB has to be updated. The query for it looks like
  UPDATE problem_test data WHERE problem_id = 200 SET inputfile_path = new_path AND inputfile_updated_at = NOW()
- Now, if submission comes and the metadata in DB does not match that of the file existing in the cache, the new file needs to be fetched from the file storage at the location new_path. The returned file will be stored in the HDD of the app server. For the next requests, it will be present on the hard disk already (if not evicted).

It can be noted that every time a submission is made, we have to go to the MySQL DB to fetch all the related information of the problem/user. The information like whether it's already solved, problem score, and user score. It's a better option to fetch the file's metadata simultaneously

while we fetch other details. If solutions pass, the related details have to be updated on DB again.

A separate cache for all machines is better than one single-layer cache.
Here(https://gist.github.com/jboner/2841832) is why.

Caching Metadata - Global Caching

Ranklist Discussion: Let's take an example of the rank list in a contest with immense traffic. During the contest, people might be on the problem list page, reading a problem, or on the rank list page (looking for the ranks). If scores for the participants are frequently updated, computing the rank list becomes an expensive process (sorting and showing the rank list). Whenever a person wants the rank list, it is fetched from DB. This causes a lot of load on the database.

The solution can be computing the rank list periodically and caching it somewhere for a particular period. Copy of static rank list gets generated after a fixed time (say one minute) and cached. It reduces the load on DB significantly.

Storing the rank list in the local server will be less effective since there will be many servers, and every minute cache miss may occur for every server. A much better approach is to store the rank list in the global cache shared by all app servers. Therefore there will be only one cache miss every minute. **Here global caching performs better than local caching**. Redis can be used for the purpose.

Redis: Redis is one of the most popular caching mechanisms used everywhere. It is a single-threaded key-value store. The values which Redis supports are:

- String
- Integer
- List
- Set
- Sorted_set

Redis

Key	Value
"Rahul_score"	500

The main scenarios where global caching is used are:

1. Caching something that is queried often
2. Storing derived information, which might be expensive to compute on DB.

And we can use Redis for either of the cases mentioned above to store the most relevant information. It is used to decrease data latency and increase throughput.

To get a sense of Redis and have some hands-on you can visit: https://try.redis.io/
You can also check the following:

- Set in Radis https://redis.io/docs/data-types/sets/
- Sorted set in Radis:https://redis.io/docs/data-types/sorted-sets/

Further lectures on Storage Layer and Redis

1. Overview of Storage Layer - https://www.scaler.com/meetings/redis-overview-2/j/a/cmtAL2JbLGgPdaowvMVhs4I9n5031S1n
2. Redis : Deep Dive - https://www.scaler.com/meetings/redis-deep-dive-4/j/a/Ayy8fo-Fwb__BAxTgEMle8X3mMyh7Iyi
3. Redis : Deep Dive 2 - https://www.scaler.com/meetings/redis-followup/j/a/w-pH_An40x9NAYCzlhTKOhG@eJfuM3EH

System Design - CAP Theorem, PACELC Theorem & Master-Slave

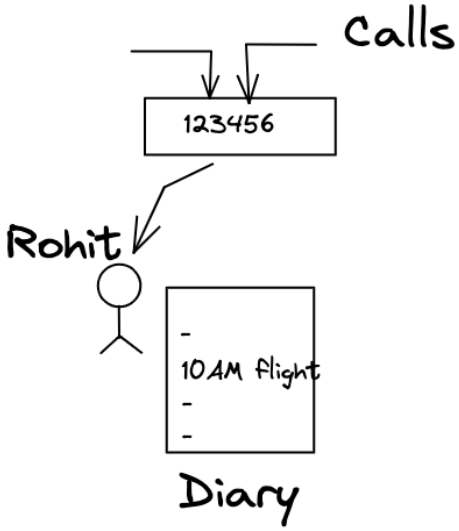
Agenda:

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

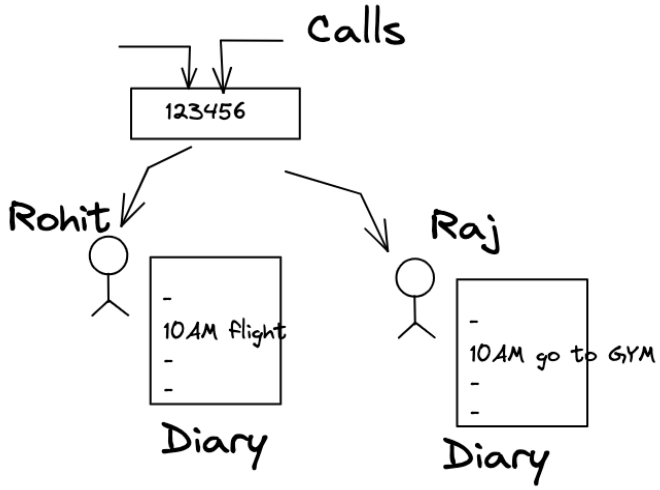
CAP Theorem:

The CAP theorem states that a distributed system can only provide two of three properties simultaneously: **C**onsistency, **A**vailability, and **P**artition tolerance.

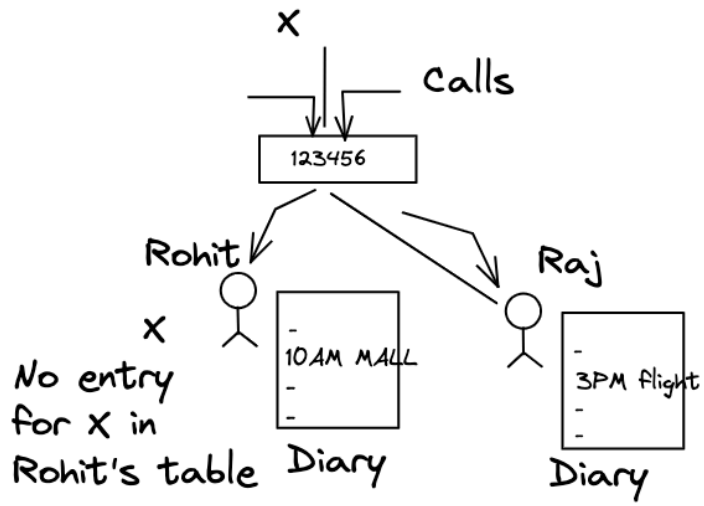
Let's take a real-life example; say a person named Rohit decides to start a company, "reminder", where people can call him and ask him to put the reminder, and whenever they call him back to get the reminder, he will tell them their reminders.
For this, Rohit has taken an easy phone number as well, 123456.
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.

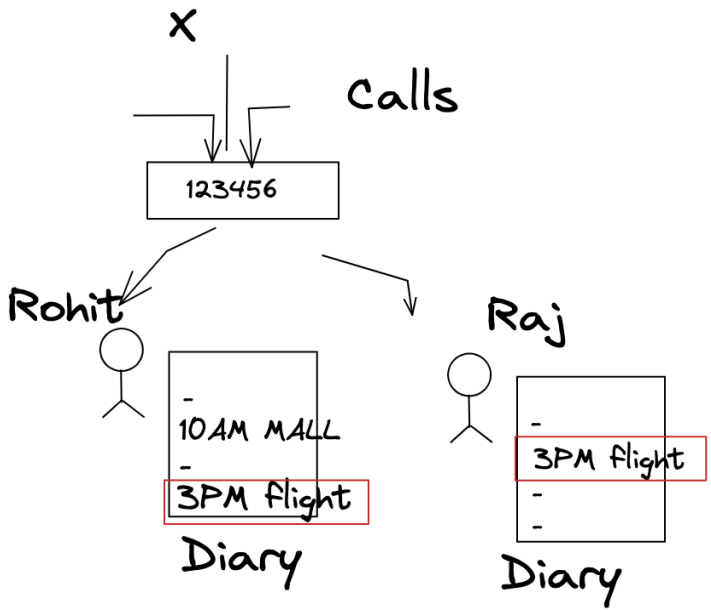


One day, Rohit gets a call from a person named “X” asking him the time of his flight, But Rohit was not able to get any entry for X. So, he says that he doesn't have a flight, but unfortunately, that person has that flight, and he missed it because of Rohit.
The problem is when person “X” called for 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.

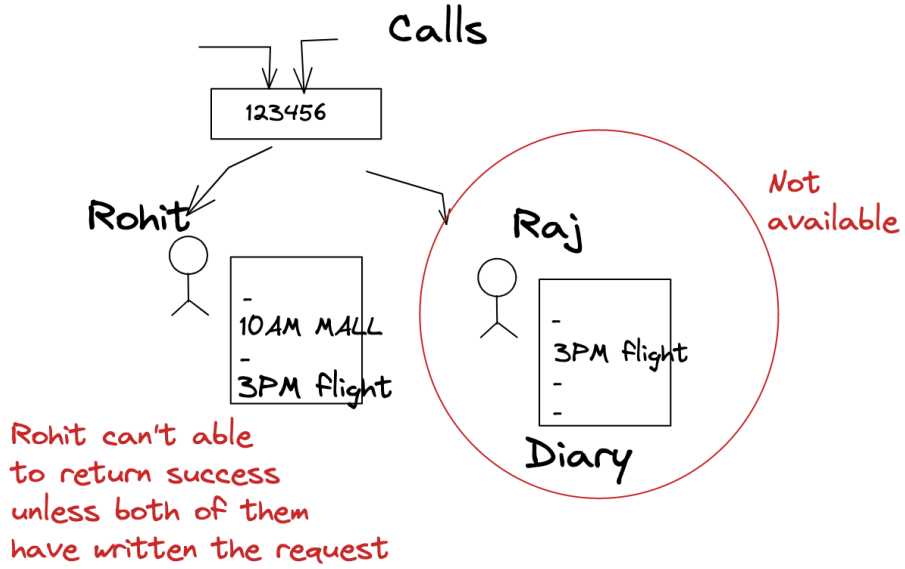


Problem 1: Inconsistency
It’s a situation where different data is present at two different machines.

Solution 1:
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.

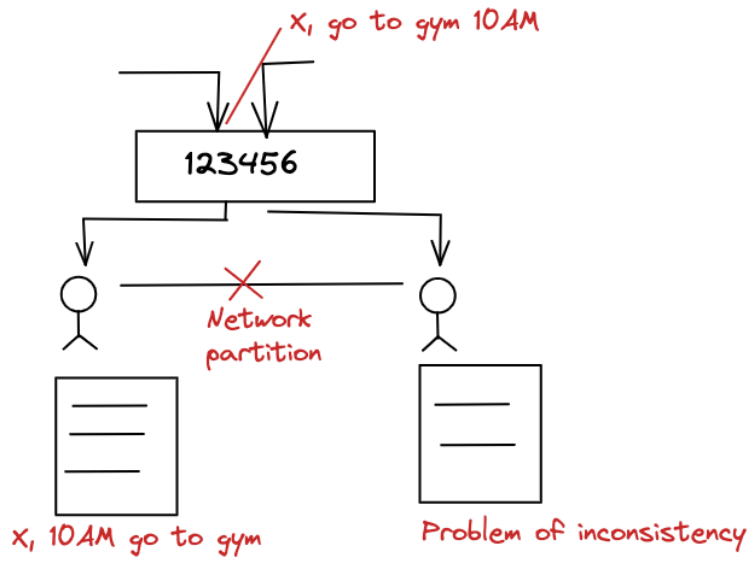


Solution 2:
When the other person is not there to take the entry, then also we will take the entries, but the next day, before resuming, the other person has to ensure they catch up on all entries before marking themselves as active (before starting to take calls).

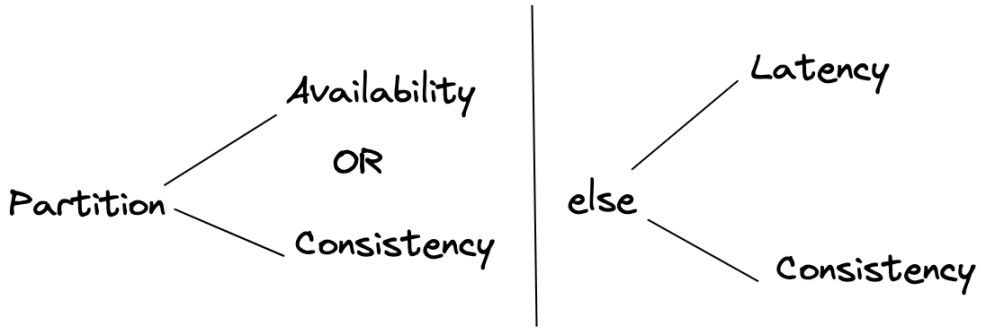
Problem 3: Network Partition.

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.

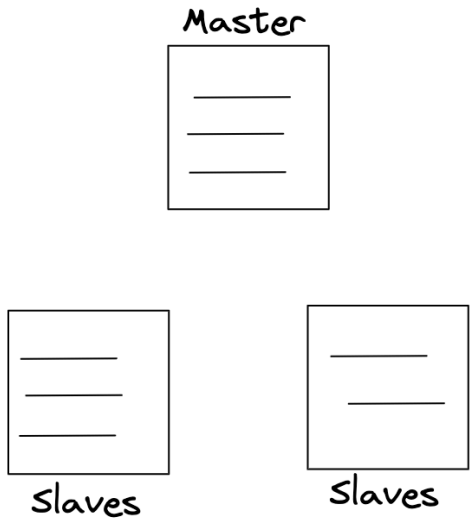
So If there is no network partition, we have to choose between extremely low Latency or High consistency. They both compete with each other.

Some Examples of When to choose between Consistency and Availability.

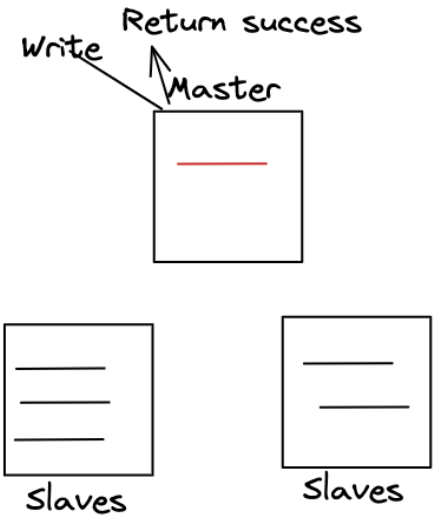
1. In a banking system, Consistency is important.
    a. so we want immediate consistency
    b. but in reality, ATM transactions (and a lot of other banking systems) use eventual consistency
2. In a Facebook news feed-like system, availability is more important than the consistency.
3. For Quora, Availability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

Master Slave System:

In Master-Slave systems, Exactly one machine is marked as Master, and the rest are called Slaves.



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



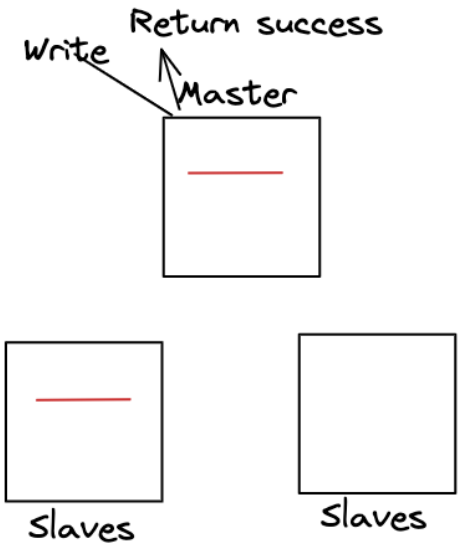
Steps:

1. Master system takes the write.
   If the write is successful, return success.

2. 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:

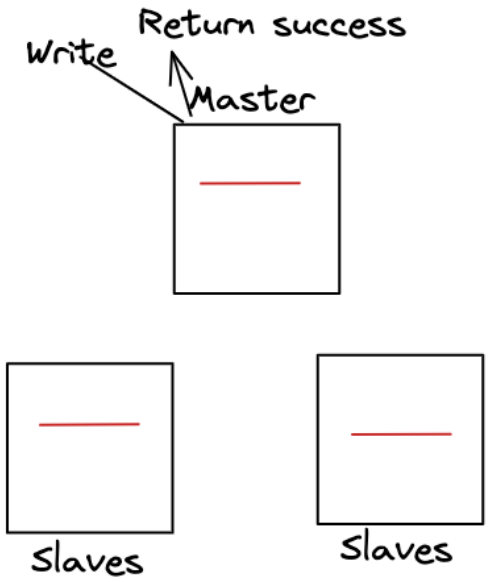


Steps:

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.

3. Master Slave that are Highly Consistent:



Steps:

1. Master and all slave take the writes, if all have written, then only return success.

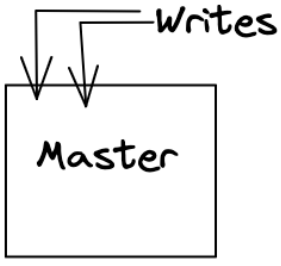
Example: The banking system.

In Master-Slave systems,

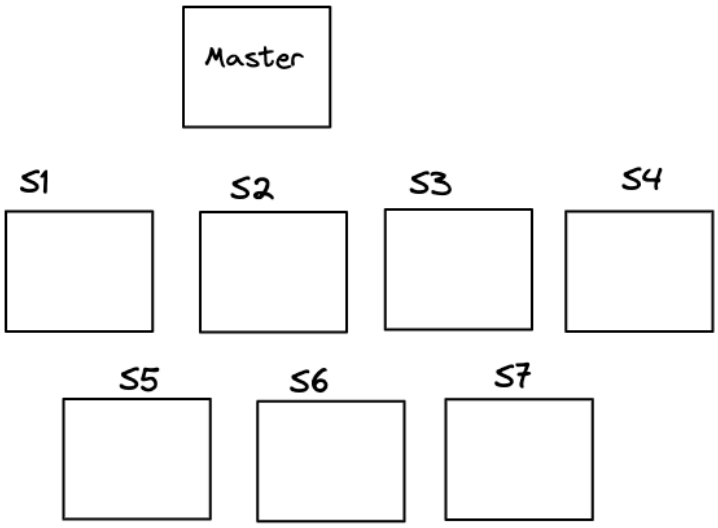
- All writes first come to Master only.
- Reads can go to any of the machines.
- Whenever the Master system dies, a new election of the master will take place based on a different elections algorithm.

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.

System Design - SQL vs NoSQL

SQL Database

SQL databases are relational databases which consist of tables related to each other and every table has a fixed set of columns. You can query across tables to retrieve related information.

Features of SQL Databases:

Normalization

One of the requirements of SQL databases is to store the data in normalized form to avoid data redundancy and achieve consistency across tables. For example, let's assume two tables are storing a particular score and one of the scores gets changed due to an update operation. Now, there will be two different scores at two different tables leading to confusion as to which score should be used.

Hence, the data should be normalized to avoid this data redundancy and trust issue.

ACID Transactions

ACID stands for Atomicity, Consistency, Isolation and Durability.

- **Atomicity** means that either a transaction must be all or none. There should not be any partial states of execution of a transaction. Either all statements of a transaction are completed successfully or all of them are rolled back.
- **Consistency** refers to the property of a database where data is consistent before and after a transaction is executed. It may not be consistent while a transaction is being executed, but it should achieve consistency eventually.
- **Isolation** means that any two transactions must be independent of each other to avoid problems such as dirty reads.
- **Durability** means that the database should be durable, i.e. the changes committed by a transaction must persist even after a system reboot or crash.

Let's understand this with the help of an example.

Let's say Rohit wants to withdraw Rs 1000 from his bank account. This operation depends on the condition that Rohit's bank balance is greater than or equal to 1000 INR. Hence the withdrawal essentially consists of two operations:

- Check if the bank balance is greater than or equal to 1000 INR.
- If yes, perform a set operation: Balance = Balance - 1000 and disperse cash.

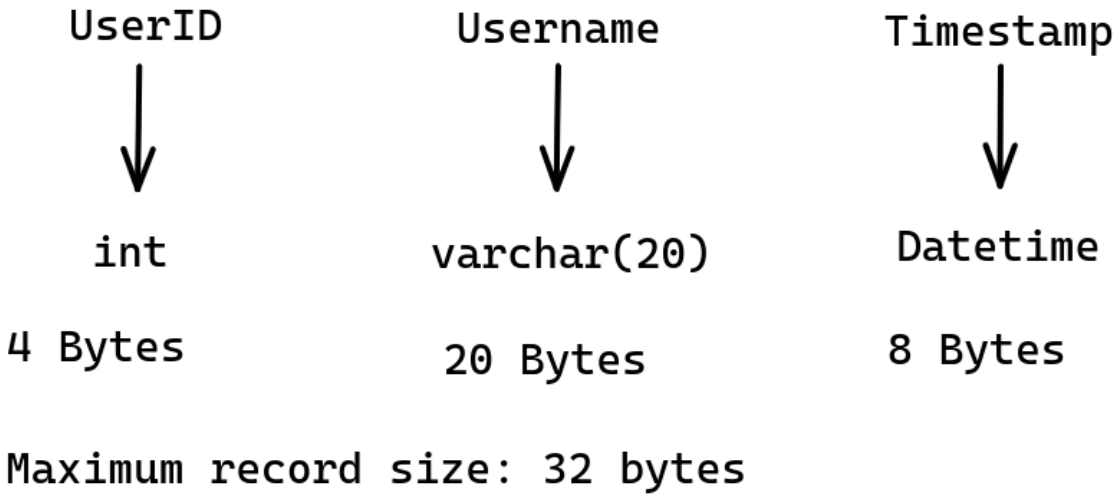
Now, imagine these operations are done separately in an app server. Let's assume that Rohit's bank balance is 1500 INR. And the first operation was completed successfully with a yes. Now, when the second operation is performed, there are chances that some other withdrawal request of 1000 INR has already changed Rohit's bank balance to 500 INR.

Now, if the second operation is performed, it would set Rohit's bank balance to -500 which does not make sense. Hence, if the database does not guarantee atomicity and isolation, these kinds of problems can happen when multiple requests attempt to access (and modify) the same node.

Now, when Rohit makes a request to withdraw 1000 INR from his account, both these operations represent a single transaction. The transaction either succeeds completely or fails. There won't be any race conditions between two transactions. This is guaranteed by a SQL database.

Defined Schema

Each table has a fixed set of columns and the size and type of each column is well-known.



Rigid Schema

However, there are a few features that are **not** supported by a SQL database.

Shortcomings of SQL Databases

Fixed Schema might not fit every use case

Let's design the schema for an ecommerce website and just focus on the Product table. There are a couple of pointers here:

- Every product has a different set of attributes. For example, a t-shirt has a collar type, size, color, neck-type, etc.. However, a MacBook Air has RAM size, HDD size, HDD type, screen size, etc.
- These products have starkly different properties and hence couldn't be stored in a single table. If you store attributes in the form of a string, filtering/searching becomes inefficient.
- However, there are almost 100,000 types of products, hence maintaining a separate table for each type of product is a nightmare to handle.
  SQL is designed to handle millions of records within a single table and not millions of tables itself.

Hence, there is a requirement of a flexible schema to include details of various products in an efficient manner.

Database Sharding nullifies SQL Advantages

- If there is a need of sharding due to large data size, performing a SQL query becomes very difficult and costly.
- Doing a **JOIN** operation across machines nullifies the advantages offered by SQL.
- SQL has almost zero power post sharding. You don't want to visit multiple machines to perform a SQL query. You rather want to get all data in a single machine.

As a result, most SQL databases such as postgresql and sqlite do not support sharding at all.

Given these two problems, you might need to think of some other ways of data storage. And that's where NoSQL databases come into the scenario.

NoSQL Databases

Let's pick the second problem where data needs to be sharded. First step to Sharding is choosing the **Sharding Key**. Second step is to perform **Denormalization**.

Let's understand this with the help of an example.

Imagine a community which exchanges messages through a messenger application. Now, the data needs to be sharded and let's choose **UserID** as the sharding key. So, there will be a single machine holding all information about the conversations of a particular user. For example, M1 stores data of U1 and M2 for U2 and so on.

Now, let's say U1 sends a message to U2. This message needs to be stored at both M1 (sent box) and M2 (received box). This is actually denormalization and it leads to data redundancy. To avoid such problems and inefficiencies, we need to choose the sharding key carefully.

Examples of Choosing a Good Sharding Key

Banking System

Situation:

- Users can have active bank accounts across cities.
- Most Frequent operations:
    - Balance Query
    - Fetch Transaction History
    - Fetch list of accounts of a user
    - Create new transactions

Why is CityID not a good sharding key?

- Since users can move across cities, all transactions of a user from the account in city C1 needs to be copied to another account in city C2 and vice versa.
- Some cities have a larger number of users than others. Load balancing poses a problem.

An efficient sharding key is the UserID:

- All information of a user at one place. Operations can be performed most efficiently.
- For Balance Query, Transaction History, List of accounts of a user, you only need to talk to one machine (Machine which stores info of that user).
- Load balancing can be achieved as you can distribute active and inactive users uniformly across machines.

Note: Hierarchical Sharding might not be a great design as if one machine crashes, all machines under its tree also become inaccessible.

Uber-like System

Situations:

- Most frequent use case is to search for nearby drivers.

CityID seems to be a good sharding key.

- You need to search only those cabs which are in your city. Most frequent use cases are handled smoothly.

DriverID is not a good choice as:

- The nearby drivers could be on any machines. So, for every search operation, there will be a need to query multiple machines which is very costly.

Also sharding by PIN CODE is not good as a cab frequently travels across regions of different pin codes.

Note: Even for inter-city rides, it makes sense to search drivers which are in your city.

Slack Sharding Key (Groups-heavy system)

Situation:

- A group in Slack may even consist of 100,000 users.

UserID is not a good choice due to the following reasons:

- A single message in a group or channel needs to perform multiple write operations in different machines.

For Slack, GroupID is the best sharding key:

- Single write corresponding to a message and events like that.
- All the channels of a user can be stored in a machine. When the user opens Slack for the first time, show the list of channels.
- Lazy Fetch is possible. Asynchronous retrieval of unread messages and channel updates. You need to fetch messages only when the user clicks on that particular channel.

Hence, according to the use case of Slack, GroupID makes more sense.

IRCTC Sharding Key

Main purpose is ticket booking which involves TrainID, date, class, UserID, etc.

Situation:

- Primary problem of IRCTC is to avoid double-booked tickets.
- Load Balancing, major problem in case of tatkal system.

Date of Booking is not a good Sharding Key:

- The machine that has all trains for tomorrow will be bombarded with requests.
- It will create problems with the tatkal system. The machine with the next date will always be heavily bombarded with requests when tatkal booking starts.

UserID is not a valid Sharding Key:

- As it is difficult to assure that the same ticket does not get assigned to multiple users.
- At peak time, it is not possible to perform a costly check every time about the status of a berth before booking. And if we don't perform this check, there will be issues with consistency.

TrainID is a good sharding key:

- Loads get split among trains. For example, tomorrow there will be a lot of trains running and hence load gets distributed among all machines.
- Within a train, it knows which user has been allocated a particular berth.
- Hence, it solves the shortcomings of Date and UserID as sharding keys.

Note: Composite sharding keys can also be a good choice.

Few points to keep in mind while choosing Sharding Keys:

- Load should be distributed uniformly across all machines as much as possible.
- Most frequent operations should be performed efficiently.
- Minimum machines should be updated when the most-frequent operation is performed. This helps in maintaining the consistency of the database.
- Minimize redundancy as much as possible.

Types of NoSQL databases

Key-Value NoSQL DBs

- Data is stored simply in the form of key-value pairs, exactly like a hashmap.
- The value does not have a type. You can assume Key-Value NoSQL databases to be like a hashmap from string to string.
- Examples include: DynamoDB, Redis, etc.

Document DBs

- Document DB structures data in the form of JSON format.
- Every record is like a JSON object and all records can have different attributes.
- You can search the records by any attribute.
- Examples include: MongoDB and AWS ElasticSearch, etc.
- Document DBs give some kind of tabular structure and it is mostly used for ecommerce applications where we have too many product categories.

Link to play with one of the popular Document DBs, **MongoDB**: MongoDB Shell. It has clear directions regarding how to:

- Insert data
- Use find() function, filter data
- Aggregate data, etc.

You can try this as well: MongoDB Playground. It shows the query results and allows you to add data in the form of dictionaries or JSON format.

Column-Family Storage

- The sharding key constitutes the RowID. Within the RowID, there are a bunch of column families similar to tables in SQL databases.
- In every column family, you can store multiple strings like a record of that column family. These records have a timestamp at the beginning and they are sorted by timestamp in descending order.
- Every column family in a CF DB is like a table which consists of only two columns: timestamp and a string.

- It allows prefix searching and fetching top or latest X entries efficiently. For example, the last 20 entries you have checked-in, latest tweets corresponding to a hashtag, posts that you have made, etc.
- It can be used at any such application where there are countable (practical) schemas instead of completely schema less.
- The Column-Family Storage system is very helpful when you want to implement Pagination. That too if you desire to implement pagination on multiple attributes, CF DBs is the NoSQL database to implement.
- Examples include Cassandra, HBase, etc.

Choose a Proper NoSQL DB

Twitter-HashTag data storage

Situation:

- With a hashtag, you store the most popular or latest tweets.
- Also, there is a need to fetch the tweets in incremental order, for example, first 10 tweets, then 20 tweets and so on.
- As you scroll through the application, fetch requests are submitted to the database.

Key-Value DB is not a good choice.

- The problem with Key-Value DB is that corresponding to a particular tweet(key), all the tweets associated with that tweet will be fetched.
- Even though the need is only 10 tweets, the entire 10000 tweets are fetched. This will lead to delay in loading tweets and eventually bad user experience.

Column-Family is a better choice

- Let's make the tweet a sharding key. Now, there can be column families such as Tweets, Popular Tweets, etc.
- When the posts related to a tweet are required, you only need to query the first X entries of the tweets column family.
- Similarly, if more tweets are required, you can provide an offset, and fetch records from a particular point.

Live scores of Sports/Matches

Situation:

- Given a recent event or match, you have to show only the ongoing score information.

Key-Value DB is the best choice

- In this situation, Key-Value DB is the best as we simply have to access and update the value corresponding to a particular match/key.
- It is very light weight as well.

Current Location of Cab in Uber-like Application

Situation:

- Uber needs to show the live location of cabs. How to store the live location of cabs?

If location history is needed: Column-Family DB is the best choice

- We can keep the cab as a sharding key and a column family: Location.
- Now, we have to simply fetch the first few records of the Location column family corresponding to a particular cab.
- Also, new records need to be inserted into the Location column family.

If location history is not needed: Key-Value DB is the best choice:

- If only the current location is needed, Key-Value makes a lot more sense.
- Simply fetch and update the value corresponding to the cab (key).

Points to remember

- Facebook manually manages sharding in its UserDB which is a MySQL database instance.
- Manual Sharding involves adding a machine, data migration when a new machine is added, consistent hashing while adding a new shard, and if a machine goes down, it has to be handled manually.
- JavaScript developers generally deal with Document DBs pretty easily. JS developers work with JSON objects quite often. Since records in Document DBs are pretty similar to JSON objects, JS Developers find it easy to start with.
- SQL databases are enough to handle small to medium sized companies. If you start a company, design the database using SQL databases. Scaler still runs on (free) MySQL DB and it works perfectly fine. NoSQL is only needed beyond a certain scale.
- ACID is not built-in in NoSQL DBs. It is possible to build ACID on top of NoSQL, however consistency should not be confused with ACID.
- ACID features can be implemented on top of NoSQL by maintaining a central shard machine and channeling all write operations through it. Write operations can be performed only after acquiring the write lock from the central shard machine. Hence, atomicity and isolation can be ensured through this system.
- **Atomicity and Isolation** are different from **Consistency**.

How a transaction is carried out between two bank accounts and how actually it is rolled back in case of failure?

Ans: Such a transaction between two bank accounts has states. For example, let A transfers 1000 INR to B. When money has been deducted from A's account, the transaction goes to **send_initiated** state (just a term). In case of successful transfer, the state of the A's transaction is changed to **send_completed**.

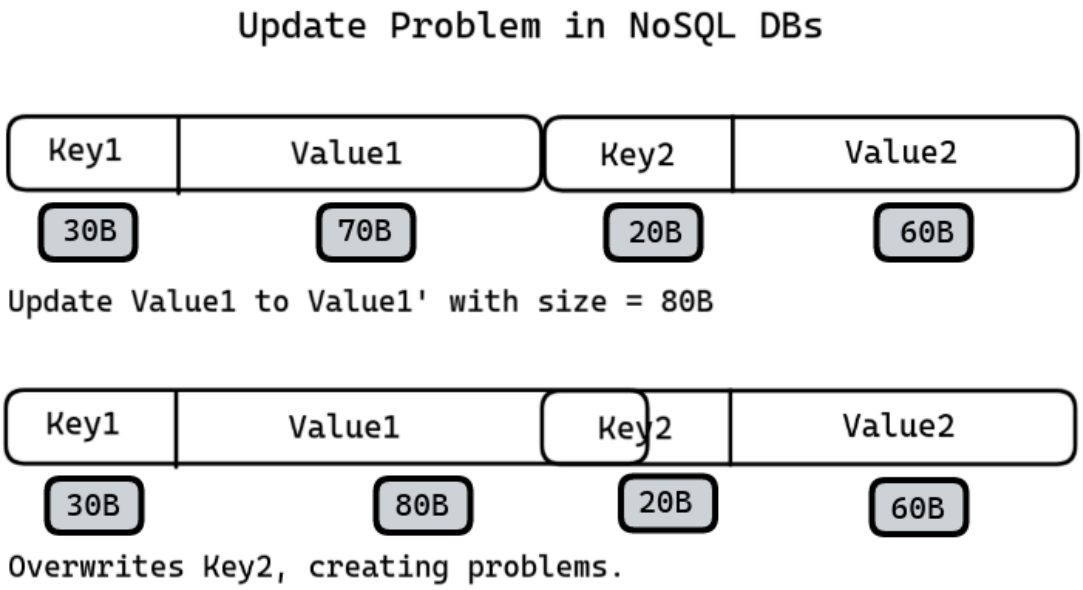
However, let's say due to some internal problem, money did not get deposited into B's account. In such a case, the transaction on A's side is rolled back and 1000 INR is again added to A's bank balance. This is how it is rolled back. You may have seen that money comes back after 1-2

days. This is because the bank re-attempts the money transfer. However, if there is a permanent failure, the transaction on A's side is rolled back.

Problem Statement 1

- Storing data in SQL DBs is easy as we know the maximum size of a record.
- Problem with NoSQL DBs is that the size of value can become exceedingly large. There is no upper limit practically. It can grow as big as you want.
    - Value in Key-Value DB can grow as big as you want.
    - Attributes in Document DB can be as many in number as you want.
    - In Column Family, any single entry can be as large as you want.
- This poses a problem in how to store such data structure in the memory like in HDD, etc.

Update Problem

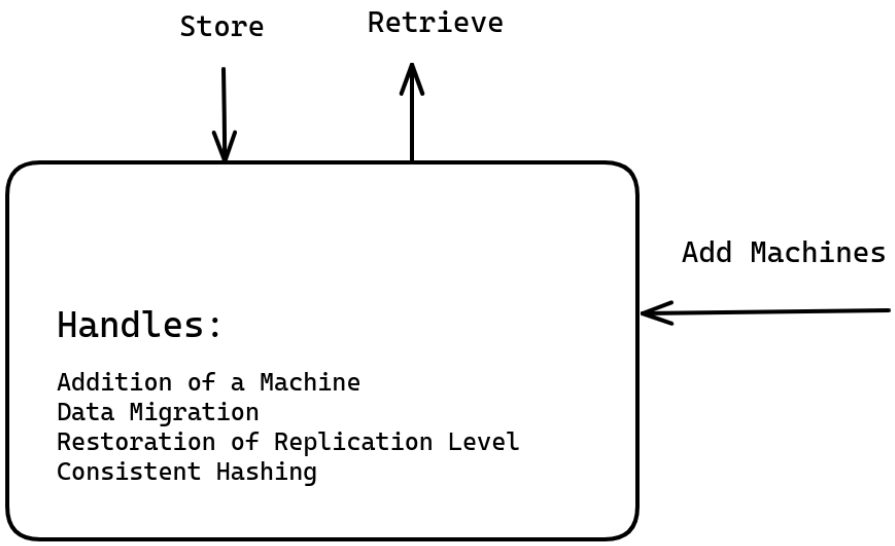


Problem Statement 2

Design a Manual Sharding system which supports:

- Adding new shard + data migration
- When a machine dies inside a shard, necessary actions are performed to restore the replication level.
- The system will keep scaling as new machines will be added.

Design the black box given below.



Manual Sharding
(Black Box)

Your task is to determine how to store data in the memory so that you are able to:

- find entries quickly &
- support updates

NoSQL Continued

Problem Discussion

Till this point, we discussed a problem statement to design a manual sharding system which supports:

- Addition and removal of machines
- Ensures even distribution of load and storage
- Maintain configuration settings such as replication level

You can assume that the system accepts the Sharding Key and Replication level as a config input.

Sharding Key: **First letter of username**

Not a good option due to:

- Uneven distribution of storage and load
    - there may be more usernames starting with ‘a’ than ‘x’.
- Under-utilization of resources
- Upper limit on the number of possible shards
    - There cannot be more than 26 shards in such a system
    - What if the website or application became really popular?
- If you proceed forward with an estimate of usernames with a particular letter, those estimates may not be fully accurate
- If the number of usernames starting with ‘a’ becomes exceedingly large, there is no way to shard further.

How to create a system which maintains the replication level (let’s say 3) automatically without you having to intervene?

Possible Solution

If the load on a machine goes beyond a certain threshold, add a new machine and transfer some part of the data to it.

Normal Drawbacks:

- Letting the sharding happen at any point in time is not desirable. What if the sharding happens at peak time? It will further degrade the response time.
- Typically, for any application, there is a traffic pattern that shows the frequency of user visits against time. It is generally seen that at a particular time, the amount of traffic is maximum, and this time is called peak time.
- In the system speculated above, it is highly likely that the data segregation to another shard will happen at peak time when the load exceeds the threshold.

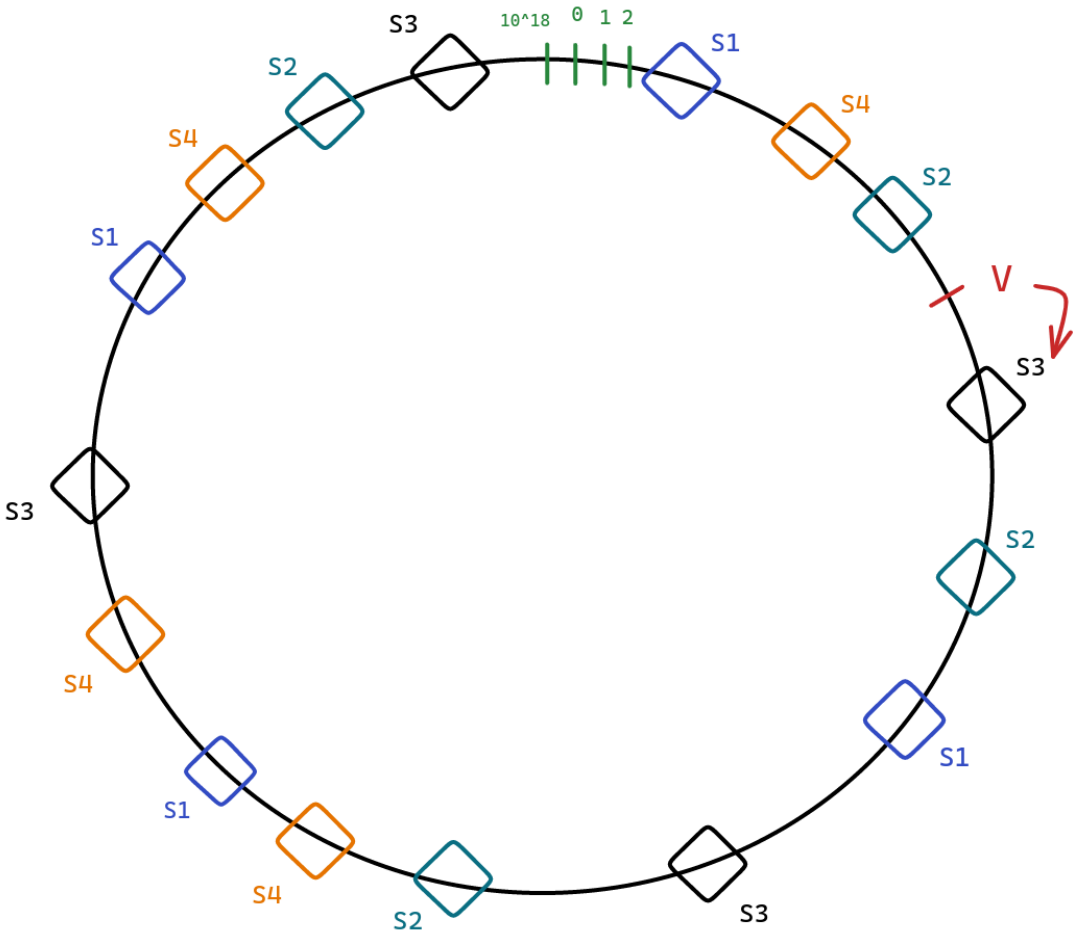
- If you are segregating the machine, then you are further adding to the load. ***Because not only now do you need to handle the peak traffic, but you also migrate data at the peak traffic time.***

Consistent Hashing Recap

- Imagine you have three shards (S1, S2 and S3) and four different hashing functions (H1, H2, H3, H4) which produce output in the range [0, 10^18].
- Determine a unique key for each shard. For example, it could be the IP of one of the machines, etc. Determine the hash values of these shards by passing their unique keys into the hashing functions.
- For each shard, there will be four hashed values corresponding to each hashing function.
- Consider the image below. It shows the shards (S1, S2, and S3) on the circle as per their hashed values.
- Let's assume **UserID** as the sharding key. Pass this UserID through a hashing function, **H,** which also generates output in the range [0, 10^18]. Let the output be **V**.
- Place this value, V in the same circle, and as per the condition, the user is assigned the first machine in the cyclic order which is **S3**.

Now, let's add a new shard **S4**. As per the outputs of hashing functions, let's place S4 in the circle as shown below.

- The addition of S4 shard helps us achieve a more uniform distribution of storage and load.
- S4 has taken up some users from each of the S1, S2 and S3 shards and hence the load on existing shards has gone down.



Note:

- Though the illustration has used a circle, it is actually a sorted array. In this sorted array, you will find the first number larger than the hashed value of **UserID** to identify the shard to be assigned.
- In the example above, UserID is used as the sharding key. In general, it can be replaced with any sharding key.

Manual Sharding

Let's consider/create a system **TrialDB** which has following properties:

- It is initially entirely empty and consists of three shards S1, S2 and S3.
- Each shard consists of one master and two slaves as shown in the image below.
- Take any sharding key which will be used to **route** to the right shard. This routing is performed by **DB Clients** running Consistent Hashing code.
- Let's assume Sulochana is directed to shard S1. Now, she can use any of the three machines if only data read is required.

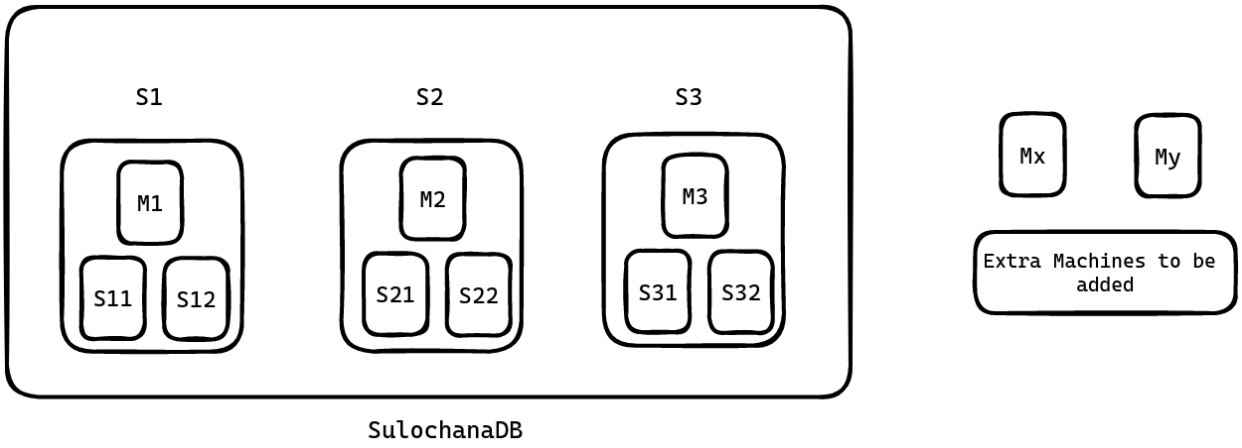
- However, the **write** behavior is dependent on what kind of system you want to create: **a highly-available** or **highly-consistent** system.

Open Questions

- How to implement the ability of adding or removing machines? Like how should the system change when new instances are added?
- What happens when a machine dies? What about the replication level?

Consider the following situation: You have two machines Mx and My which are to be added to **TrialDB**.

- What to do with these two machines?
  Options:
  - Add them to an existing shard.
  - Keep them in standby mode.
  - Create a new shard.



Another basic Algorithm

- Define an order of priority as follows:
  - Maintain the replication level (replace the crashed machines first). We have to first address the issue of under-replication. Reason behind this is we cannot afford the unavailability of the website. (Topmost priority)
  - Create a new shard.
  - Keep them in standby.
- Let's say we have **N** new machines and each shard consists of **M** machines.
  - Then N % M number of machines will be used for replacing crashed machines to maintain the replication level.
  - Remaining machines(divisible by M) will be used to create new shards.

Minor modifications discussed

- Let N = 3, M = 3 and currently one machine in S1 has died.
- But according to the algorithm, N % M = 0 machines are available to replace the dead machine.
- To solve this issue, we can decide a threshold number of machines, **X** which are always in standby to cater to our topmost priority, i.e. replacing dead machines and regaining replication level. This threshold can be a function of the existing number of shards.
- And from the remaining machines (**N - X**) we can create new shards if possible.

Note: Orchestrator implements these functionalities of maintaining reserve machines and creating new shards with the remaining ones. This Orchestrator goes by various names such as **NameNode** (Hadoop), **JobTracker**, **HBase Master**, etc.

Utilization of Standby Machines

- Contribution to existing shards by being slaves in them (additional replica).
    - If a slave dies in one shard containing one of these standby machines, you don't have to do anything as a backup is already there.

Now that we have got an idea of where to use additional machines, let's answer two questions:

- How are shards created?
- What is the exact potential number of reserve machines needed based on the number of shards?

Seamless Shard Creation

While adding a new shard, **cold start** (no data in the beginning) is the main problem. Typically, data migrations are done in two phases:

Staging Phase

- Nobody in the upper layer such as DB clients knows that there is going to be a new shard.
- Hence, the new shard does not show up in the Consistent Hashing circle and the system works as if the new shard does not exist at all.
- Now, a **Simulation** is executed to determine the **UserIDs** which will be directed to the new shard once it comes online.
- This basically determines the hash ranges that would get allocated to the new shard. Along with this, the shards which store those hashes are also determined.
- Now, let's say Staging phase starts at T1 = 10:00:00 PM and you start copying the allocated hash ranges. Assume at T2 = 10:15:00 PM the copying process is complete and the new shard is warmed up.
- However, notice it still may not have the writes which were performed between T1 and T2.

    - For example, if Manmeet had sent a write request at 10:01:00 PM then it would have gone for shard S1.
    - Let's assume Bx and By bookmarks were added by Manmeet at 10:01:00 PM. Now, there is no guarantee that these bookmarks have made their way to the new shard.

Real Phase

In this phase, the new shard is made live(with incomplete information).

- Hence you have to catch up with such relevant entries (made between T1 and T2). This catch up is really quick like a few seconds. Let's say at T3 = 10:15:15 PM, the catch up is complete.
- However, at T2 you made S4 live. Now, if Manmeet again asks for her bookmarks between **T2 and T3**, there are two choices:
    - **Being Highly Available:** Return whatever the new shard has, even if it is stale information.
    - **Being Highly Consistent:** For 15 seconds, the system would be unavailable. However, this is a very small duration (15 mins to 15 seconds downtime).

Timelines:
T1: Staging Phase starts.
T2: New shard went live.
T3: Delta updates complete. Missing information retrieved.

After T3, S4 sends signals to relevant shards to delete the hash ranges which are now served by itself (S4). This removes redundant data.

Note: Existing reserved shards are better tied to shards where it came from. Hence, these existing reserved machines could be utilized to create new shards. And the new machines can take up the reserve spot.

Estimate of the number of Reserved Machines

- Reserved Machines = X * Number of Shards
- Number of required reserved machines actually depends on the maximum number of dead machines at a time.
- Maximum number of dead machines at a time depends on various factors such as:
    - Quality of machines in use
    - Average age of machines
- Now, the approach to determine this number is to calculate
    - The probability of failing of X machines simultaneously
    - Expected number of machines dead at the same time

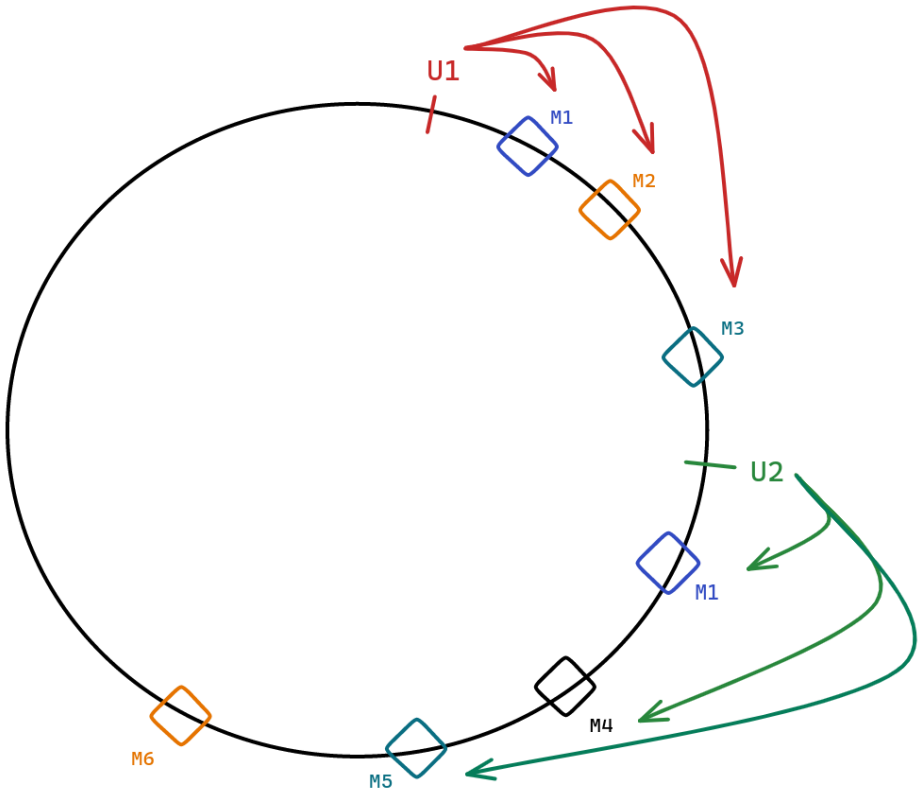
- There is another approach to this problem: Multi-master approach deployed by DynamoDB, Cassandra, etc.

Multi-Master

Consider a system of multi-master machines i.e. every machine in the system is a master. There is no slave. In the Multi-Master system, there is no need for reserved machines. Every single machine is a master and it is brought to the consistent hashing circle as and when it is live.

Master machines M1, M2, M3, etc. are shown in the Consistent Hashing circle.

- Now, let's say replication level = 3. Imagine a user U1 as shown below. Since you want to maintain three replicas, what are the optimal two machines where you should put bookmarks of U1?
- If M1 dies and U1 makes a request, which machine gets U1's request? M2 right. Hence, it would be better if M2 already had the second replica of U1's bookmarks.
- Finally, the third replica of U1 should be in M3 so that even when both M1 and M2 die, there is no struggle to find U1 data, it's already in M3.
- Remember, M2 should have a replica of only U1's bookmarks and not a complete replica of M1. Similarly for M3.
- To complete, U1's second replica should be in M2 and third replica should be in M3.
- Similarly, U2's second replica should be in M4 and third replica should be in M5.
- So, it makes sense that for a user, the three replicas should be in the next three **unique** machines in cyclic order.



Read, Write Operations

- In Multi-Master, you can have tunable consistency. You can configure two variables: **R** and **W**.
- **R** represents the minimum number of read operations required before a read request is successful.
- **W** represents the minimum number of write operations required before a write request is successful.
- Let **X** be the replication level. Then **R <= X** and **W <= X**.
- When R = 1 and W = 3, it is a highly consistent system.
    - Till all the machines are not updated, a write operation is not considered successful. Hence, highly consistent.
    - Even if one of these machines is taking time to respond or not available, writes will start failing.
- If R = 1 and W = 1, it is a highly available system.
    - If a read request arrives, you can read from any of the machines and if you get any information, the read operation is successful.
    - Similarly, if a write request arrives, if any of the machines are updated, the write operation is considered successful.
    - After the successful update of any one machine, other machines can catch up using the **Gossip** protocol.

    - This system may be inconsistent if a read request goes to a non-updated machine, you may get non-consistent information.
- In general,
    - As you increase R + W, Consistency increases.
    - **Lower R + W => Lower Consistency, Higher R + W => Higher Consistency**.
    - If **R + W > X**, you have a **highly consistent** system.
    - Because by just changing **R** and **W** it is possible to build a highly consistent or available system, it is also called **tunable consistency**.

The value of R and W depends on the type of application you are building. One of the frequent uses of DynamoDB is the Shopping Cart checkout system. Here:

- The shopping cart should be as available as possible.
- But if there should not be frequent cases of inconsistency and X = 5, then keeping R = 2, and W = 2 suffices. That way, you are writing to two different machines.
- If anytime you receive inconsistent responses from two machines, you have to merge the responses using the timestamps attached with them.

Example:
Response 1:
Lux Soap 10:00 PM
Oil: 10:15 PM

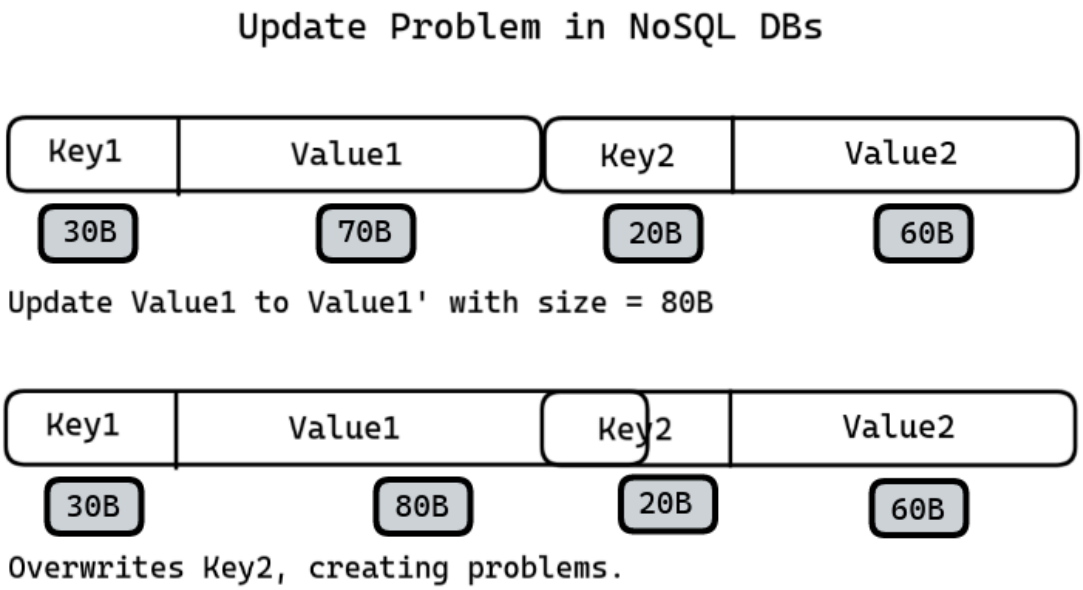
Response 2:
Lux Soap: 10:00 PM
Mask: 10: 20 PM

Merge:
Lux Soap
Oil
Mask

Questions for next class

- Storing data in SQL DBs is easy as we know the maximum size of a record.
- Problem with NoSQL DBs is that the size of value can become exceedingly large. There is no upper limit practically. It can grow as big as you want.
    - Value in Key-Value DB can grow as big as you want.
    - Attributes in Document DB can be as many in number as you want.
    - In Column Family, any single entry can be as large as you want.
- This poses a problem in how to store such data structure in the memory like in HDD, etc.

Update Problem



So, the question is how to design the data storage structure of NoSQL databases given the variable sizes of its records?

Points during discussion

- Sharding key is used to route to the right machine.
- One machine should not be part of more than one shard. This defies the purpose of consistent hashing and leads to complex, non-scalable systems.
- **Heartbeat** operations are very lightweight. They consume minimal memory (a few bytes) and a small number of CPU cycles. In the Unix machine, there are 65536 sockets by default and it uses one socket for its functioning. Hence, saving on heartbeat operations does not increase efficiency even by 0.1 percent.
- **CAP theorem** is applicable on all distributed systems. Whenever you have data across two machines and those two machines have to talk, then CAP is applicable.
- Rack aware system means the slaves are added from different racks. Similarly, Data Center aware system implies the slaves are added from different data centers.

Complexity of Write and Delete Operations

Delete operations are very fast as compared to write operations. You may have observed this when you transfer a file vs when you delete the same file. It is because:

- Delete operations do not overwrite all the bits involved. They simply remove the reference which protects the bits from getting overwritten.
    - That's why deleted files can be recovered.

- However, write (or overwrite) operation involves changing each of the bits involved, hence costlier.

System Design - Case study 2 (Typeahead)

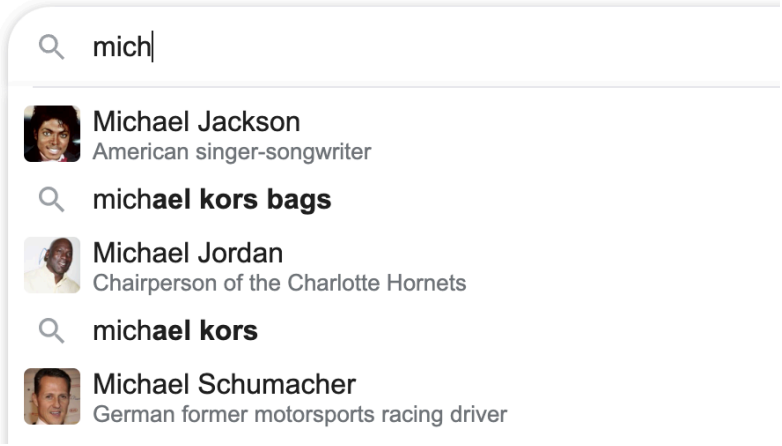
System Design - Getting Started

Before you jump into design you should know what you are designing for. The design solution also depends on the scale of implementation. Before moving to the design, ponder over the following points in mind:

- Figure out the MVP (Minimum Viable Product)
- Estimate Scale
    - Storage requirements (Is sharding needed?)
    - Read-heavy or write-heavy system
        - Write operations block read requests because they acquire a lock on impacted rows.
        - If you are building a write-heavy system, then the performance of reads goes down. So, if you are building both a read and write heavy system, you have to figure out how you absorb some of the reads or writes somewhere else.
    - Query Per Second (QPS)
        - If your system will address 1 million queries/second and a single machine handles 1000 queries/second, you have to provision for 1000 active machines.
- Design Goal
    - Highly Consistent or Highly Available System
    - Latency requirements
    - Can you afford data loss?
- How is the external world going to use it? (APIs)
    - The choice of sharding key may depend on the API parameters

Typeaheads

Typeaheads refers to the suggestions that come up automatically as we search for something. You may have observed this while searching on Google, Bing, Amazon Shopping App, etc.



Design Problem

- How to build a Search Typeahead system?
- Scale: Google

Minimum Viable Product

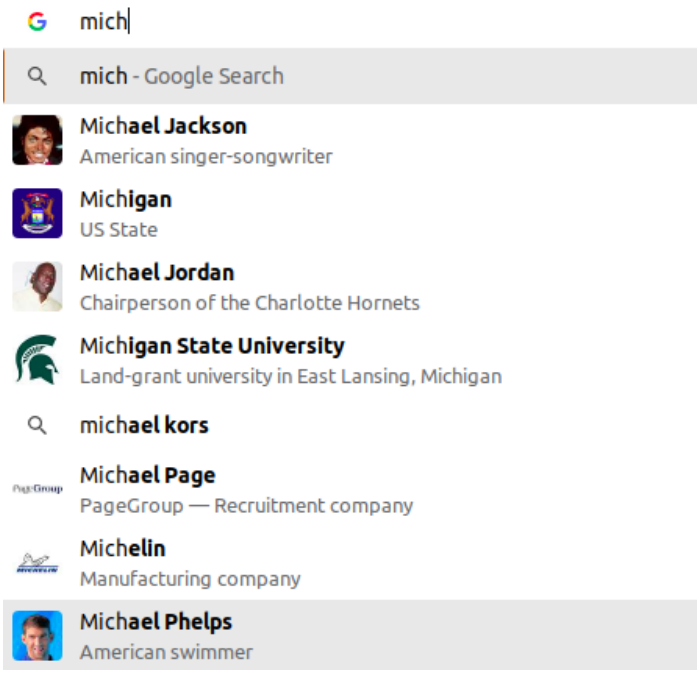
Consider Anshuman as the CEO of Google and he comes to Swaroop asking for building a typeahead system. Questions from the **Engineering Architect** (Swaroop):

- Maximum number of suggestions required?
    - Let's say five.
- Which suggestions? How to rank suggestions?
    - Choose the most popular ones. Next question-> Definition of Popularity.
    - Popularity of a search phrase is essentially how frequently do people search for that search phrase. It's combination of frequency of search, and recency. For now, assume popularity of a search term is decided by the number of times the search phrase was searched.
    - Strict prefix
- Personalisation may be required. But in MVP, it can be ignored. (In a real interview, check with the interviewer)
- Spelling mistakes not entertained.
- Keep some minimum number of characters post which suggestions will be shown.
    - Let's say 3.
- Support for special characters not required at this stage

Note:

- MVP refers to the functional requirements. Requirements such as latency, etc. are non-functional requirements that will be discussed in the Design Goal section.
- The algorithm to rank suggestions should also consider **recency** as a factor.

- For example, Roger Binny has the highest search frequency: 1 million searches over the last 5 years. On a daily basis, it receives 1000 searches.
- But, yesterday Roger Federer won Wimbledon and he has received 10000 queries since then. So, the algorithm should ideally rank Roger Federer higher.
- However, for now let's move forward with frequency only.



Estimate Scale

Assumptions:

- **Search terms** or **Search queries** refers to the final query generated after pressing Enter or the search button.
- Google receives 10 billion search queries in a day.
- The above figure translates to 60 billion typeahead queries in a day if we assume each search query triggers six typeahead queries on average.

Need of Sharding?

Next task is to decide whether sharding is needed or not. For this we have to get an estimate of how much data we need to store to make the system work.

First, let's decide what we need to store.

- We can store the search terms and the frequency of these search terms.

Assumptions:

- 10% of the queries received by Google every day contain new search terms.

    - This translates to 1 billion new search terms every day.
    - Means 365 billion new search terms every year.
- Next, assuming the system is working past 10 years:
    - Total search terms collected so far: 10 * 365 Billion
- Assuming one search term to be of 32 characters (on average), it will be of 32 bytes.
- Let's say the frequency is stored in 8 bytes. Hence, total size of a row = 40 bytes.

Total data storage size (in 10 years): 365 * 10 * 40 billion bytes = 146 TB (Sharding is needed).

Read or Write heavy system

- 1 write per 6 reads.
    - This is because we have assumed 10 billion search queries every day which means there will be 10 billion writes per day.
    - Again each search query triggers 6 typeahead queries => 6 read requests.
- Both a read and write-heavy system.

Note: Read-heavy systems have an order of magnitude higher than writes, so that the writes don't matter at all.

Design Goals

- Availability is more important than consistency.
- Latency of getting suggestions should be super low - you are competing with typing speed.

APIs

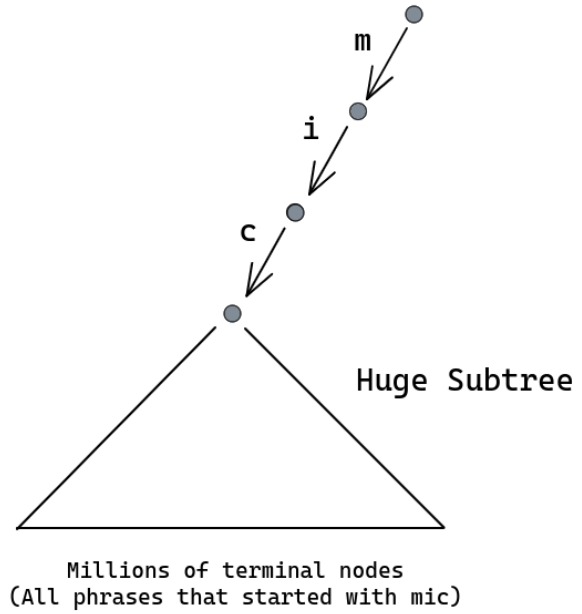
- getSuggestion(prefix_term, limit = 5)
- updateFrequency(search_term)
    - Asynchronous Job performed via an internal call
    - The Google service which provides search results to a user's query makes an internal call to Google's Typeahead service to update the frequency of the search term.

Trie Approach

- Construct a trie where each node is an English alphabet (or alphanumeric if digits are also considered)
- Each node has utmost 26 children i.e. 26 letters of the English alphabet system
- Each terminal node represents a search query (alphabets along **root** -> **terminal node**).

getSuggestions API

Consider **“mic”** as the search query against which you have to show typeahead suggestions. Consider the diagram below: The subtree can be huge, and as such if you go through the entire subtree to find top 5 suggestions, it will take time and our design goal of low latency will be violated.



Solution

* Store the top five suggestions in each node

* The getSuggestions API becomes super fast

* Fetch top five suggestions in O(prefix_term)

updateFrequency API

With the trie design suggested above where we are storing top five suggestions in every node, how will updateFrequency API work?

- In case of **updateFrequency** API call, let **T** be the terminal node which represents the **search_term**.
- Observe that only the nodes lying in the path from the **root to T** can have changes. So you only need to check the nodes which are **ancestors** of the terminal node **T** lying the path from **root to T**.

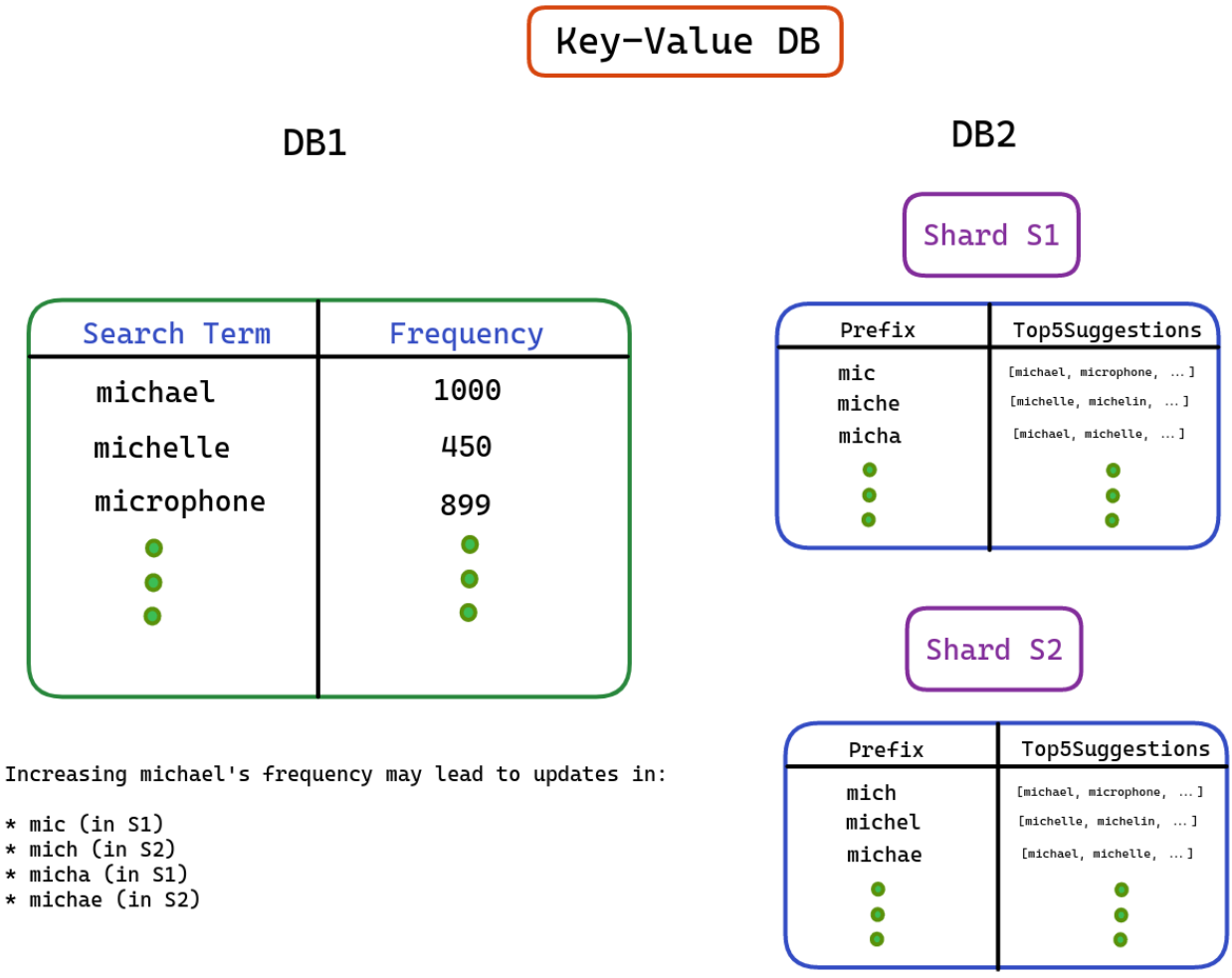
Summarizing, a single node stores:

- The frequency of the search query. [This will happen if the node is a terminal node]
- Top five suggestions with **string “root -> this node”** as prefix

HashMap Approach

- We can maintain two **HashMaps** or **Key-Value** store as follows:
    - **Frequency** HashMap stores the frequency of all search terms as a key-value store.
    - **Top5Suggestions** HashMap stores the top five suggestions corresponding to all possible prefixes of search terms.
- **Write:** Now, when **updateFrequency** API is called with a search term **S**, only the prefixes of the search term may require an update in the **Top5Suggestions** key-value store.

- **Write:** These updates on the **Top5Suggestions** key-value store need not happen immediately.
    - These updates can happen asynchronously to the shards storing the prefixes of the search term.
    - You can also maintain a queue of such updates and schedule it to happen accordingly.
- **Read:** In this system, if the search query is "**mich"**, through consistent hashing, I can quickly find the shard which stores the top five suggestions corresponding to "mich" key and return the same.
    - Consistent Hashing does not guarantee that "**mic**" and "**mich**" will end up on the same machine (or shard).
- In a Key-Value DB, Sharding is taken care of by the database itself. The internal sharding key is the key itself.
    - In any generic key-value store, the sharding happens based on the key.
    - However, you can specify your own sharding key if you want to.




```
Increasing michael's frequency may lead to updates in:

* mic (in S1)
* mich (in S2)
* micha (in S1)
* michae (in S2)
```

Optimize writes (Read and write heavy -> Read Heavy)

Reads and writes compete with each other. In the design above, a single frequency update is leading to multiple writes in the trie (on the ancestral nodes) / hashmap. If writes are very large in number, it will impact the performance of reads and eventually **getSuggestions** API will be impacted.

Can we reduce the number of writes?

- Notice that exact frequency of the search query is not that important. Only the relative popularity (frequencies) of the search queries matter.

Threshold Approach

How about you buffer the writes in a secondary store. But with buffer, you risk not updating the trending search queries. Something that just became popular. How do we address that? How about a threshold approach (detailed below):

- Maintain a separate HashMap of additional frequency of search terms. This is to say, updateFrequency does not go directly to the trie or the main hashmap. But this secondary storage.
- Define a threshold and when this threshold is crossed for any search term, update the terminal trie node representation of the search term with **frequency = frequency + threshold**.
    - Why? You don't really care about additional frequency of 1 or 2. That is the long tail of new search terms. The search terms become interesting when their frequency is reasonably high. This is your way of filtering writes to only happen for popular search term.
    - If you set the threshold to 50 for example, you are indicating that something that doesn't even get searched 50 times in a day is not as interesting to me.
- As soon as one search item frequency is updated in the trie, it goes back to zero in the HashMap.
- Concerned with the size of the HashMap?
    - Let's estimate how big can this HashMap grow.
        - We have 10 billion search in a day. Each search term of 40 bytes.
        - Worst case, that amounts to 400GB. In reality, this would be much lower as new key gets created only for a new search term. A single machine can store this.
    - If you are concerned about memory even then, flush the HashMap at the end of the day
- So basically we are creating a write buffer to reduce the write traffic and since you do not want to lose on the recency, a threshold for popularity is also maintained.

Sampling Approach

Think of exit polls. When you have to figure out trends, you can sample a set of people and figure out trends of popular parties/politicians based on the results in the sample. Very similarily,

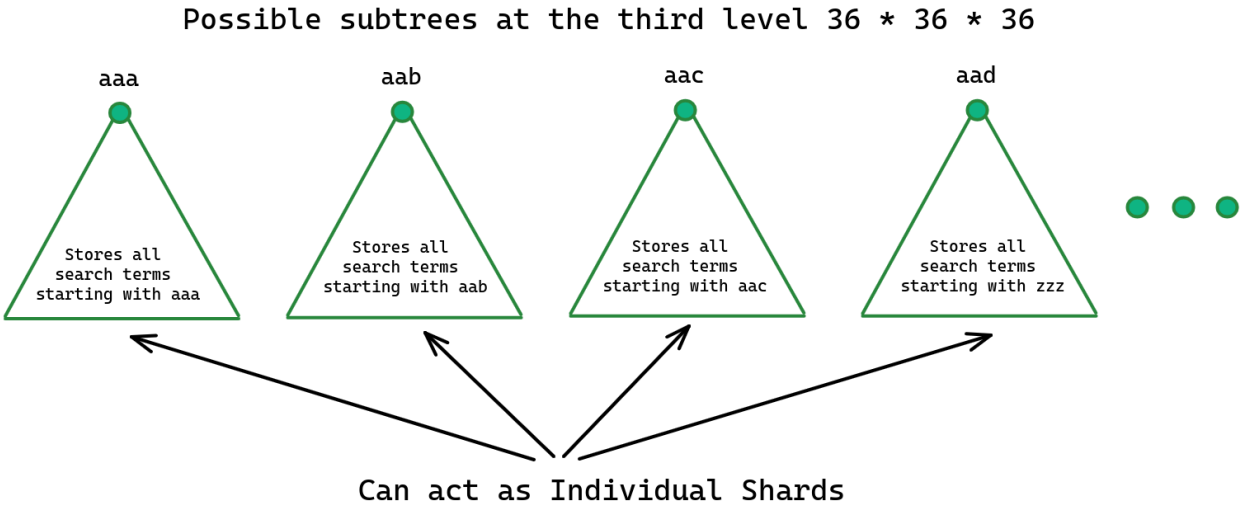
even here you don't care about the exact frequency, but the trend of who the most popular search terms are. Can we hence sample?

- Let's not update the trie/hashmap on every occurrence of a search term. We can assume that every 100th occurrence of a search term is recorded.
- This approach works better with high frequency counts as in our case. Since you are only interested in the pattern of frequency of search items and not in the exact numbers, Sampling can be a good choice.

Sharding

Sharding the trie:
Sharding key: Trie prefix

- The splitting or sharding should be on the basis of prefixes of possible search terms.
- Let's say someone has typed 3 characters. What are the possible subtrees at the third level?
    - The third level consists of 26 * 26 * 26 subtrees or branches. These branches contain prefix terms "aaa", "aab", "aac", and so on.
    - If we consider numbers as well, there will be 36 * 36 * 36 branches equivalent to around 50k subtrees.
    - Hence, the possible number of shards required will be around 50000.



Disproportionate Load Problem

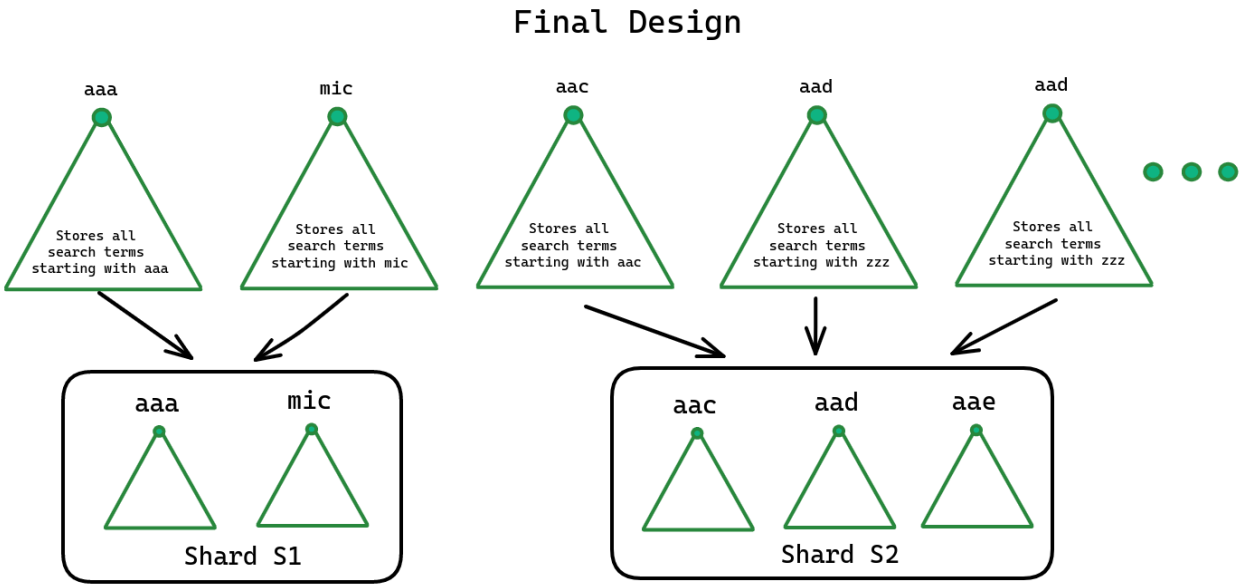
- The problem with the design above is that some shards or branches have high traffic while others are idle.
    - For example, the shard representing **"mic"** or **"the"** will have high traffic. However, the **"aaa"** shard will have low traffic.

- To solve this issue, we can group some prefixes together to balance load across shards.
    - For example, we can direct search terms starting with “**aaa”** and “**mic”** to the same shard. In this way we can better utilize the shard assigned to **“aaa”** somewhere else.
- So, we are sharding on the first three characters combined.

Example:

- Let’s say the search query is “**mich”**, you find the right shard based on the hash generated for the first three characters “**mic”** and direct the request there.

Consistent Hashing code will map to the correct shard by using the first three characters of the search term.



Sharding the Hashmap DB

- Hashmap DB is easier to shard. It’s just a collection of key and value.
- You can choose any existing key value DB and it automatically takes care of sharding
    - Consistent Hashing with sharding key as the key itself.

Recency Factor

How to take recency of search queries into consideration? How to reduce the weightage of search queries performed earlier as we move ahead?
For example, “Shania Twain” might have been a very popular search term 5 years back. But if no one searches for it today, then it’s unfair to keep surfacing it as the most popular search term in suggestions (Less likelihood of people selecting that suggestion).

- One idea is to decrease a fixed number (absolute decrement) from each search term every passing day.
    - This idea is unfair to search terms with lower frequency. If you decide to decrease 10 from each search term:
        - Search term with frequency two is already on the lower side and further decreasing will remove it from the suggestions system.
        - Search terms with higher frequency such as 1000 will have relatively no effect at all.
- To achieve this, we can apply the concept **Time Decay.**
    - Think in terms of percentage.
    - You can decay the frequency of search terms by a constant factor which is called the **Time Decay Factor**.
    - The more quickly you want to decay the frequencies, the higher the **TDF**.
    - Every day, **Freq = Freq/TDF** and when **updateFrequency** is called, **Freq++**.
        - New frequency has a weight of one.
        - Frequency from yesterday has a weight of half.
        - Frequency from the day before yesterday has a weight of one-fourth and so on.
    - According to this approach, if a search term becomes less popular, it eventually gets kicked out of the system.
- Using the concept of Time Decay, every frequency is getting decreased by the same percentage.

Summarizing the class:

- Tries cannot be saved in databases, unless you implement one of your own. Hence, you can look at the Hashmap approach as a more practical one (All key value stores work).
- Reads and writes compete, hence you need to think of caching reads or buffering writes. Since, consistency is not required in this system, hence buffering writes is a real option.
- Sampling cares only about the trend and not about the absolute count. A random sample exhibits the same trend.
    - It is like if a whole city is fighting, you pick 1% of the city randomly, even if it would be fighting as well.
    - Best example is the Election Exit Poll. Based on the response of a random set of population, we determine the overall trend.
    - Hence, the same trend is exhibited by a random sample of search queries. Using this approach, the number of writes gets reduced.
    - Basically, if you choose to sample 1% of the queries, the number of writes got reduced by 100x.
- Time Decay factor reduces the weightage of search queries performed in the past in an exponential fashion.

Schema Design- NOSQL Internals

Problems discussed earlier:

1. Unlike SQL, NOSQL is unstructured and has no fixed size.
2. How to design the system so that Updates don't take too much of time, they are feasible to do.
3. In SQL, both write and read take log(N) time.
   Therefore, how can we design our NOSQL system? Additionally how do we tweak such a system for read heavy vs write heavy system?

Solution:

Most NOSQL systems have 2 forms of storage :

1. WAL (Write ahead Log): This is an append only log of every write (new write / update) happening on the DB. Theoretically, even if you start from zero, you can replay all of these logs to arrive at the final state of the DB.
   a. Think of this as a really large file. You only append to this file and in most cases, never read from this file.
   b. Reads if done are mostly asking for a tail of this file (entries after timestamp X which are the last Y number of entries in this file).
2. The current state of data.
   We will discuss how to store the current state of data effectively.

Key: Value / RowKey: Column Family:

If we had fixed size entries, we know we could use B-Trees to store entries.
For the sake of simplicity, let's assume we are only talking about a key-value store for now.

What is the brute force way of storing key values?
Maybe I store all keys and values in a file.

Key	Value
ID 001	John
ID 002	Karen
ID 005	Bill
ID 003	Scott

Now, imagine, there is a request to update the value of "ID 002" to "Ram". Brute force would be to go find "ID 002" in the file and update the value corresponding to it. If there is a read request for "ID 002", I again will have to scan the entire file to find the key "ID 002".

This seems very slow. Both reads and writes will be very slow. Also, note that the value is not of fixed size. Also, note that when there are multiple threads trying to update the value of ID 002, they will have to take write lock (which will make things even slower). Can we do something better?

What if all new writes were just appended to the file.

Key	Value
ID 001	John
ID 002	Karen
ID 005	Bill
ID 003	Scott
ID 002	Ram

This will cause duplicate keys, but if you notice my write will become super fast. For reads, I can search for keys from the end of the file, and stop at the first matching key I find. That will be the latest entry. So, reads continue to be slow, but I have made my writes much faster.
One downside is that now I have duplicate entries and I might require more storage. Essentially, in this approach, we are indicating that every entry is immutable. You don't edit an entry once written. Hence, writes don't require locks anymore.

Reads are still super slow. O(N) in the worst case. Can we do something better?

What if we somehow index where the keys are. Imagine if there was an in-memory index (hashmap) which stored where the keys were in the file (offset bytes to seek to, to read the latest entry about the key).

Key	Value
ID 001	John
ID 002	Karen
ID 005	Bill
ID 003	Scott
ID 002	Ram

In-Memory Key	Offset
ID001	0
ID002	80
ID005	40
ID003	60

This way, the read has the following flow:

```
def read(key):
  offset = InMemoryHashmap[key]
  bytes = Read `n` bytes from file with key-value starting at offset `offset`
  key, value = parse(bytes)
  return value
```

And write is no more just a simple append to the file. It has an additional step of updating the in-memory hashmap.
This would ensure the read need not go through the entire file, and is hence no more O(N).

But there is a big flaw here. We are assuming all keys and offset will fit in-memory. In reality, a key value store might have billions of keys. And hence, storing such a map in memory might not even be feasible. So, how do we address that? Also, note that we still need a lot of memory to store duplicate older entries that are lying around.

Let's solve both one by one. How do we make the storage more efficient?

One simple answer is that we can have a background process which reads this file, removes the duplicates and creates another file (and updates the in-memory hashmap with new offsets). However, while the idea is correct, the implementation is easier said than done. Note that these are really large files. How do you even find duplicates quickly? Also, you cannot read the entire file at once. So, how do you do that in chunks that you can read all at once?

If I were to read the file in chunks of 100MB, then why have the entire thing as one single file. Why not have different files for these chunks. This will enable me to have the latest file (latest chunk) in memory, which I can write to disk when it is about to be full [Let's call this file as the **"memTable"**]. The latest chunk gets all the writes and is most likely to have the most recent entries for frequently asked items. Also, I can avoid appending to MemTable, as it is in-memory HashMap and I can directly update the value corresponding to the key [memTable will not have duplicates].

In parallel, we can merge the existing chunks [chunkX, chunkY - immutable files as new entries only affect memTable] into new chunks [chunkZ]. Delete after removing duplicate entries [Easier to find the latest entry from the in-memory hashmap which tells you whether the entry you have is duplicate or not]. Note that chunkX and chunkY are deleted, once chunkZ is created and in-memory hashmap updated. Let's call this process **"compaction".**

So, while storage might temporarily have duplicates across older chunks, compaction time to time will ensure the duplicate entries are compacted. *Compaction process can run during off-peak traffic hours so that it does not affect the performance during peak times.*

Ok, this is great! However, we still have not addressed the fact that our in-memory hashmap storing the offset for keys might not fit in memory.

Question: Given now new writes are coming to memTable, is storing keys in random order really optimal? How do we optimize searching in file without a hashmap which stores entries for all keys? Hint: Sorting?

What if memTable had all entries sorted ? [What data structure should we use then - TreeMap? Internally implemented through Balanced binary trees]. What if memTable had all entries stored

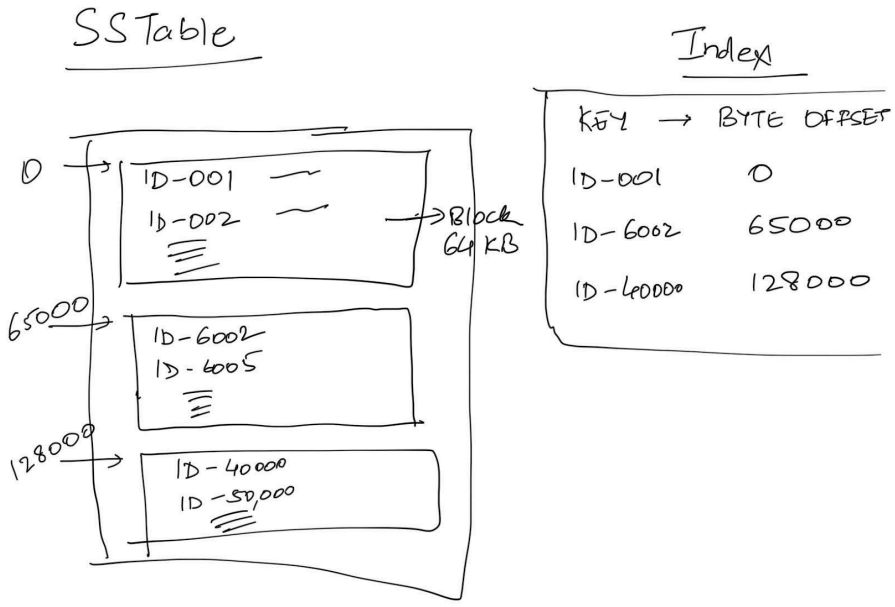
in a balanced binary tree (like Red Black Tree or AVL trees or Binary Search Tree with rotations for balancing).

That way, whenever memTable is full, when flushing content to disk, I can flush in sorted order of keys (Just like in TreeMap, you can iterate in sorted order). Let's call these files **SSTables [Sorted String Table].** With sorted order, I can do some form of binary search in the file.

But, how do I do binary search because I can land on some random byte in the file in binary search and I would have no way of finding which key/value this byte is from.

So, how about I split the file into blocks of 64Kb each. So, a 1GB file will have ~16k blocks. I store one entry per block in my index which is the first key in the block (So, index also has sorted entries - TreeMap again?).

Something like the diagram below:



In the above diagram, imagine if a request comes for ID-1234, then you would binary search for the last entry / highest entry which has block_key <= current_key I am looking for *[The block before index.upper_bound(current_key)].* In that case, I know which block my key lies in and I only have to scan 64Kb of data to find what I need. Note that this index is guaranteed to fit in memory.

What we described above is also called the LSM **Tree**. Summarizing:

- An in-memory MemTable which has entries stored as a TreeMap: All new writes go here and overwrite entry in MemTable if key exists.
- A collection of SSTable, which are sorted keys broken down into blocks. Since there can be multiple SSTable, think of them linked together like a LinkedList (newest to oldest).

- An in-memory index of blocks in SSTable.
- Time to time, a compaction process runs which merges multiple SSTables into one SSTable, removing duplicate entries. This is exactly like doing merge sort of multiple sorted arrays on disk.

Write: This is plainly an addition/update to the MemTable TreeMap.

```
def write(key, value):
    memTable[key] = value;
```

Flush MemTable to Disk:

```
def flushProcess():
    if memTable.size() <= THRESHOLD:
        return
    ss_new = new SSTable
    block = []
    key = memTable first key
    for k,v in memTable.items():
        #memTable is a TreeMap. So sorted order.
        newEntry = entry(k, v))
        if size(block) + size(newEntry) > BLOCK_THRESHOLD:
            index[ss_new.id][key] = size(ss_new) + 1 #last byte in ss_new file
            ss_new.write(block) # Flush block to SSTable
            block = [] # Start a new block
            key = k
        block.append(newEntry)

    if size(block) > 0:
        index[ss_new.id][key] = size(ss_new) + 1
        ss_new.write(block)
    ss.writeToDisk() # SSTable is ready.
    listOfSSTables.add(offset of ss, number of blocks in ss).
```

Read: If the entry is found in MemTable, great! Return that. If not, go to the newest SSTable, try to find the entry there (Find relevant block using upper_bound - 1 on index TreeMap and then scan the block). If found, return. Else go to the next SSTable and repeat. If the entry not found in any SSTable, then return “Key does not exist”.

```
def read(key):
    if memTable.hasKey(key):
        return memTable[key]
    for ss in listOfSSTables:
        relevantKeyBlock = the block before index[ss.id].upper_bound(key)
        if key found in scan of block:
            return value
    return "key does not exist"
```

Further questions:

- **What happens if the machine storing this entry reboots / restarts? Everything in the memTable will be lost since it was RAM only. How do we recover?**

  WAL comes to our rescue here. Before this machine resumes, it has to replay logs made after the last disk flush to reconstruct the right state of memTable. Since all operations are done in memory, you can replay logs really fast (slowest step being reading WAL logs from the disk).

- **How does this structure extend to column family stores where updates are appended to a particular CF and reads ask for last X entries (last X versions).**

  Mostly everything remains the same, with some minor modifications:
  - Compaction merges the 2 entries found instead of using the latest entry only.
  - Write appends in memTable to the rowKey, columnFamily.
  - Read asking for last X entries: You look for the number of entries available in memTable. If you find X entries there, return, If not, keep going and reading from SSTable, till you find X entries or you are left with no more SSTables.

- **How does delete a key work?**

  What if delete is also another (key, value) entry where we assign a unique value denoting a tombstone. If the latest value you read is a tombstone, you return “key does not exist”.

- **As you would have noticed, read for a key not found is very expensive. You look it up in every sorted set, which means you scan multiple 64Kb blocks before figuring out the key does not exist. That is a lot of work for no return (literally). How do we optimize that?**

  Bloom Filter. A filter which works in the the following way:
  Function:
  - doesKeyExist(key) : return false -> Key definitely does not exist.
    return true -> Key may or may not exist.

So, if the function returns false, you can directly return “key does not exist” without having to scan SSTables. The more accurate your bloom function, the more optimization you get.
Also, another prerequisite is that the bloom filter has to be space efficient. It should fit in memory and utilize as little space there as possible.
https://llimllib.github.io/bloomfilter-tutorial/ has a simple, interactive explanation of BloomFilter (also explained in class).

Additional Resources

https://llimllib.github.io/bloomfilter-tutorial/
https://hur.st/bloomfilter/
https://dev.to/creativcoder/what-is-a-lsm-tree-3d75

System Design Case Study: Design Messenger

Again, We follow the same structure and broadly we divide it into 4 sections,

1. Defining the MVP
2. Estimation of Scale: Primarily to determine 2 things,
   2.1 Whether we need Sharding.
   2.2 Whether it's a read heavy system, or write heavy system or both.
3. Design goals.
4. API+Design.

MVP:

- Send a message to the recipient.
- Realtime chat.
- Message history.
- Most recent conversations.

Estimation of Scale:

Let's say starting with 20 billion messages/day.
Every single message is 200 bytes or less.
That means 4TB/day.
If we want to let's say save our messages for 1 year. So for 1 year, it will be greater than PB. In reality, if we are building for the next 5 years, we need multiple PB of storage.

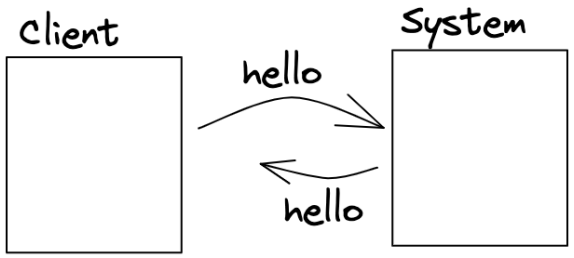
1. We definitely need Sharding!!
2. It's both a read + write heavy system.

Design Goals:

System should be **Highly Consistent** because inconsistency in communications can lead to issues.
Latency should be low.

APIs

In a case of an app like messenger where consistency is super important, one thing to consider should be that your write APIs are idempotent. You need to consider this because your primary caller is a mobile app which could have an intermittent network. As such, if the API is called multiple times, due to application level retries, or if data reaches multiple times due to network level retries, you should not create duplicate messages for it.



Let's say we have a client that is trying to talk to your backend.
Imagine I send a message "Hello" to the backend. The backend gets the message, successfully stores the message, but the connection breaks before it could return me a success message. Now, it's possible I do a retry to ensure this "Hello" message actually gets sent. If this adds 2 messages "Hello", followed by another "Hello", then the system we have is not idempotent. If the system is able to deduplicate the message, and understand it's the same message being retried and hence can be ignored, then the system is idempotent.

How to make the system Idempotent:

We can use a messageId - something that is different across different messages, but same for the same message retried.
Imagine everytime we send a message "hello", the moment "hello" is generated, a new messageId is generated.
Now, when we send this message to the backend, instead of saying user A is sending user B a message "Hello", we say user A is sending userB a message "Hello" with messageId as xyz. Then even if the system gets the same message again then it can identify that it already has a message with messageID xyz and hence, this new incoming message can be ignored.
This however, won't work if messageID is not unique across 2 different messages (If I type "Hello" twice and send twice manually, they should be considered 2 different messages and should not be deduplicated).

How to generate Unique messageId:

We can possibly use the combination of:

- Timestamp(date and time)
- senderID
- deviceID
- recipientID (To be able to differentiate if I broadcast a message).

APIs

Before thinking of the APIs, think of the usecases we would need to support. What kind of views do we have?
First view is the view when I open the app. Which is a list of conversations (not messages) with people I recently interacted with (has name of friend/group, along with a snippet of messages. Let's call that getConversations.

If I click into a conversation, then I get the list of most recent messages. Let's call that getMessages.
And finally, in that conversation, I can send a message.
So, corresponding APIs:

1. SendMessage(sender, recipient, text, messageId)
2. getMessages(userId, conversationId, offset, limit)

Offset: Where to start
Limit: How many messages after that is limit. Offset and limit are usually used to paginate (if page sizes can be different across different clients).

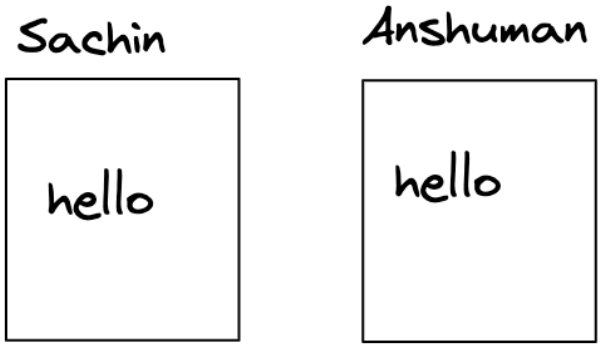
3. getConversations(userId, offset, limit)
4. CreateUser(---).

System Design:

Problem #1: Sharding:

1. userId: All conversations and messages should be on the same machine. Essentially, every user has their own mailbox.
2. ConversationId: Now all messages of a conversation go on the same machine.

userID based sharding:
So every user will be assigned to one of the machines.



Now, Let's do each of the operations:

1. getConversation: It's pretty easy.
   Imagine if Sachin says, get the most recent conversation. We go to a machine corresponding to Sachin and get the most recent conversations and return that.

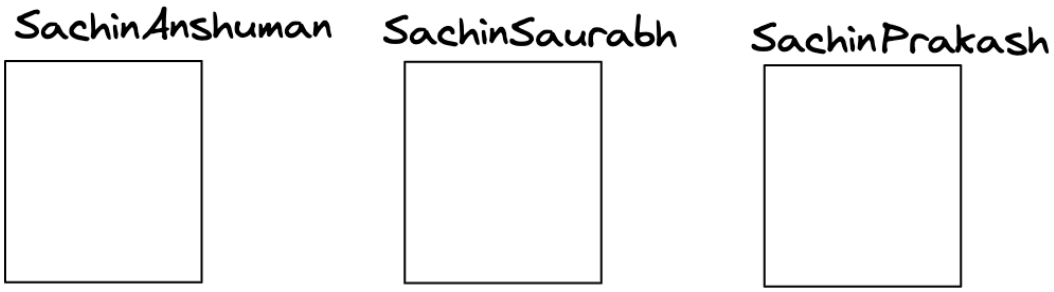
2. getMessages:
   Same, We can go to a machine corresponding to Sachin and for that we can get the messages for a particular conversation.

3. sendMessage:

Imagine if Sachin sends a message “hello” to Anshuman, Now this means we have 2 different writes, in Sachin’s machine and in Anshuman’s machine as well and they both have to succeed at the same time.
So, for sendMessage in this type of sharding, there should be 2 writes that need to happen and somehow they still need to be consistent which is a difficult task.

conversationID based sharding:
Here for every conversation we will have a separate machine.

For example,



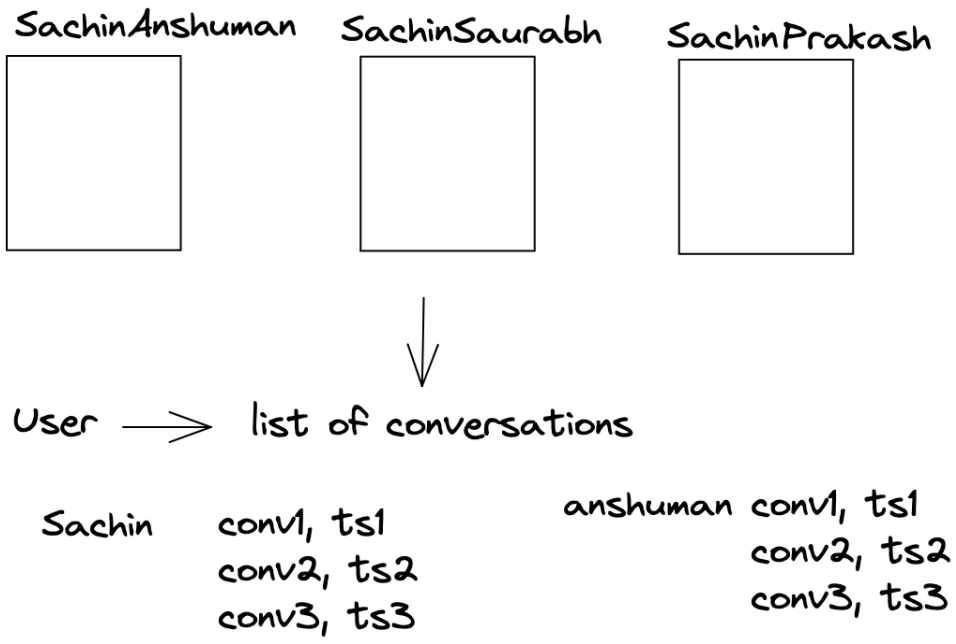
Now, Let’s do each of the operations again:

1. getMessages:
   Say, We want to get the last 100 messages of conversation b/w Sachin and Anshuman.
   So we will go to the corresponding machine which has Sachin/Anshuman messages and fetch the messages inside.

2. sendMessage:
   This is also fairly simple. If Sachin wants to send a message to Anshuman, we go to the machine corresponding to Sachin/Anshuman, and add a message there.

3. getConversations:
   For example, we want to get the latest 10 conversations that Sachin was part of.
   Now in this case, we need to go to each and every machine and fetch if there is a conversation which has the name Sachin in it. ***That is very inefficient.***

One solution might be to have a **Secondary database:**
In this database we can have user to list of conversations (sorted by recency of the last message sent - along with metadata of conversations - snippet, last Message Timestamp, etc.).



Now again if we do these operations:

1. getMessage: it will work fine.
   If we say get the last 10 messages with the conversation of Anshuman and Sachin. Since they are sharded by conversationId, it will have one machine which has all the messages.

2. getConversations: Now again we can't go to any one of the databases, we have to go to the secondary database and have to read from here.

3. sendMessage:
   If let's say Sachin sends the message in the conversation b/w Sachin and Anshuman, In this case, we have to add the message to SachinAnshuman Database and then in the secondary database we have to change the ordering of conversations in both Sachin and Anshuman's list of conversations.
   Therefore a single send message has 3 writes.

For systems like, Slack, MS Team, Telegram that can have large groups, userID based sharding will be ridiculously expensive as every single sendMessage will leads to 1000 writes in a 1000 member group. Hence, they relax on the ordering of threads in getConversations (best effort) and instead use conversationId based sharding.

For 1:1 messages-> UserId seems to be a better choice (2 writes vs 3). That being said, you can't go terribly wrong with conversationID either. For the purpose of this doc, we will use userID.

In sharding based on userId, 2 operations are working perfectly fine: getMessage, gerConversation.
But the problem is with sendMessage, when we send a message hello, it was written to 2 different machines and if one of those writes fails then probably both the machines become inconsistent.

Problem #2: sendMessage consistency

Consistency means: If user1 sends a message “Hi”, and does not get an error, then it should imply that the message has been delivered to user2. User2 should get the message.

If user1 sends a message to user2, how should we sequence the write between user1 DB and user2 DB to ensure the above consistency?

Case1: write to sender/user1 first.

1. If it fails then we return an error.
2. If it succeeds, then:
   We write to recipient / user2 shard: again it can have 2 possibilities:
   2.1: Success: Then the system is in consistent state and they return success.
   2.2: Failure: Rollback and return error.

Case2: Write to recipient/user2 shard:

1. Failure: Simply return an error.
2. Succeed: It has reached to recipient so system is in consistent state.
   Then we write to sender shard:
   1. Success: System is in consistent state, return success.
   2. Failure: Add it to the queue and keep retrying so that eventually it gets added to the sender’s shard.

Out of these 2 cases, case2 is much better because the sender sends the message and the recipient gets the message. The only problem is when the sender refreshes he cannot see the message. Not the good behavior but better of the two behaviors.
Because case1 is very dangerous, Sender sends the message and when he refreshes the message is still there but the recipient never got it.

Problem #3: Choosing the right DB / cache:

Choosing the right DB here is very tricky as this system is both read heavy and write heavy. As we have discussed in the past, both compete with each other, and it’s best to reduce this system to either read heavy or write heavy on the storage side to be able to choose one DB.
Also, this requires massive amount of sharding. So, we are probably looking for a NoSQL storage system that can support high consistency.

Reduction to read vs write heavy
If we were building a loosely consistent system where we cared about trends, we could have looked to sample writes / batch writes. But here, we need immediate consistency. So, absorbing writes isn't feasible. You'd need all writes to be immediately persisted to remain highly consistent.

That means, the only read option is to somehow absorb the number of reads through heavy caching. But remember that you'd need to cache a lot for it to absorb almost all of reads (so much that it won't fit on a single machine) and this cache has to be very consistent with the DB. Not just that, you'd need to somehow handle concurrent writes for the same user to not create any race condition.

Consistency of cache: We can use write-through cache.
Lots of data to be cached: We would need to shard cache too.
Handle write concurrency in cache: How about we use appservers / business logic servers as cache. We can take a write lock on user then.

A simple way to do this might be to use appservers as cache, and have them be tied to a set of users (consistent hashing of users -> appservers). This would also let you take a write lock per userID when writes happen, so that writes happen sequentially and do not create race condition in the cache.
Since you cannot cache all the users, you can look at doing some form of LRU for users.

Pros:
- Can scale horizontally. Throw more appservers and you can cache more information.
- Race conditions and consistency requirements handled gracefully.

Cons:
- If the server goes down, things are unavailable till reassignment of the user happens to another app server. Since, this might take a few seconds, this causes unavailability of a few seconds every time the appserver is down.
- When the app server restarts, there is a cold cache start problem. The initial few requests are slower as the information for the user needs to be read from the DB.

Right DB for Write heavy, consistent system:
If we successfully absorb most of the reads, so that they rarely go to the DB, then we are looking for a DB that can support write-heavy applications. HBase is good with that. It allows for column family storage structure which is suited to messages/mailbox, and is optimized for high volumes of writes.

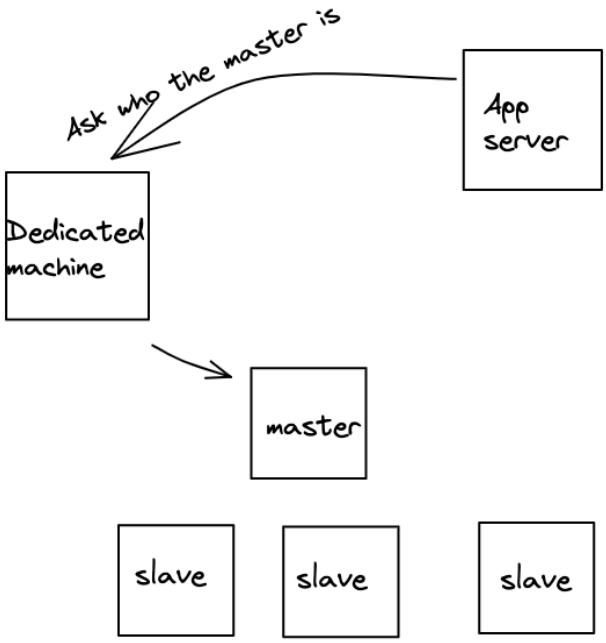
State tracking - Zookeeper :

In a Master Slave architecture, all writes must come to the master and not the slave machines. This means all clients (appservers) must be aware of who the master is. As long as the master is the same, that's not an issue.

The problem is, If the master might die, and in that case we want to select a new master and all the machines should be aware of it, they should be in sync.

If you were to think of this as a problem statement, how would you solve it?

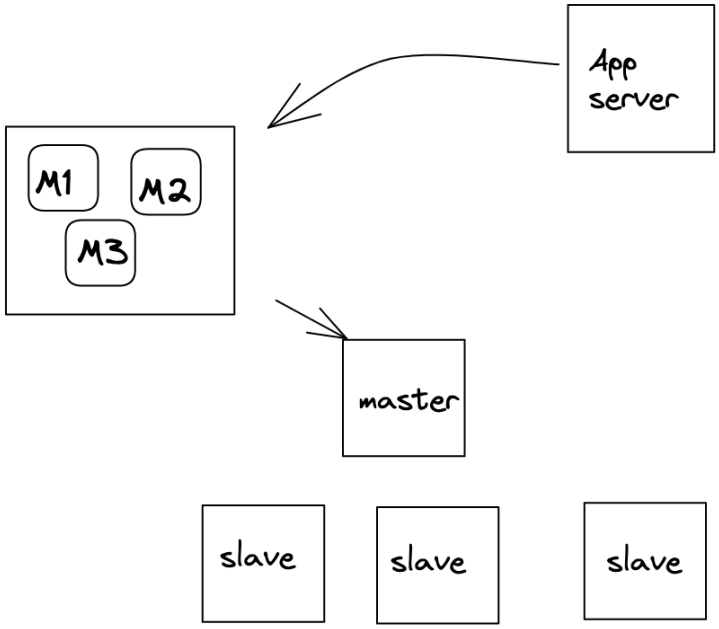
A naive approach might be to say that we will have a Dedicated machine and the only job of this machine is to keep track of who the master is. Anytime an appserver wants to know who the master is, they go and ask this dedicated machine.



However, there are 2 issues with this approach.

1. This dedicated machine will become the single point of failure. If the machine is down, no writes can happen - even though the master might be healthy.
2. For every request, we have introduced an additional hop to find out who the master is.

To solve the issue #1, maybe instead of 1 machine we can use a bunch of machines or clusters of machines.

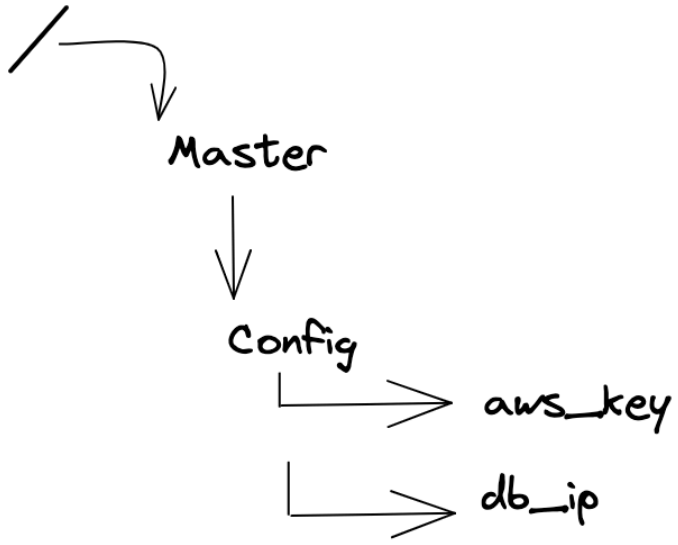


- *How do these machines find out who the master is?*
- *How do we make sure that all these machines have the same information about the master?*
- *How do we enable appservers to directly go to master without the additional hop to these machines?*

Solution: Zookeeper:

Zookeeper is a generic system that tracks data in **strongly consistent** form. More on this later.

Storage in Zookeeper is exactly like a file system.
Example, we have a root folder inside that we have bunch of files or directories.



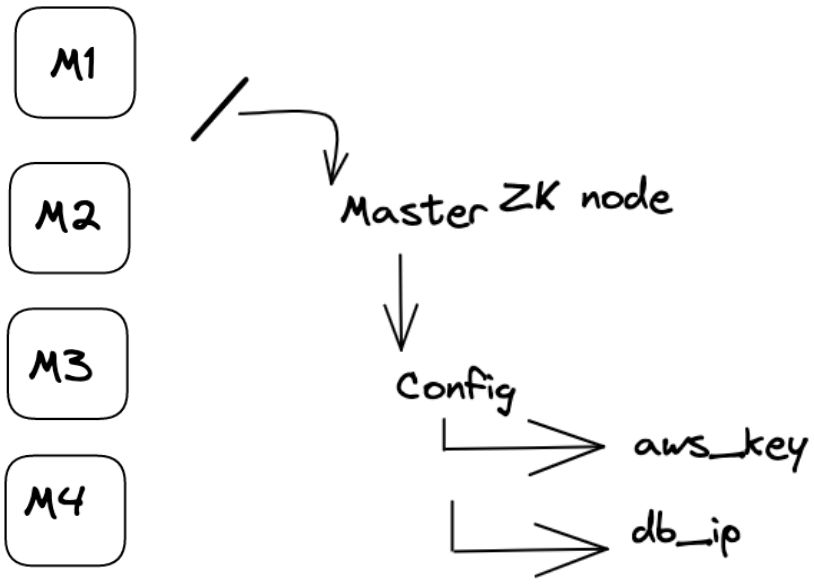
All these files are known as ZK nodes in zookeeper.

ZK Nodes:
Every file in zookeeper is of one of two kinds:

1. Ephemeral : Ephemeral nodes ***(do not confuse node to mean a machine. Nodes are files in the context of Zookeeper)*** are files where the data written is only valid till the machine/session which wrote the data is alive/active. This is a fancier way of saying that the machine which wrote on this node has to keep sending heartbeat to ensure the data on this node is not deleted.
    a. Once an ephemeral node is written, other machines / sessions cannot write any data on it. An ephemeral node has exactly one session/machine as the owner. Only the owner can modify the data.
    b. When the owner does not send a heartbeat, the session dies and the ephemeral node is deleted. This means any other machine can then create the same node/file with different data.
    c. These are the nodes which are used to track machine status, master of a cluster, taking a distributed lock, etc. More on this later.

2. Persistent : Persistent nodes are nodes where the node is not deleted unless specifically requested to be deleted. These nodes are used to store configuration variables.

ZK Node for consistency / Master Election:
To keep things simple, let's imagine that Zookeeper is a single machine (we will move to multiple machines later). Let's imagine there are a bunch of storage machines in a cluster X.



They all want to become master. However, there can only be one master. So, how do we resolve the “kaun banega master” challenge. We ask all of them to try to write their IP address as data to the same ephemeral ZK node (let’s say /clusterx/master_ip).

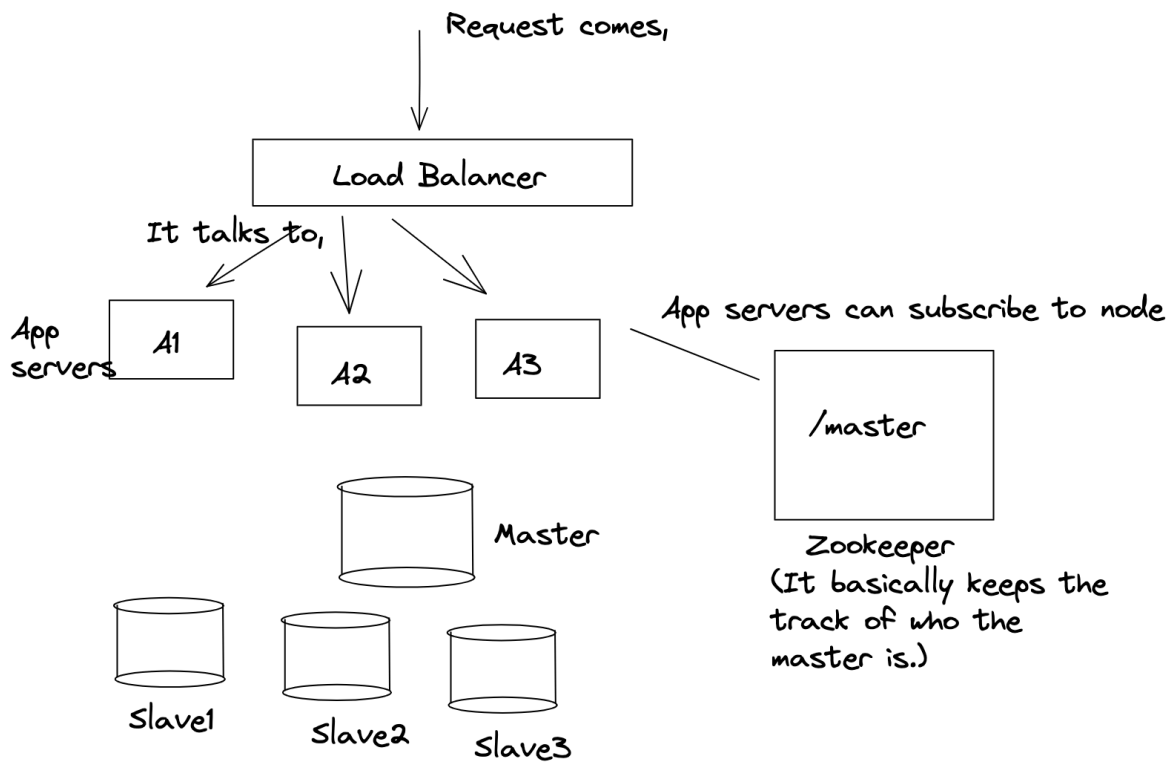
Note that only one node will be able to write to this ephemeral node and all other writes will fail. So, let’s say M2 was able to write M2’s ip address on /clusterx/master_ip.
Now, as long as M2 is alive and keeps sending heartbeat, /clusterx/master_ip will have M2’s ip address. When any machine tries to read data on /clusterx/master_ip, they will get M2’s ip address in return.

ZK: Setting a watch:
There is still the additional hop problem. If all appservers and other machines have to talk to the zookeeper on every request to find out who the master is, not only does it add so much load to zookeeper, it also increases hops of every request.

How can we address that?
If you think about it, the data on ephemeral node changes very less frequently (probably like once in a day - not even that). It seems stupid that every client has to come to ZK to ask for the master value when it does not change most of the time.
So, how about we reverse the process. We tell people, “Here is the value X. No need to keep asking me again and again. Keep using this value. Whenever this value gets updated, I will notify you.”.

Zookeeper does a similar thing. It solves that using a “subscribe to updates on this ZK node” option.
On any ZK node, you can **set a watch** (subscribe to updates). In ZooKeeper, all of the read operations have the option of setting a watch as a side effect.
If I am an appserver, and I set a watch on /clusterx/master_ip, then when this node data changes or this node gets deleted, I (and all other clients who had set a watch on that node) will be notified. This means when clients set a watch, zookeeper maintains a list of subscribers (per node/file).

ZK: Architecture:
All of this is great. But we were assuming ZK is a single machine. But ZK cannot be a single machine. How does this work across multiple machines?



Now the problem is Zookeeper is a single machine and if it's a single machine it becomes a single point of failure.
Hence zookeeper is actually a bunch of machines **(odd number of machines**).

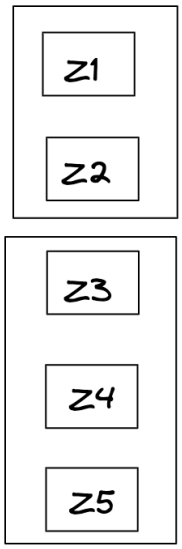
Zookeeper machines also select a leader/master among themselves. When you setup the set of machines (or when the existing leader dies in case of running cluster), the first step is electing the leader. [How is a leader elected in Apache ZooKeeper? - Quora / ZK Leader Election Code for the curious ones to explore how leader election happens in Zookeeper].

Now, let's say Z3 is elected as leader, whenever any write is done, for Eg, change /master ip address to ip,x then it is first written to the leader and leader broadcasts that change to all other machines. If at least the majority of the machines (including the leader) acknowledge the change, then the write is considered successful, otherwise it is rolled back.
So, in a cluster of 5 machines, 3 machines need to acknowledge for the write to succeed (in a cluster of 7, 4 machines need to acknowledge and so forth). *Note that even if a machine dies, the total number of machines still stays 5, and hence even then 3 machines need to acknowledge.*

Hence, if 10 machines were trying to become the master and they all sent requests to write to /clusterx/master simultaneously, all those requests would come to a single machine - ***the leader,*** first. The leader can implement a lock to ensure only one of those requests goes

through first, and the data is written if majority ZK machines acknowledge. Else data is rolled back, lock released and then the next request gets the lock.

But why the majority number of machines?
Let's imagine we let the write succeed if it succeeds on X/2 number of machines (X being the total number of machines). For this let's imagine we have 5 zookeeper machines, and because of network partition z1 and z2 become disconnected from the other 3 machines.



Let's say write1 (/clusterx/master_ip = ip1) happens on z1 and z2. .
Let's say another write write2 (/clusterx/master_ip = ip2) happens for the same ZK node on z4 and z5.

Now when we try to read (/clusterx/master_ip) then half of the machines would suggest ip1 is master, and the other half would return ip2 as master. This is called **split brain.**
Hence we need Quorum / Majority so that we don't end up having 2 sets of machines saying x is the answer or y is the answer, there should be consistency.
So till the write is not successful on majority of the machines we can't return success, now in this case both ip1 and ip2 try to write in Z3 and whoever succeeds the master will have that address and the other one fails.

ZK: Master dies
Imagine master had written it's ip address to */clusterx/master_ip*. All appservers and slaves had set watch on the same node (noting down the current master IP address).

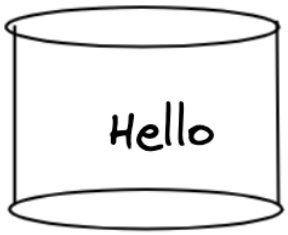
Imagine the master dies. What happens?

- Master machine won't be able to send heartbeat to the zookeeper for the ephemeral node */clusterx/master_ip*
- The ephemeral node */clusterx/master_ip* will hence be deleted.
- All subscribers will be notified of the change.

- Slaves, as soon as they get this update, will try to become masters again. Whoever is the first one to write on Zookeeper becomes the new master.
- Appserver will delete the local value of master_ip. They would have to read from the zookeeper (+set new watch + update local master_ip value) whenever the new write request comes.
    - If they get back null as value, the request fails. New master is not selected yet.
- Old master whenever it comes back up will read from the same ZK node to find out the new master machine and will become a slave itself.
    - Unless it comes back up quickly, finds ZK node to be null and tries along with other slaves to become the new master.

Async tasks - Messaging Queues :

Let's take an example of the messenger,
Imagine whenever a message comes for a user, say abhi sends a message to raj, so the message is written in the raj database.



After this we want to do a couple of things.

1. Notify raj
2. Email to raj
   (If raj is not reading messages for the last 24 hrs).
3. Update relevant metrics in analytics

Now whenever a message comes, we have to do these things but we don't want that the sender of the message to wait for these things to happen. Infact, if any of the above fails, it does not mean that the message sent itself failed.
So how to return success immediately?

To solve these types of problems where we have to do a few things asynchronously we use something known as **Persistent Queue.**
Persistent Queue is durable which means we are actually writing it in a hard disk so that we won't lose it.

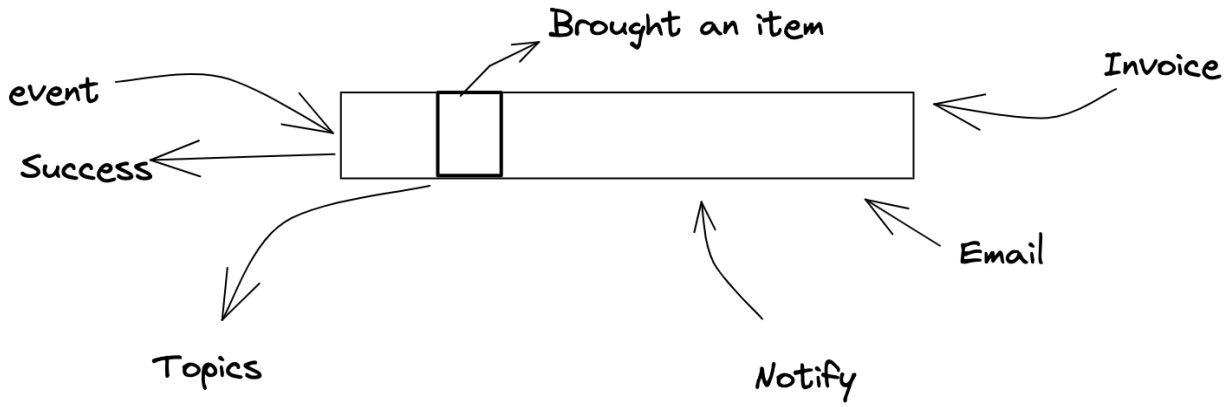
Solution: Persistent Queue:
Persistent Queues work on a model called pub-sub (Publish Subscribe).

PubSub:
Pubsub has 2 parts:

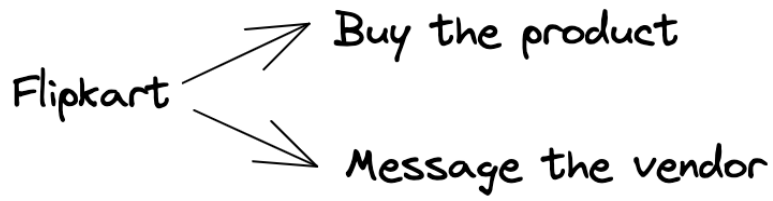
- **Publish:** You look at all events of interest that would require actions post it. For example, a message being sent is an event. Or imagine someone buys an item on Flipkart. That could be an event. You publish that event on a persistent queue.
- **Subscriber:** Different events could have different kind of subscribers interested in that event. They ***consume*** events they have subscribed to from the queue. For example, in the above example, message notification system, message email system and message analytics system would subscribe to the event of “a message sent” on the queue.
    - Or an invoice generation system could subscribe to the event of “bought an item on Flipkart”.

There could be multiple types of events being published, and each event could have multiple kind of subscriber consuming these events.

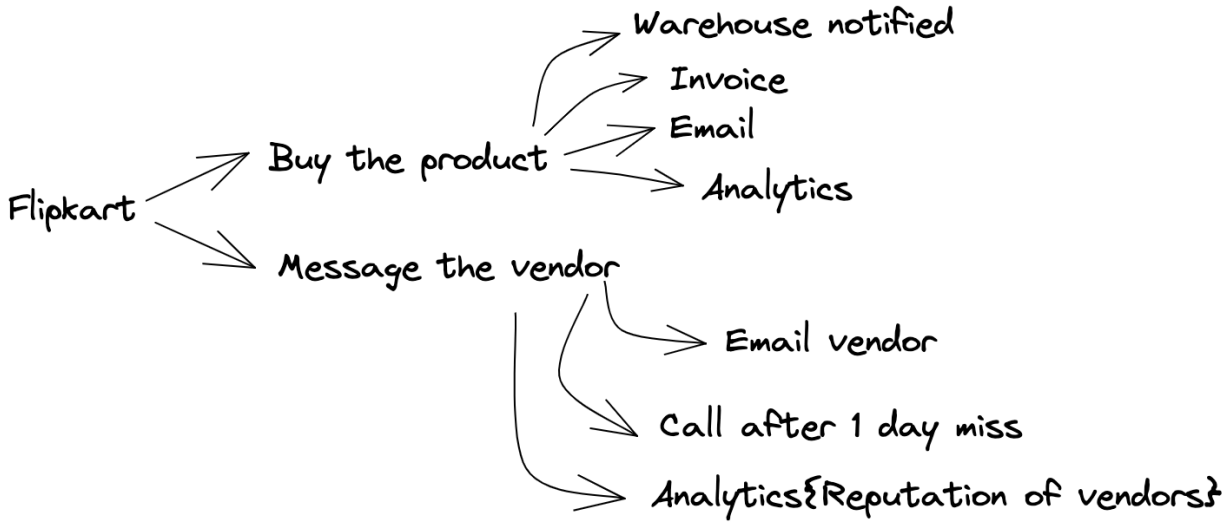
Topics:
Now within a queue also we need some segregation of topics because the system doesn't want to subscribe to the whole queue, they need to subscribe to some particular type of event and each of these events is called topic.



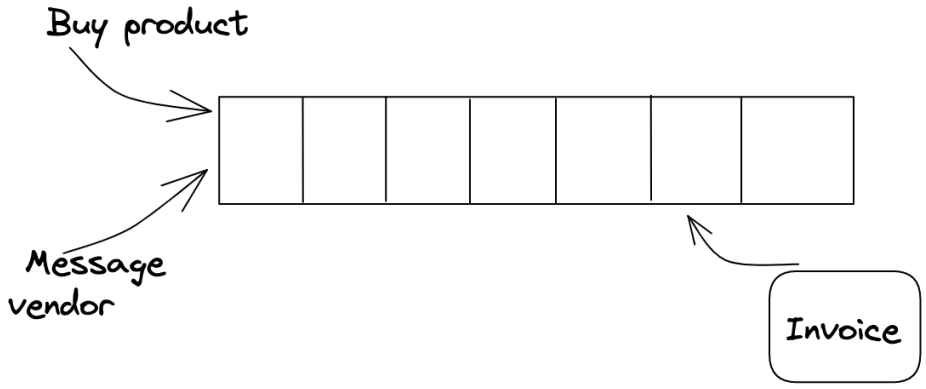
Let’s take an example of Flipkart,
Say, flipkart also has an inbuilt messaging service; we can message the vendor about the quality and feedback of the product.



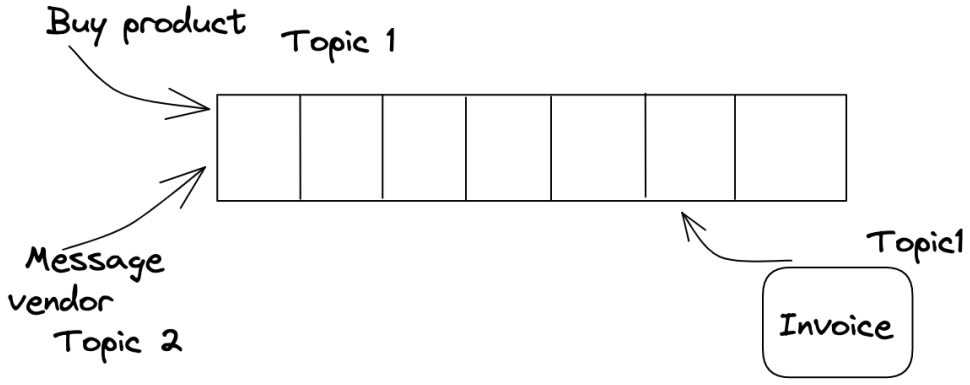
These are the two events, now after that, we want certain things to happen,



Here both the events are very different.
If we publish both of the events in a single persistent queue, and let's say invoice generation has subscribed to the queue then it will get a lot of garbage.



Hence we say all the events are not the same, and we classify them in different topics.

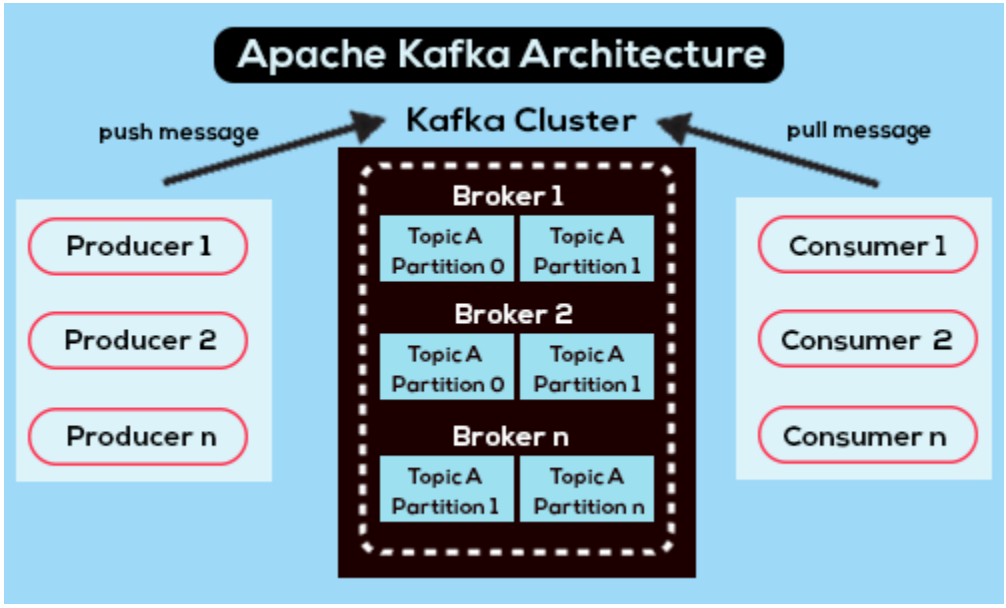


Now the invoice generation has only subscribed to Topic1 and would only get messages from Topic1.

One such high throughput system that implements persistent queue and supports Topics is **KAFKA.**

In general, persistent queues help handle systems where producers consume at a different rate and there are multiple consumers who consume at a different pace asynchronously. Persistent Queues take guarantee of not loosing the event (within a retention period) and let consumers work asynchronously without blocking the producers for their core job.

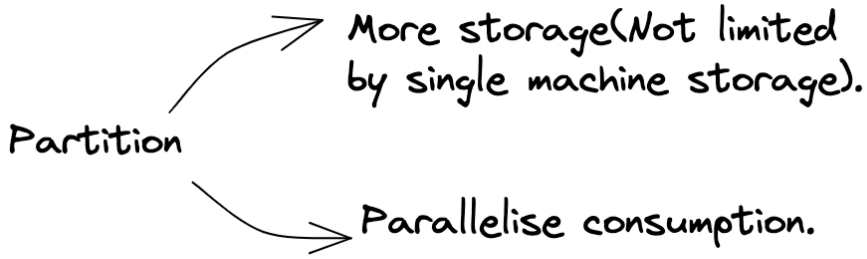
Kafka:



Terminologies:

- **Producer:** Systems that publish events (to a topic) are called producers. There could be multiple producers.
- **Consumer:** Systems that consume events from subscribed topic (/topics) are called consumers.
- **Broker:** All machines within the Kafka cluster are called brokers. Just a fancy name for machines storing published events for a topic.
- **Partition:** Within a single topic, you can configure multiple partitions. Multiple partitions enables Kafka to internally shard / distribute load effectively. It also helps consumers

consume faster. *More on this later.*



- **Event retention period:** Kafka or any persistent queue is designed to store events transiently and not forever. Hence, you specify event retention period, which is the period till which the event is stored. All events older than retention period are periodically cleaned up.

Problem statements:

- ***What if a topic is so large (so there are so many producers for the topic), that the entire topic (even for the retention period) might not fit in a single machine. How do you shard?***

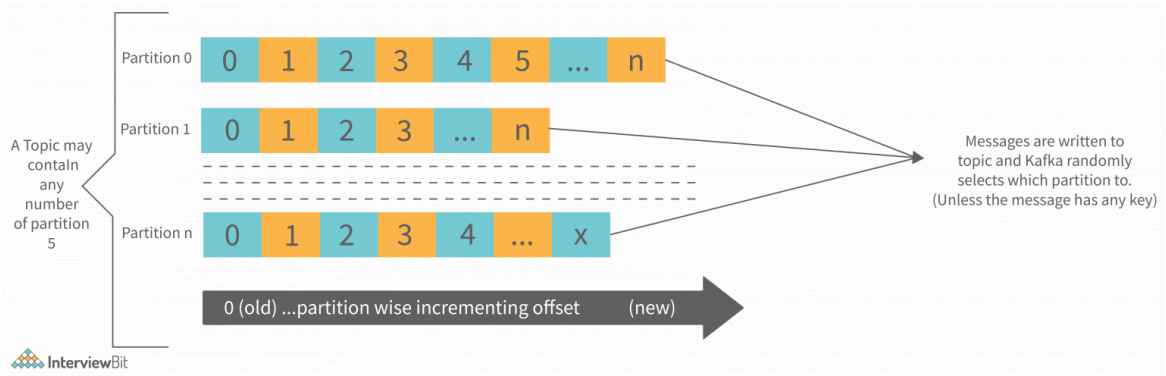
Kafka lets you specify number of partitions for every topic. A single partition cannot be broken down between machines, but different partitions can reside on different machines.
Adding enough partitions would let Kafka internally assign topic+partition to different machines.

- ***With different partitions, it won't remain a queue anymore. I mean wouldn't it become really hard to guarantee ordering of messages between partitions?***
    - For example, for topic messages, m1 comes to partition1, m2 comes to partition2, m3 comes to partition 2, m4 comes to partition 2 and m5 comes to partition 1. Now, if I am a consumer, I have no way of knowing which partition has the next most recent message.

Adding ways for you to know ordering of messages between partitions is an additional overhead and not good for the throughput. It is possible you don't even care about the strict ordering of messages.
Let's take an example. Take the case of Flipkart. Imagine we have a topic Messages where every message from customer to vendor gets published. Imagine we have a consumer which notifies the vendors.
Now, I don't really care about the ordering of messages for 2 different users, but I might care about the ordering of messages for the messages from the same user. If not in order, the messages might not make sense.

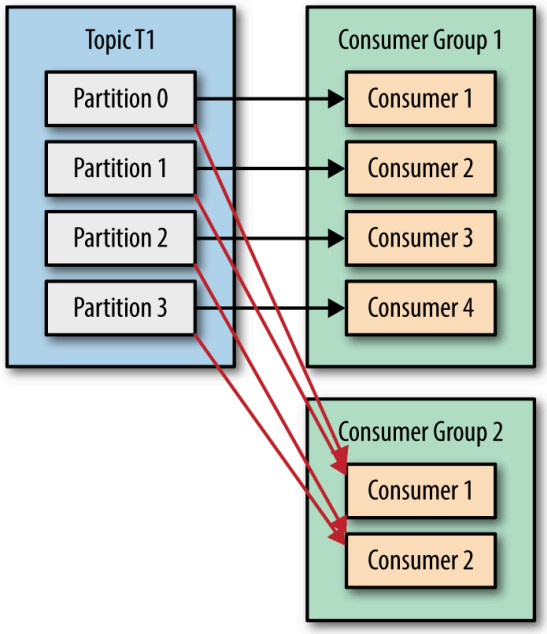
What if there was a way of making sure all messages from the same user ends up in the same partition. Kafka allows that.
Producers can optionally specify a key along with the message being sent to the topic.
And then Kafka simply does hash(key) % num_partitions to send this message to one of the

partition. If 2 messages have the same key, they end up on the same partition. So, if we use sender_id as the key with all messages published, it would guarantee that all messages with the same sender end up on the same partition. Within the same partition, it's easier to maintain ordering.



- **What if a topic is super big. And hence it would take ages for a single consumer to consume all events. What do you do then?**

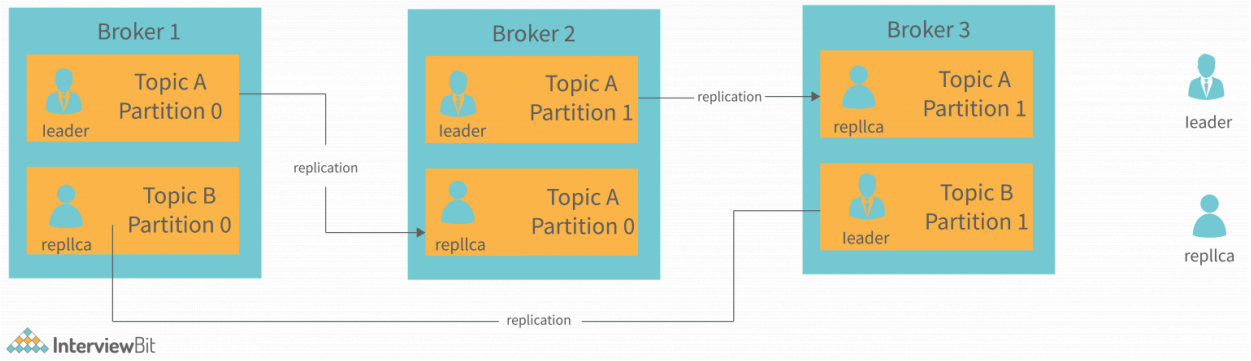
In such a case, the only way is to have multiple consumers working in parallel working on different set of events.
Kafka enables that through consumer groups.
A consumer group is a collection of consumer machines, which would consume from the same topic. Internally, every consumer in the consumer groups gets tagged to one or more partition exclusively (this means it's useless to have more consumer machines than the number of partition) and every consumer then only gets messages from the relevant partition. This helps process events from topics in parallel across various consumers in consumer group.



- **If one or more machines(broker) within Kafka die, how do I ensure I never lose events?**

Same solution as every other case. **Replicate**.
Kafka lets you configure how many replica you wish to have. Later, for every partition, primary replica and other replicas are assigned between machines/brokers.

Example: *See below image, when Kafka has 3 machines, 2 topics and each topic has 2 partitions. Replication configured to be 2.*



- **If I am a producer or a consumer, how do I know which Kafka machine to talk to?**

Kafka says it does not matter. Talk to any machine/broker in Kafka cluster and it will redirect to the right machines internally. Super simple.

- **If I am a consumer group, and I have already consumed all events in a topic, then when new events arrive, how do I ensure I only pull the new events in the topic?**

If you think about it, you would need to track an offset (how much have I already read) so that you can fetch only events post the offset.
More reading: https://dattell.com/data-architecture-blog/understanding-kafka-consumer-offset/

Elasticsearch

Agenda

1. Full-text search
2. Elastic Search
3. Sharding keys

Imagine you are working for LinkedIn. And you are supposed to build a search feature which helps you search through all matching posts on LinkedIn.

Why is SQL not a good choice for designing such a system?

In SQL, we would be constructing a B+ tree of all posts. But this approach works fine when the posts are small in size (single word or 2 words). For large text, you will have to go to every tree node, which is time-consuming.

Someone might search for “job posting” which means that you are looking for all posts that contain “job posting” or “job post” in their body. This would require you to write a query like the following:

```
SELECT * FROM posts WHERE content LIKE “%job post%”.
```

The above query will end up doing the entire table scan. It will go to every row and do a string search in every post content. If there are N rows and M characters, the time complexity is O(N*M^2). Not good. Will bring LinkedIn down.

NoSQL DB?

Using a NoSQL DB would have a similar issue. In key value store or column family store, you would have to go through every row and search for matching values/columns which would take forever. Same for document store.

Correct Data Structure that can be used to solve this problem:

The data structure that we can use to solve this problem is:
Hashmap Or Trie

If we use a hashmap, the key-value pair can be as follows. The key will be the word to be searched, and the value will be the list of IDs of the document (or the review) where the queried word is present.

This is called as **INVERSE INDEX.**

APACHE LUCENE performs a similar thing**.** Every entry (the entire post for example) is called as a document. The following are the steps performed in this:

1. **Character elimination**
   In this phase, we remove characters such as “a”, “an”, “the”, etc. Although the name is character elimination, but it also includes word elimination.
2. **Tokenization**
   The entire post is broken down into words.
3. **Token Index**
   All of the tokens are broken down into their root word.
   Consider the following sentences for example,
   → When I ran the app, the app crashed.
   → While running the app, the app crashes.
   Here the pair of words
   → "ran" and "running"
   → "crashed" and "crashes"
   Carry the same meaning but in a different form of sentence. So this is what reduction to root word means. This process is also called **stemming.**
   So the words "running" and "ran" are converted to the root word "run"
   The words "crashes" and "crashed" are converted to the root word "crash"
4. **Reverse Indexing**
   In this phase, we store the (document id, position) pair for each word.

For example,
If for document 5, the indexed words after 3rd phase look as follows
– decent - 1
– product - 2
– wrote - 3
– money - 4
Then in the reverse indexing phase, the word "decent" will be mapped to a list that will look as
[(5,1)]
Where each element of the list is a pair of the (document id, position id)

Full-Text Search

Use cases of full-text search:

1. Log processing
2. Index text input from the user
3. Index text files / documents (for example, resume indexing to search using resume text).
4. Site indexing

Elastic Search

Apache Lucene is great. But it's just a software built to run on one single machine.
Single machine could however:

- Become single point of failure
- Might run out of space to store all documents.
- Might not be able to handle a lot of traffic.

So, ElasticSearch was built on top of Lucene to help it scale.

Should ES be more available vs more consistent?
Most search systems like LinkedIn post search is not supposed to be strongly consistent. Hence, a system like ElasticSearch should prioritize high availability.

Terminologies:

- **Document:** An entity which has text to be indexed. For example, an entire LinkedIn post is a document.

- **Index:** An index is a collection of documents indexed. For example, LinkedIN posts could be one index. Whereas Resumes would be a different index.
- **Node:** A node refers to a physical / virtual machine.

Sharding:
How would you shard if there are so many documents that the entire thing does not fit in a single machine?

1. Elastic search shards by document id.
2. Given a lot of document_ids, a document is never split between shards, but it belongs to exactly one shard.
3. **Sharding algorithm:** ElasticSearch requires you to specify the number of shards desired at the time of setup. If number of shards is fixed or does not change often, then we can use something much simpler than consistent hashing:
    a. A document with document_id will be assigned to shard no. `hash(document_id)%number of shards.`

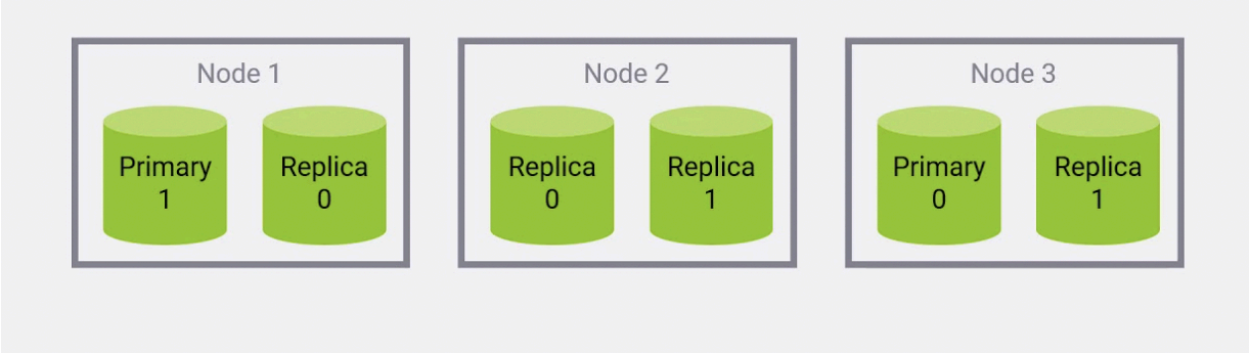
Replication in ElasticSearch:

Just like number of shards, you can also configure number of replicas at the time of setup.
You need replicas because:

- Machines die. Replicas ensure that even if machines die, the shard is still alive and data is not lost.
- More replicas help in sharing the load of reads. A read can go to any of the replicas.

Just like in master-slave model, one of the replicas in the shard is marked as primary/master and the remaining replicas are followers/slaves.

So, imagine if num_nodes = 3, num_shards = 2(0 and 1), num_replicas = 3, then it could look like the following:



Given there are less number of nodes, hence, multiple shards reside on the same node. You can reduce that by adding more nodes into the cluster. With more nodes in the cluster, you can also configure and control the number of shards per node. Further reading at https://sematext.com/blog/elasticsearch-shard-placement-control/

Read / write flow

Write (Index a new document): This finds the right shard for the document_id and the node containing primary replica. Request to index the document (just as writes happen in Lucene as detailed earlier) is sent to that node (primary replica). Updates from primary replica are propagated to slaves async.

Read (Given a phrase, find matching documents along with matching positions): Given documents are spread across shards, and any document can match, read in ElasticSearch is read in every shard.

When a read request is received by a node, that node is responsible for forwarding it to the nodes that hold the relevant shards, collating the responses, and responding to the client. We call that node the coordinating node for that request. The basic flow is as follows:

- Resolve the read requests to the relevant shards.
- Select an active copy of each relevant shard, from the shard replication group. This can be either the primary or a replica.
- Send shard level read requests to the selected copies.
- Combine the results and respond.

When a shard fails to respond to a read request, the coordinating node sends the request to another shard copy in the same replication group. Repeated failures can result in no available shard copies.

To ensure fast responses, some ElasticSearch APIs respond with partial results if one or more shards fail.

Additional Detailed Explanation for ElasticSearch

Elasticsearch is a highly scalable and distributed search and analytics engine built on top of Apache Lucene. It is designed to handle and process large volumes of data in near real-time and provide fast search capabilities.

High-level overview of how Elasticsearch works:

1. Data Storage: Elasticsearch stores data in a distributed manner across multiple nodes forming a cluster. Each node can hold a subset of the data and performs various operations independently.

2. Document Indexing: Data in Elasticsearch is organized into documents, which are JSON objects. Documents are grouped into indices, which are logical namespaces that define a collection of similar documents. Before data can be searched, it needs to be indexed. Indexing involves parsing the document, extracting the relevant fields, and storing them in inverted indices.

3. Inverted Indices: Elasticsearch uses inverted indices to enable efficient full-text search. Inverted indices store a list of terms along with their locations in the indexed documents. This allows Elasticsearch to quickly locate documents that match a given query term.

4. Distributed Architecture: Elasticsearch distributes data and queries across multiple nodes in a cluster. This distribution allows for horizontal scaling and improved performance. The cluster is responsible for automatically managing the allocation and replication of data across the nodes.

5. Sharding and Replication: Elasticsearch uses a technique called sharding to break down indices into smaller parts called shards. Each shard is an independent index that can be stored on a separate node. Sharding allows

Elasticsearch to distribute data and queries across multiple nodes, enabling parallel processing and improved performance. Additionally, Elasticsearch supports replication, where each shard has one or more replicas. Replicas provide redundancy and high availability.

6. Querying: Elasticsearch provides a powerful query DSL (Domain-Specific Language) that allows you to construct complex queries for searching and filtering data. Queries can be executed across one or multiple indices, and Elasticsearch uses its distributed nature to parallelize and optimize query execution.

7. Near Real-Time Search: Elasticsearch offers near real-time search capabilities, which means that data is indexed and made searchable within a short period after being ingested. This makes Elasticsearch suitable for applications where data needs to be available for search immediately after ingestion.

8. RESTful API: Elasticsearch exposes a RESTful API that allows you to interact with the cluster. You can perform operations such as indexing documents, searching, aggregating data, updating, and deleting documents using HTTP requests.

9. Scalability and Resiliency: Elasticsearch is designed to scale horizontally by adding more nodes to the cluster. It automatically distributes data across nodes and balances the load. It also provides resiliency by replicating data, so even if a node fails, the data is still available from other nodes.

Overall, Elasticsearch's distributed and scalable architecture, combined with its powerful search and analytics capabilities, make it a popular choice for various use cases, including log analysis, application monitoring, e-commerce search, and more.

Sharding details

In Elasticsearch, sharding is the process of dividing an index into smaller parts called shards. Each shard is an independent index that can be stored on a separate node within a cluster. Sharding is a crucial aspect of Elasticsearch's distributed architecture, enabling horizontal scalability, parallel processing, and improved performance.

Detailed explanation of how sharding works in Elasticsearch:

1. Shards and Indices

a. An index in Elasticsearch is a logical namespace that contains a collection of documents sharing similar characteristics.
b. When you create an index, you specify the number of primary shards it should have. The default is five primary shards.
c. Each shard is a self-contained index with its inverted index, storing a subset of the data for the entire index.
d. The total number of shards across all nodes determines the maximum amount of data that an Elasticsearch cluster can hold.

2. Sharding Process:
    a. When a document is indexed, Elasticsearch determines which shard it should be stored in using a hashing algorithm.
    b. Elasticsearch calculates a shard ID for each document based on a sharding key, which is typically the document's ID or a specific field value.
    c. The shard ID is a numeric value that determines the shard to which the document will be assigned.
    d. Elasticsearch uses a consistent hashing algorithm to distribute the documents evenly across the available shards.
    e. The hashing algorithm generates a hash value from the shard ID, and then the shard assignment is determined based on this value and the total number of shards.

3. Shard Allocation:
    a. Elasticsearch automatically manages the allocation of shards across nodes in the cluster.
    b. When a new node joins the cluster, it receives a copy of some existing shards to balance the data distribution.
    c. Elasticsearch uses a shard allocation algorithm to determine the best location for each shard, considering factors like node availability, shard size, and data rebalancing.

4. Primary and Replica Shards:
    a. Each shard can have one or more replica shards, which are copies of the primary shard.
    b. Replicas provide redundancy and high availability in case of node failures or network issues.
    c. Replicas are distributed across different nodes to ensure that if one node fails, the data is still accessible from other nodes.

    d. The number of replica shards can be configured when creating an index, and the default is one replica.

5. Benefits of Sharding:
    a. Horizontal Scalability: Sharding allows Elasticsearch to distribute the data and workload across multiple nodes, enabling the cluster to handle large volumes of data and scale horizontally by adding more nodes.
    b. Parallel Processing: Shards can be processed in parallel, allowing Elasticsearch to handle concurrent queries and operations efficiently.
    c. Improved Performance: By distributing data and queries across multiple shards, Elasticsearch can distribute the computational load and reduce response times.
    d. It's important to note that the sharding key used in Elasticsearch is typically the document's ID or a specific field value. The choice of sharding key depends on your data and access patterns. If you have a natural unique identifier for your documents, such as a product SKU or a user ID, using that as the sharding key can lead to an even distribution of data across shards. However, it's essential to consider potential data skew and access patterns to avoid hotspots where a particular shard receives a disproportionate amount of requests.

By using sharding, Elasticsearch provides a scalable and distributed architecture that can handle large amounts of data, support parallel processing, and ensure high availability and fault tolerance.

Querying

Elasticsearch's search queries can be multi-shard queries. When you execute a search query in Elasticsearch, it automatically routes the query to the relevant shards that contain the indexed data. This process is transparent to the user, and Elasticsearch handles it internally.

When a query is executed across multiple shards, Elasticsearch parallelizes the search operation, distributing the workload across the shards. Each shard performs the search operation independently on its local data and returns the results to the coordinating node.

The coordinating node receives the results from all the shards, merges them, and returns a unified response to the user. The merging process includes sorting, aggregating, and ranking the results as specified in the query.

By parallelizing the search operation across multiple shards, Elasticsearch can achieve faster response times and handle larger datasets. This distributed approach allows Elasticsearch to scale horizontally by adding more nodes to the cluster and balancing the search workload across them.

It's worth noting that while multi-shard queries can provide improved performance, they may also introduce additional complexity and network overhead, especially when the query involves aggregations or sorting across a large number of shards. Therefore, it's important to design your Elasticsearch cluster and index settings appropriately, considering factors such as shard size, query complexity, and data distribution, to ensure efficient search operations.

When executing a search query, Elasticsearch determines the relevant shards for the query through a process called query routing. It uses the following steps to determine the shards to which the query should be sent:

1. Shard Allocation:
    a. Elasticsearch automatically distributes shards across the nodes in the cluster during shard allocation.
    b. Each shard is assigned to a specific node, and the node becomes the primary holder of that shard.
    c. The allocation process considers factors such as node availability, shard size, and data rebalancing to ensure an even distribution of shards across the cluster.

2. Shard Mapping:
    a. Elasticsearch maintains a cluster state, which includes the mapping of shards to nodes.
    b. This mapping information is used to route queries to the appropriate shards.
    c. The cluster state is continuously updated as nodes join or leave the cluster, or as shards are allocated or moved.

3. Shard Metadata:

a. Each shard contains metadata that describes its index and shard ID, including information about the range of document IDs or values it holds.
b. This metadata allows Elasticsearch to quickly identify the relevant shards for a given search query.

4. Query Parsing:
    a. When a search query is received, Elasticsearch parses the query to understand its components, such as the search terms, filters, aggregations, sorting criteria, and other parameters.
    b. Based on the query structure, Elasticsearch determines the necessary shards to be involved in the search operation.

5. Shard Filtering:
    a. Elasticsearch uses the shard metadata and the parsed query to filter out shards that are irrelevant to the query.
    b. For example, if the query specifies a filter on a specific field value, Elasticsearch can exclude shards that don't contain documents matching that value.

6. Query Routing:
    a. After filtering the relevant shards, Elasticsearch routes the search query to those shards for execution.
    b. The query is sent to each relevant shard independently, and each shard processes the query on its local data.

7. Coordinating Node:
    a. The coordinating node receives the results from all the shards involved in the query execution.
    b. It merges the results, performs sorting, aggregations, and other post-processing steps specified in the query, and returns the final result set to the user.

By following these steps, Elasticsearch efficiently determines the shards that hold the relevant data for a given search query. This process ensures that the query is executed in parallel across the appropriate shards, allowing for scalable and distributed search operations.

MongoDB vs Elasticsearch

MongoDB and Elasticsearch have different strengths and use cases, but MongoDB does offer some features that can provide search functionality similar to Elasticsearch.

MongoDB has a built-in text search feature called "text indexes." With text indexes, you can perform full-text search queries on specific fields or across multiple fields within your documents. MongoDB's text search supports various query operators, such as exact phrase matching, fuzzy matching, and language-specific stemming.

While MongoDB's text search is capable of handling basic search requirements, it may not provide the same level of advanced search capabilities, relevance scoring, and scalability as Elasticsearch. Elasticsearch is designed specifically for search and excels at handling large volumes of data and complex search queries.

If your search requirements are relatively simple and your application already uses MongoDB as the primary database, you can leverage MongoDB's text search capabilities to provide basic search functionality. However, if search is a critical component of your application and you require more advanced search features, scalability, and performance, Elasticsearch is generally a more suitable choice.

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.

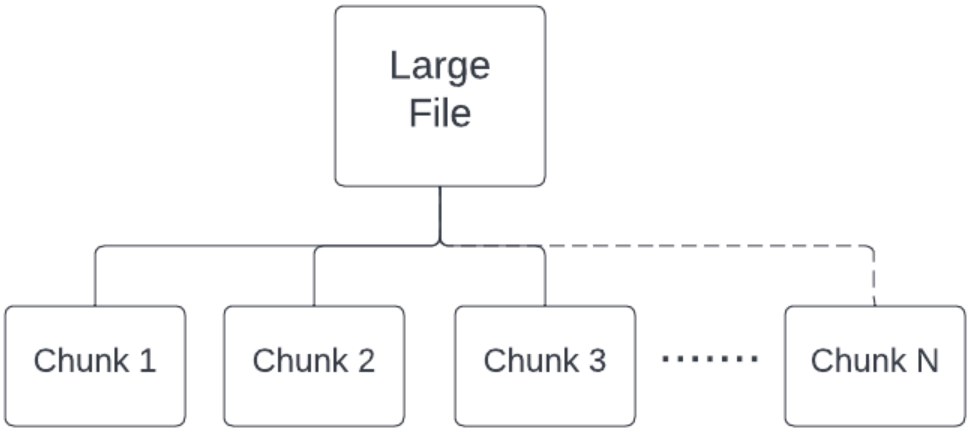
We discussed that for a post made by the user on Facebook with images(or some other media), we don't store the image in the database. We only store the metadata for the post (user_id, post_id, timestamp, etc.). The images/media are stored on different storage systems; from that particular storage system,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

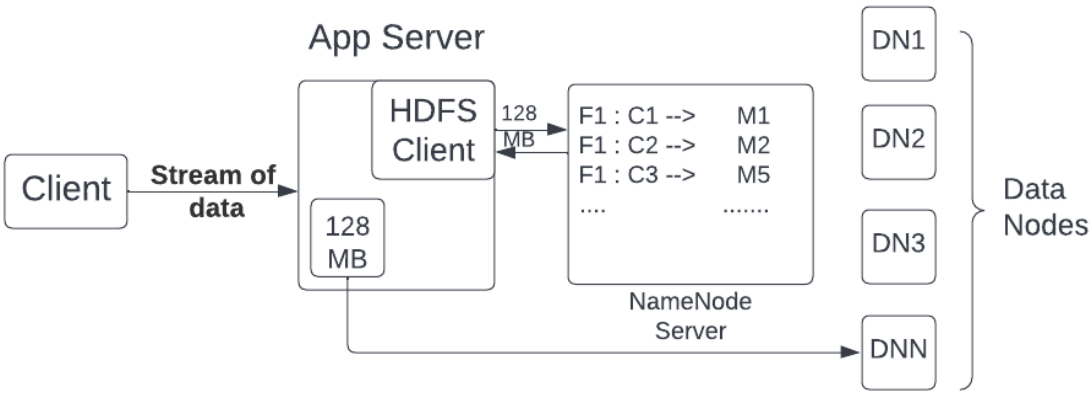
Making System reliable: We know that to make the distributed system reliable, we never store data on a single machine; we replicate it. Here also, a chunk cannot be stored on a single machine to make the system reliable. It needs to be saved on multiple machines. We will keep chunks on different data 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 NamNode 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)**.

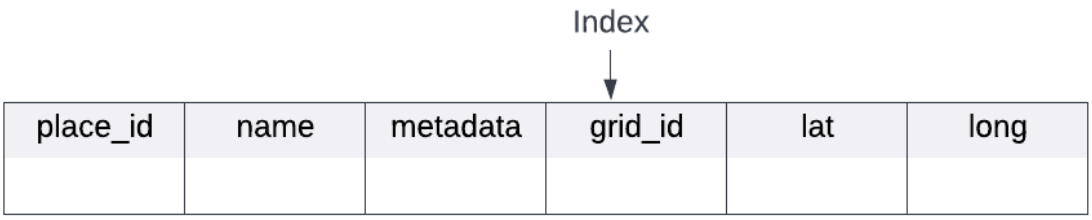
However, this approach has some issues:

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

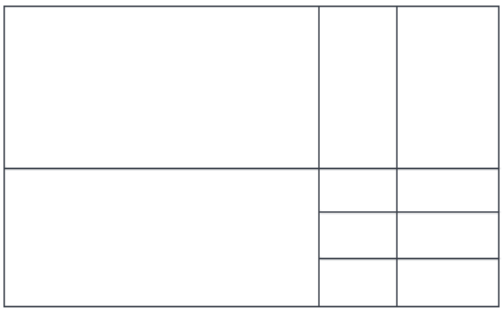
We can break the entire world into small grids (maybe 1 km sq. grids). Then to get all the points, we only need to consider the locations in the grid of our current location or the points in the adjacent grids. If there are enough points in these grids, then we can get all the nearest neighbors. The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

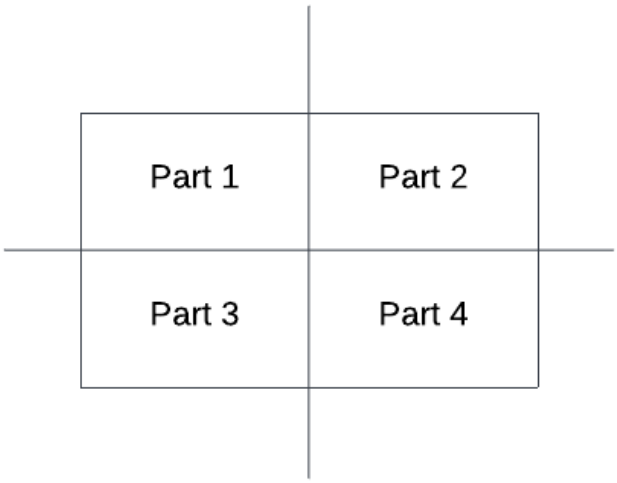
This can be achieved using **quadtrees**.

QuadTree

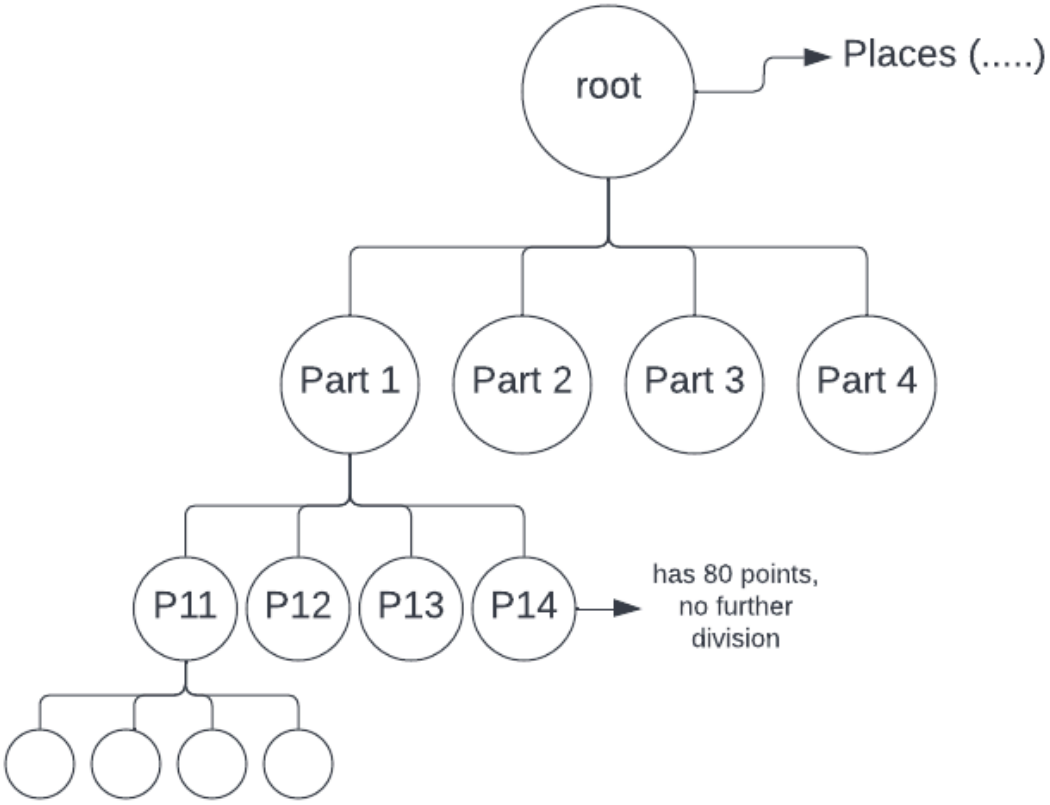
Creation

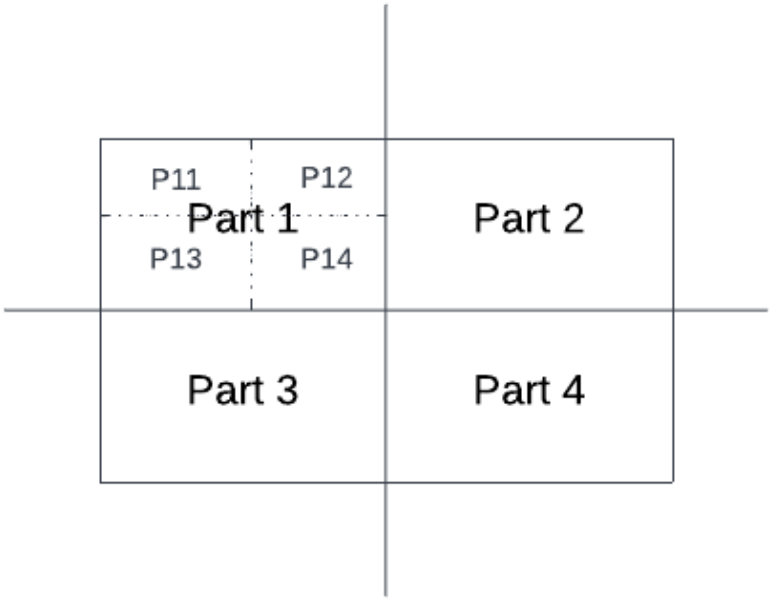
Imagine the entire world with billions of points (think of a world map, a rectangle with points all over).

- 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.

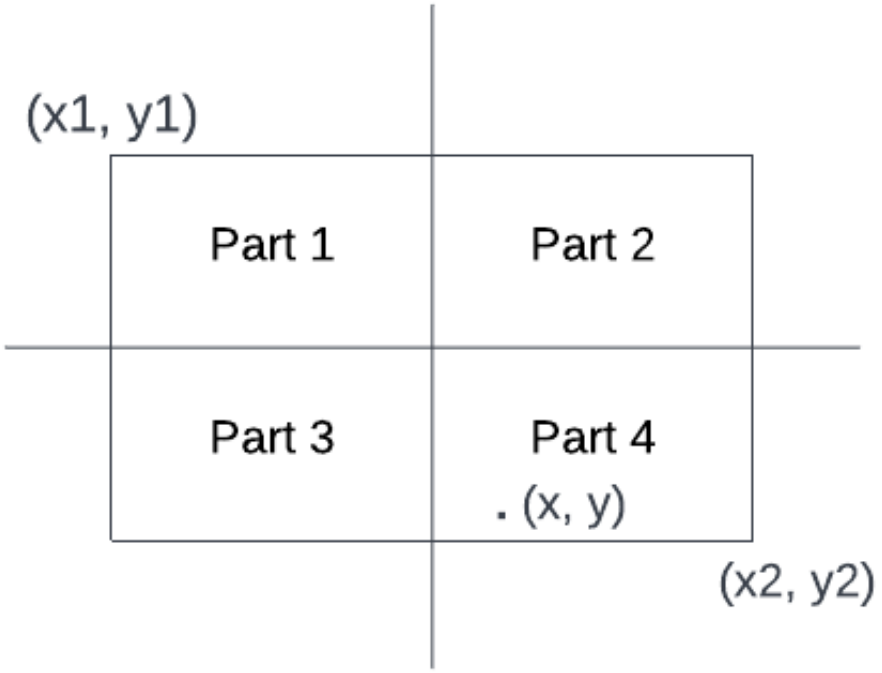

root
Places (.....)
Part 1
Part 2
Part 3
Part 4
P11
P12
P13
P14
has 80 points,
no further
division


P11
P12
Part 1
P13
P14
Part 2
Part 3
Part 4

- 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
```

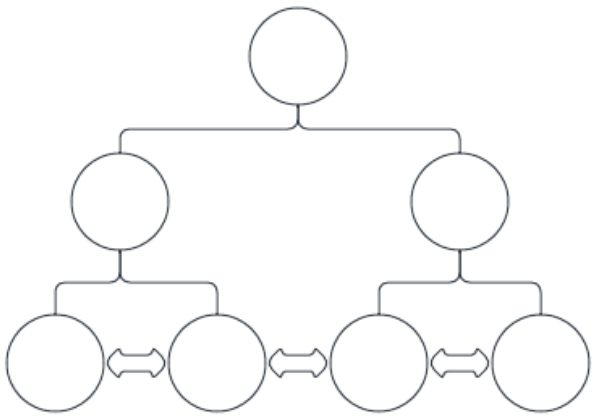
```
If x > (X1 + X2) / 2:
        If y > (Y1 + Y2) / 2:
                findgrid(x, y, root.children[1])
        Else:
                findgrid(x, y, root.children[3])

 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

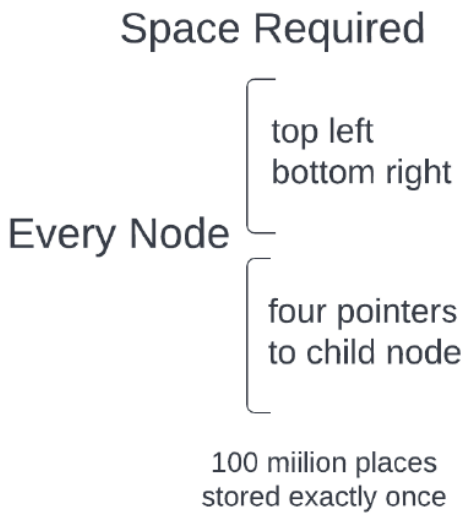
Deletion is exactly the opposite of addition. If I delete a point and the summation of all the points in the four children becomes less than 100, then I can delete the children and go back to the parent. However, the deletion part is not that common.

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.

System Design - Case Study 5 (Design Uber)

Prerequisite: *System Design - S3 + Quad trees (nearest neighbors)*

We discussed nearest neighbors and quadtrees in the last class and left our discussion with a problem statements:

1. How to design for Uber? (earlier nearest-neighbor problem worked well with static neighbors, here taxis/cabs can move)

Uber: Case Study

We are building uber for intracity, not intercity.

Uber has users and cabs. Users can book a cab, and a cab has two states, either available for hire or unavailable. Only the available cabs will be considered when a user is trying to book.

Use case: I am a user at location X(latitude, longitude); match me with the nearest available cabs. (The notification goes to all nearby cabs for a rider in a round-robin fashion, and a driver can accept or reject). How will you design this?

Let's start our discussion with a question: Suppose there are 10 million cabs worldwide, and I am standing in Mumbai then do I need to care about all the cabs? What is the best sharding key for uber?

You might have guessed that the best sharding key is the city of the cab it belongs to. Cities can be a wider boundary (including more than one region). Every city is a hub and independent from the others. By sharding cabs by city, the scope of the problem will be reduced to a few hundred thousand cabs. Even the busiest cities in India will have around 50 thousand cabs.

Now suppose I am in Mumbai and requesting a cab. If the whole Mumbai region is already broken into grids of equal size, then that would not work well for finding the nearest cabs.
In the Mumbai region, only some areas can be dense (high traffic) while others have less traffic. So breaking Mumbai into equal size grids is not optimal.

- If I am in a heavy traffic area, then I want to avoid matching with cabs that are far(5km) away, but it's fine for sparse traffic areas because cabs will be available very soon.
- So we can say a uniform grid approach of quadtree will not work because different areas have different traffic. It's a function of time and location.

So what can be other approaches?

We can use the fact that cabs can ping their current location after every fixed time period (60 seconds), cab_id → (x,y), and we can maintain this mapping.

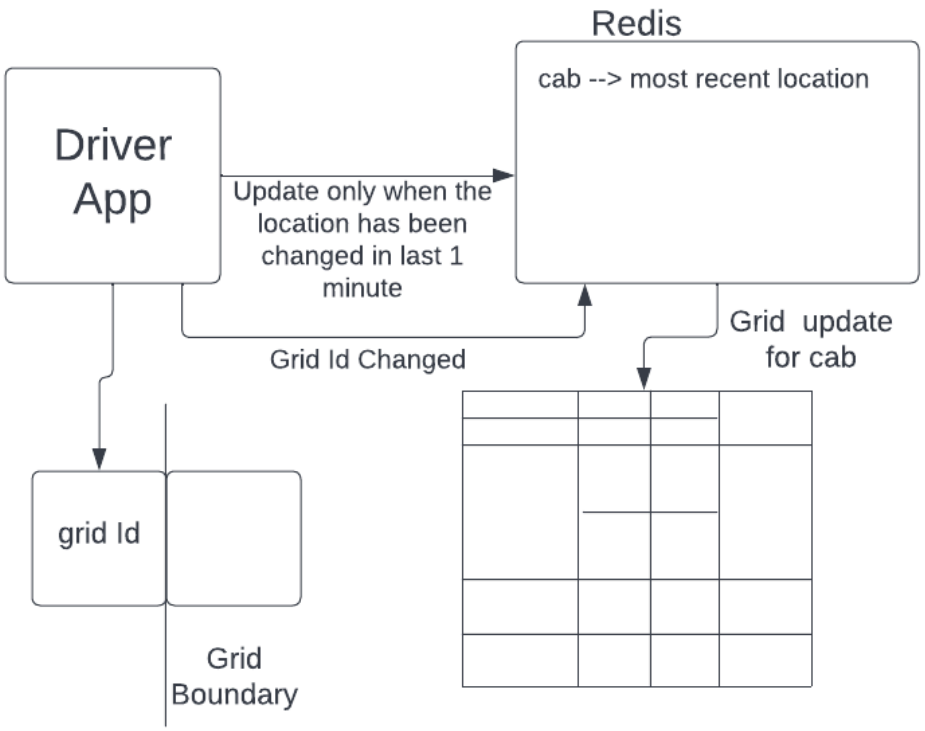
Bruteforce: We go through all the drivers/cabs in the city (since there will only be a few thousand) and then match with the nearest ones.

For an optimized approach, consider this: *If cabs are moving all the time, do we need to update the quadtree every time?*

Initially, we created a quadtree on the traffic pattern with some large and some small grids. When cabs move and change their grids (we get notified by the current location), then we have to update the quadtree (the previous grid and the new grid). There will be a lot of write queries for the update. *How can we optimize it?*

Note: Cabs will only send updates when their location has been changed at the last 1 minute. We can track this on the client side only.

Optimizations:
The driver app tells when the location changes and is aware of grid boundaries.
The optimization can be: cabs already knew the grid they are part of and the boundary. If the cab is moving within the boundary, then there is no need to update the quadtree. When the cab goes into a new grid, the cab_id and corresponding grid can be updated at the backend. We know we can maintain a memory map of cabs and the most recent know locations inside Redis.



We can modify the quadtree by using the above knowledge. However, one problem can occur if we start creating the grids instantaneously: If the number of cabs in a grid is fixed (say 50) and we create four children or merge four children if cabs move in/out of the grid. This will become an expensive operation and will cause more writes. We can also optimize the creation and merging of grids by not changing the grid dimensions instantly.

For example, the threshold for a cab is 50, and the moment we have more the 50 cabs in the grid, we split it into four parts (this was the usual rule).

- What we can do is, instead of fixing the threshold, we can have a range (like 50 to 75). If cabs exceed the range, then we can change the dimensions.

- Another approach could be running a job after every fixed time (3 hours or more) and checking for the number of cabs in leaf nodes. And we can consolidate or split the nodes accordingly. This would work because the traffic pattern doesn't change quickly.

To conclude the design of uber, we can say we need to follow the following steps:

Step 1: For a city, create a quadtree based on current cab locations. (At t = 0, dynamic grids will be created based on the density of cabs).
Step 2: Maintain **cab → (last known location)** and **quadtree** backend for nearest cabs.
Step 3: Don’t bombard quadtree and cab location updates. You can use optimizations:

- Driver app only sends location if cab location is changed in last 1 minute, handling this at client side only
- While sending a new location driver app also checks whether the grid has changed by checking the boundary. For a grid with boundaries (a,b) and (c,d), we can check location (x,y) is inside or not simply by a<=x<=c && d<=y<=b.
- If the grid Id changes, then delete it from the old grid and update it in the new grid.
- Another optimization is, in a quadtree, don't update grids' dimensions immediately. Instead, do it periodically.

Popular Interview Questions

1. Design a Unique ID generator
2. Design a Rate Limiter

Design a Unique ID generator

In this problem, we are required to generate IDs (numeric values - 8 bytes long) that satisfy the following criteria:

1. The ID should be unique.
2. The value of the ID must be incremental.

FAQ: What does incremental mean?

Let us suppose two IDs (numeric values), i1 and i2, are generated at time t1 and t2, respectively. So incremental means here that if t1 < t2, it should imply that i1 < i2.

Ques: What options do we have to construct these IDs?

1. Auto increment in SQL
2. UUID
3. Timestamp
4. Multi-master.
5. Timestamp + Server ID
6. Multi-Master

Ques1: Why can't we directly use IDs as 1,2,3,... i.e., the auto-increment feature in SQL to generate the IDs?

Sequential IDs work well on local systems but not in distributed systems. Two different systems might assign the same ID to two different requests in distributed systems. So the uniqueness property will be violated here.

Ques2: Why can't we use UUID?

We can not use UUID here because

1. UUID is random
2. UUID is not numeric.
3. The property of being incremental is not followed, although the values are unique.

Ques3: Why can't we use timestamp values to generate the IDs?

The reason is, again, the failure to assign unique ID values in **distributed systems**. It may be possible that two requests land on two different systems at the same timestamp. So if the timestamp parameter is utilized, both will be given the same ID based on epoch values (the time elapsed between the current timestamp and a pre-decided given timestamp)

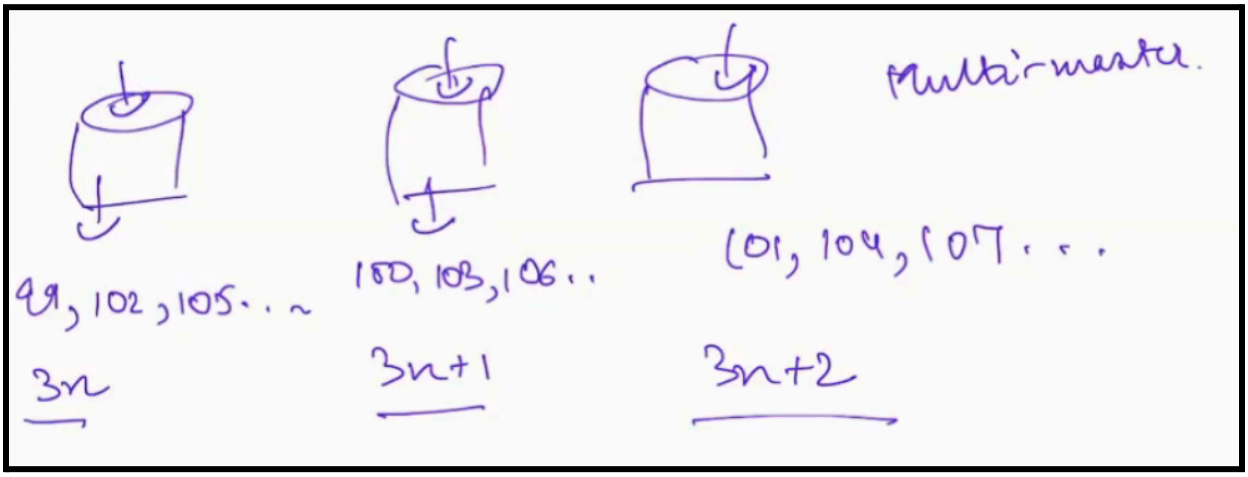
Next Approach: **Timestamp + ServerID**

The question here would be how many bits are to be assigned to the Timestamp and serverID. This situation would be tricky.

Next Approach: **Multi-Master**

Let us assume there are three different machines in a distributed computing environment. Let us number the machines as **M1**, **M2**, and **M3**. Now suppose the machine M1 is set to assign the IDs, which are a **multiple of 3n**, machine M2 is set to assign IDs that are a **multiple of 3n+1**, and machine M3 is set to assign IDs that are a **multiple of 3n+2**.

This situation can be represented diagrammatically as follows:

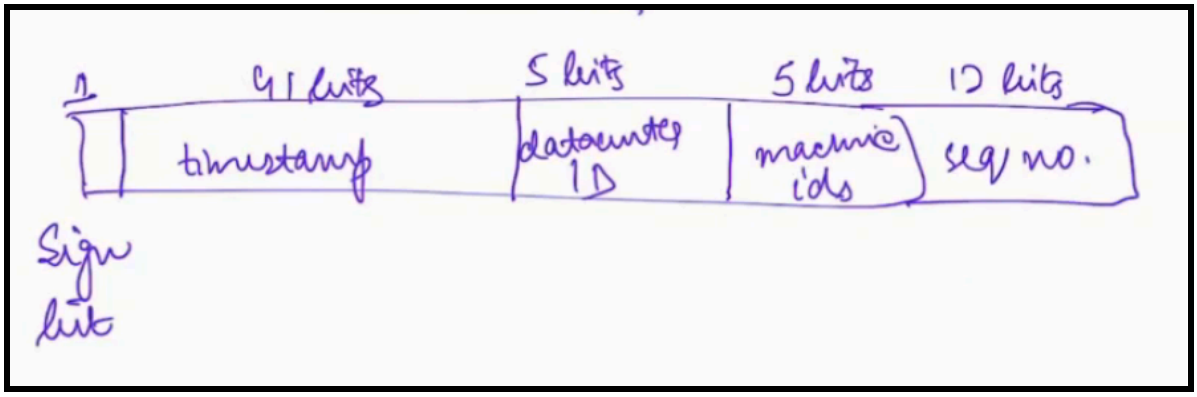


If we assign IDs using this technique, the uniqueness problem will be solved. But it might violate the incremental property of IDs in a distributed system.

For example, let us suppose that request keeps coming consecutively on M1. In this case, it would assign the IDs as 99,102,105,... . Now, after some time, the request comes on some other machine (e.g., M2), then the ID assigned would have a numeric value lower than what was assigned previously.

Twitter Snowflakes Algorithm:

In this algorithm, we have to design a 64-bit solution. The structure of the 64 bits looks as follows.



64-bit solution generated by Twitter Snowflakes Algorithm

Number of Bits (from left to right)	Purpose for which they are reserved
1 bit	Sign
41 bits	Timestamp
5 bits	Data center ID
5 bits	Machine ID
12 bits	Sequence Number

Let us talk about each of the bits one by one in detail:

1. Sign Bit:
The sign bit is never used. Its value is always zero. This bit is just taken as a backup that will be used at some time in the future.

2. Timestamp bits:
This time is the **epoch time**. Previously the benchmark time for calculating the time elapsed was from **1st Jan 1970**. But Twitter changed this benchmark time to **4th November 2010**.

3. Data center bits:
5 bits are reserved for this, which implies that there can be 32 (2^5) data centers.

4. Machine ID bits:

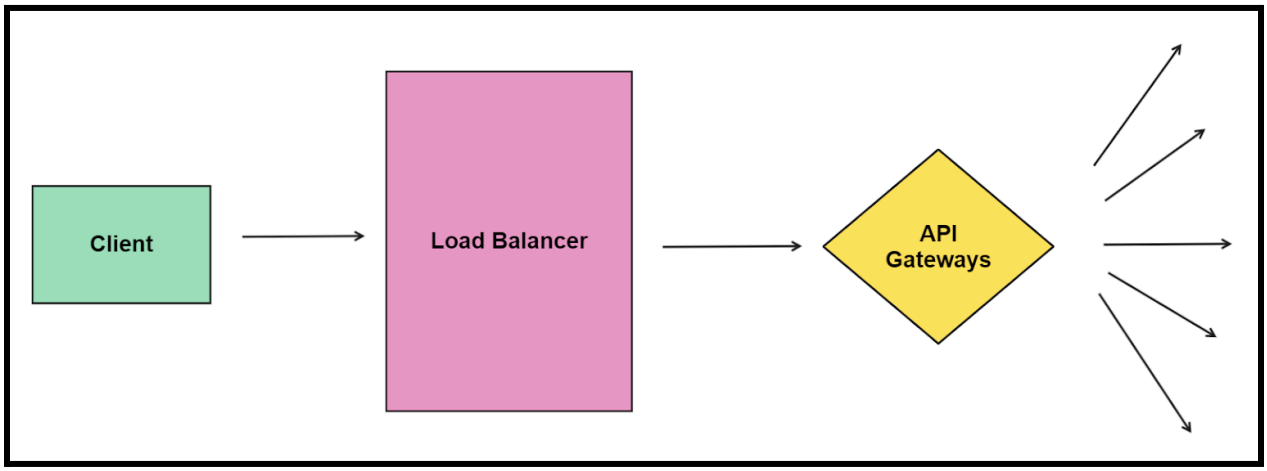
5 bits are reserved for this, which implies that there can be 32 (2^5) machines per data center.

5. Sequence no bits:
These bits are reserved for generating sequence numbers for IDs that are generated at the same timestamp. The sequence number is reset to zero every millisecond. Since we have reserved 12 bits for this, we can have 4096 (2^12) sequence numbers which are certainly more than the IDs that are generated every millisecond by a machine.

Further reading (optional): https://en.wikipedia.org/wiki/Network_Time_Protocol

Design a Rate Limiter

Rate limiter controls the rate of the traffic sent from the client to the server. It helps prevent potential DDoS attacks.

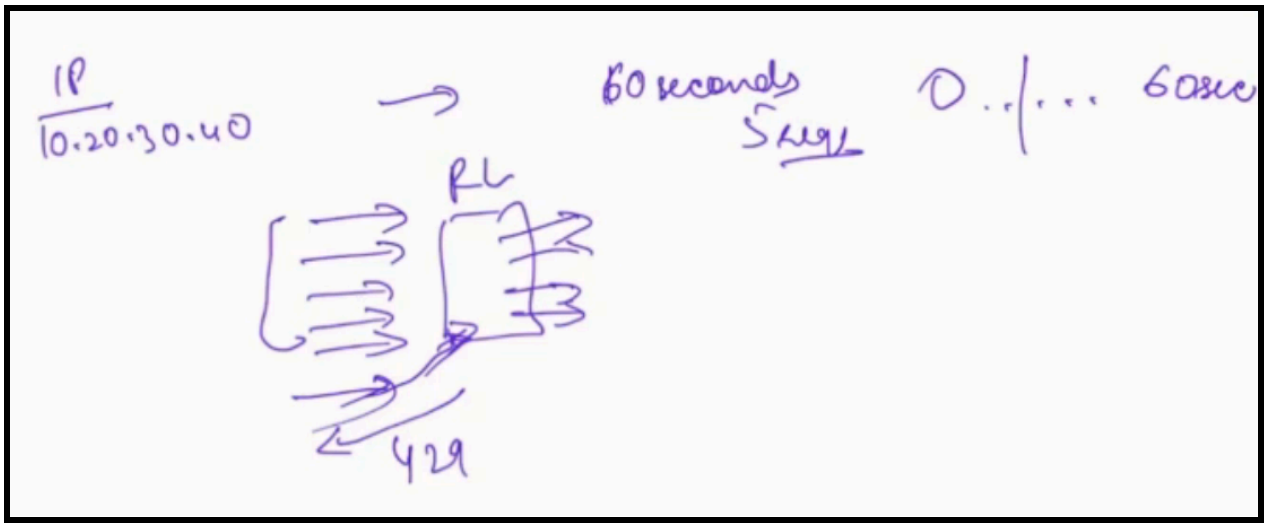


Throttling: It is the process of controlling the usage of the APIs by customers during a given period.

Types of Rate Limiter:

1. Client Side Rate Limiter
2. Server Side Rate Limiter

Let us consider an example. Suppose a user with IP address **10.20.30.40** sends requests to a server where a rate limiter is installed, and a rate of a maximum of 5 requests has been set every 60 seconds. In this case, if the user sends more than five requests in 60 seconds, its 6th request will be rejected, and an error code of 429 will be sent to him. The user can send this request in the next 60 seconds time frame. The situation is diagrammatically represented as follows:



Algorithms for rate limiting

1. Token Bucket Algorithm
2. Leaking Bucket Algorithm
3. Fixed Window Counter
4. Sliding Window Counter

Rate limiting algorithms are used to control the rate at which a system or application can handle incoming requests or data. They are essential for managing resources and preventing overload or abuse. Here's a detailed explanation of each of the rate limiting algorithms you mentioned:

Token Bucket Algorithm

The Token Bucket Algorithm is a widely used rate limiting mechanism. It is based on the concept of a token bucket, which holds a fixed number of tokens. Tokens are continuously added to the bucket at a predetermined rate, and each incoming request or data unit requires a certain number of tokens to be processed.
Here's how the Token Bucket Algorithm works:

The token bucket has a maximum capacity (maximum number of tokens it can hold) and a token refill rate (tokens added per unit of time).
At regular intervals (e.g., every second), a certain number of tokens (defined by the refill rate) are added to the bucket until it reaches its maximum capacity.

When a request or data unit arrives, the algorithm checks if there are enough tokens in the bucket to process it. If there are enough tokens, the request is allowed, and the required number of tokens is removed from the bucket. If there aren't enough tokens, the request is delayed or rejected.
The token bucket can have a burst mode where the number of tokens in the bucket can briefly exceed the maximum capacity. However, sustained requests beyond the bucket's capacity will eventually be limited.
The Token Bucket Algorithm provides a smooth and controlled rate limiting mechanism, as it allows for bursts of requests while still maintaining the overall rate.

Leaking Bucket Algorithm

The Leaking Bucket Algorithm is another rate limiting method that uses a similar concept to the Token Bucket Algorithm. However, in this case, the bucket "leaks" tokens at a constant rate, regardless of whether there are requests arriving.
Here's how the Leaking Bucket Algorithm works:

The leaking bucket has a fixed capacity and a constant leak rate (tokens removed per unit of time).
Whenever a request arrives, the algorithm checks if there are enough tokens in the bucket to process it. If there are enough tokens, the request is allowed, and the required number of tokens is removed from the bucket. If there aren't enough tokens, the request is delayed or rejected.
Unlike the Token Bucket Algorithm, where tokens are added at regular intervals, the Leaking Bucket continuously removes tokens at the specified leak rate, even if there are no incoming requests.
The Leaking Bucket Algorithm is effective at maintaining a steady output rate, even if requests arrive in bursts.

Fixed Window Counter

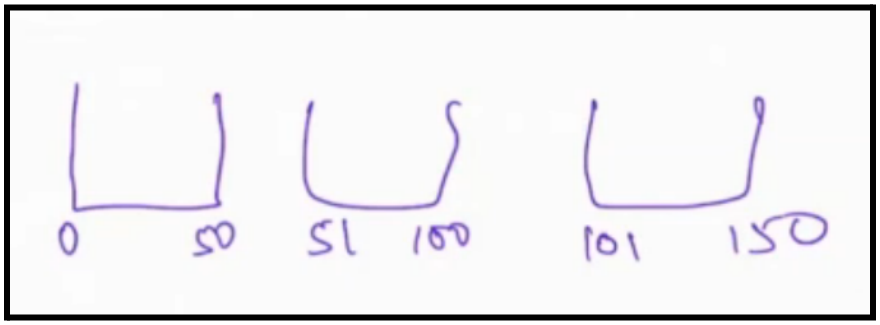
The Fixed Window Counter is a simple rate limiting algorithm that tracks the number of requests or data units received within fixed time windows.
Here's how the Fixed Window Counter works:

The rate limiter divides time into fixed intervals (e.g., 1 minute, 1 hour, etc.).
Each time window has a maximum allowed number of requests or data units that can be processed.

When a request arrives, the algorithm checks if the current time window is full. If it is not full, the request is allowed, and the counter for that window is incremented. If the window is already full, the request is delayed or rejected.
The Fixed Window Counter can lead to uneven request distribution if requests arrive in bursts near the beginning of each time window.

Suppose the threshold is set to 100 requests per 50 seconds.

We will maintain buckets of 50 seconds like this.



Now consider the following scenario. Suppose we receive 100 requests in the interval 40-50 seconds and again 100 requests in the interval 51-60 seconds. So this would lead to 200 requests in 40-60 seconds, which is **twice the threshold** value we have set. This is a common challenge that is faced when using this solution.

Sliding Window Counter

The Sliding Window Counter is an improvement over the Fixed Window Counter as it provides a more dynamic approach to rate limiting.
Here's how the Sliding Window Counter works:

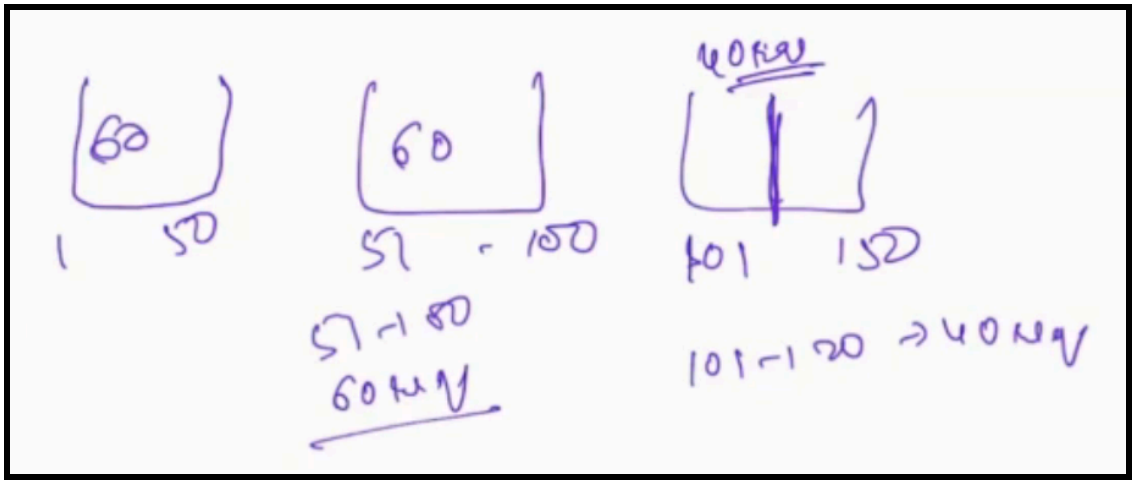
Like the Fixed Window Counter, time is divided into intervals, but these intervals form a sliding window that moves over time.
Each interval within the sliding window has a maximum allowed number of requests or data units that can be processed.
As time progresses, intervals outside the sliding window are discarded, and new intervals are added at the end of the window.
When a request arrives, the algorithm checks if the current interval is full. If it is not full, the request is allowed, and the counter for that interval is incremented. If the interval is already full, the request is delayed or rejected.

The Sliding Window Counter provides a more even distribution of requests and can handle bursts more effectively than the Fixed Window Counter.

Sliding Window is an approximation solution.

Let us suppose that we have again divided the timeline into buckets of size 50, i.e., the first interval is from t = 1 to 50, the second interval is from 51- 100, and the next is from 101 - 150, and so on.

Suppose we receive 60 requests in the interval t = 51 to 100 and 40 requests in the first half of the third interval ranging from t = 101 to 120.



Now we will use an approximation as follows.

Calculate what percentage of the interval (101,150) does the sub-range (101,120) constitute?

- Length of (101,120) is 20
- Length of (101,150) is 50
- So the percentage coverage would be (20/50)*100% = **40%**

So we have data for the 40% interval. Now to obtain data for the remaining 60% interval, we will use the last interval data and estimate it.

Thus 60% of the previous_count = 60 % of 60 = 36 requests.
The current_count is 40 requests.

Thus total count approximation for this interval is 36 + 40 = 76

In summary, rate limiting algorithms are crucial for maintaining system stability and ensuring fair resource allocation. Each algorithm mentioned above has its advantages and use cases, and the choice of which one to implement depends on the specific requirements and characteristics of the system being rate-limited.

Case Study - System Design of Hotstar or any Streaming Service in general

Frequently Asked Interview Questions

Q1: Given that different clients have different internet speeds, how do you ensure a good experience for all?

Answer: If clients have different speeds, and they all had to download the same heavy video, then obviously the slower clients will have no option but to keep buffering till they have some part of the video to show. In general, the rate at which video plays is always going to be higher than the rate at which those bits can be downloaded and hence, the client for slow internet will lag.

So, what do we do? Can we play with the resolution of the video? A smaller resolution video would be low on quality, but would require much less number of bits to be downloaded to show the same scenes. That way, even the slower connections can show video without lag.

You would however want to give the best resolution possible at the speed of the internet available. For example, if you know 1080p cannot be loaded at the current speed, but 720p can be loaded, then you would not want to degrade the experience by loading 240p. Note that the internet speed can be changing as well. So, you'd need to do detection continuously and adapt to the best resolution possible at the current internet speed available.

Most OTTs do exactly the above by using **Adaptive bitrate streaming (ABS)**. ABS works by detecting the user's device and internet connection speed and adjusting the video quality and bit rate accordingly.

The Adaptive Bitrate Streaming (ABS) technology of Hotstar detects Internet speed by sending data packets to the user's device and measuring the time it takes to respond. For example, if the response time is low, the internet connection speed is good; hence, Hotstar servers can stream a higher video quality. On the other hand, if the response time is high, it indicates a slow internet connection; hence, Hotstar servers can stream a lower video quality. This way, Hotstar ensures that all its viewers have an uninterrupted streaming experience, regardless of device or internet speed.

But is the internet speed alone enough? For example, if I am on a good internet connection but on mobile, does it make sense to stream in 1080p resolution? At such a small screen size, I might not even be able to decipher the difference between 480p, 720p and 1080p resolution. And if so, streaming 1 080p is bad for my data packs :P

Hotstar can detect a user's client device to provide an optimal streaming experience. Hotstar uses various methods, such as user-agent detection and device fingerprinting.

User-agent detection involves analyzing the string sent by a user's browser when they visit a website. This string contains information about the browser version and operating system, which Hotstar can use to identify the device type.

Device fingerprinting works by analyzing specific parameters of a user's device, such as screen size, plugins installed, and time zone settings. Then Hotstar uses this data to create a unique "fingerprint" for each user's device to identify their device type.

Using these two methods, Hotstar is able to accurately identify the user's device and provide an optimal streaming experience. This ensures that users are able to enjoy uninterrupted viewing, no matter which device they are using.

Q2: For non-live/recorded content, what optimizations would you do on the client to ensure smoother loading?

Observation #1: It does not make sense to load the entire file on the client. If I decide to view 5 movies, watch their first 5 mins and end up closing them, I would end up spending a lot of time and data downloading 5 entire movies.

So, we do what Torrent does. We break the file into small chunks (remember HDFS?). This way, a client can only request for a specific chunk (imagine 10:00 - 11:00 min chunk).

Observation #2: But how does the client know which chunk to request? Or if you notice, as you try to scroll to the future/past timestamp on the video, it shows a preview. How would it do that? When you first click on a video, the metadata of these chunks (timestamp boundaries, chunk_id, a low res thumbnail per chunk) can be brought to the client (and the client caches it). This could enable the jumping from one timestamp to another.

Observation #3: Ok, but what happens when I am done with the current chunk? If I would have to wait for the next chunk to be downloaded, then it would lead to a buffering screen at the end of every chunk. That's not a good experience, is it? How do we solve that?
What if we pre-load the next chunk (or next multiple chunks) as we near the end of the current chunk. For example, if the current chunk is 60 seconds, then we might start to preload the next chunk when I am at 30 seconds / 40 seconds. It happens in the background, and hence you'd never see buffering.

Obviously, I cannot cache too many chunks as it takes up my RAM space. So, I keep evicting from cache chunks I have seen in the past (or just simple LRU on chunks).

Note that the chunk downloaded is of the right bitrate, depending on the adaptive bit rate we talked about above. So, it is possible I download chunk 1 of very high quality, but if my internet speed has degraded, the next downloaded chunk is of lower quality.

Chunks make it easier to upload files (easier to retry on failure, easier to parallelise uploads) and make it easier to download (2 chunks can be downloaded in parallel).

And obviously, it's better to fetch these chunks from CDN for the right resolution of the file instead of directly from S3/HDFS.

More to think about here(Homework): Is there a better way of creating chunks other than just timestamps? For example, think of the initial cast introduction scenes which most people skip. If most people skip that, how would you break that down into chunks for most optimal data bandwidth utilization? Also, what future chunks to load then? Or could you create chunks by scenes or shot selection?

Observation #4: If you notice, if Netflix has 10,000 TV shows, most people are watching the most popular ones at a time. There is a long tail that does not get watched often. For most popular shows, can we do optimisations to ensure their load time is better?
What if we did server side caching of their metadata, so that it can be returned much faster. In general, LRU caching for movies/TV show metadata does the job.

Summary: Hotstar uses various technologies and techniques to ensure that viewers can access high-quality video streams quickly, reliably, and without interruption. To further optimize the user experience for non-live content, Hotstar could employ the following optimizations:

- **Chunking:** Dividing video content into small sections or "chunks" allows for improved streaming and delivery of video files.
- **Pre-emptive loading** of the future chunks.
- **Browser Caching** of relevant metadata and chunks to enable a smoother viewing experience.

- **Content Delivery Network (CDN):** A CDN can help distribute the load of delivering video files to users by caching content closer to the users. It can significantly reduce the distance data needs to travel, thereby reducing load times. The browser can download resources faster as they are served from a server closer to the user.
- **Server-side Caching:** By caching frequently-accessed video files and metadata, the system can reduce the number of times it has to retrieve data from a slower storage system like disk or network storage.
- **Encoding optimization:** Using video encoding, Hotstar reduces the size of the video files without affecting the perceived quality.
- **Adaptive Bitrate Streaming:** This technique allows the video player to adjust the video quality based on the user's network conditions.
- **Minimizing HTTP requests:** By reducing the number of resources that need to be loaded, the browser can load the page faster. It can be done by consolidating files, using CSS sprites, and lazy loading resources.

Q3: In live streaming, given a bunch of clients could be lagging by a few seconds/minutes, how do you handle those?

Answer:

Now that recorded videos are done, let's think about how this should work for a live streaming use case.

For the recorded case, we had the entire file with us upfront. Hence, we could create chunks, transform to different resolutions as an offline task at our convenience. Publish chunk metadata. And then make the video live once all of it was ready.

For live use cases, we are getting video on the fly. While a delay of a minute is fine, you wouldn't want the delay to be higher than that.

Problem statement: How does a client tell CDN/Hotstar where does it need to get the next sequence of bytes from?

Imagine if we do create chunks. If chunks are large (1-2 mins), then that means, our clients will have large lag. They will have to wait for the 2 min chunk to be created (which is only after events in those 2 mins have occurred) and then that chunk goes to the client via CDN.
Imagine we create 15-30 second chunks. This is also called stream segmentation.

Steps involved:

- From the source, the video gets to the Hotstar backend using RTMP.

- For every 15-30 second chunk, immediately, jobs are scheduled to transform these chunks to different resolutions. Metadata for these chunks is updated in primary DB (In memory cache and MySQL?).
- Via another job, these chunks are uploaded to CDNs.

On the client side,

- You request for chunk metadata (Backend can control - till how long back I send the metadata for, and what you send in metadata)
- Based on metadata, you request for the most recent chunk from CDN.
- You keep asking for incremental metadata from the backend, and keep going to CDN for the next chunk. *[Note that in the case of recorded videos, you need not have asked for incremental metadata again and again].*

The segmentation in the above case helps with:

- Smoother handling of network lag, flaky internet connection.
- Being able to support lagging clients (not every client needs to be at the same timestamp).
- Being able to show history of chunks in the past, so you can scroll to earlier chunks if you want to watch a replay.

Do note that there are clients like Scaler class streaming / Zoom / Meet where delay has to be less than 2-3 seconds. However, you don't need to support lagging clients there. You cannot scroll to a history of the meeting. You might not even be recording the meeting.
The number of clients would be limited. That is a very different architecture.

Homework: Think about how Google Meet streaming would work if you were building it.

Size of a segment: It's essential to balance the number of segments you create and the size of each segment. Too many segments will mean a lot of metadata, while too few segments will make every segment larger and hence increase the delay.

In addition, Hotstar also utilized **AI** and **data mining techniques** to identify trends in the streaming behavior of its users to improve the user experience. For example, they scale the number of machines up and down based on certain events (since autoscaling can be slow). Dhoni coming to bat increases the number of concurrent viewers, so ML systems can detect that pattern and scale to a larger number of machines beforehand.

Do note that **CDN** delivers the chunks of video fast as it's at the heart of the design.

To sum it up, Hotstar's system design uses techniques like chunking, encoding, dynamic buffer management, CDN, ABS, and AI-powered platforms; it ensures users have an enjoyable streaming experience regardless of their device or internet connection speed.

Q4: How do you scale from 100 clients to 50 million clients streaming simultaneously?

Let's look at what are the common queries that scale as the number of users concurrently live streaming increases.

In a live stream, as discussed above, 3 things happen:

1. **Upload:** Get the video from the source, encode and transform it to different resolutions and update metadata of chunks. This is independent of the number of users watching.
2. **Metadata fetch:** Fetch updated metadata of chunks to be loaded, so that clients can keep requesting the right stream of data from the CDN, from the right checkpoint.
3. **Streaming the actual video:** from the CDN.

2 and 3 scales with the number of concurrent users. [Note that we have assumed a MVP here, so no chat/messaging feature assumed].
For 2, you'd need to scale the number of appservers and number of caching machines which store these metadata. However, the maximum amount of load is generated due to 3. As it is a massive amount of data being sent across to all the concurrent users.

CDN infrastructure plays a major role here. Akamai (Hotstar's CDN provider) has done a lot of heavy lifting to let Hotstar scale to the number of concurrent users that they have. A CDN stores copies of web assets (videos, images, etc.) on servers worldwide so that users can quickly access them no matter where. As the number of users of Hotstar increases, the CDN will have to scale accordingly.

If most of the users are expected to be from India, then for edge servers (CDN machines closer to the user) that are close to India region, more capacity is added there (more number of machines). Clients do an ANYCAST to connect to the nearest available edge server.

Additional Detailed Explanation for Streaming Platforms

Designing the architecture for a system like Hotstar or Netflix requires various components and their interactions. High-level overview of the architecture for such a streaming platform:

1. User Interface:
    a. Web Interface: A responsive web application allowing users to access the platform using a web browser.
    b. Mobile Applications: Native mobile apps for iOS and Android platforms.

    c. Smart TV and Set-top Box Applications: Applications designed for smart TVs and set-top boxes.
    d. Other Devices: Applications for gaming consoles, streaming devices, and other platforms.

2. Content Ingestion and Management:
    a. Content Ingestion: A system to ingest and process different types of content, including movies, TV shows, documentaries, etc.
    b. Content Management: A central repository to store and manage metadata related to content, including title, description, duration, genre, cast, crew, ratings, etc.
    c. Content Encoding and Transcoding: Media encoding and transcoding pipelines to convert video and audio files into various formats suitable for streaming across different devices and network conditions.

3. Content Delivery:
    a. Content Delivery Network (CDN): A distributed network of edge servers strategically placed across different regions to deliver content efficiently.
    b. Adaptive Bitrate Streaming: The ability to dynamically adjust the quality of the video stream based on the user's network conditions, using technologies like MPEG-DASH or HLS.
    c. Caching and Load Balancing: Caching frequently accessed content at edge servers and load balancing the incoming traffic across multiple servers to handle scalability and improve performance.

4. User Management and Personalization:
    a. User Authentication and Authorization: Systems to handle user registration, login, and session management.
    b. User Profiles and Preferences: Storing and managing user profiles, viewing history, watchlists, and personalized recommendations based on user behavior and preferences.
    c. Social Integration: Allowing users to connect their social media accounts, share content, and engage with other users through comments, ratings, and reviews.

5. Search and Discovery:
    a. Search Engine: A powerful search engine to enable users to find specific content based on keywords, filters, and advanced search options.

b. Recommendation Engine: Utilizing machine learning algorithms to analyze user behavior, preferences, and content metadata to provide personalized recommendations and discover new content.
c. Categories and Genres: Organizing content into categories, genres, and curated collections to assist users in discovering relevant content.

6. Payment and Subscription:
    a. Payment Gateway Integration: Integrating with various payment gateways to facilitate secure payment transactions for subscription plans or individual content rentals/purchases.
    b. Subscription Management: Managing user subscriptions, including plan selection, recurring billing, and handling cancellations or upgrades/downgrades.

7. Analytics and Monitoring:
    a. Usage Tracking: Collecting and analyzing user data to gain insights into user behavior, content popularity, and system performance.
    b. Error Logging: Logging and monitoring system errors, exceptions, and issues for proactive debugging and troubleshooting.
    c. Performance Monitoring: Monitoring system performance metrics, including response times, server load, network latency, etc., to ensure optimal user experience.

8. Backend Infrastructure:
    a. Microservices Architecture: Decomposing the system into smaller, loosely coupled services for scalability, maintainability, and independent deployment.
    b. Containerization: Utilizing containerization technologies like Docker to package and deploy services consistently across different environments.
    c. Orchestration: Employing container orchestration platforms like Kubernetes to manage and scale the containerized services effectively.
    d. Database Management: Utilizing databases for storing user data, content metadata, session information, and system configurations.

Additional Resources

- Scaling hotstar.com for 25 million concurrent viewers
- Building a scalable data platform at Hotstar

Microservices

Microservices vs Monoliths

Microservices and monoliths are two different architectural approaches for building applications. Each approach comes with its own set of advantages and disadvantages.

Advantages of Microservices:

1. Modularity and Scalability: Microservices break an application into small, independent services that can be developed, deployed, and scaled independently. This modularity allows teams to work on different services simultaneously and scale individual services based on demand.
2. Flexibility and Technology Diversity: Each microservice can use the most appropriate technology stack for its specific functionality. This enables developers to choose the best-suited technology for each service without being tied to a single technology stack across the entire application.
3. Continuous Delivery and Deployment: Microservices enable continuous delivery and deployment practices. Since services are independent, updates and changes to one service do not affect others. This allows for faster release cycles and the ability to respond quickly to user feedback.
4. Fault Isolation: If a microservice fails, the rest of the application can continue to function independently. This fault isolation ensures that failures in one part of the system do not bring down the entire application.
5. Scalability: Microservices architecture enables horizontal scaling by allowing the addition of more instances of a specific service as needed, making it easier to handle varying loads.

Disadvantages of Microservices:

1. Complexity: Microservices introduce additional complexity, such as inter-service communication, service discovery, and managing distributed systems. This complexity can increase the operational overhead and development effort.
2. Distributed Systems Challenges: Managing a distributed system introduces challenges such as network latency, data consistency, and ensuring reliable communication between services.

3. Testing and Debugging: Testing and debugging can become more complex due to the distributed nature of microservices. Ensuring end-to-end testing and tracking down issues can be challenging.

Advantages of Monoliths:

1. Simplicity: Monolithic applications are generally simpler to develop, test, and deploy compared to microservices. The codebase is usually contained within a single project, making it easier for developers to understand and work on the application.
2. Performance: Monolithic applications may have lower overhead compared to microservices since there is no need for inter-service communication over the network.
3. Easier Data Consistency: Data consistency is easier to manage within a monolithic application since all code operates within the same database transaction.
4. Deployment Simplicity: Deploying a monolithic application is typically simpler than deploying a distributed microservices architecture since there are no interdependent services.

Disadvantages of Monoliths:

1. Limited Technology Flexibility: In a monolithic architecture, all components of the application must use the same technology stack, limiting the ability to use the best tool for each specific task.
2. Scalability Bottlenecks: Scaling a monolithic application can be challenging since the entire application needs to be scaled as a whole, even if certain parts experience higher demand.
3. Team Coordination: In large monolithic applications, coordination between development teams can become difficult, leading to slower development and release cycles.
4. Maintainability: As monolithic applications grow larger, they can become harder to maintain, understand, and modify, leading to increased technical debt.

Choosing between microservices and monoliths depends on the specific needs and characteristics of the project. Microservices are well-suited for large and complex systems with independent, cross-functional teams, and a need for scalability and continuous deployment. Monoliths may be more appropriate for smaller applications,

simple projects, or teams with limited resources and expertise in managing distributed systems.

Communication between Microservices

In microservices architecture, various protocols can be used to facilitate communication between microservices. Here are some commonly used protocols:

1. HTTP (Hypertext Transfer Protocol):
    a. HTTP is widely used for communication between microservices just like monoliths as well due to its simplicity and compatibility with the web. It leverages standard methods (GET, POST, PUT, DELETE) to perform actions on resources and supports different data formats like JSON or XML for message payloads.

2. gRPC (Google Remote Procedure Call):
    a. gRPC is a high-performance, open-source framework developed by Google. It uses the protocol buffers (protobuf)- language-agnostic data serialization format and supports bi-directional streaming and efficient binary communication. gRPC provides strong typing and supports multiple programming languages, making it suitable for inter-service communication in microservices architectures.

    b. gRPC is a specific implementation of Remote Procedure Call (RPC). RPC is a communication protocol that enables the execution of procedures or functions on a remote server or service, allowing clients to invoke methods on remote objects as if they were local. It abstracts the complexities of network communication and provides a more seamless way to interact with remote systems. gRPC is an open-source framework developed by Google that implements RPC using modern technologies and techniques. It uses the protocol buffers (protobufs) language-agnostic data serialization format and supports bi-directional streaming and efficient binary communication. gRPC provides strong typing and supports multiple programming languages, making it easier to define and implement services and clients.

    c. Protocol Buffers offer a compact, efficient, and extensible way to serialize structured data, making it suitable for high-performance communication

between microservices.

d. gRPC supports bi-directional streaming, allowing clients and servers to send and receive multiple messages over a single connection. This enables efficient real-time communication and is particularly useful in scenarios where both the client and server need to send and receive continuous streams of data.

e. gRPC uses a service contract definition language to specify the interface and methods exposed by the service. This contract makes it easier for clients to understand and interact with the service by generating strongly-typed client stubs/protos from the contract.

f. gRPC provides language-specific bindings for multiple programming languages, including Java, C++, Python, Go, and others. This allows services and clients to be implemented in different languages while maintaining interoperability.

3. Message Queues / Kafka (Apache Kafka):
    a. As you know, Kafka is a distributed streaming platform that enables high-throughput, fault-tolerant, and real-time event streaming. It is often used as a central event bus or message broker for inter-service communication in microservices architectures. Kafka allows producers and consumers to publish and subscribe to streams of records. Microservices can implement event driven architecture using these message queues for communication between services.

Distributed Transactions in Microservices

Distributed transactions in the context of microservices refer to the coordination and management of transactions that span multiple services. In a microservices architecture, an application is broken down into a collection of loosely coupled and independently deployable services, each responsible for a specific business capability. In traditional monolithic architectures, transactions often involve multiple operations on a single, centralized database. In microservices, however, each service typically has its own database.

This decentralization introduces challenges when it comes to maintaining consistency across services during a transaction. Distributed transactions address this challenge by ensuring that a transaction involving multiple services either succeeds as a whole or fails entirely.

There are two main approaches to handling distributed transactions in microservices:

1. Two-Phase Commit (2PC):
    a. Two-Phase Commit is a more traditional approach but is less favored in microservices due to its potential for blocking and scalability issues.

    b. It involves a coordinator that sends a prepare message to all participating services. If all services acknowledge readiness, the coordinator sends a commit message. If any service fails to prepare, a rollback message is sent to undo the changes made by the services.

    c. The main drawback is that it introduces blocking, as all services must wait for the coordinator's decision, which can impact system responsiveness and scalability.

2. Saga Pattern - Compensatory Transactions / Transactional Messaging:
    a. In this approach, services communicate through messages and a message broker. When a transaction involves multiple services, the initiating service sends a message to other services to perform their part of the transaction.

    b. If any part of the transaction fails, a rollback message is sent to undo the changes made by the services - This is called "compensatory event" or "compensatory transaction"

    c. This approach relies on reliable messaging systems and is often described as Saga pattern to manage distributed transactions. It breaks a global transaction into a series of smaller, localized transactions (sagas) that are each managed by individual services. These smaller transactions ensure that the system can maintain consistency, even in the face of failures.

    d. Each saga step is associated with a compensating transaction that can undo the effects of the original transaction in case of a failure. This

ensures that the system can recover to a consistent state, even if some steps fail.

It's important to note that handling distributed transactions in microservices introduces complexities and trade-offs. Hence, this should only be used when necessary.