Components of System Design - I

- **Scaling**
- **Load Balancer**
- **Caching**
- **Content Delivery Networks (CDN)**
- **Message Queues**
- **Stateless Web Tier**

Scaling

Horizontal and Vertical Scaling
Vertical scaling, also called scaling up, means adding more power to a machine, for example we can add more CPU, RAM etc. to our server and make it more powerful, thus able to handle larger loads.

However, Vertical Scaling has a hardware limit. It is impossible to add unlimited CPU / RAM to a machine

In addition Vertical Scaling suffers from Single Point of Failure, despite how powerful the server might be if it goes down the entire system will go down, i.e. there is no redundancy or failover.

Horizontal Scaling, also called scaling up, allows you to add more machines to your system to handle the greater load. For example instead of just having a single Server, we can have 2 and load balance the traffic over them. This allows for Redundancy and Failover.

Separating Web and Data Tier - In a single server application, the server is responsible for everything i.e. running the application, running the web logic as well as running the database.
With growth of the user base we'll need to separate the web and data tier, i.e. having a separate set of servers for each of the tier.

The app tier will run the application and core business logic while the data tier can run the database. This approach allows for decoupling, and allows for the web and data tier to be scaled independently.

Load Balancer

With horizontal scaling we can get multiple servers to handle the traffic, however how do we actually distribute traffic between them. This is where a Load Balancer is used, a Load Balancer evenly distributes traffic among multiple servers downstream. For example AWS Load Balancer (ALB / NLB / CLB or GLB) we need to specify the servers in a target group. The Load Balancer will load balance the traffic between these servers.

Now the user (web browser / mobile app) instead of connecting to a specific server will connect to a public Endpoint provided by the Load Balancer.
Thus the client doesn't directly connect to the web server anymore, rather it connects to the Load Balancer.
Load balancer will forward the traffic to downstream servers, and to increase security the Load Balancer will communicate with the servers using Private IPs.

Load Balancer helps in increasing the availability of the application. For example if load balancer has 2 downstream servers S1 and S2. If S1 goes down, then it'll forward all the traffic to S2. This prevents the website from going offline. Additionally we can put the instances in an ASG (Auto Scaling group), which will always ensure we have a certain desired number of servers running.

Ofcouse if the load increases, the ASG will detect this and add more instances to the server pool, and the Load Balancer will distribute the traffic b/w all those servers.

Load Balancer Algorithms

The main purpose of the load balancer is to distribute traffic across multiple downstream servers, so that the traffic is distributed evenly and none of the servers are overwhelmed.

Types of Load Balancers:

1. Application load Balancers: ALBs operate at layer 7 of the OSI model. They can read the headers, session, data, response, cookies and route traffic based on this information. ALB can perform caching (because it can read the response). For example AWS ALB can route

HTTP / HTTPS or WebSocket traffic to multiple downstream servers (EC2 instances) using the following strategies:

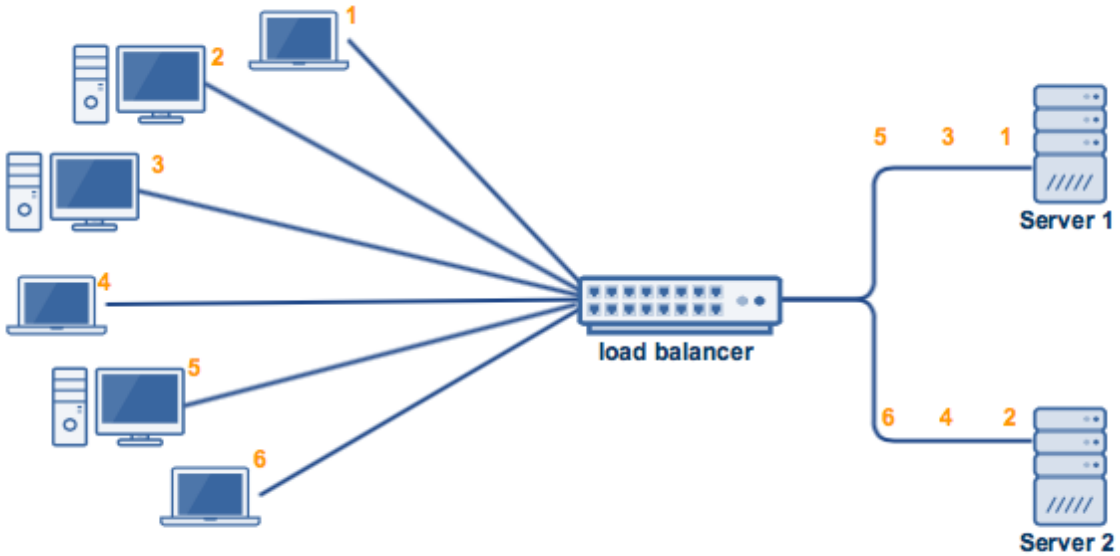
a. Routing based on path in URL
b. Routing based on hostname in URL
c. Routing based on query strings, headers

2. Network load Balancers: NLB operate at layer 4 of the OSI model, i.e. they can read src / dest port numbers. NLB's are faster than ALBs. For example AWS NLB (Network Load Balancer)

Load Balancer Algorithms

Static Algorithms (Static Load Balancing)

Round Robin: The load balancer routes traffic to the different servers in an alternative round robin manner, say we have 3 servers and the first request is routed to server 1, then the second request will be routed to server 2, third will be routed to server 3. Next the fourth request will be routed to server 1 again, and so on.



Advantages of Round Robin Algorithm:
=> Easy to implement

=> Equal load distribution across all the servers.

Disadvantages:

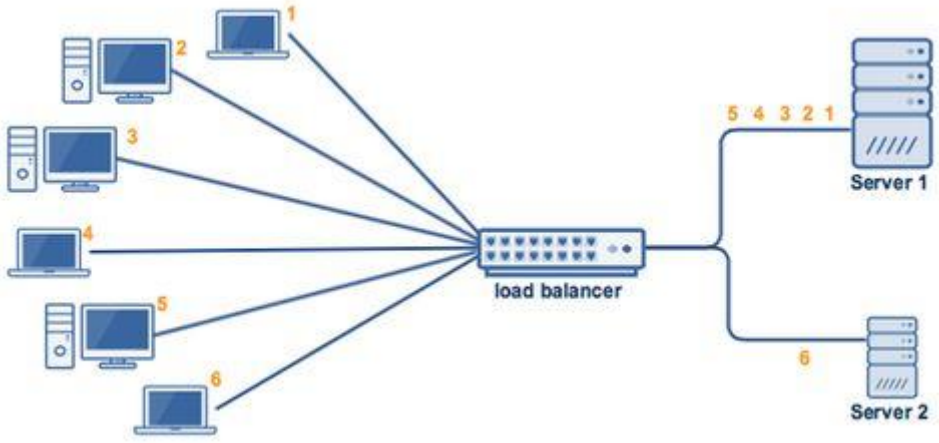
=> If one of the servers has lower capacity than the other, even then Round Robin load balancing will treat them similarly, hence load balance the traffic equally among the two servers. However since one of the servers has a lower capacity it might be overloaded, and it may go down.

Weighted Round Robin: Each of the servers is associated with a weight, so for example server S1 has a weight of 3 and S2 has a weight of 1, this implies S1 can handle thrice as many requests as S2. However important to note, the requests will still be distributed in a round robin manner.

See the weight as the capacity of the server. The weight determines the proportion of incoming requests the server will receive.

For example if we have 2 servers, S1 with weight of 3 and S2 with a weight of 1, then S1 will receive 75% of the incoming requests, while S2 will receive 25%.

S1 will keep receiving requests until it's capacity is exhausted, once it is, the next request will be sent to S2, in a round robin manner.



Advantages:

=> Traffic is distributed in accordance with the weight of the servers, so the lower capacity servers will not be overwhelmed.
=> Easy to implement as weights are static in nature.

Disadvantages:

However it is still possible to overburden the lower capacity server. If the requests have different processing times, then it is possible for the lower capacity server to be overwhelmed if it receives a high number of large requests, i.e. ones which need more computation time.

IP Hash: Each of the clients has a source IP address, the load balancer uses this IP address to compute a hash value (using a hash function), the hash value will determine which server to route the request to.

For example if there are S servers, then the request will be routed to the server x, where x = hash(src_ip) % S

Advantages:

=>Provides stickiness, where the same client needs to connect to the same server.
=> Easy to implement.

Disadvantages:

=> If the client requests are coming through a proxy, then all the clients will have the same source IP address, and this will overload the server to which these requests are being routed.
=> Cannot guarantee even distribution.

Dynamic Algorithms (Dynamic Load Balancing)

Least Connection: The load balancer will check which server has the least number of active connections and will route the request to that server.

Advantageous:

=> Dynamic, it considers the load on each server, and routes the request to the server with the least load so the chance of overwhelming any server is less, when each server has equal capacity.

Disadvantages:
=> It is possible for a TCP connection to be active, despite no traffic over that connection, so the purpose of Least Connection is failed.
=> Additionally it does not differentiate between low capacity and high capacity servers, so the low capacity server might get overwhelmed.

Weighted Least Connection: Each server has a weight associated with it. When a new request comes in the load balancer computes the ratio of number of active connections to the weight for each of the servers. Server with the minimum ratio is assigned the request.

Least Response Time: It uses the concept of TTFB (Time to First byte). TTFB is the time interval between sending a request and receiving the first byte of response from the server.

Each server has 2 attributes, number of active connections and TTFB. To compute the TTFB the load balancer will send probes to the servers, and track the response time.

The load balancer calculates the value of (Number of active connections * TTFB) for each of the servers, the server having the minimum value is assigned the request. What if this value is the same for 2 or more servers, in this case the load balancer will round robin the requests b/w them.

Caching

https://aws.amazon.com/nosql/in-memory/

Caches are in-memory databases with really high performance and low latency, Caches are used to store frequently accessed data so that subsequent requests for that data can be served more quickly.

Caches are suitable for storing data that is frequently read but modified infrequently.
A cached data is stored in volatile memory i.e. it is not persistent.

Advantages of using a cache

1. Serve frequently requested data quickly
2. Reduce the load of the database, database calls are expensive thus frequent database calls degrade application performance, Caching helps to overcome this issue.
3. Avoid Recomputations, by storing results of aggregate queries.
4. Decrease number of network calls.

Cache Invalidation

Cache Invalidation is needed because the data that is there in the cache is going to change at some point in time, and if that data is changing then we need to update the cache as well. The process of removing the old entry in the cache, or updating it with the new value is called Cache Invalidation.

When should an entry be invalidated?

=> It could be done based on TTL, *the TTL or the time to live, is the expiry time for an item in the cache, beyond which it will be evicted.*
=> Further the application can explicitly remove an item from the Cache.

Cache Eviction

A Cache has a limited size, for example 1000 entries. If the cache is full and a new item needs to be added then one of the older items will need to be deleted (evicted) from the cache. This is called Cache Eviction.
Cache Eviction policies: FIFO, LRU, LFU

When will Cache Eviction Occur?

Cache eviction can take place when the cache is full and new data needs to be added to the cache.
In addition Cache Eviction will take place if the TTL for an item has expired.

Cache Policies / Patterns

1. Cache-Aside: The Cache talks only to the application, and never to the database. In this pattern, the application talks to both cache and database.

   If the application wants to read some data, it will first go and query the Cache to see if it has the required data, if yes then it's a Cache hit and the required data is returned to the application.

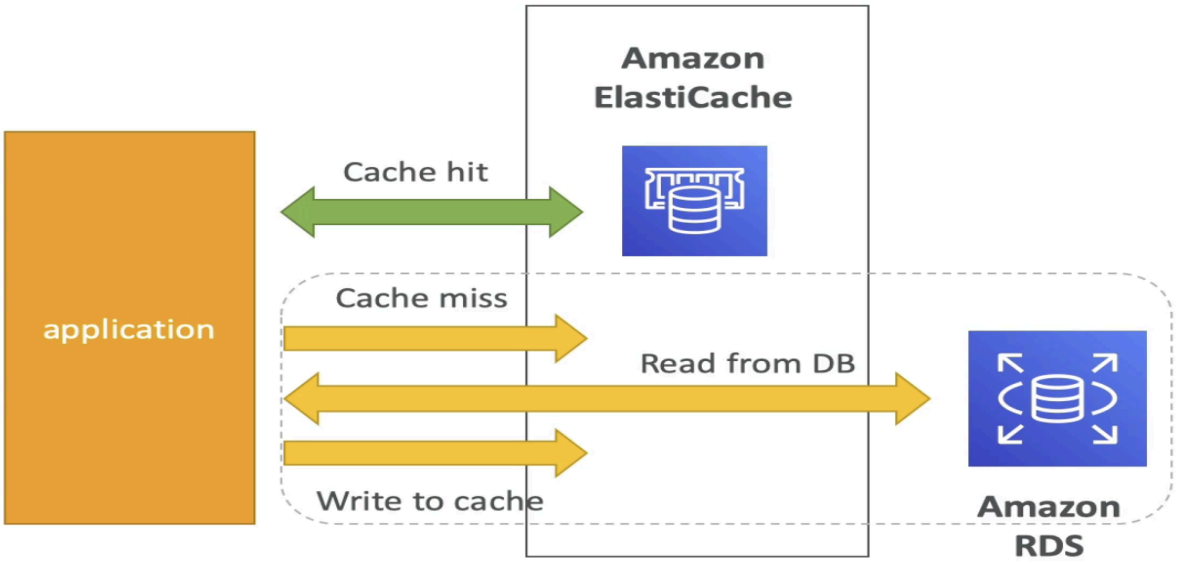
   In case of a cache miss, the application will send a read request to the main database, get the result and update the cache with it as well so that in future if that data is requested again, there will be a cache hit.

   Advantages:

   Even if the cache fails, the application can still keep serving the data from the main database, however it will be slower, since querying the db is more expensive.

   Disadvantages:

   With the cache-aside strategy the data in cache might become stale, and thus the user might be served outdated data. To fix this we'll need to invalidate the older data if it's updated in the main database. This could be done by either setting a TTL for the item, or the application could explicitly remove / or update an item from the cache.



2. Read Through / Write-Through: In this pattern, the application does not talk to the database, it only talks to the cache. The cache sits in b/w application and database and talks to both.

   In this strategy the application writes to the cache and it is the responsibility of the cache to update the database.

   Similarly for read requests, the application will query the cache, if the data is found then it will be returned to the application. However in case of a cache miss, it will be the responsibility of the cache to request the data from the database, i.e. application doesn't talk to the database the cache does. Once db returns the data, it will be stored inside the cache and then returned to the application.

   Advantages:
   - Suitable for read-intensive applications.

   Disadvantages:
   - Cache is a single point of failure

3. Write Around: This approach is similar to Read Through / Write Through as the cache sits in between the application and the database. However, here the application writes directly to the database. If there is a read request it will first query the cache, if the data is there (ie cache hit), it will be returned to the application. Else the cache will talk to the

database, get the data and populate the cache and return the data to the application. Useful for write-intensive applications.

Advantages:

Suitable for write-intensive applications.

Disadvantages:

Cache is a single point of failure

4. Write Back Strategy: Write requests are buffered in the cache for some time and then sent to the database in batches. Useful for write heavy workloads, and this pattern can handle db failures for some time.

   Advantages:

   As writes are buffered in cache and written to db in batches, the performance is improved, additionally we can withstand temporary outages.

   Disadvantages:

   Cache is a single point of failure, if Cache goes down all the reads are lost

Other Caching Concepts

Thrashing: If you have a very small cache size, it is possible you are inserting and deleting items from the cache without ever using them. If there are too many evictions, consider scaling the cache out or up.

Content Delivery Networks (CDN)

CDN is a network of geographically dispersed servers used to deliver static content. The CDN servers cache static content like images, videos, CSS, java Script files etc.

When a user visits a website, the CDN server closest to the user will deliver the static content, the closer the user is to the CDN server the faster the content will load.

What if CDN doesn't have the required data cached?
Imagine a user requests an image called img.png, and the CDN server does not have img.png in its cache, in this case the CDN server will request the file from the origin, the origin could be a web server or online storage like S3.

The origin returns the required data along with a TTL value, the image is cached in the CDN server for future use and then it is returned to the user who requested it.
Next, if another user B requests for the same file img.png, then that will result in a cache hit as the file is present in the CDN server's cache, and thus the image is returned to the user immediately.

Considerations of using a CDN

1. CDNs are run by third party providers, like AWS CloudFront and you are charged for data transfers in and out of CDN. Thus only caching the frequently used assets makes sense.
2. TTL: The TTL or the time to live is the expiry time for an item in the CDN server cache, beyond which it will be evicted.
3. Invalidating data in the CDN: We can remove a file from the CDN by:
    a. Explicitly asking the CDN to remove an object by using the APIs provided by the CDN vendor.
    b. Use object versioning to serve a different version of the object.

Hence with the addition of a CDN, it takes over the responsibility of serving static content, i.e. the web servers no longer need to serve static assets.
Fetching static content from CDN improves application performance.

Message Queues

A Message Queue is a durable component that makes asynchronous communication possible.
It acts as a buffer and distributes asynchronous requests.

The producers can keep pushing messages to the queue, and the consumer can process them as and when possible. This is not possible with synchronous communication where immediate response is expected. If there are a lot of requests and there is no queue, then the system will be overwhelmed and it won't be able to serve the requests, so we can't handle a lot of load.

Basic Architecture: Services called Producers / Publishers create messages and publish them to the message queue. Other services called Consumers / Subscribers poll the messages, and perform the actions requested in the messages.
A message could, example: Process this video, Process this order, Insert something into the database

The Consumer will poll the Message Queue for messages, process it and delete it from the queue.

We can have multiple producers and consumers, and the Message Queue decouples them, i.e. it decouples components of an application.

https://medium.com/@erickgallani/how-to-apply-a-heartbeat-pattern-to-your-aws-sqs-consumers-in-net-f87399c1231c

Message Queues: Amazon SQS, Rabbit MQ, Kafka.

A Message Queue helps in making a system scalable and reliable. The Queue is durable, i.e. even if the producer or consumer goes down, the messages still remain on the Queue. Additionally the producer can push messages into the queue even when the consumer is not available. Similarly a Consumer can read (poll) messages from the queue even when the producer is not available.

Message Queue Models

1. Producer Consumer Model
2. Pub Sub Model, Publish Subscribe Model

A note on Message Consumption:
In a Producer consumer model, the consumption is one-to-one i.e. each message pushed by the producer in the queue will be processed only once, and once a consumer has processed the message it will delete the message from the queue. So in this model, each message is processed only once. It doesn't matter if there is one consumer or multiple, each message is processed only once.

In a Pub-Sub model, the consumption is one-many, i.e. the same message will be processed by multiple consumers.

Note, on the ordering of messages

I) *Ordering Does Not Matter*
Amazon SQS Standard Queue offers 'Best Effort Ordering' (i.e. the queue can have out of order messages).

Suppose the queue looks like:

m2	m7	m3	m5	m9	m1	m6	m8	m4

As can be seen the queue contains out of order messages. What happens if we are not able to process a message, say m6. In the case of SQS Standard Queue since ordering of messages is not important hence we can go about processing the other messages, as for m6 we can put in the dead letter queue.

Consider for example, a queue containing customer requests for 'invoices for the last x months', in this case the ordering doesn't matter much and hence SQS can be used, if we are not able to process the message for a particular customer, even then we can process the other customer messages, **since all these messages are independent.**

Additionally we add the messages which couldn't be processed into a dead letter queue, for closer examination and debugging later on.

II) *Ordered Queue*
Now consider AWS SQS FIFO Queue, which guarantees a FIFO ordering of the messages. A FIFO Queue is useful when the order of messages

matters, and the messages must be processed in the same order as they were added into the queue.

Consider an example of a chat application, obviously we want the messages to be sent in the correct order, we cannot use a SQS queue for this purpose as it can send out of order messages, and hence the messages might be sent in the wrong order!!

To fix this we use a FIFO queue, which ensures that the messages are in the correct order.

Now let's consider the case where we are not able to process a message. In the SQS example, we just moved the message 'm6' (which we were not able to process) into the dead letter queue and continued processing other messages.

Now let's consider the case of FIFO queue

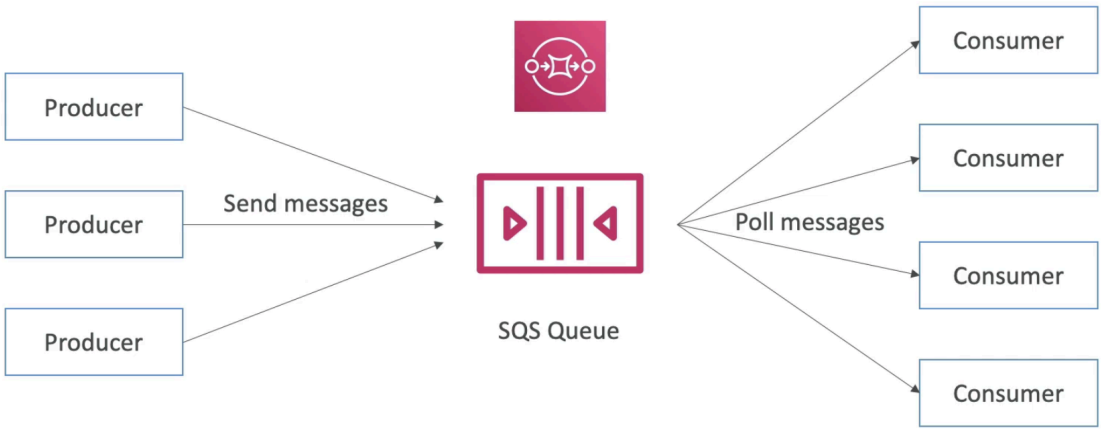
m9	m8	m7	m6	m5	m4	m3	m2	m1

As we know we use a FIFO queue when the ordering of messages matters, i.e. the messages are not independent of each other. For example m2 cannot be processed before m1 and m3 cannot be processed before m2.

Thus if for some reason we are not able to process the page m4, then we cannot process the messages: m5, m6, m7, m8, m8 (or any m[i], where i > 4). This is because the messages need to be processed in order, and we cannot process m5 if we haven't processed m4.

Thus in such a situation the FIFO Queue would block the processing of messages here (once it determines that m4 cannot be processed), it will not let other messages get processed until m4 is processed.

Interlude: Amazon SQS



SQS Offerings

1. Standard Queue
2. FIFO Queue

SQS Standard Queue

- Message Retention: 4 - 14 days
- Message Size Limit: 256KB
- Best Effort Ordering, i.e. the queue can have out of order messages.
- Unlimited Throughput
- Consumers poll the SQS queue for messages, and may receive up to 10 messages at a time. Once a consumer has processed a message it will delete the message from the Queue using the DeleteMessage API.
- For horizontal scaling we can put our consumers in an Auto Scaling Group (ASG) and scale it on some SQS metric like 'Approximate Queue length'.
- Message Visibility Timeout: Once a message is polled by a consumer it becomes invisible to other consumers for a period of time called the Message Visibility Timeout. During this period if any consumer requests the queue for messages then this message won't be polled. If the consumer is able to process the message in this time period, then it will delete the message from the queue, otherwise once the timeout period is over, the message will again become "visible" in the queue, i.e. it can be polled by any consumer now.
- Dead Letter Queues (DLQ): If a message cannot be processed by a consumer within the message visibility timeout period, then it will be put back into the queue. If this happens repeatedly it means there might be some issue with the message. To solve this problem

we set a threshold indicating how many times a message can go back to the Queue, if this threshold is exceeded then the message is put in a Dead Letter Queue, this helps in debugging. Note DLQ of standard queue must also be a standard queue, and the DLQ of FIFO queue must also be a FIFO queue.

SQS FIFO Queue

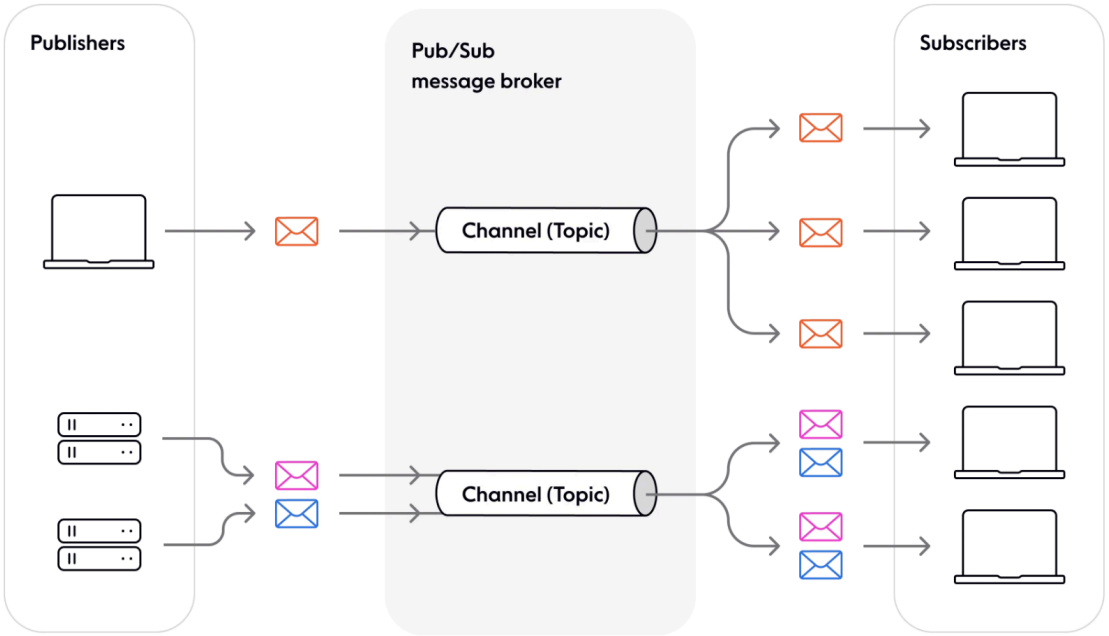
- FIFO Ordering (Messages are ordered inside the queue)
- Lower Throughput
- Messages will be processed in order by the consumer and exactly once.

Pub Sub Model

Pub Sub Model is an architectural design principle which enables asynchronous communication, in the case of Publish Subscribe model the consumption is one to many, i.e. the same message is processed by many consumers.
Pub Sub Model provides a framework for exchanging messages b/w publishers and subscribers.

The Pub Sub Model consists of the following parts:

1. Publisher: Components that create and send messages, the Publisher isn't aware about the subscribers, instead it will publish the message to a topic.
2. Subscriber: Components that receive and consume messages, Subscribers will subscribe to the topics they are interested in. Whenever a message is published to the topic (by the publisher) all the subscribers of the topic will receive the message.
3. Topics: Topics are essentially named communication channels, to which the subscribers will subscribe to. Subscribers can subscribe to one or more topics. Publishers will publish messages to the topic.
4. Message Broker: The Message Broker is an intermediary component which manages the routing of messages between publishers and subscribers. The message broker ensures that the messages are delivered correctly to subscribers of the topic.



AWS SNS (Simple Notification Service)
=> The Publisher only sends messages to the SNS topic.
=> Subscribers subscribe to a SNS Topic.
=> Each Subscriber will get all the messages published to that SNS topic.

Stateless Web Tier

To scale the web tier horizontally, we need to move state (for example user session data) out of the web tier.

First, we discuss what a stateful web tier is and what are some of the problems with it.

A stateful server remembers client data (state) from one request to the next, a stateless server keeps no information.

Consider an example:

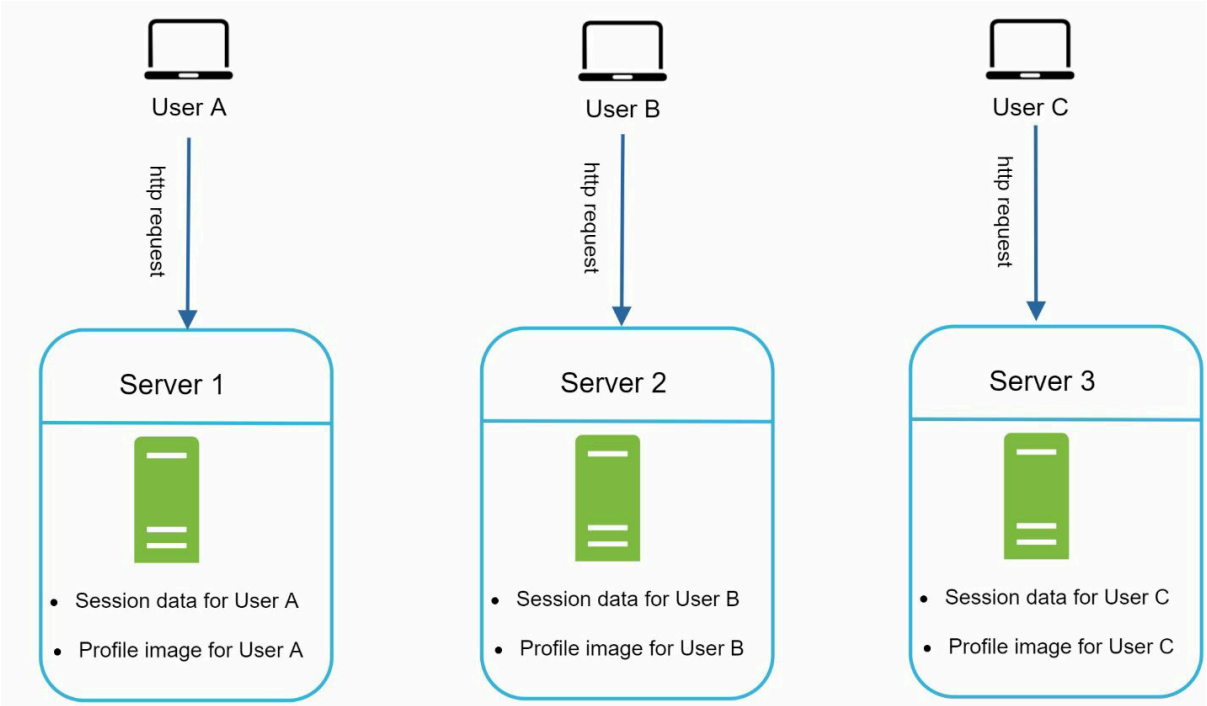


Figure 1-12

User A has authenticated to the application, and his session data and profile image are stored on server 1, i.e. server1 has information regarding User A. Now what happens if a request from User A is sent to another server (Say Server 2), since server 2 has no information on User A, it doesn't have A's session data. Hence A would be required to log in again.

To fix this issue, we must ensure that every request from the same client is always routed to the same server. This can be done by using sticky sessions in the load balancer. However with sticky sessions enabled it is

difficult to add or remove servers, it is also challenging to handle server failures.

In a stateless architecture, the state is moved out of the web tier, i.e. the web servers do not store any information regarding the client like session data.

Instead we store the state (session data) in persistent storage such as relational databases or NoSQL. Each web server in the cluster can access the state from the databases, resulting in a stateless web tier.

Now, in a stateless architecture the HTTP requests from the client can be sent to any of the web servers. Which will fetch the state (Session data) from the shared storage.
In essence state data is moved out of the web servers to a shared data store.
Stateless architecture is simpler and scalable.

Components of System Design - II

- **Proxies: Forward and Reverse Proxy**
- **Consistent Hashing**
- **Rest APIs**
- **High Availability, Active-Passive and Active-Active architectures**
- **API Gateway and Service Discovery**

Proxies: Forward and Reverse Proxy

The proxy sits between the client and the server and all the requests from the client to the server (or for that matter multiple multiple clients) are funneled through the proxy.

So as can be seen the proxy can handle requests for multiple clients, and sends those requests to the server (asking for the required data), the server will send the desired data to the proxy, which will in turn send it downstream to the clients. Hence the proxy server sits b/w the client and server and acts as an intermediary, all the requests pass through the proxy. Client and server can't communicate directly.

Types of Proxies

Differ in the direction of communication.

Forward Proxy (Simple proxy): When we say proxy we usually mean forward proxy.

Forward Proxy hides the client network, i.e. the server or the outside network / servers (do note the communication happens over the public internet) do not have any information about the client (internal network), i.e. the proxy hides the client from the outside world, no server can talk directly to the client, it can talk to the proxy only.

Advantages:

=> Gives anonymity to the internal network, so the client's IP address, location are hidden from the server.

=> Grouping of requests: As mentioned before a proxy can handle multiple clients, which allows the proxy to perform connection pooling. For example if multiple clients are requesting for www.google.com, then the proxy can club these requests and send them as one (Remember RDS Proxy).

=> Access restricted Content: With proxy in place we can bypass access restrictions. For example the user trying to access some restricted content in India won't be able to access it directly, but can access it through a proxy server which is located in a different geographical location, for example Australia.

=> Provides security: Since the requests from all the clients will be funneled through the forward proxy, we can add certain rules or logic to safeguard the client from accessing vulnerable websites or unsecure content networks for example. Additionally we can add rule based checking to the forward proxy.

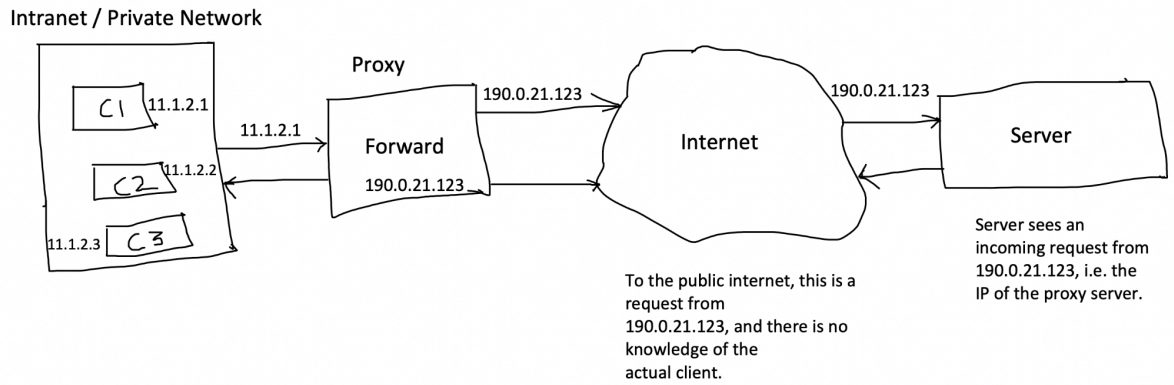
=> Caching: Forward Proxy performs caching of static content.

Disadvantages:
=> Forward proxy works at the application level, so for each application we need to set up the proxy.
This means the forward proxy inspects and modifies the packets based on the Application layer protocols (like HTTP, FTP, SMTP or DNS etc).

Hence the forward proxy can filter traffic based on the application layer protocols like the ones mentioned above, allowing for targeted security policies and content control.

So for example we can have a:
Web Proxy which can inspect the HTTP / HTTPS traffic, block malicious URLs, encrypt sensitive data,
An Email Proxy which can scan and filter emails, detecting and preventing spam and other malicious emails.
A Database Proxy can pool connections, and encrypt the data in transit.

Example diagram:



Reverse Proxy

As mentioned before, forward and reverse proxies differ in the direction of communication, the forward proxy sits towards the client and protects the client (providing security). Reverse Proxy sits towards the server side and provides security to servers. No incoming request can talk directly to the server.

Instead the client needs to talk to the reverse proxy, which in turn will relay the request to the backend servers.

So in other words the Forward Proxy works on behalf of the client, while the Reverse proxy works on behalf of the server.

Examples of Reverse Proxy: CDN

Advantages

=> Security: The outside world is not aware of the server's IP address, the clients only know the IP of the reverse proxy. This protects the end servers from DDOS attacks. For example if we have a CDN (reverse proxy) in place the attacker can only attack the CDN and not the actual origin servers.

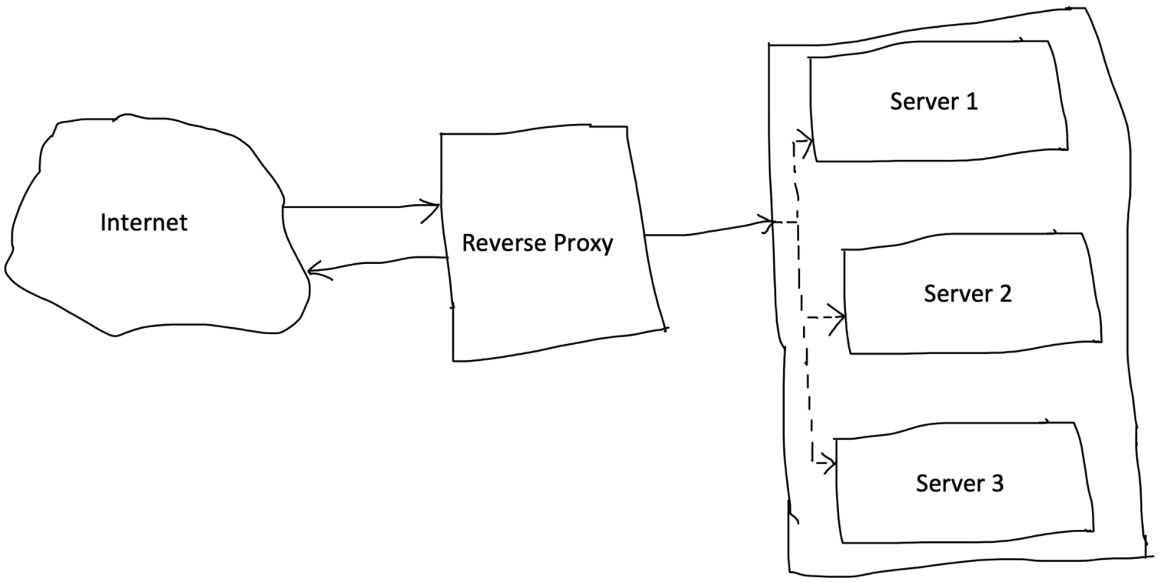
=> SSL / TLS termination.

=> Caching, for example by CDN.

=> Latency Reduction: Again a CDN can be placed in a location geographically nearer to the customers hence decreasing the latency.

=> Load Balancing: In the backend we might have multiple server instances handling the requests, the reverse proxy needs to balance these requests across all the servers. The client cares about the request being fulfilled and not which backend server instance is fulfilling it.

Example diagram:



Difference b/w Proxy and VPN

Proxy	VPN
Proxy acts as an intermediary between the client and the server and provides anonymity (hides IP address) to the client, similarly for the server in case of Reverse Proxy. In addition it provides features like caching, logging	When a client wants to communicate with a server over VPN, first a VPN tunnel is created b/w a VPN client and the VPN server. So when a client makes a request, first it has to connect to a VPN client, all the requests will go through the VPN client. The VPN client will do encryption on the data, before sending it over the VPN tunnel, i.e the data in transit in the VPN tunnel is encrypted. The VPN server will decrypt the data and relay it to the desired server. Hence VPN provides a way for the client and server to communicate in a secure manner (using

	encryption) over an unsecure network like the internet. The Proxy can mask the IP addresses however it cannot encrypt the data (encryption / decryption is not possible). However VPN creates a safe VPN tunnel through which the data flows.

Difference b/w Reverse Proxy and Load Balancer

Reverse Proxy	Load Balancer
Reverse Proxy can act as a Load Balance Reverse Proxy provides: IP Anonymity Caching Logging However Load Balancer does not provide these features.	Load Balancer cannot act as a Reverse Proxy. Consider a situation wherein we have only one server, here we don't need a Load Balancer but we will still need a Reverse Proxy.

Difference b/w Proxy and Firewall

A firewall has a set of rules which controls traffic flow. A firewall performs packet scanning, i.e it examines various headers of the packet and collect information like Src / Dest IP addresses, Src / Dest Ports Numbers etc. and will check the set of rules to see if the packet matches any of them, accordingly action will be taken (whether the packet should be allowed in / out or not).

=> One important difference is that Proxy works on the Application level, while firewall works on the packet.
=> Proxy Firewalls are proxy which can perform blocking, i.e. it has the capabilities of a firewall hence the proxy can act as a firewall also.

Proxy	Firewall
Proxy works on the application protocol level	Firewall works on the packet level, it perform packet scanning
Proxy works at the layer 7 of the OSI model	Firewall works at the layer 3 (Network layer) and Layer 4 (Transport layer) of the OSI model.
Performs client side request filtering	Performs IP packet filtering
Proxy can perform caching	Firewalls cannot perform any caching.
Proxies can act as Firewall	Firewall cannot act as Proxy.

Why do we say the proxy works at the application level?
Proxy server filters the client side requests, for example it can examine and modify HTTP requests and responses, i.e. it can filter web traffic, cache web resources etc.

Related Concepts:

Thick Client: Most of the Logical manipulation and processing happens at the client side. Eg. Outlook, Video Editing Softwares.

Thin Client: Most of the Logical manipulation and processing happens at the server side. Eg. Streaming Platforms like Netflix.

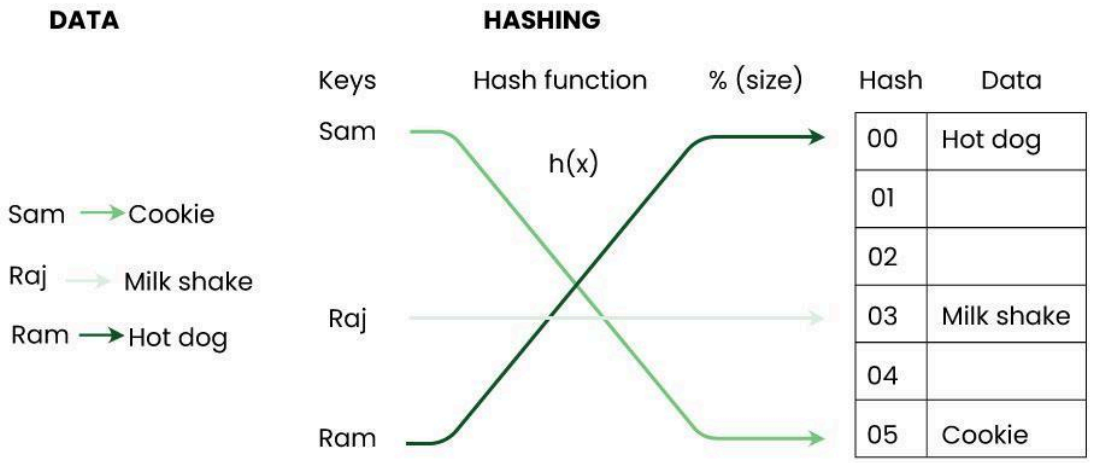
Summary:
Forward Proxy: Sits between client and the internet, and it communicates to the server on behalf of the client. Forward Proxy provides anonymity to the client, as the server does not see the IP address of the client. Caching the response at the forward proxy site. Can be used to access restricted content.

Reverse Proxy: Sits between the Internet and the server, and acts as a middleman for all the servers. Client talks to the proxy, and the proxy talks to the servers. In this way the servers are not directly exposed. Used for load balancing, caching responses from servers, SSL / TLS offloading. If the reverse proxy fails, it can become a single point of failure.

Consistent Hashing

Hashing is the process of using a hash function to produce a pseudo-random number given a key. Next this number is modulated with the size of the memory space (for example a hash table), this gives us the index or the position where the key, value pair needs to be stored.

Hash Functions are any functions that map value from an arbitrarily sized domain to another fixed-sized domain, usually called the Hash Space. For example, mapping URLs to 32-bit integers or web pages' HTML content to a 256-byte string.



Hashing

Problems with traditional hashing: If the size of the hash space (or size of hashtable) is fixed then hashing works perfectly fine. However if the size is not fixed, then Hashing runs into some problems.

Say we are building a storage distributed system, where users can upload and fetch a file. Now in the backend we have multiple storage nodes on which the files are stored. Say we have 5 such storage nodes. When the user uploads a file the system needs to determine which node to store the file on. It does so by using a hash function which produces a value in the range [0 .. 4] indicating which node to store the data on.

$$h(x) = f(x) \% 5$$

The hash value is the index of the storage node that will hold the file.

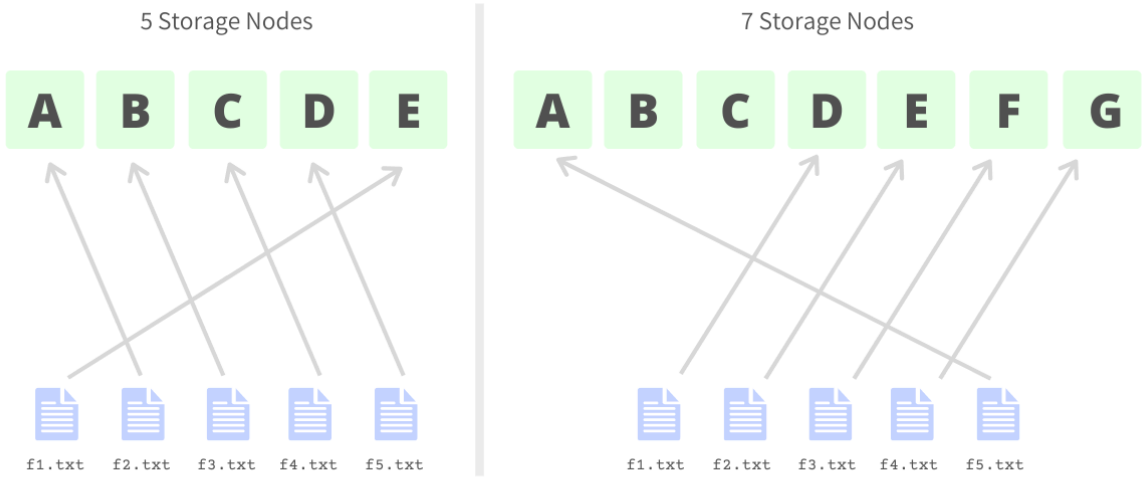
Say our system is gaining traction and now needs to be scaled from 5 to 7 storage nodes. To make use of these 2 additions nodes, our hash function will need to change to

$$h(x) = f(x) \% 7$$

Changing the hash function implies changing the mapping and association of files with the storage nodes, this involves moving the files from one node to another, as the mappings will need to be updated as the hash function has been changed. This movement is known as rebalancing.

This is an expensive operation as it will need to move a lot of files across nodes, to adapt to the new hash function. However the system needs to be scaled up and down quite frequently in response to the actual load, and if, every time the system scales we update the hash function then it will lead to movement of a lot of data across nodes each and every time, this process is super expensive and infeasible.
Another example could be load balancer for Application servers.

The goal of consistent hashing is to minimize data movement / transfer required when the system scales up or down.



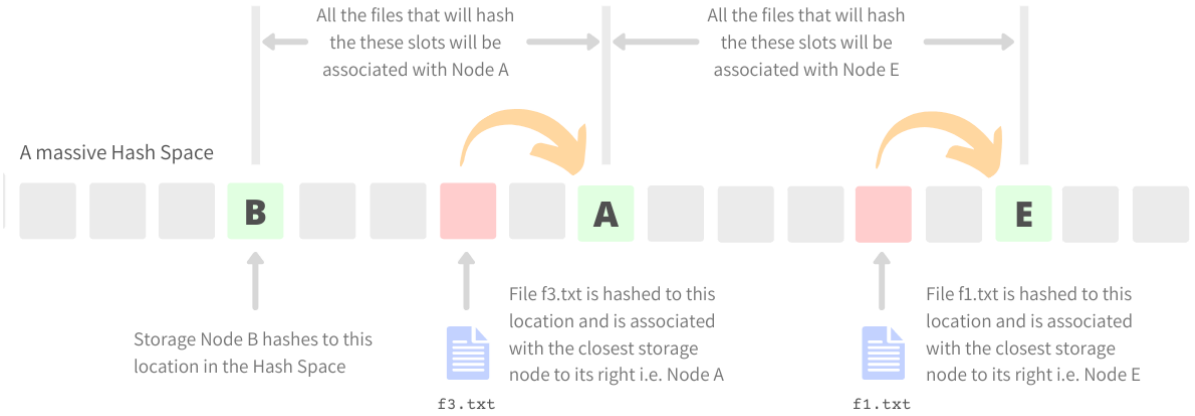
Scaling up the Storage Nodes

Consistent Hashing - As mentioned above the goal of consistent hashing is to minimize data movement / transfer across nodes when the system

scales up or down. It does so by making the hash function independent of the number of nodes in the system.

Consistent hashing addresses the drawbacks of hashing by maintaining a huge Hash Space. The requests and nodes (servers) are both mapped to slots in this hash space.

How do we define an association between the request and a node?
The request will be associated with the server node which is present to the immediate right of its hashed location. This keeps the hash function independent of the number of server nodes.

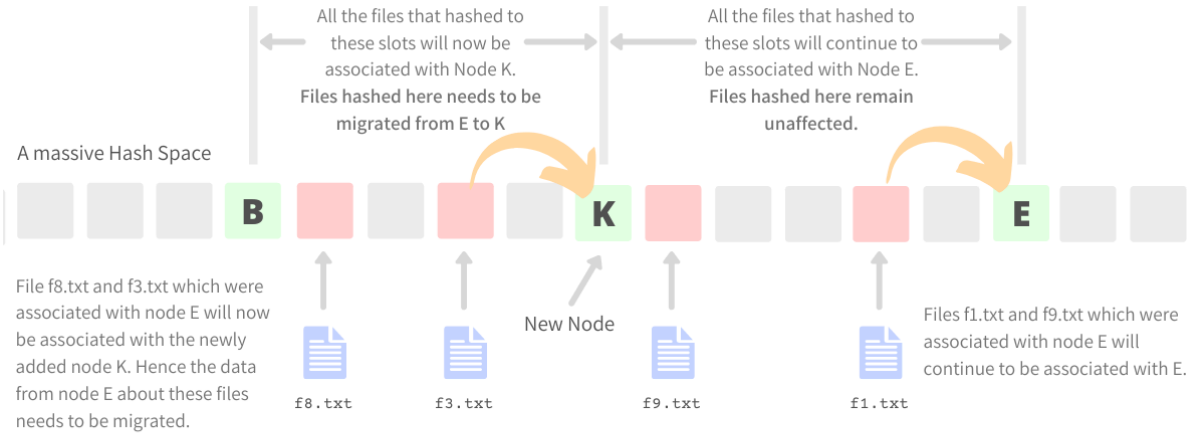


Associations in Consistent Hashing

Consistent hashing on average requires only k / n units of data to be migrated during system scale up and down.
Where k is the total number of keys and n is the number of nodes in the system.

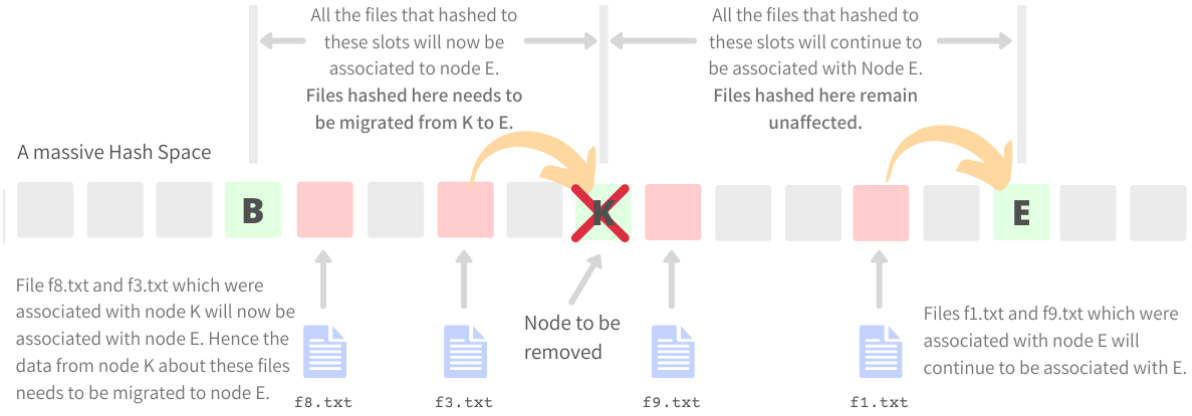
This large hash space can be represented as a virtual ring.

Work flows:



When a new node is added to the system only the files to the left of the node, associated with the node to the right of the newly added node, are affected and needs migration; associations of all other files remain intact.

Adding a new node in Consistent Hashing

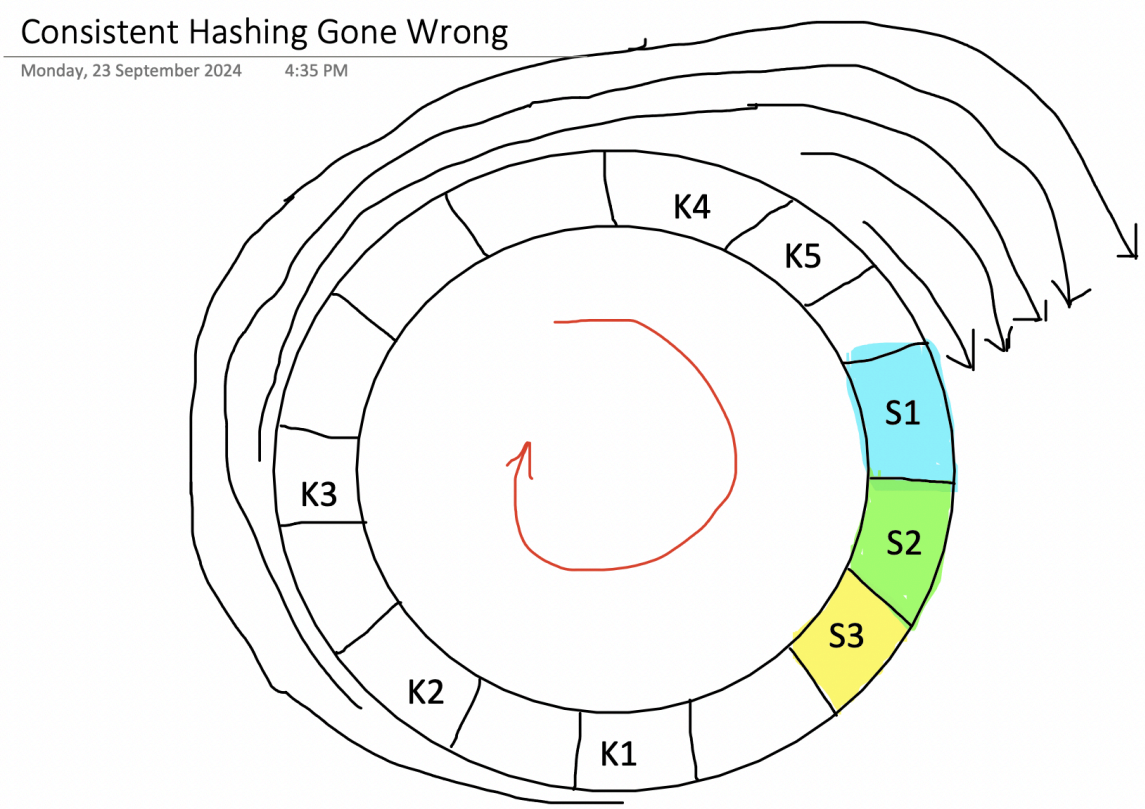


When a node is removed from the system only the files associated to that node will be migrated to an existing node to its right, all other files, nodes and associations remain unaffected.

Removing a node in Consistent Hashing

https://arpitbhayani.me/blogs/consistent-hashing/

Disadvantages of Consistent Hashing:
Since we are hashing the server it is possible that they are hashed very close to each other



Here all the requests will be routed to S1, overwhelming it. To fix this we need to use the concept of virtual servers.

We can pass each of the server nodes through K hash functions, and map all of these locations (hash values) to slots in the hash space, i.e. we replicate the server nodes at different locations in the hash space.

Virtual servers provide a better distribution of keys, and prevent any of the nodes from getting overwhelmed.

Say there are k keys and n servers, if distribution is even then each server handles k / n keys, so the load factor is (k / n) / k = (1 / n)

Consistent hashing is dynamic in nature.

Additional / Summary points on Consistent hashing

Regular mod hashing strategy works well if we have a fixed (static) number of nodes or servers, i.e. size of the server pool is fixed. However problems arise when the number of nodes is changed (increased - scale

up or decreased - scale down, or node failures), as regular hashing strategy results in widespread redistribution of keys across nodes. Suppose we have a distributed cache and one of the servers goes down, in this case most keys will need to be redistributed, not just the ones stored on the failed server.

So in this case even if one of the servers goes down, most keys will need to be redistributed, as a result the client connects will need to be re-routed to the other nodes, other side most clients will be connected to the wrong cache server to fetch data, resulting in a storm of cache misses.

The Hash Ring: The hash ring is a very big hash space which is circularly connected, how big is the hash ring?
That depends on the hash function, for example if we use a Cryptographic function like MD5 our hash space will be of 128 bits, i.e. addresses from 0 to $2^{128} - 1$
If we use SHA1, the hash address space will of 160 bits, i.e. addresses from 0 to $2^{160} - 1$.

Using the same hash function f we will map the servers to the ring, and then the keys to the ring (we don't use mod hashing in case of consistent hashing).

On What basis are the servers mapped: We can use the server's IP, server name or ID for mapping it onto the ring.

Concept of Partition: A partition is the space in the hash ring between adjacent servers. For even distribution of keys, the partition size for all the servers must be kept nearly the same. We don't want a situation where the partition size for a server is very large, as more keys can be mapped into this large partition size and hence the load on this server will be very high.

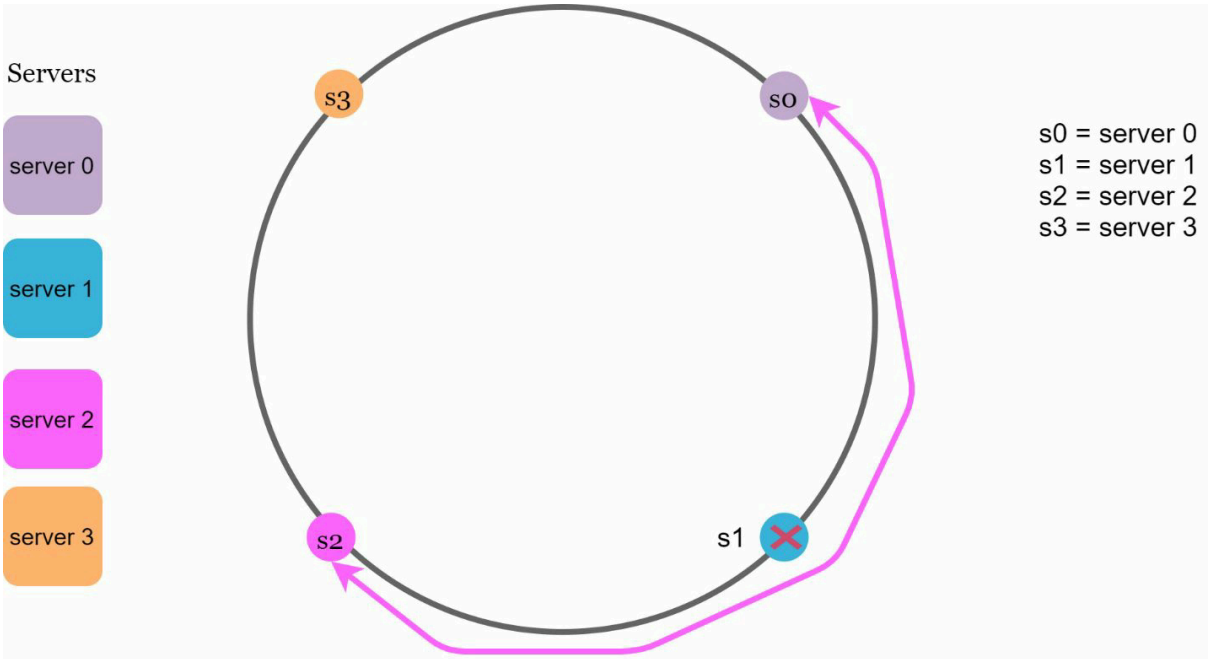


Figure 5-10

To address this problem we use the concept of Virtual servers / nodes / replication.

Virtual servers also solve the problem of uneven distribution of keys, a non-uniform distribution of keys can cause high load on a set of servers, while the others have very few keys.

Real world systems using Consistent hashing:

1. Amazon DynamoDB, for its partitioning component.
2. Apache Cassandra, for data partitioning across the cluster.
3. Discord
4. Maglev NLB (Network load balancer)

REST Apis

REST stands for Representational State Transfer

1. Client server Model
2. Cacheable: Response should implicitly or explicitly label itself as Cacheable or non-cacheable.
3. Layered: In a RESTful architecture each component cannot see beyond the immediate layer it is interacting with.
4. Statelessness: Stateless client-server communication, meaning no client information is stored between get requests and each request is separate and unconnected, i.e. the server should not save any state.
5. Uniform Interface: This guideline states that all requests and all responses must follow a common protocol, or a way of formatting their messages. Applications and servers are written in all sorts of different languages that don’t do a great job of working together without an intermediary. A uniform interface is a common language for any client to communicate with any REST API.

High Availability, Active-Passive and Active-Active architectures

=> Design an architecture with 99.999 (five 9's) availability.
=> Design resilient Architecture: A system can work even with failures, and recover from it.
=> There should be no single point of failure.

Issues with just using a single node:
A single server node suffers from a single point of failure, as if the node fails the entire application will be affected.

Active-Passive Architecture

The Active passive architecture consists of a primary active system and a secondary passive (backup) system that remains inactive until the primary system fails. This architecture is known as a failover or standby system.

The primary system handles all the incoming requests while the passive system remains on standby, ready to take over if the primary system fails and becomes unavailable.

Synchronization is established b/w the primary and standby to keep the standby up-to-date with the latest data, so that it can take over at any time. This synchronization can be synchronous or async (eventual consistency)

In the event of the primary system failure, the passive system takes over and is promoted to the primary system, and it will handle all the incoming requests. So this architecture relies on a failover mechanism to switch to the passive system in case the primary system fails, leading to potential downtime during failover.

For example: We can have an active DB (primary DB / Live DB or R/W DB) which is handling all the read / write requests and a passive database, which just sits idle.

Why can't we perform write to the standby DB: Databases like MySQL, Oracle, Postgres are not multi-master, i.e. they can only write to one database instance, which is the primary / live db, this fact also explains

why this architecture is useful, specially with relation to databases, as we cannot configure certain databases to work in the active - active architecture.

Relaxations of the Active-Passive architecture: We can set up the standby database to handle read requests, which will lessen the load on the primary instance. So we can have one primary DB taking all the writes, and multiple replicas handling read requests, which will scale the system. Here the type of synchronization b/w the active and passive db instance becomes very important.

Disadvantages with this architecture:

1. Latency Addon
2. Failover time
3. Lower resource utilization, since the standby db just idle most of the time.

Active-Active Architecture

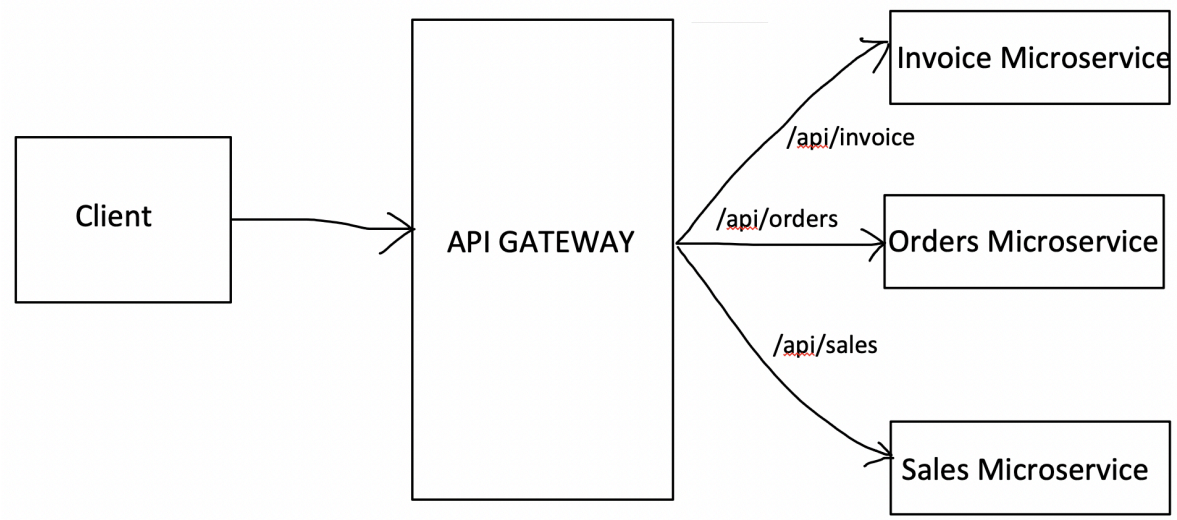
In the active-active architecture multiple identical resources like nodes or servers are simultaneously active and serving requests. In this setup, incoming requests can be handled by any of the active resources, allowing for load balancing and maximizing resource utilization.

Databases like Cassandra, Aurora that support the Multi-Master mode, i.e. more than one primary db / live db can be present, and the data can be written to any of them. Bidirectional synchronization is needed b/w all the primary / live instances. Obviously each of these active db instances can take reads and writes.

The advantage of this architecture is that it can scale to high write workloads, however concurrency control is a major challenge.
This architecture provides better resource utilization.

API Gateway and Service Discovery

API Gateway is a component which accepts the client API requests and routes them to the correct backend service based on the API endpoint.



In contrast a load balancer simply distributes the traffic across the multiple instances of a microservice, based on factors like server load, health etc. Load Balancers do not have the capability of understanding the API, or make routing decisions based on it.

Functions of the API Gateway: Routing, Authentication, Rate Limiting, analytics, API composition, SSL / TLS termination, HTTP to HTTPS redirects.

API Composition:

API Composition involves using an API composer or aggregator to perform a query by getting the data from various microservices that own the data. The aggregator or API composer then combines the results by performing in-memory joins. This strategy is widely used by Netflix, Conditional composition is also possible.

Authentication:

API gateway can implement Authentication, for example by using OAuth 2.0, i.e. it can authenticate the clients.

So instead of placing authentication logic at each service, we can just put it at the API gateway, thus preventing duplication.

Rate Limiting:

API gateway have natively built in support for rate limiting, we can configure the burst rate (maximum number of concurrent requests allowed), or API throttling (finer grained control, for example there should be only 10 calls per minute to a specific API, or it can be called only 5 times a day by a user in a particular day).

=> In addition we can perform IP based blocking.
=> API Gateways also support API Queue which holds the requests to the API which cannot be serviced immediately, this is used to prevent the thundering herd issue.

Service Discovery:

We mentioned the API gateway route the request to the correct backend service, however how does it know where each service is located, i.e, what is the communication endpoint of the service?

This information is managed by the Service Discovery component, for example Eureka, Zookeeper.

The Service discovery will keep track of the communication endpoints of the various services, the communication endpoint is basically the IP + Port for communicating with that service.

Service discovery is very important as the microservices can rapidly scale up and scale down.

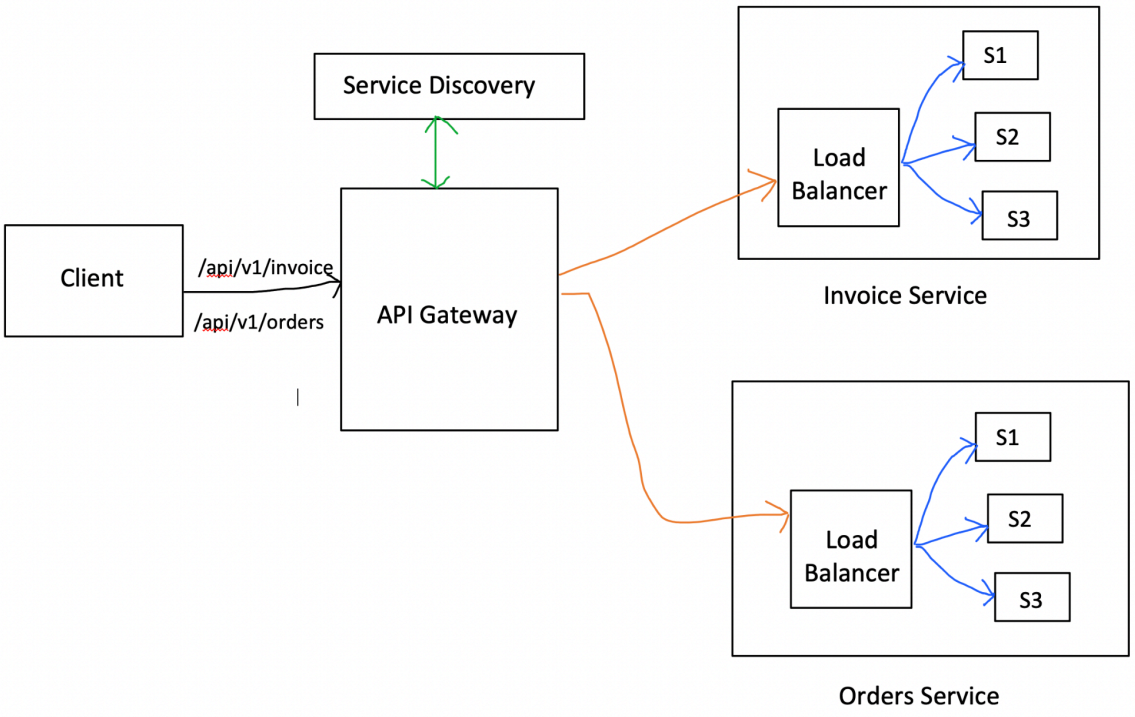
The microservices must register with the Service discovery component. Additionally, the service discovery will perform health checks on all the registered microservices, and track only the healthy ones.

The Service discovery in addition can contain policies which allow it to make smart decisions regarding traffic distribution. For example if we have 2 availability zones, when a request comes in the API gateway needs to route it to the correct backend service, which is running both the data Centers, so it’ll check with the Service discovery as to where it should

route the request, the Service discovery will consider factors like Geographical proximity, latency, compliance etc. and make an informed decision regarding which data center to route traffic it. Finally it'll return this IP+Port to the API gateway.

Example of the Service Discovery:

A request for the API endpoint /api/invoice comes into the API gateway. The API gateway will need to route this request to the Invoice service, so it'll check with Service discovery to get the communication endpoint for the Invoice service. If the Invoice service is running on a single server, the Service discovery returns it's IP+Port, however in most cases we'll have multiple nodes all fronted by a Load Balancer, so Service discovery will return the IP+Port of this Load Balancer, and API gateway will talk to it.



How can the API Gateway handle millions of requests, when we say it is a single point of entry into the system?

The term "single" is misleading here, we can horizontally scale our API gateway across multiple servers, and we load balance the traffic across the different API Gateway instances using DNS based load balancers, like

AWS Route 53, or Azure Traffic Manager. This DNS load balancer has similar intelligence like the Service Discovery.

DNS based load balancers help to route traffic b/w multiple regions.

Protocols and Network Security

- **Networking**
- **Hashing and Cryptographic Hash Functions**
- **DNS**
- **Notes on Encryption**
- **Diffie Helman**
- **OAuth**
- **RPC**
- **Brief Notes on Cryptography**

Networking

Application Layer Protocols:

Client Server Protocol:
- HTTP / HTTPS
- FTP (File Transfer Protocol)
- Web Socket
- SMTP

Note: Web Socket is a client server protocol that enables bi-directional real time communication between a client (for example a web browser) and a server. Consider Web Socket when designing any sort of chat application like WhatsApp.

SMTP is used in conjunction with IMAP. SMTP is used to send emails, while IMAP is used to receive emails, POP3 is not used now. So SMTP is used in conjunction with IMAP.

HTTP / HTTPS, FTP, SMTP and Web Socket rely on TCP as the L4 protocol

Peer2Peer Protocols
- WebRTC: Clients can communicate with each other directly without going through a server.

Transport Layer / Network Layer

=> TCP / IP
=> UDP / IP

IP Protocol Numbers (Quick Reference)

ICMP: 4
TCP: 6
UDP: 17
ICMPv6: 58

https://en.wikipedia.org/wiki/List_of_IP_protocol_numbers

These numbers are used in the IP / IPv6 header.

OSI Model

The Open Systems Interconnect Model is a conceptual model, used to describe how computers communicate to each other over a network. It has seven layers.

Upper Layers:

7. Application Layer
6. Presentation Layer
5. Sessions Layer

Lower Layers:

4. Transport Layer
3. Network Layer
2. Data Link Layer
1. Physical Layer

Each layer adds its own header to the packet, encapsulating it.
L4 Header contains info regarding the L4 protocol, TCP / UDP / RAW and src and destination port numbers.
L3 Header contains info regarding l3 protocol, IPv4 / IPv6 / CLNS and src and destination IP addresses
L2 header contains the MAC address

Application Layer
It provides network services to applications of the end user, it does not provide any service to other layers of OSI model. It receives information from the Presentation layer and provides it to the user in a meaningful way.

Presentation Layer

The Presentation Layer ensures that the information that is sent from the application layer of one system, is readable by the other system. Presentation layer can translate multiple data formats into a common format, for example computers using different encoding schemes.

Session Layer
The Sessions layer establishes, manages and terminates sessions between two communicating hosts. For example, a web server might have many users and it keeps track of them via sessions.

Data is prepared by the upper layers

Data

Transport Layer
The transport layer provinces the following functionality:

1. Segmentation and Reassembly of Data: Breaks large pieces of data into smaller pieces called segments which can be more easily sent over the network, and are less likely to cause transmission problems if errors occur.
2. Provides end to end communication: The transport layer is said to provide end to end communication because it works on both the end hosts. It is responsible for taking packets from the IP layer and delivering them to the correct application using the port numbers. Ports do not identify processes, they identify sockets and each socket is attributed to a single process.
3. Additionally it performs sequencing, checksum acknowledgments and flow control.
4. Important information at this layer includes: which l4 protocol is being used and src and destination port numbers.

Transport layer provides end to end delivery of data, additionally it provides flow control and error recovery. Flow control is the process of adjusting the rate of flow of data from the sender so that the receiver can handle all of it.

Session Multiplexing: Transport layer allows the host to handle multiple traffic streams (or sessions) over a single link. It does so by using port numbers.

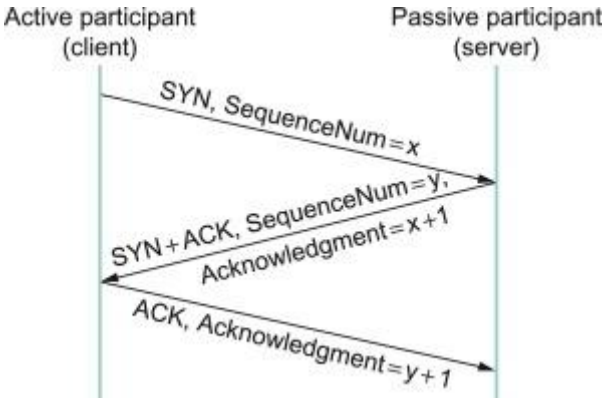
Port Numbers: Port Numbers are used to identify upper layer protocols.
HTTP uses 80
SMTP uses 25
The sender also adds its Port number as the source port in the layer 4 header.
The combination of source and destination ports can be used to track sessions.

Layer 4 Protocols

TCP (Transmission Control Protocol) -

1. TCP is connection oriented, once a connection is established data can be sent bidirectionally over it.
2. TCP carries out sequencing, i.e. it adds sequence numbers to the segments, to ensure they are processed by the receiver in the correct order and that none of the segments are missing.
3. TCP is reliable, the receiver sends acknowledgment back to the sender, lost segments are retransmitted.
4. TCP performs flow control.

TCP establishes connection between 2 hosts via 3-way handshake



The TCP segment is composed of the TCP header and the Data (from upper layers), The TCP header can range in size from 20 - 60 bytes, with 20 bytes being the minimum possible value.

TCP 3-way Handshake

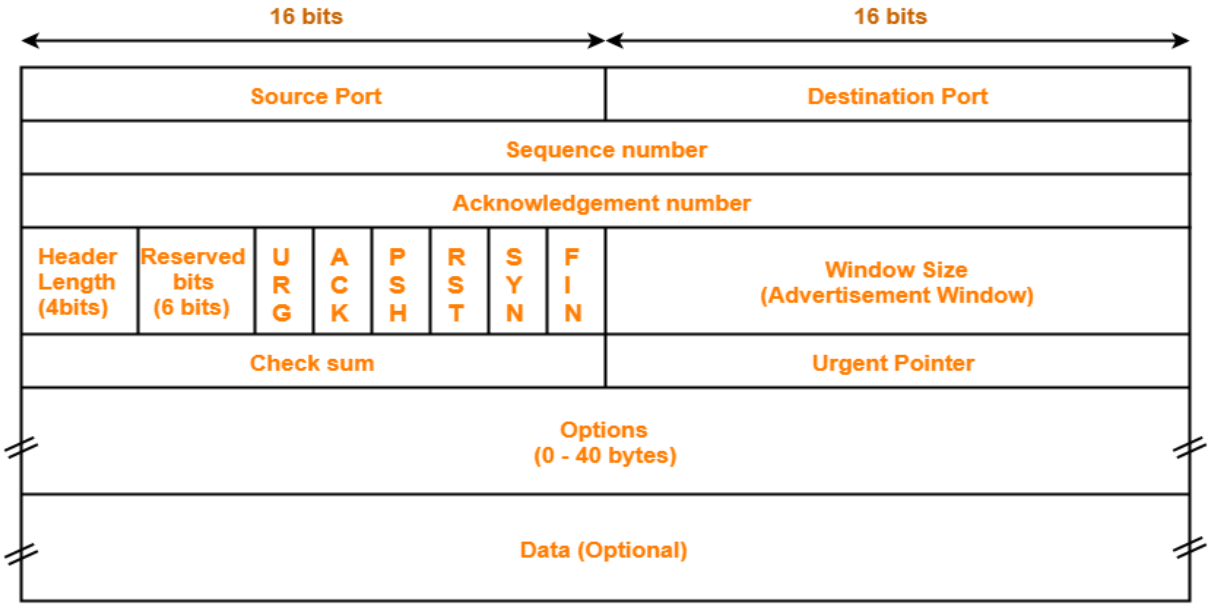
The TCP 3-way handshake is a fundamental process used by the TCP protocol to establish a reliable connection between the client and the server. The handshake ensures that both parties are synchronized and ready for communication.

The steps involved in the handshake:

Step 1: Synchronize (SYN) - The client or the initiator that wants to establish a connection with the server sends a segment with the SYN flag set, to the server (or receiver). The SYN (Synchronize Sequence Number) informs the server that the client wishes to start a communication and with what sequence number the client will start the segments with (i.e. the ISN - initial sequence number)

Step 2: Synchronize-Acknowledge (SYN-ACK) - The server responds with a TCP segment having both the SYN and ACK flags set. This acknowledges the client's SYN request and sends its own ISN (initial sequence number) indicating with what sequence number the server will start the segments with.

Step 3: Acknowledge (ACK) - The client acknowledges the response of the server (SYN-ACK) by sending a TCP segment with the ACK flag set. Now, they both establish a reliable connection, through which actual data transfer will take place.

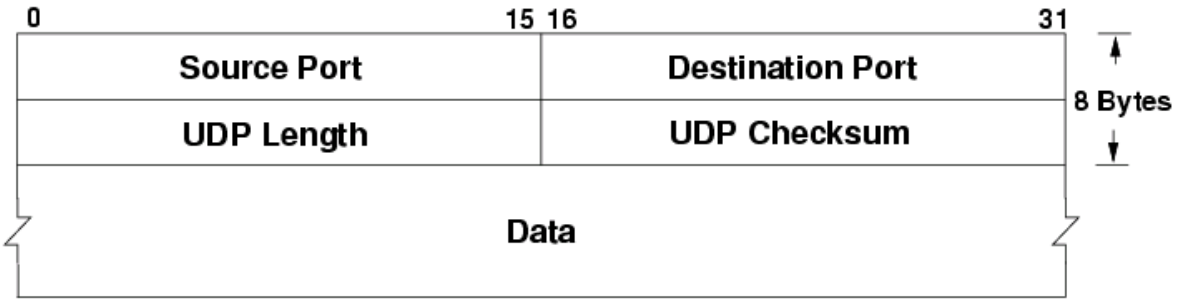


TCP Header

Segment, PDU at Transport layer

UDP (User Datagram Protocol) -

1. UDP performs best effort delivery. It is not connection oriented and hence, there is no handshake mechanism for setting up the connection.
2. UDP does not perform sequencing
3. UDP is not reliable, the receiver does not send acknowledgements to the sender.
4. UDP does not perform flow control.
5. If error detection and recovery is needed, then upper layers need to provide it.



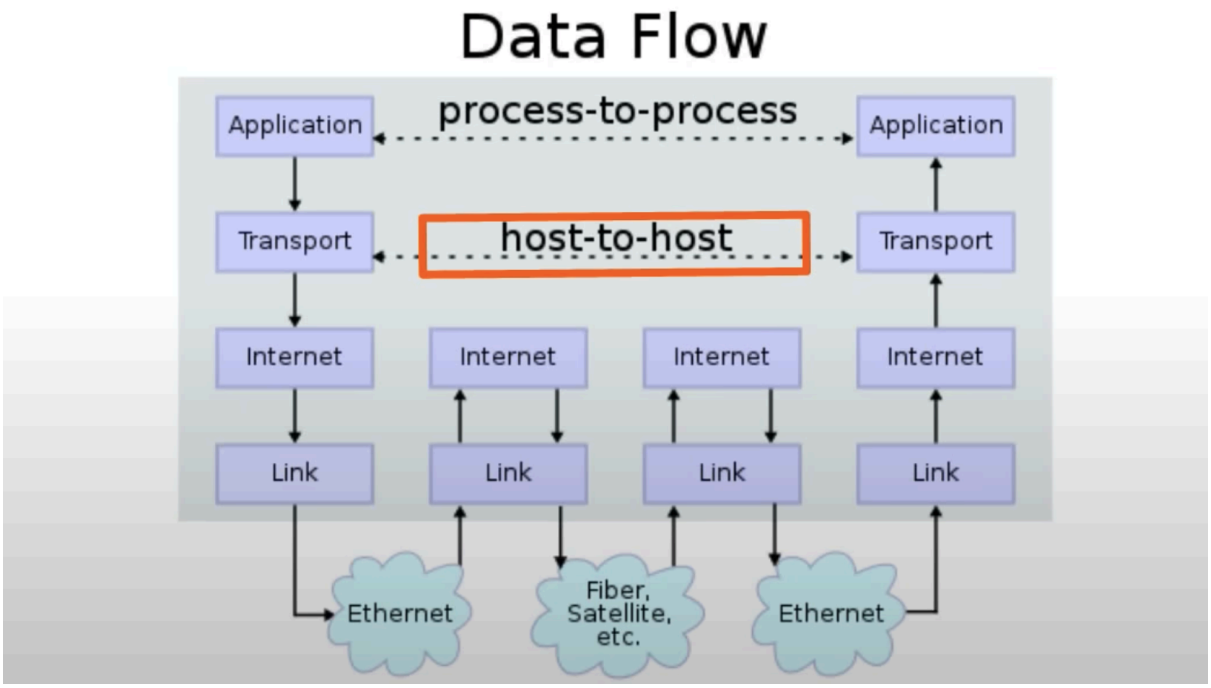
The UDP header is 8 bytes in size. UDP has much less overhead as compared to TCP.

Choice b/w TCP and UDP
=> TCP is preferred for traffic which requires delivery with high reliability.
=> UDP is used in Real time applications (like Voice, video) which are sensitive to delays and cannot afford the extra overhead of TCP.
=> Some applications use both TCP and UDP
=> Still overall TCP is the most common layer 4 protocol.

Ports used by some common applications.

TCP	SSH (22) Telnet (23) FTP(21) HTTP(80) HTTPS(443)
UDP	SNMP(161) TFTP(69) RDP(3389)
Both	DNS(53)

Important Point: Transport Layer works on the end hosts (endpoints), and not in the network itself.
(https://stackoverflow.com/questions/69385655/in-tcp-ip-model-why-transport-layer-functionality-referred-to-host-to-host-but-n#:~:text=The%20main%20purpose%20of%20the,%2C%20checksums%2C%20and%20flow%20control.)



Observed in above diagram, how the segment remains untouched during routing.

Network Layer

The most important information at this layer is the source and destination IP addresses.

1. Network layer provides connectivity between end hosts on different networks. (ie outside of LAN).
2. Provides logical addressing (IP addressing)
3. Path Selection: Identifying the best path for the data to reach the destination from the source
4. Routers operate at L3

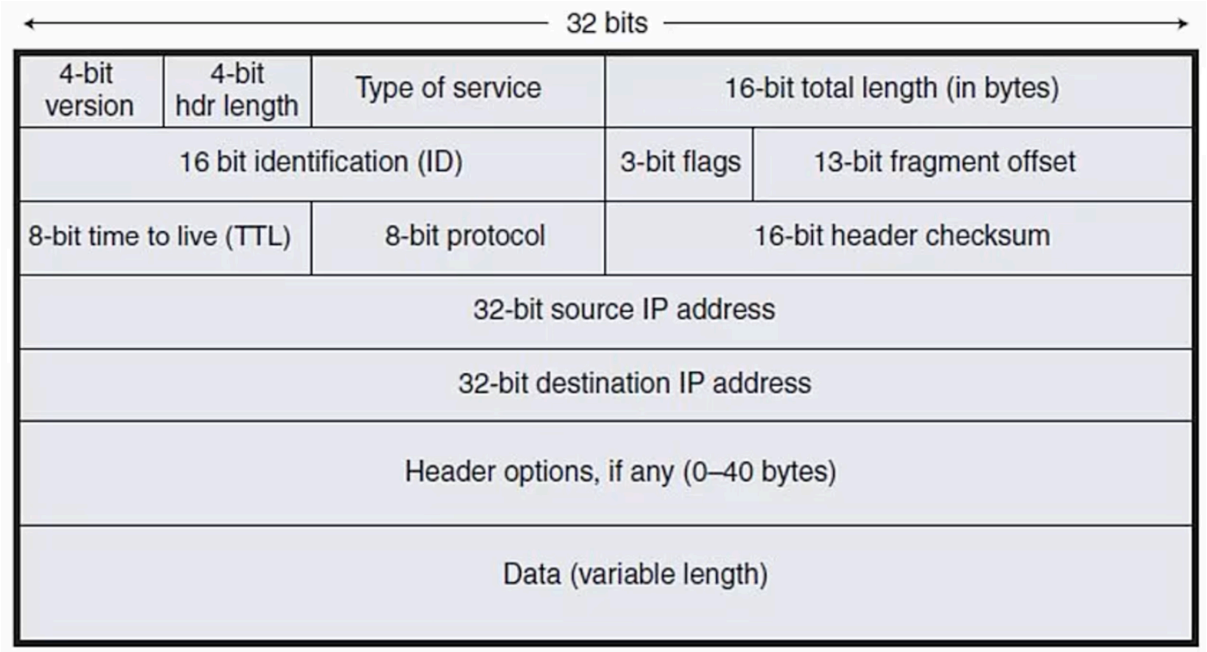
Network layer is responsible for routing of packets to their destination and QoS. Internet Protocol (IP) is the best known L3 protocol, other protocols include ICMP and IPSec.

IP is a connectionless protocol, with no acknowledgement at layer 3.

IP Addressing

IP Addressing is a logical addressing scheme which is implemented at layer 3.
=> Using IP addressing the overall network is divided into smaller subnets.
=> This improves performance and security and makes troubleshooting easier.

IP Header
size: 20 - 60 bytes. 20 bytes being the minimum size.



Data	L4 Header	L3 Header

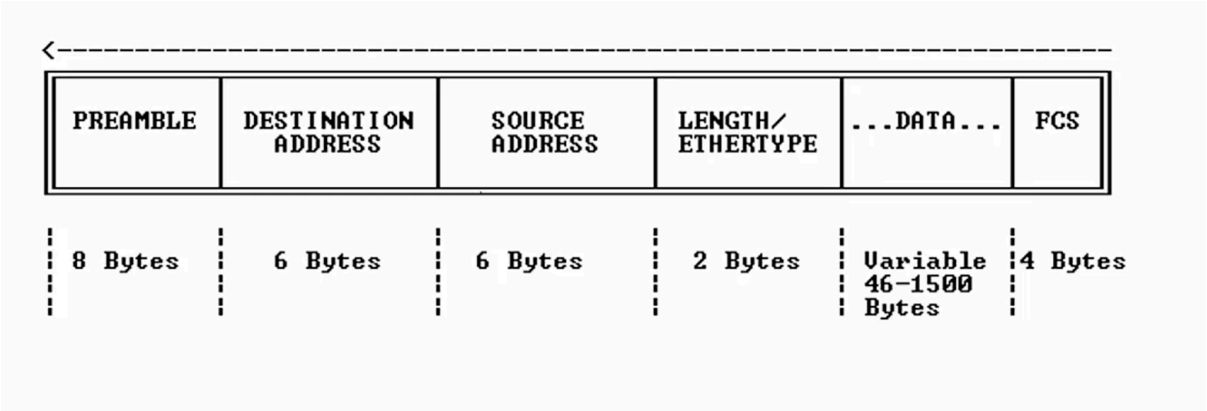
Packet, PDU at Network layer

Data Link Layer
The most important information at this layer is the source and destination layer 2 addresses, for example if Ethernet is the layer 2 technology then the layer 2 header will contain the source and destination MAC addresses.

1. Provides node to node connectivity, for example connectivity b/w PC to switch, switch to router, Router to Router etc.
2. Defines how data is formatted for transmission over the physical medium. (for example Copper UTP cables)
3. Detects and corrects physical layer errors, to ensure reliable delivery of data.
4. Switches operate at layer 2.

Frames are encoded and decoded into bits at layer 2.
=> Ethernet is the layer 2 medium used on Local Area Networks.

An Ethernet frame looks like the following:



Note, this is an Ethernet frame, not the Ethernet header.

The source address, destination address and Length / Ethertype fields combined together are known as the Ethernet header.

The Ethernet header is 14 bytes in size

Other fields in the above diagram are part of the Ethernet frame but not part of the Ethernet header.

The length / Ethertype field indicates the l3 protocol being used, it is a 2-bytes field. Possible values of Ethertype: https://en.wikipedia.org/wiki/EtherType#Values

https://stackoverflow.com/questions/14812979/what-does-an-ethernet-header-look-like

Ethernet header is also called MAC header.

MAC Address (Media Access Control Address)

Ethernet uses 48-bit hexadecimal MAC address
=> First 24 bits are the OUI (Organizationally Unique Identifier) which uniquely identifies the manufacturer of the Ethernet port, it is assigned by IEEE.
=> Next 24 bits are Vendor Specific
=> The burned in address (BIA) on every NIC port in the world is unique.

MAC addressing is just one big flat address space, i.e. there is no logical addressing in case of MAC addresses.

Number of MAC addresses possible = 2^48

Seeing the MAC address configured on the interfaces =>
- Windows: ipconfig /all
- Linux: ifconfig

Cisco Routers: show int <>

Data	L4 Header	L3 Header	L2 Header

Frame, PDU at Data link layer

Physical Layer
Defines the physical characteristics of the medium, which is being used to transfer data b/w devices.
For example: cable specifications, voltage levels, maximum transmission distances etc.
=> Digital bits are converted into electrical signals (for wired connections) or radio signals (for wireless connections).

TCP / IP Stack

The OSI model as mentioned above is conceptual, however TCP/IP stack is actually used in production networks. It has 4 layers

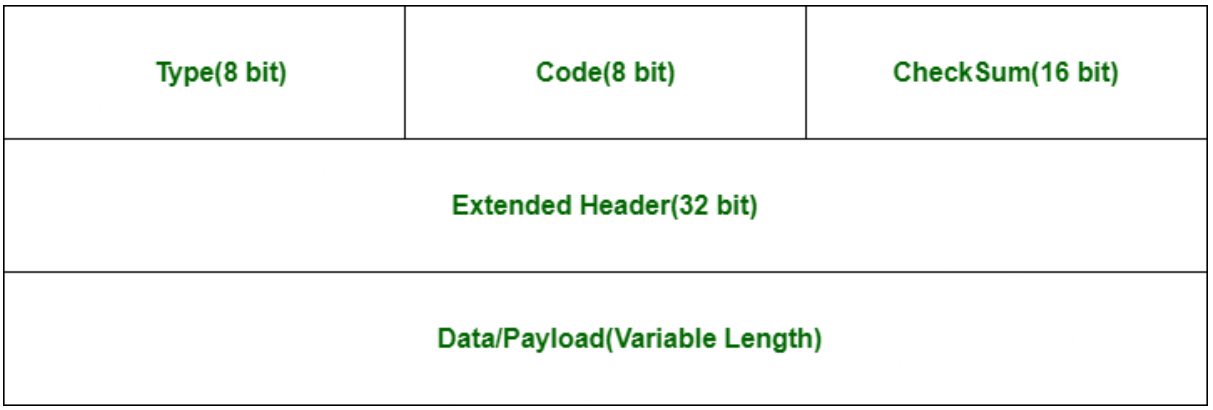
1. Application Layer - Corresponding to Application, Presentation and Session layer of OSI model. The Protocol Data unit is ‘Data’.
2. Transport Layer - Corresponds one to one with the Transport layer of OSI model. PDU is segment
3. Internet Layer - Corresponds one to one with Network Layer of OSI model. PDU is a packet.
4. Network Access Layer - Corresponds to Physical and Data Link Layer of OSI model. PDU is a frame.

ICMP (Internet Control Message Protocol)

ICMP is an auxiliary protocol that is an integral part of IP.
=> All IP devices must support ICMP
=> ICMP messages are encapsulated in the IP packets, with a protocol field of 1, however ICMP is not considered a layer 4 protocol.

ICMP is used by network devices to send error messages, make Routing suggestions etc.

ICMP packets have an 8 byte header and a variable sized data section.



Type: Indicates type of icmp message
Code: Provides additional information for some message types.
Check Sum: Used for error detection in ICMP header and encapsulated data
Extended Header: Contents depend on the type of ICMP message and code.

The header is used to encapsulate ICMP data. The contents depend on the ICMP message type:

1. In ICMP error messages, the data section will contain the IP header + first 8 bytes of the IP payload of the packet that caused the error.
2. In ECHO messages, the data in the ECHO request must be echoed back in ECHO replies.
3. Not all ICMP messages encapsulate data, for example ping can have a 0-byte payload (ICMP header only).

ICMP Message Types

Error Messages
1. Destination Unreachable
2. Redirect
3. Time Exceeded

Query Messages
1. Echo Request
2. Echo Reply

Destination Unreachable message: These messages are used to inform the host that the packet couldn't reach its destination.

ICMP type: 3

ICMP code: In this case the ICMP code tells why the packet couldn't reach the destination.
For example

ICMP Code: 0 => Destination Network unavailable
1 => Destination host unavailable
3 => Destination protocol unavailable.
13 => Communication administratively prohibited (for eg by ACL)

The data section will contain the IP header + first 8 bytes of the IP payload of the packet which couldn't be sent. [ie data section will tell which packet couldn't reach its destination]

Redirect message: Used to inform a host that there is a better route available for a particular destination.
=> Used by Routers to tell a host to use a different Router to reach the destination.

ICMP type: 5
ICMP Code: Indicates type of redirect.
Extended Header: IP address of the Router which should be used.

The data section will contain the iP header + first 8 bytes of the packet which caused the redirect.

Time Exceeded message: These messages are used in 2 conditions

1. Inform the sender that a router couldn't forward the packet because the TTL (time to live) expired.
2. Inform the sender that a device had to discard a fragmented packet, because not all fragments arrived in time.

ICMP Type: 11
ICMP Code:

0 => TTL exceeded in transit
1 => Fragment reassembly time exceeded.

The data section will contain the iP header + first 8 bytes of this packet, which caused the issue.

Each Request and Echo Replies (Ping)
Ping uses 2 ICMP message types, ICMP Echo Requests and ICMP Echo Replies.
Echo Request: ICMP Type = 8
Echo Reply: ICMP Type = 0

If a host receives an Echo Request it should send an Echo Reply message to the sender of the request.

ICMP Code: 0

The extended header is divided into 2 fields: The identifier and the sequence number.

Identifier: Used by a device to keep track of the pings it sends.
=> Incremented by one after a series of pings is sent. (not per ECHO request, it is increment per ping command)
=> Cisco Routers send 5 Echo Requests as part of one ping command.

Example:

ping 1.1.1.2 = SEND 5 ECHO Requests with identifier 0
ping 1.1.1.2 = SEND 5 ECHO Requests with Identifier 1

The Reply message will use the same identifier as the request message.

Sequence Number: Used to keep track of Request and Replies exchanged in a single series of pings.

Example:

Request 1 = Seq 0, Reply 1 = Seq 0
Request 2 = Seq 1, Reply 2 = Seq 1
Request 3 = Seq 2, Reply 3 = Seq 2

=> The entire payload (data section) of the request must be sent back in the reply.
=> However, a ping can be sent with no payload. In this case the message size will be 28 bytes (IPv4 + ICMP).
Summary of ICMP Type and code for different messages.

Type	Code	Description
0 – Echo Reply	0	(used for ping)
3 – Destination Unreachable	0	Destination Network Unreachable
	1	Destination Host Unreachable
	2	Destination Protocol Unreachable
	3	Destination Port Unreachable
	4	Fragmentation required, but DF-bit set
	13	Communication Administratively Prohibited
5 - Redirect	0	Redirect for Network
	1	Redirect for Host
	2	Redirect for ToS & Network
	3	Redirect for ToS & Host
8 – Echo Request	0	(used for ping)
11 – Time Exceeded	0	TTL expired in transit
	1	Fragment reassembly time exceeded

ICMPv6
ICMPv6 is ICMP's implementation for use with IPv6.
=> Like ICMP and IPv4, ICMPv6 is an integral part of IPv6.

ICMPv6 uses an IP protocol number of 58.

Some ICMPv6 types:

- 1 = Destination Unreachable
- 2 = Packet too big
- 3 = Time exceeded
- 128 = Echo Request
- 129 = Echo Rely

How does Ping Work:
https://images.globalknowledge.com/wwwimages/whitepaperpdf/WP_Mays_Ping.pdf

Switches

Switches are used to provide connectivity between hosts which are on the same LAN.
ie end hosts in Local Area network can communicate with each other via a Switch (previously hubs were used). Switches are layer 2 devices.

Switches have many interfaces / ports to which the end hosts can connect to. Switches can be connected to each other as well. Ethernet cables are used for connecting a switch to the end hosts or a switch to another switch.

Hub	Switch
Operates at layer 1	Operates at layer 2, and is layer 1 aware as well as they have physical ports too.
Hubs operate in half-duplex, i.e. a host cannot send and receive data at the same time, it can only do one of the two.	Operates in full duplex, a host can send and receive data at the same time.
All the hosts share the same collision domain, hence only one host can transmit at a time.	Each host has a dedicated collision domain
CSMA/CD [Carrier Sense Multiple Access with Collision Detection] is used for detecting collisions.	No collision detection needed.

Hubs operate at layer 1, so they are not MAC address aware. Hence when a frame comes the hub will flood it out on all ports except the one it was received on. As a result of which all the attached hosts must process all the packets.

When a frame is received, the switch will check, look at the source MAC address, and add it to its MAC address table.
=> The MAC address table is a mapping b/w the MAC address and the port on which the frame came in.
If a unicast frame is later received with a known destination MAC address (i.e. the destination MAC is in the MAC address table), then the switch will only forward the frame out on the relevant port only.
This is better for performance and security, as the frames go only where they are required and the other hosts don’t need to do additional processing.

Whenever the switch receives a broadcast frame or an unicast frame with an unknown destination MAC address (i.e. the MAC is not in the MAC address table), then the switch will flood the frame out on all ports except the one it was received on.

Routers

Network layer provides connectivity b/w end hosts on different networks. Routers are Network Layer (L3) devices:

=> Routers are required if we want to send traffic from one subnet to another.

=> Routers know the paths to get to different subnets.

=> Routers operate at layers 2 and 1 as well, and have awareness up to layer 7

Routers	Switches
Layer 3 device	Layer 2 device
Can route traffic between different networks	Can switch traffic between hosts on the same LAN (Local area network)
Routers support many types of interfaces, for eg. Ethernet, ADSL, Serial etc.	Switches generally support only Ethernet interfaces, however Switches have more ports than Routers.
Don’t forward broadcast traffic	Forward broadcast traffic.

Layer 3 Switches: Advanced Switches are Layer 3 aware and can be used to route traffic between different IP subnets. However like traditional switches they generally only support Ethernet interfaces and have more ports than Routers.

Domain Name System (DNS)

Domain name System is used to resolve a Fully Qualified Domain (FQDN) into an IP address. Eg, of FQDN: www.cisco.com

=> Enterprises generally have an internal DNS server which can resolve the IP addresses of internal hosts.

=> Hosts send DNS queries to their DNS server.

=> If the internal DNS server cannot resolve a query, it will forward the query to a public DNS server on the internet.

=> DNS requests are sent using UDP port 53. (however DNS can fail over to TCP as well)

Address Resolution Protocol (ARP)

Just like DNS maps FQDN (Fully qualified domain name) to IP address, ARP is used to map IP addresses to MAC addresses.

Where do DNS and ARP come into play?
When the sender is creating the packet it wants to send. The sender starts at the top layers, from where it gets the data to be sent in the packet. Next the sender adds L4 header (source, destination port numbers, layer4 protocol).
The next step is the addition of the L3 IP header, which has fields for source and destination IPs. It is rare that we directly have the value for the destination IP, instead most of the time, FQDNs are used. However we cannot put the FQDN as is in the packet l3 header. We first need to convert the FQDN into a normal IP, this is done via DNS. Once we get the IP address corresponding to this FQDN, we put it into the l3 header. Basically DNS maps destination FQDN to destination IP.

Next comes the layer 2 header. Which contains the source and destination MAC addresses, but the sender doesn't know the MAC address of the destination, here ARP comes into play as it can map an IP address to the corresponding MAC address. Once we get the MAC address for the destination using ARP, we can put those into the header. Basically ARP maps the destination IP address to the destination MAC address.

So in summary, the sender needs to provide among other things, destination IP, destination MAC and destination port number.

ARP replies are saved in the ARP cache, so that the host doesn't need to send ARP requests every time it wants to communicate.

View arp cache
Win: arp -a
Linux: arp -n

https://en.wikipedia.org/wiki/EtherType

Routing

A router has 2 mail functions:

1. Finding the best path to available networks
2. Forwarding traffic to those networks

Routing Table: The best available path or paths to a destination network are stored in the Routing table, and will be used for forwarding traffic.
Routing table consists of:

1. Directly Connected Routes
2. Routes configured statically
3. Routes learned dynamically via Routing protocols.

Connected Routes: The Networks configured on the Router interfaces.
Local Routes: Actual IP address configured on the Router interfaces.

Static Routes: If the router needs to forward traffic to a network, which it is not directly connected to, then it needs to know how to get there to forward the traffic. This can be achieved by either:

1. The admin configuring static route to the destination on the router
2. Learning routes via a routing protocol.

Route Selection:
When there are overlapping routes, the one with the longest prefix match will be selected, in other words the most specific one will be selected.

Load Balancing:
If there are multiple paths to the same destination with the same prefix, then the Router will load balancer between them.

Traffic for the same flow will always take the same path, i.e. there will be no load balancing for the same flow. In summary: the traffic for the same flow will always go over the same path.

IPV6 Addressing

Note: IP Phones use embedded IP addresses for communication, if we assign a private IP address to the IP phones and use NAT to convert them to public IPs, then communication won't be possible, as even though the phones will have connectivity over their NATd public IP addresses, they still won't have connectivity over their embedded IPs (as they are private IPs). The IP phones are unaware of the network translation and hence they keep using the private IPs assigned to them.

NAT breaks the way how IP normally works, and it might affect devices which work with embedded IPs or port numbers.

IPv6 address size = 128bits [IPv4 address size = 32bits]
IPv6 supports more than 7.9x10^28 times the devices as compared to IPv4

An interface can be configured with both IPv4 and IPv6 addresses, this is called the Dual Stack implementation.

IPv6 address format

As mentioned before IPv6 address size if 128 bits (128 / 8 = 16bytes)

The address is written as

X:X:X:X:X:X:X:X

Where each X is a 16 bit, hexadecimal field.

Note: 1 hex digit = 4bits, 4hex digits = 16 bits.
Eg 2001:db8:3333:4444:5555:6666:7777:8888

Hex values = 0-9, A-F

Each 16 bit segment of an IPv6 address is known as ‘hextet’ or ‘pieces’

Address Shortening
=> Leading zeros in each segment (field) can be removed.
=> Successive all zero fields can be shortened to “::”

Eg.
Original IP: 2001:0DB8:0000:0001:0000:0000:0000:0001
After removing leading zeros in each field
=> 2001:DB8:0:1:0:0:0:1

After shortening all successive zero fields to “::”
New IP: 2001:DB8:0:1::1

All 3 are valid IPs

Note: Successive all zero fields can be shortened only once in an IPv6 address.

Types of IPv6 address

1. Global Unicast
2. Link Local
3. Unique Local

1. Global Unicast Addresses
Global Unicast Addresses are similar to IPv4 public addresses, they are assigned to an individual host and have global reachability, they are assigned from the range 2000::/3

A common assignment for a company is a /48 block. Eg 2001:10:10::/48

Every individual (host) should use a /64 mask as per IPv6 standards.

Eg, if a company is assigned a /48 block and it uses /64 for host addresses, then -
Number of subnets = 2^(64-48) = 2^16 = 65536
Number of hosts = (2^64) - 2

Broadcast and multicast addresses:
IPv4 supports broadcast to all hosts on 255.255.255.255
Routers do not forward broadcast traffic so this stays on the local subnet. However IPv6 does not support broadcast, it instead supports Multicast.

For eg. *ff02::1 is the multicast address for all hosts on the local subnet, which is functionally equivalent to broadcast to 255.255.255.255*
Many services which use broadcast on IPv4, use more specific multicast addresses in IPv6, for example DHCP uses the multicast address ff05::1:3

Configuration of IPv6 addresses
conf
int Gi0/0/0/0
no shut
ipv4 add 1.1.1.2/24 => Dual stack implementation
ipv6 add 2001::9999/64
commit
end

Verification

show ipv6 interface brief

RP/0/0/CPU0:ios#sh ipv6 int br
Tue Apr 2 09:53:30.339 IST
GigabitEthernet0/0/0/0 [Up/Up]
fe80::b5:1ff:fe76:dff8
2001::8888

Note: IPv6 address is not case sensitive.

EUI-64 Addresses

A Cisco Router can generate the full IPv6 address for itself when given the interface (for which the generated) and the /64 network to use, i.e. it will automatically generate the host portion of the address.
However, remember MAC address is 48 bits, while the host portion of IPv6 address is 64 bits, so we need to pad up the MAC address to bring it up to 64 bits. To do this

FF:FE is injected into the middle of the MAC address and the 7th bit is inverted, this will generate the host part.

EUI-64 Address Configuration

```
RP/0/0/CPU0:ios(config)#int GigabitEthernet 0/0/0/2
RP/0/0/CPU0:ios(config-if)#no shut
RP/0/0/CPU0:ios(config-if)#ipv6 add 2001:db8:2:1::/64 eui-64
RP/0/0/CPU0:ios(config-if)#commit
Tue Apr  2 10:13:40.799 IST
RP/0/0/CPU0:ios(config-if)#end

RP/0/0/CPU0:ios#show ipv6 int br
Tue Apr  2 10:13:46.914 IST
GigabitEthernet0/0/0/0 [Up/Up]
   unassigned
GigabitEthernet0/0/0/1 [Shutdown/Down]
   unassigned
GigabitEthernet0/0/0/2 [Up/Up]
   fe80::7b:8aff:fe09:e20
   2001:db8:2:1:7b:8aff:fe09:e20
```

Note:
=> If we try to configure eui-64 addresses on a non-Ethernet interface (for example serial), then the Router will borrow the MAC address from the first Ethernet port, and use it to generate the host part of the address.
=> EUI-64 addresses are generally used for end hosts (like desktop pc's)

2. Unique Local Addresses
They are similar to IPv4 private addresses (RFC 1918), i.e. they are not publicly reachable.
=> These addresses are assigned from the range FC00::/7
=> Hosts are assigned /64 addresses.

3. Link Local Addresses
Link local addresses are valid for communication on that link only, i.e. they cannot communicate to hosts on other links. In simpler words, the traffic will not be forwarded on the other side of the router (other link).

=> These addresses are assigned from the range FE80::/10 - FEB0::/10
=> Host are assigned /64 addresses

Link local addresses are used for traffic which should not be forwarded beyond the local link, for example Routing protocol hello packets.

=> Link local addresses are mandatory on Cisco Routers
=> EUI-64 Link local addresses are automatically generated (These addresses will be auto-generated when the interface has been configured with a Global unicast address, either manually or using eui-64)
=> Can be overwritten, by manually configuring a link local address.

Example:

No IPv6 Configuration

```
RP/0/0/CPU0:ios#show ipv6 int br
Tue Apr  2 11:26:00.677 IST
GigabitEthernet0/0/0/0 [Up/Up]
   unassigned
GigabitEthernet0/0/0/1 [Shutdown/Down]
   unassigned
GigabitEthernet0/0/0/2 [Up/Up]
   unassigned
GigabitEthernet0/0/0/3 [Shutdown/Down]
   unassigned
GigabitEthernet0/0/0/4 [Shutdown/Down]
   unassigned
GigabitEthernet0/1/0/0 [Shutdown/Down]
   unassigned
GigabitEthernet0/1/0/1 [Shutdown/Down]
   unassigned
GigabitEthernet0/1/0/2 [Shutdown/Down]
   unassigned
GigabitEthernet0/1/0/3 [Shutdown/Down]
   unassigned
GigabitEthernet0/1/0/4 [Shutdown/Down]
   unassigned
```

```
RP/0/0/CPU0:ios#conf
Tue Apr  2 11:26:05.536 IST
RP/0/0/CPU0:ios(config)#int GigabitEthernet 0/0/0/0
RP/0/0/CPU0:ios(config-if)#no shut
RP/0/0/CPU0:ios(config-if)#ipv6 add 2001:db8:2:2::/64 eui-64
RP/0/0/CPU0:ios(config-if)#commit
Tue Apr  2 11:26:45.548 IST
```

```
RP/0/0/CPU0:ios(config-if)#end



RP/0/0/CPU0:ios#show ipv6 int br
Tue Apr  2 11:26:52.100 IST
GigabitEthernet0/0/0/0 [Up/Up]
   fe80::b5:1ff:fe76:dff8     => link local address
   2001:db8:2:2:b5:1ff:fe76:dff8
GigabitEthernet0/0/0/1 [Shutdown/Down]
   unassigned
…….
```

Note: Since Link Local Addresses are valid only on the local link, hence we can have multiple interfaces configured with the same link local address if they are on different links.

Configuring link local address:

```
RP/0/0/CPU0:ios(config)#int GigabitEthernet 0/0/0/0
RP/0/0/CPU0:ios(config-if)#ipv6 add feb0::5 link-local [no need to specify /64 here, it's implicit]
RP/0/0/CPU0:ios(config-if)#commit
Tue Apr  2 11:34:58.036 IST
RP/0/0/CPU0:ios(config-if)#end
RP/0/0/CPU0:ios#show ipv6 int br
Tue Apr  2 11:35:02.099 IST
GigabitEthernet0/0/0/0 [Up/Up]
   feb0::5
   2001:db8:2:2::5
GigabitEthernet0/0/0/1 [Shutdown/Down]
   unassigned
……
```

Note:
An interface can have a maximum of 2 IPv4 addresses, one of which needs to be explicitly configured as secondary. In comparison an interface can have any number of IPv6 addresses.

Summary of different types of address

Global Unicast	Unique Local	Link Local

Optional	Optional	Mandatory on IPv6 enabled interfaces
Assigned from the range 2000::/3	Assigned from the range FC00::/7	Assigned from the range FE80::/10 - FEB0::/10

One link local address for routing protocols and One global unicast address for normal routing is typical.

SLAAC (Stateless Address Auto Configuration)

Hosts can be assigned IPv6 addresses via static addressing, DHCPv6 or SLAAC.
DHCP is stateful, ie it maintains a mapping b/w IP addresses given out and the MAC address of the device to which it was given.
SLAAC is stateless.

With SLAAC, hosts learn the /64 subnet (network address) their interface is on and use this information to generate their own IPv6 EUI-64 addresses. However for privacy concerns, here the host part is randomized.
This is stateless because it does not track which hosts have which IP address.

When a global unicast address is configured on an interface, it sends out Router Advertisements to advertise the network prefix by default. These advertisements are sent to the ‘All Nodes’ multicast address with the link local address of the Router’s interface as the source. In addition to this, the hosts can also request information about the network prefix (subnet) via Router Solicitations.
Router Advertisements and Router Solicitations are ICMP messages.

Apart from telling the subnet (on which the IPv6 address needs to be generated by the host), Router also tells the hosts to use itself (the Router) as their default gateway. No other information is provided. For example information regarding DNS server. To provide information related to DNS, a DHCP server will still be needed.

If the IP addresses are assigned by SLAAC and the DNS server is assigned by DHCP, then the overall combination is still stateless.

Note:
=> Unknown or unspecified address (::)
=> ::/0 is equivalent to the default route 0.0.0.0 0.0.0.0 in ipv4

=> :: is used as the source when an interface is trying to acquire an IPv6 address via SLAAC or DHCP.

=> DHCP is applicable to both IPv4 and IPv6 addresses, however SLAAC is exclusive to IPv6

Neighbor Discovery

Neighbor Discovery protocol is the IPv6 version of ARP, and works in the same way. Instead of ARP requests and replies, ND uses ICMP Neighbor Advertisements and Neighbor Solicitations.
Neighbor Solicitation messages (ARP requests) are sent to the Solicited-node multicast address, which reaches all hosts on the subnet.

HTTP / HTTPS

HTTP or the Hypertext transfer protocol is the foundation of any data exchange on the web, it is an application layer protocol. HTTP protocol is used for fetching resources such as HTML documents.

It is a client server protocol, meaning that the communication is initiated by the client, in the form of a request to the server, and the server responds to the client in the form of a response, a client could be a web browser for example.

Clients and servers communicate by exchanging individual messages. HTTP runs over TCP, on port 80.

Further HTTP is not restricted to fetching HTML documents, it can be used to fetch other resources like images, videos etc, or post content to servers.

When we try to load a page on the web browser, the browser will send a request to the server to fetch the HTML document for that page, the browser will then parse the file making additional requests for sub-resources contained within the file (example image, videos, css, js scripts). The browser finally combines all these resources to present the complete document, the web page.

HTTP request, response messages are human-readable.

HTTP is a stateless protocol, i.e. it doesn't store any information regarding the client, so http will treat 2 successive requests on the same connection as if there is no link between them. This is problematic, for example if a user logs in to some website and tries to access some information or perform any action then HTTP will treat the second request totally independently, and since it doesn't store any state on the user hence the user will asked to login again, this is because HTTP has not stored any session data to indicate that the client is already logged in.

This is where a cookie comes in. When a user logs onto a website an entry will be created in the sessions database for that user, the sessions table is generally a No-sql database which stores information regarding the active user sessions. Among other things it stores the session ID. Now when a user is logged in, the website will create a cookie containing the

session ID and store it on the file system of the user's machine. Next time the user opens the website, it'll find the cookie and send it to the application, the application will get the session ID from the cookie and check if it exists in the sessions database, if it does the user will be logged in automatically (without having to provide login details again). A cookie has an expiration period, beyond which the user will need to relogin to the website.

Q. How is the cookie sent to the application?
Cookies are added as part of the HTTP header

Cookies make it possible to establish and maintain sessions, despite the stateless nature of HTTP.

Note: Each cookie has a unique ID, and a website can only access its cookies.

HTTP Flow

When a client wants to communicate with a server, it performs the following steps:

1. Open a TCP connection: The TCP connection will be used to send requests and receive responses.
2. Send a http request message

   GET / HTTP/1.1
   Host: developer.mozilla.org
   Accept-Language: fr

3. Parse the response sent by the server
   HTTP/1.1 200 OK
   Date: Sat, 09 Oct 2010 14:28:02 GMT
   Server: Apache
   Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
   ETag: "51142bc1-7449-479b075b2891b"
   Accept-Ranges: bytes
   Content-Length: 29769
   Content-Type: text/html

<!DOCTYPE html>… (here come the 29769 bytes of the requested web page)

4. Close the TCP connection or reuse it for further requests.

What does an http request message consists of

1. HTTP Request Method: GET, POST, PUT, DELETE
2. Version of the HTTP protocol being used, for example 1.1 or 2
3. Path of the resource
4. Optional headers
5. A body, some request methods like POST have a body.

HTTP Status Codes
Common HTTP Status Codes

200 - OK
201 - New Resource was created, returned on successful POST requests
301 - Permanently Moved
302 - Temporarily Moved / Found
400 - Bad Request (Incorrect Syntax)
403 - Forbidden
404 - Not Found
429 - Rate Limited, Too many requests
500 - Internal Server Error
503 - Service Unavailable

ResponseCodes

1xx Informational
[Server sends some information to the client]

2xx Success
[Request from client is received and processed successfully]

Code	Reason	Used by which HTTP methods
200	Ok	GET POST (idempotent calls)

201	Created, Request is successful and new resource is created.	POST
202	Accepted (Task will go on in the background, delayed execution)	POST
204	No Content, Delete request is successful and no data is returned in the response body.	DELETE

3xx Redirection
[Client must take additional action to complete the request]

301 - Moved Permanently
302 - Found [Temporarily moved]

When the client sees a return code like 301, it will understand that it needs to go to another URL to get the desired information. The information regarding the other URL is present in the header, the 'Location' field in the HTTP header specifies where the client needs to go. Clients are smart, and they perform this redirection automatically.

304 - Modified

4xx Validation Error
[Client passed a wrong request to the server]
400 - Bad Request
401 - Unauthorized (Authentication needed)
403 - Forbidden
404 - Not Found
405 - Method Not Allowed (hitting a get api with post request)
429 - Too many requests

5xx Server Error
500 - Internal Server Error
502 - Bad Gateway

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Hashing and Cryptographic Hash Functions

Searching in arrays at best takes O(logN) time, which can become problematic as the dataset becomes larger and larger, this is where hashing comes in which enables us to perform fast lookups in near constant time, hashing is the process of transforming a variable sized input into a fixed sized output, it involves using a "hash function" to transform the input key or data to a hash key (also called hash code), this hash key is the index or the location wherein the underlying data "value" is actually stored.

key -> hash(key) -> hash_key

Components of hashing:

a. Key: A key can be anything int, string, custom object etc., this is the value inputted to the hash function which produces the corresponding hash key.
b. Hash function: The hash function is a mathematical formula that takes as input the key or data and produces a unique value, this value also called the hash value is the index where the data is stored in the hash table (index).
c. Hash Table: The underlying data is stored in the hash table, which stores the data in an associative manner in an array, each value is mapped to a unique index in this table by the hash function.

Example of hash function

$$h(x) = f(x) \% 4$$

As can be seen the values generated by this hash function can be 0, 1, 2 or 3. So it transforms a variable sized input to a fixed size output, these values will be used as an index to the hash table and will be used for storing and retrieving values.

Properties of a hash function:

1. Efficiently Computable
2. Should uniformly distribute keys
3. Should minimize collisions
4. Should have a low load factor

5. For the same key must always produce the same hash value (consistency).

Methods for designing / creating the hash function

1. Division Method: h(k) = k % m, where k is the input key and m is a prime number. Simple to implement, but results in poor distribution if m is not chosen wisely.

2. Multiplication Method: h(k) = floor(m(kA % 1)), where k is the input key, m is a prime number and A ∈ (0, 1). This method is less sensitive to choice of m hence produces a more even distribution, however it is more complex than the division method.

3. Mid-Square Method: In this method the key is squared and the middles digits of the result are taken as the hash value. This method produces a good distribution of hash values, however it is computationally more expensive.

4. Folding Method: The key is divided into equal parts, then the parts are summed up and then take the modulo with m of the sum, this method depends heavily on the choice of partitioning scheme.

Cryptographic hash algorithms like CRC32, MD5, SHA1, SHA256 are used in Network Security for authentication and data integrity checks. The hash function takes an input of any size and returns a hash value called message digest. The size of the message digest can vary from algorithm to algorithm.

For example MD5 generates a 128 bit message digest.
SHA1 generates a 160 bit message digest.

Use Case: Suppose you are downloading some file from the internet and want to verify it's integrity, then you can compare the SHA1 hash value of the file with what the server claims to be providing, If the hash values do not match, then the file might be malicious, or might have been changed via a man in the middle attack for example. Hence SHA1 / MD5 provide data security.

MD5, SHA1 are older algorithms and not used widely anymore, instead more modern algorithms like SHA256 and SHA512 are used.

https://stackoverflow.com/questions/6220756/why-hashing-functions-like-sha1-use-only-up-to-16-different-char-hexadecimal

Collisions

Collision is said to occur when two or more items / objects are mapped to the same hash key, i.e the hash function produces the same value for these items. Now assume we are trying to insert an item, however the slot identified by the hash function has already been taken by another item (which got mapped to the same key), this is known as a collision. The hashing process generates a small number for a big key hence there is a probability that two keys are mapped to the same value by the hash function.

Methods for Handling Collisions

1. Separate Chaining (Open Hashing)
2. Open Addressing (Closed Hashing)

Separate Chaining

The idea is to make each cell of the hash table point to a linked list of records that have the same hash key. Chaining requires additional memory outside the table. Hence the separate chaining (or Open Hashing) method is used as a collision resolution technique.

Suppose we have the hash function: h(x) = f(x) % 5
and the following data to insert: [12, 15, 22, 35, 37], then with chaining the data will be stored as follows:

0	-> 15 -> 35
1	-> NULL
2	-> 12 -> 22 -> 37
3	-> NULL
4	-> NULL

Open Addressing
In case of open addressing all elements are stored in the hash table itself, each table entry will either store a record or be NULL.

Types of Open Addressing

1. Linear Probing: Let hash(x) be the slot index computed by the hash function and S be the table size.
   If slot (hash(x) % S) is full, then we try (hash(x) + 1) % S
   If slot (hash(x) + 1) % S is full then we try (hash(x) + 2) % S
   If slot (hash(x) + 2) % S is full then we try (hash(x) + 3) % S
   and so on …

2. Quadratic Probing: Let hash(x) be the slot index computed by the hash function and S be the table size.
   If slot (hash(x)) % S is full, then we try (hash(x) + 1*1) % S
   If slot (hash(x) + 1*1) % S is full, then we try (hash(x) + 2*2) % S
   If slot (hash(x) + 2*2) % S is full, then we try (hash(x) + 3*3) % S
   and so on…

3. Double Hashing: We use an additional hash function hash2(x) and look for i*hash2(x) slot where i >= 1.
   If slot (hash(x)) % S is full, then try (hash(x) + 1 * hash2(x)) % S
   If slot (hash(x) + 1 * hash2(x)) % S is full, try (hash(x) + 2 * hash2(x)) % S
   If slot (hash(x) + 2 * hash2(x)) % S is full, try (hash(x) + 3 * hash2(x)) % S
   and so on…

Linear probing is simple to implement however it can lead to clustering, i.e. a situation where the keys are stored in long contiguous runs, which degrades performance.
Quadratic probing is more spaced out, but it can also lead to clustering and can lead to a situation where some slots are never checked.
Double Hashing is more complex, but can provide a more even distribution of keys, hence better performance.

=> In terms of caching, open addressing performs better than Chaining.

Universal Hashing: Using a family of hash functions to map the key to a hash value, using multiple hash functions minimizes chances of collisions for any given input set.

Load Factor and Rehashing

Load factor is defined as the total number of items in the hash table divided by the size of the hash table.

$$load\ factor\ =\ (Total\ Number\ of\ items\ in\ the\ hash\ table)\ /\ (Size\ of\ the\ hash\ table)$$

If the load factor exceeds a predefined value (generally 0.75), then rehashing is performed. Rehashing involves increasing the size of the hash table array (doubling it), and all the values are hashed again by using a new hash function and stored in the new hash table array, this decreases the load factor, hence decreasing the complexity.

DNS

DNS (Domain Name System) - When an application accesses a website it does so by using the website's domain name. The DNS is a service which is used to convert these Domain Names to IP addresses.

Example: Route 53, GoDaddy etc.

OAuth

OAuth is an Authorization framework. It is used to enable secure third-party access to user protected data.

For example, when logging into a website using gmail, we authorize that third party website to access the gmail's user protected data.

OAuth Roles

=> Owner: In the above example it'll be the owner of the Gmail account.
=> Client: The client is the entity which initiates the request for user protected data. In the above example, the client is the third party website.
=> Authorization Server: In the above example, Gmail will have a component called the authorization server which takes care of authorizing.
=> Resource Hosting Server: In the above example, Gmail hosts all of the user protected data like Name, email, DOB etc. Hence it is the Resource Hosting Server.

So in the above example Gmail has separate components which act as Authorization servers or Resource Hosting servers.

RPC

RPC or Remote Procedure calls are used to perform inter service communication over a network. RPC is designed to make a network call look just like a local function call; it does so by abstracting details such as Serializing / Deserializing and transport.

=> RPC (Remote Procedure Call) protocol is used for inter-service communications, which are physically running on different machines. So RPC provides a way for a process on one machine to communicate with / or request a service from a process on another machine.

=> RPC provides a layer of abstraction, the client does not need to worry about the details of the connection, and the client can work with remote procedures as if they are local procedures. So RPC allows us to invoke functions / methods on a remote server.

Stubs: A stub is an integral component of RPC. Take an example where we have an authentication service and a notification service. The authentication service wants to send an OTP to the user, it does through the notification service. For doing this it'll call the remote procedure notification_otp(email). Now this invocation encapsulates the following details:

- The routine to call on the notification service end (notification_otp)
- The email to which the OTP needs to be sent.

The above pieces of information need to be serialized (marshalling) into a format that is understandable by both services. This job is done by stub. So a stub performs serializing / deserializing of data, converts it into a native object (which is understood by the corresponding service) and then gives that object to the service. In this way the services (in our case Authentication and Notification) need only focus on their business roles and not on the details of communication. Stubs allow the 2 services to be written in different programming languages.

=> Conversion of Request and Response is handled by Stub, this includes conversion of method, request type, response type to the form used by the RPC system and to a native object format.

How to define the Stub?
RPC Implementations provide a Stub Interface Description Language.
Using this language we specify the types.
gRPC uses the protobuf format, i.e. we specify the types / interfaces in a .proto file.

Next we run a generator. The Stub generator will take this interface definition (for example .proto) and convert it to code in the programming language being used by the service. This contains all the details of making the network call, retries, exponential backups, marshalling / unmarshalling etc. Once stubs are generated we just need to implement the necessary interfaces.

gRPC is a framework which implements the RPC protocol.

Notes on Encryption

Encryption is the process of converting readable text into ciphertext (text which is difficult to read).

Encryption makes use of an Encryption key, which is used by the encryption algorithm to convert the data into an unreadable format and decrypt it back.

Types of Encryption

Symmetric Encryption	Asymmetric Encryption
Uses the same key for Encryption and Decryption	Uses 2 different keys Public key and Private key => The public key is used by the sender to encrypt the data => The private key is used by the receiver to decrypt the data.
Important Symmetric Encryption Algorithm: => DES (Not recommended) => AES [Advanced Encryption Standard] DES Fixed key length: 56 bits AES Supports key length: 128, 192 and 256 bits => The bigger the key, the more the security and so would be the compute time. => AES processes data in blocks of 128 bits length By key we mean the encryption key which is used to encrypt the data	Important Asymmetric Encryption Algorithm: => RSA (Uses 1024 or 2048 bits long key length) => DSA => Diffie Hellman => ECDHE The Public key is known by everybody, when A wants to send some data to B, it'll encrypt the data with the public key of B. However to decrypt the data, the private key of B is needed which is only known by B, private key is never transferred over the network. Note: As can be seen in case of Symmetric Encryption one of the major challenges is how to securely distribute the key b/w the

	client and server. To achieve this the Diffie-Helman algorithm is used. However the algorithm itself makes use of Asymmetric encryption.
Advantages: Symmetric Encryption (AES) is faster than Asymmetric Encryption, as the key length is smaller. Hence it is useful when we need to do a lot of encryption on some bulk data, for example a Chat application.	Advantages There are no issues relating to Key distribution in case of Asymmetric encryption. Digital Signatures can be created using Asymmetric Encryption. Digital Signatures are used for authentication and data integrity purposes. Use of digital signatures: When the client C sends a message to the server S 1. The server needs to be sure that the data has been sent by the client C only, and not some imposter pretending to be C, or someone in the network has intercepted the data and sent it (Authentication). 2. The server needs to validate if the message has been modified in transit (Data Integrity)
Disadvantages: => Key distribution i.e. how to securely distribute the key between the client and server. => Consider client / server interaction, each of these communications need to be encrypted with their own unique	Disadvantages => Compute intensive, as the key length is longer. => Not suitable for encryption of bulk data.

keys, hence the server needs to keep track of the client and Encryption key mapping, as the number of clients increases this can become a performance bottleneck. So as the number of clients increases the server has to manage more and more Symmetric keys [The server distributes the Encryption keys] and their distribution too.	

A set of 4 bytes is known as word

Diffie Hellman Algorithm

The Diffie Hellman Key Exchange protocol is used for securely sharing a secret key between two parties over an insecure communication channel. Diffie Hellman is an Asymmetric Encryption algorithm (as it makes use of a pair of keys: Public and Private).

As mentioned before in symmetric encryption the same key is used for encryption and decryption and the client and server need a way to share the secret key, this is where Diffie Hellman is used. The security of Diffie Hellman relies on the difficulty of computing the Discrete Logarithms. For example if we have an equation like:

$$3^{x}\%19 = 5$$

In such an equation computing the value of x is very difficult, and there is no proven, efficient solution for it.

So, How does it work?

The parties, let's say Alice and Bob first decide and agree on 2 numbers, G and P. These numbers are shared directly over the communication channel, hence an eavesdropper can potentially see these numbers.

G, known as the primitive root, is a large prime number, while P is a large prime modulus.

Next Alice picks a private key a, and Bob picks a private key b

Now Alice will generate her public key using the formula:

$$x = G^a mod\ P$$

While Bob computes his public key as follows:

$$y = G^b\ mod\ P$$

x and y are public keys. These keys will be shared over the network, again they will be shared over an unsecure channel and hence can be potentially seen or copied by an eavesdropper.

So, Alice has shared her public key x to Bob
And Bob has shared his public key y to Alice.

Now both parties can generate the shared private key as follows:

Alice will calculate:

$$ka = y^a\ mod\ P$$

and Bob calculates

$$kb = x^b\ mod\ P$$

Now, ka will be equal to kb, since the original base (G) was raised to the power of both the original secrets just in the opposite order.

Hence ka = kb = $G^{ab}\ mod\ P$

And now both Alice and Bob have the shared private key, without the key itself ever being sent over the network.

DBMS Concepts

- **Choice of Database (Choosing Between SQL and NoSQL database)**
- **Choice of Non-Relational Database**
- **Transactions and ACID Properties**
- **CAP Theorem**
- **DB Indexing**
- **Normalization**
- **How do Databases store data / BTrees / Indexing cont.**
- **Database Replication**
- **Distributed Transactions - 2PC, 3PC and SAGA**

Choice of Database (Choosing Between SQL and NoSQL database)

https://medium.com/geekculture/choosing-the-right-database-for-system-design-sql-vs-nosql-and-beyond-d58fde5a6fe3

Choosing the right database for a particular use case and the reasoning behind it.

Section I - Overview of SQL and NoSQL databases

Databases can be broadly divided into 2 categories: SQL and NoSQL databases.

SQL databases also known as Relational Databases are based on the relational model, where the data is stored in tables (also called relations), Each table contains rows and columns. The data is organized in a structured manner. Different tables can be related to each other via the foreign key attribute. SQL databases have a fixed predefined schema.
Examples of Relational databases - MySQL, Oracle, PostgreSQL, Microsoft SQL Server.

NoSQL databases also known as Non Relational Databases prioritize flexibility and scalability. They have a flexible schema. NoSQL databases are of the following types: Document databases, key-value stores, Graph Based, Column oriented databases.
Examples of Non Relational Databases - MongoDB, Cassandra, DynamoDB, Redis.

Section II - SQL Databases Characteristics

As mentioned SQL databases represent data in the form of tables (also called relations). A table consists of rows and columns. Each row represents a unique record, and each column represents an attribute of the record. Relational databases support relationships, transactions and constraints.

Relational databases use SQL as the query language, for defining, manipulating and querying the data.

Here are some of the characteristics / properties of a Relational Database:

1. Relational Databases are ACID compliant, i.e. they support the ACID properties. ACID refers to Atomicity, Consistency, Integrity and Durability. These properties ensure the consistency and integrity of the data, and reliability of transactions. An important advantage of Relational Databases is that they support Transactions.

2. Structured Schema: As mentioned before SQL databases store the data in a well structured manner. The Schema needs to be predefined prior to adding any data to the table, and it cannot be changed i.e. the Schema is not flexible.

3. Query Language - Relational Databases use SQL as the query language. SQL is a powerful language that allows users to perform complex operations like Filtering, Sorting, grouping and Joins. Hence SQL databases are suitable for performing complex queries.

4. Scalability - Relational Databases can be scaled vertically by adding more compute resources like CPU / RAM / Disk to a single server. However horizontal scaling (also called sharding) is much more challenging because of the relational nature of the data and the constraints imposed by the ACID properties. It is difficult to scale the database to meet the needs of a large scale application which has a massive amount of data and heavy write load, hence in such scenarios the Relational Databases can be a performance bottleneck.

Section III - NoSQL Databases Characteristics

NoSQL (Not only SQL) databases are also known as Non-Relational databases. NoSQL databases do not follow the relational model, and do not use SQL as the query language. Instead they employ a variety of data models and query languages.

NoSQL databases do not have a rigid pre-defined schema, hence are more flexible in terms of data stored. Another important feature of NoSQL databases is Horizontal Scalability, NoSQL databases can be easily scaled horizontally (sharding). These databases provide native support for sharding.

Here are the characteristics / properties of Non-Relational Databases:

1. Flexible and Schema-less Design: NoSQL databases have a schema-less design and thus offer greater flexibility in handling various data models, i.e. these databases can accommodate new data types without extensive Schema modifications like in the case of Relational Databases

2. Horizontal Scalability - NoSQL databases are designed to scale horizontally. They have built-in support for features like Sharding, Replication etc. Hence NoSQL databases are suitable for large scale applications with massive amounts of data and high write load.

3. Performance - NoSQL databases offer superior performance than SQL databases under some workloads, such as systems which have high write load, or applications which need to support large scaled data storage and retrieval, under such situations NoSQL databases are more performant, and thus a better choice.

4. CAP Theorem Trade Offs - NoSQL databases prioritize Availability and Partition Tolerance over Consistency. They only support "eventual consistency", as a result NoSQL databases are not a good choice if we have a requirement for strong consistency, where data integrity at all points is topmost priority.

5. Complex Queries - Some NoSQL databases offer powerful query languages, however these are generally not as powerful and versatile as SQL, which supports complex operations like filtering, ordering, grouping, joins and aggregation etc. thus allowing complex data manipulation and analysis. Hence if the application needs to perform such complex queries then a NoSQL database might not be the right choice.

Section IV - Factors to consider when choosing the database

1. Data Model and Structured:
    a. If the data is structured and the data model is primarily tabular with well defined relationships, then a SQL database might be a better choice, as the relational model used by SQL is particularly suited for structured tabular data, and supports relationships via foreign keys and joins.

    b. If the data is unstructured or hierarchical, or if the data schema is dynamic (evolves over time) then a NoSQL database is a better choice, as it offers a Schema-Less design, and it will allow a high degree of flexibility. Different types of NoSQL databases offer a variety of data models like Document, Key Value, Graph or Column Oriented. Hence we can choose a data model which better accommodates the data.

2. Scalability Requirements
    a. Vertical Scalability - SQL databases are generally more adept at scaling vertically. Vertical scaling involves adding more compute resources to the single server to make it more powerful, for example by adding more CPU / RAM / Disk. However there is a hardware limit to it and we cannot keep adding more resources after a certain point.

    b. Horizontal Scalability - NoSQL databases are designed to scale horizontally (sharding), they have in-built support for features like Sharding, Partitioning etc. This makes NoSQL databases suitable for large applications with massive amounts of data, and high write load. Traditional SQL databases are difficult to scale horizontally so they may struggle to keep up their performance in such workloads. Hence for such workloads NoSQL databases are a better choice.

3. Consistency and Transactions
    a. ACID Properties and Strong Consistency - If strong consistency and ACID properties are essential to the application then a SQL database might be a better choice. SQL databases enforce these properties to ensure the consistency of data and reliability of transactions. If the

application needs transaction support then a SQL database could be chosen.

   b. Eventual Consistency - NoSQL databases are not ACID compliant, further they prioritize availability and fault tolerance over Consistency. Instead, NoSQL databases offer eventual consistency. This might be acceptable in some applications, however if our application requires strong consistency then NoSQL database might not be the right choice.

4. Query Complexity
   a. Complex Queries: Relational Databases use the SQL language for querying the database. SQL is a powerful and versatile language which supports complex operations like filtering, sorting, aggregation, grouping, joins etc, which helps to perform complex data manipulation and analysis.

   b. Simple Lookups and Updates: If your application performs simple lookups or updates then using a NoSQL database might offer better performance.

5. Performance and Latency
   a. High Performance and low latency - If our application desires high performance for read / write operations along with low latency, then using a NoSQL database that is optimized for that workload would be a better choice. NoSQL databases offer superior performance over SQL databases under certain workloads like large scale applications which store massive amounts of data and retrievals, and have high write loads.

   b. General Purpose Performance - If the performance requirements of our applications are not too high, then we can use a SQL database. Relational databases offer reliable, consistent General Purpose Performance suitable for a wide variety of workloads.

6. Other factors to consider
   a. Cost
   b. Ease of deployment
   c. Security

d. Community Support
e. Backup
f. Monitoring and Analytics

Choice of Non-Relational Database

Non Relational Databases

- **Document Databases**
- **Key Value Databases**
- **Column Oriented Databases**
- **Graph Based Databases**

Document Databases

These databases store data in a semi-structured format such as JSON or BSON documents. A document can contain complex data types like nested fields and arrays, allowing a high degree of flexibility in terms of data storage. Document databases are well suited for storing hierarchical and related data, further it is useful for applications which need to support complex data types and have a diverse and dynamic data model. For example: Content Management Systems, User Profiles, Event logging etc. Examples of Document Databases - MongoDB, CouchDB.

Key Value Databases

A Key Value database is a Non-relational (No SQL) database which uses a key value method to store the data, i.e. it stores data as a collection of key-value pairs, where the key serves as the unique identifier, both the key and value can be anythings.

Key Value databases are highly scalable and allow horizontal scaling (or sharding) at a level that none of the other types of databases can offer.

Use Cases:

1. Use Key Value stores when the amount of data which needs to be stored is very large, as key value databases are highly scalable and provide built in support for features like sharding.
2. Key Value databases offer very high performance with low latency for both read and write operations.
3. Flexible Schema
4. Session Management: Web Applications generally use a key value database to store sessions data.
5. Caching: Key Value databases are used by in-memory data caches.

Examples: Cassandra, Apache HBase, Amazon DynamoDB, Google BigTable.

DynamoDB Introduction
DynamoDB stores data in a table structure, due to the flexibility of the schema the Key Value databases don't need to perform expensive Join operations instead they can accommodate all the needed information in a single table.

When creating a table we need to specify a primary key.
The primary key consists of a partition key and a sort key. Do note the sort key is optional.

i.e. Primary Key could be

1. Partition key
2. Partition key + Sort Key [Composite key]

Regardless of the way we define the primary key it should be unique for all the records in the database.

Querying Database
DynamoDB provides following methods to retrieve data:

1. GetItem
2. Query
3. Scan

Get Item: Retrieve a single item from the database using its **primary key**. Operation is highly efficient, as dynamoDB has direct access to the physical location where the item is stored.

Query: Retrieve all items from the database matching the **partition key**. Operation is highly efficient because dynamoDB has direct access to the physical location where the data is stored.

Scan: Retrieve all items from the database, generally coupled with a filter to remove noise. Inefficient when only a subset of data is required.

Notes on the Query method
Query() allows us to retrieve all the items from the database matching the partition key.
However using the Query() method we can only query data based on the primary key (partition key + sort key), querying data based on non-primary key attributes is not possible. However in many situations we

will need to perform such queries, to do this we need to use a feature provided by DynamoDB called Global Secondary Index (GSI).
Using GSI we can treat a non-key attribute like the partition key, hence making the data retrieval much faster (other approach would have been to use Scan).

GSI consists of the attribute (non-key) which we specified, alongside the primary key attributes of the table, which dynamoDB will automatically add to the index.
Thus GSI allows us to query the database based on non-key attributes in a very efficient manner. We can have multiple GSIs in a table.

Note: The list of items returned by the Query() method is ordered by sort key in ascending order.

https://medium.com/ssense-tech/stop-using-scan-speed-up-your-dynamodb-reads-by-leveraging-global-secondary-indexes-407d076cc126

Column Oriented Databases
These databases store data in columns rather than rows, making them highly efficient for read / write operations on a specified column. Column oriented databases are well suited for large scale distributed applications with high write loads, such as Time Series data, IOT Devices, Log Analysis and recommendation engines. Examples of Column Oriented Databases - Cassandra, HBase

Graph Based Databases
Graph databases are used to store highly connected data, and perform graph based queries on the data, making Graph databases ideal for social media applications, fraud detection, knowledge graphs etc. Examples of Graph Databases - Amazon Neptune, Neo4j.

Transactions and ACID Properties

Transaction - A Transaction is a collection of Database Reads and write operations that only succeeds if all the operations within it succeed. A transaction can impact a single database record or multiple such records.

For example transferring funds from bank account A to B is an example of a transaction, it consists of following operations, suppose A wants to send B 50$

=> Update A's bank account to cur_balanace_A - 50 :- (1)
=> Update B's bank account to cur_balance_B + 50 :- (2)

Both these operations need to succeed for the transaction to succeed, we should not have a situation where money got deducted from A's account but didn't get credited to B's account, or a situation where B's account got credited but A's account didn't get deducted. In a transaction if any of the operations fails then the entire transaction fails.

If any part of the transaction fails then the entire transaction needs to be rolled back, this restores the database to the state it was in before the transaction was attempted, hence maintaining the integrity of data.

The transaction is committed when all the operations within the transaction are successfully completed. Committing the transaction permanently saves the changes to the database, and the locks acquired by the transaction are released.

ACID Properties of Transactions

ACID is an acronym for Atomicity, Consistency, Isolation and Durability. The ACID properties ensure that a transaction leaves the database in a valid state, even in the case of unexpected errors.

What are the ACID Properties?

1. Atomicity: The Atomicity property guarantees that all the operations / commands that make up a transaction are treated as a single unit, a transaction can only succeed if all of these operations are successful. If even one operation fails then the entire transaction fails. If any part of the transaction fails then the entire transaction needs to be rolled back. Atomicity thus ensures that if any of the

operation fails for whatever reasons (hardware, application or networking problem) then the transaction is rolled back and the database returns to its previous state as if nothing had happened.

Thus atomicity is a guarantee that either the entire transaction takes place or it doesn't happen at all. There are no intermediate states, hence if a transaction failed we know it failed entirely.

2. Consistency - The property of consistency guarantees that the changes made within a transaction are consistent with the database constraints, i.e. the changes made as part of the transaction should follow the database constraints, rules, Referential Integrity Checks etc.

   If the data gets into an illegal state then the entire transaction is rolled back.

   For example a banking system has a constraint that the bank balance should be a positive number. If we try to overdraw money then the constraint on the balance will be violated, in such cases the property of Consistency ensures that the transaction is rolled back so as to maintain the integrity of the data.

   Hence the property of consistency ensures that if the information in a database is in a consistent state prior to performing a transaction then it should remain in a consistent state post performing the transaction as well.

3. Isolation - The property of Isolation ensures that all transactions run in an isolated environment, this enables running transactions concurrently because the transactions don't interfere with each other.

   For example: An account has a balance of Rs. 200 and 2 transactions for Rs. 100 withdrawal arrive at the same time, then Isolation ensures that each of the transactions runs in an isolated environment. When both the transactions complete the final balance will be 0, not 100 (which it could be in case of Race Conditions).

4. Durability - The property of Durability guarantees that once the transaction completes and the changes are written to the database,

they are persisted. This ensures that the data within the system will persist even in the case of system failures like Crashes or Power Outage.

Database Locking and types of Locks

Database Locking ensures that no other transaction can update the locked rows or tables, in other words when one transaction takes a lock on a particular row then no other transaction or query can update it.

Types of Locks:

1. Shared Locks (S): Shared locks allow you to read the rows or tables that have been locked, hence it is also called a read lock. Multiple transactions can acquire a shared lock on the same resource (R) and read from it. No transaction is allowed to update (write to) the resource while it has a shared lock.

2. Exclusive Locks (X): Exclusive locks lock the row or table entirely and allows a single transaction to update the row or table in isolation. As can be seen only one transaction can acquire an exclusive lock on a certain resource at any point in time. While one transaction holds the exclusive lock on the resource, the other transactions will need to wait for the lock to be released before they can update the resource. Once the lock is released one of the waiting transactions acquires it and goes about making its modifications.

Important points:

=> Multiple shared locks can be acquired on a certain resource at any point in time.
=> If a transaction has acquired a shared lock on resource R, then another transaction cannot acquire an exclusive lock on R, however other transactions can acquire a shared lock on R.
=> If a transaction has acquired a shared lock on R, then no other transaction can acquire either a shared or exclusive lock on R.

Isolation and Levels of Isolation

ACID compliant databases need to ensure that each transaction is carried out in isolation, i.e. the transactions should not be aware of each other. The results of a transaction should only become visible after a commit to the database has been performed. Other transactions should not be aware about what's going on with the records while the transaction is happening.

Problems encountered when Transaction Isolation is not done:

1. Dirty Read: A dirty read basically refers to reading stale or incorrect data. Example, consider a transaction which updates a row or table but does not commit the changes, if the database lets another transaction read these changes (before it's committed) then it's called a dirty read. The reason is that suppose the first transaction is rolled back in this case the second transaction has stale data.

2. Non-Repeatable Read: A side effect of concurrent execution of transactions is that consecutive reads can retrieve different results, if we allow other transactions to do updates in between. For example a transaction queries a row twice, however between the two reads there is another transaction which updates the same row, hence resulting in the reads giving different results.

3. Phantom Read: A Phantom Read is similar to Non-Repeatable read as it too involves the situation where consecutive reads produce different results. For example a transaction performs a read query twice, however between the two reads there is another transaction which inserts or deletes rows (records) from the table, leading to a difference in the number of rows fetched by the first read and the second read for the same transaction. This situation is known as Phantom Read. The difference between Phantom Read and Non-Repeatable read is that a Non-Repeatable read results in inconsistency in the values of a row (record), however in case of Phantom read the number of rows retrieved by the queries will be different.

Levels of Isolation:

1. Read Uncommitted: This level of isolation lets other transactions read the data that has not yet been committed to the database by the appropriate transaction. Example: Transaction A updates the data, however before it commits Transaction B tries to access the

data and reads this uncommitted new data. There is no isolation happening here, and it does not solve any of the problems mentioned in the last section. Locking Strategy:
    a. READ: No Lock acquired
    b. WRITE: No Lock acquired

2. Read Committed: This level of isolations lets the transactions only read the committed data. If a transaction is updating a row and another transaction tries to access it then it won't be able to. Read Committed solves the problem of Dirty Read, however since it applies only to updates and not to read queries, hence Non-Repeatable reads and Phantom reads are still possible. Locking Strategy:
    a. READ: Shared Lock acquired and released as soon as the Read is done.
    b. WRITE: Exclusive Lock acquired and released only at the end of the transaction.

3. Repeatable Read: This level of isolation locks the resource in question throughout the duration of the transaction. Example: If a transaction consists of two read queries and in between if another transaction tries to update the same rows then it won't be able to do so, i.e. it will be blocked from updating these rows. Repeatable read is the default level of isolation in many databases, however it does suffer from Phantom Reads. Locking Strategy:
    a. READ: Shared Lock acquired and released only at the end of the transaction.
    b. WRITE: Exclusive Lock acquired and released only at the end of the transaction.

4. Serializable: This is the highest and strongest level of isolation, where all the concurrent transactions appear to be executed serially or sequentially. Serializable isolation level solves the problem of Phantom reads as well.

Optimistic Concurrency Control

In the case of Optimistic Concurrency control multiple transactions can update the records in parallel, however when one transaction successfully commits its changes the other transactions are told that there exists a

conflict due to which these transactions will be rolled back and attempted again later.

Optimistic Concurrency Control uses the Read Committed Isolation Level.

This approach makes use of versions, in a database like MySQL each row has built in versions and whenever we make any changes to the row the version number increases, however in case of databases like Oracle which don't have versions, we'll need to add our own Version attribute to the table.

Example and Flow:
Consider two transactions A and B, running concurrently. Consider this is an example of a Movie Ticket Booking application where both these transactions are user requests to book the same seat (let's say seat number 1).

Transaction A	Transaction B	DB
BEGIN TRANSACTION	BEGIN TRANSACTION	Seat ID: 1 STATUS: FREE VERSION: 1
READ ROW Seat ID: 1 STATUS: FREE VERSION: 1	READ ROW Seat ID: 1 STATUS: FREE VERSION: 1	Seat ID: 1 STATUS: FREE VERSION: 1
PERFORM UPDATE – acquire Exclusive lock - Version Validation: Check that the version of the row in the database is the same as that read in by A: Here it is 1, 1.		Seat ID: 1 STATUS: FREE VERSION: 1
UPDATE Seat ID: 1 STATUS: BOOKED VERSION: 2		Seat ID: 1 STATUS: FREE VERSION: 1
COMMIT		Seat ID: 1

- Release Exclusive Lock		STATUS: BOOKED VERSION: 2
	PERFORM UPDATE - acquire exclusive lock - Version Validation: Check that the version of the record in the database is same as that read in by transaction B: Here it is not 1, 2	Seat ID: 1 STATUS: BOOKED VERSION: 2
	ROLLBACK - Release Exclusive Lock	Seat ID: 1 STATUS: BOOKED VERSION: 2

Optimistic concurrency control provides a high level of concurrency.

Pessimistic Concurrency Control

Pessimistic Concurrency Control uses the Repeatable Read or Serializable isolation level.

In this approach we lock the record (or row) as soon as one transaction attempts to modify it, the other transactions need to wait for this transaction to complete before doing anything.

As mentioned above that Repeatable read isolation level solves the problem of Non-Repeatable read, it does so by holding the shared locks till the end of the transaction so that no other transaction can modify those resources. For example if a transaction needs to perform a read query on row X, then it will acquire a shared lock on X and hold it till end of transaction, so that when another transaction tries to modify X, it won't be able to as it can't acquire the exclusive lock, this way we can solve the problem with consecutive reads as there won't be any inconsistency. Serializable isolation level further solves the problem of Phantom Reads by using Range Locks.

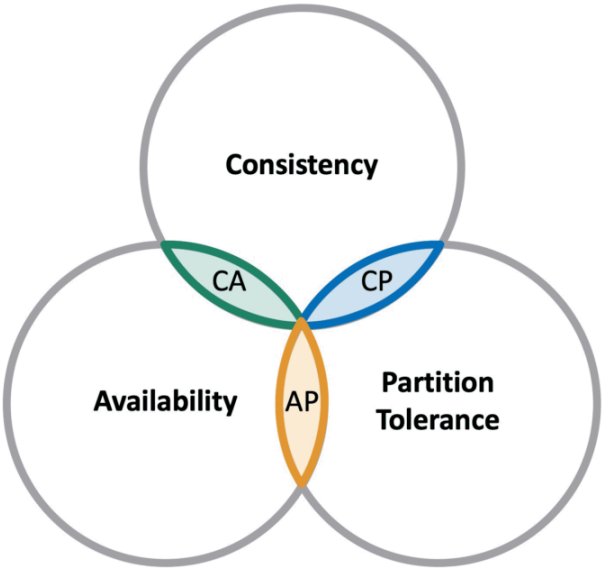
Pessimistic concurrency control suffers from deadlocks, for example suppose we have 2 concurrent transactions A and B. Transaction A has a

shared lock on resource X and is waiting to get an exclusive lock on resource Y while Transaction B has a shared lock on Y and waiting to get an exclusive lock on X. However A cannot get an exclusive lock on Y, since B has a shared lock on it, similarly B cannot get an exclusive lock on X, since A has a shared lock on it. This is a deadlock, and ultimately these transactions will need to be aborted, in this situation the issue which leads to the deadlock is that in Pessimistic concurrency control the shared lock is held till the end of the transaction rather than just till the read operation is completed.

Optimistic Concurrency control doesn’t suffer from this issue as the shared locks are released as soon as read is completed.

CAP Theorem

The CAP Theorem states that it is not possible to guarantee all three of Consistency, Availability and Partition Tolerance at the same time in a distributed system, according to the CAP theorem a distributed system can at best provide two of the three properties.



The 3 Properties -

1. Consistency - The view of the data is up-to-date on all members of the distributed system.

   Any read that happens after the latest write, should return the value of that latest write, on all the nodes.

2. Availability - The data is always accessible for reading and writing.

   Every available node in the system should respond in a non-error format to any read request without the guarantee of returning the latest write.

3. Partition Tolerance - The distributed system continues to operate despite network failures, where not all members can reach the

other members, i.e. the system continues to respond to read / write requests even in the presence of Network Partitions.

Distributed systems cannot be made immune to Network Failures.

AP Systems: If a system chooses to prioritize Availability and Partition Tolerance over Consistency then it is known as an AP system.
Such a system will continue to service requests even though the data might be in an inconsistent state due to Network Partitioning, the system cannot ensure that the data is completely up-to-date on each node. In such systems temporary discrepancies in data among different nodes are acceptable, the primary goal being to ensure high availability. As can be seen AP systems compromise on Consistency.

CP Systems: If a system chooses to prioritize Consistency and Partition Tolerance over Availability then it is known as a CP system. Here, since consistency is our priority, the system will make sure that all the nodes in the network have an up-to-date view of the data. If the consistency of the data cannot be guaranteed then the system will stop serving requests (become unavailable). As can be seen CP systems compromise on Availability, as some data might not be accessible in the event of a network partition.

CA Systems: CA system prioritizes Consistency and Availability. However these systems struggle with Network Partitions, potentially making the system less available and consistent.
Hence pure CA systems are very rare as distributed systems cannot be made immune to Network failures, hence they must handle Network Partitions to some degree.

Under normal operations (lack of Network Partitions) the CAP theorem does not impose constraints on Availability or Consistency.

Eventual Consistency: It means the system can become temporarily inconsistent, typically during a Network partition. The idea is that after the system is reconnected, the nodes can reconcile their views of the data, and construct an up-to-date view of the data, thus the system becomes consistent again.

Since partitions are generally short, hence the system will be consistent most of the time. This way we can achieve an Available system which is consistent for most of the time, such systems are called Eventual Consistent.

DB Indexing

Indexing
When we want to read some data from the disk, we need to read in the entire block. Each block is generally 4KB, so when we issue a disk I/O to read certain bytes, the entire block containing the required bytes is read into memory. Block is the unit of data read from the disk.

Example - Consider the database record has the following attributes:

ID - INTEGER: 4 BYTES
USERNAME - VARCHAR: 60 BYTES
AGE - INTEGER: 4 BYTES
BIO - VARCHAR: 128 BYTES
BLOGS PUBLISHED - 4 BYTES

Each record has a total size of 200 bytes, suppose the database table has 100 such records, then the total space required is 200 bytes * 100 = 20000 bytes.

This data will need to be stored on the disk, as mentioned above the data in the disk (HDD, SSD) is stored in blocks. Assume the block size is 600 bytes.

How many disk blocks will our data need?
Each disk block can accommodate 3 (600 bytes / 200 bytes = 3) records from our database, hence total number of blocks on disks
= Total number of records / 3 = 100 / 3 = 33.3 ->34

Hence we need 34 blocks to store the data in the disk.

Time estimation of queries:
SELECT * FROM user_blogs_info WHERE user_age == 23

Assumption: Each disk block access takes 1 second.

To get the result for this query

1. Read all db records from the disk (i.e. all 100 rows if it’s a Relational db)
2. Store the result in an Output buffer
3. Filter the buffer to only contain records with age == 23

4. Return the output buffer to the user.

To read in all records, we'll need to fetch all of them from the disk. Assume for now we have no caching mechanism for now, thus to fetch 100 records we'll perform 34 block accesses.

Total time taken = (# blocks read) * (time taken for disk i/o of one block)
Here we assume that to read a block into memory it takes 1 second.

Therefore total time taken = 34 * 1 s = 34 seconds.

What are Indexes and how can they make this query faster?
Indexes are smaller referential tables that hold row references against the indexed value, i.e. they map the indexed value to the row reference (Or the Primary key of the row), i.e. the row which contains that indexed value.

Since we want to optimize the queries based on age, we can put an index on the age attribute.
We can think of the index as a 2 column table, where the first column is the indexed value and the second column contains the row reference corresponding to each indexed value.

The index is sorted by the indexed value (ascending).

How the Index looks like -

Age	Row Reference (ID)
21	3
22	1
23	8
23	2
24	5
25	7
25	4
28	6

As can be seen the index is sorted by the Indexed Value (age in this case)

The index will be stored on the disk, how many disk blocks do we need for the index?

A record in the index needs 4 bytes (Age: INT) + 4 bytes (ID: INT) = 8 bytes (as compared to 200 bytes per record in the original table).

Each disk block is 600 bytes.
So 1 disk block can accommodate 75 index records.
We have 100 index records hence we'll need 2 disk blocks for the index.

Another way to calculate this
No of disk blocks = (Total space requirement) / Size of a single disk block = (8 * 100) / 600 = 1.33, hence we need 2 disk blocks.

Same Query with Index on Age, note: This is the worst case where we read in the entire index.

1. Read all the records in the index from the disk one by one (i.e. block by block)
2. Check if age == 23, if yes add the row reference to a buffer, if no discard that record.
3. For all the relevant id's (row references) in the buffer, read the corresponding record from the disk and add it to an output buffer.
4. Return the output buffer.

Total time taken = Time taken to read the entire index + Time taken to read in the corresponding records from the disks.

Time taken to read the index:
The index is stored in 2 blocks, hence we'll need to perform 2 block accesses, time taken = 2 seconds.

Now, assume we find k records with age == 23,
then time taken to read these k records = time taken to read the k blocks corresponding to the records.

Hence t = k * 1 s = k seconds

Total time taken = 2 + k = k + 2

Where k is the number of relevant id’s in the buffer (i.e. number of index records with age == 23).

If k == 2, then Total time taken = 4, which is an 8x improvement in performance as compared to the approach where we did not use an Index.

Database Scaling

Scaling the data tier

Vertical Scaling - In Vertical Scaling we add more power to an existing machine, in the case of our database server we can add additional RAM / Disk, CPU etc. to improve its performance.
Disadvantages with Vertical Scaling:

1. We can add CPU / RAM etc. to the database server but there is a hardware limit, beyond which you cannot add CPU / RAM / Disk.
2. Vertical scaling suffers from a single point of failure, because even though the machine is much more powerful, however, if it goes down the entire system will go down.
3. Very powerful database servers do exist, but they are very expensive.

Horizontal Scaling - Also known as Database Sharding.
Sharding involves adding more database servers, in other words Database Sharding is the process of storing a large database across multiple machines. As the data stored becomes larger and larger it might not be possible to store all of it in a single database server, hence we need to store the data across multiple Database servers. Sharding achieves this by splitting the data into smaller chunks called 'shards', and storing these shards across multiple database servers.

Each shard has the same schema, though the actual data on each shard is unique.

Sharding is applicable to both Relational and Non Relational databases,as mentioned before the data is splitted into smaller chunks called shards, in case of a relational database each shard will contain a unique set of rows from the original database.

Logical and Physical Shards - The partitioned data chunks are called logical shards. The machine that stores the logical shard is called a physical shard (or database node). A physical shard can contain multiple logical shards.

Shard Key - The Sharding key (or Partition key) determines how to partition the dataset, or how the data is distributed, the Shard Key can be a column from the dataset (or a combination of multiple columns) or we can define our own Shard key.

shard_number = f(shard_key)

Algorithmic vs Dynamic Sharding

Algorithmic Sharding makes use of a Sharding Function to determine the shard to which a particular request must be sent to.
The sharding function takes some columns as input and returns the shard to which the read / write request must go to.
The sharding function is kept inside the application code, hence the client knows where the read or write request has to be routed.

Dynamic Sharding uses a separate module called Locator service, which tells where the queries have to be routed to. The client will query the locator service to determine where the read / write request has to be routed to. Dynamic Sharding is useful because it allows us to add or remove shards.

Advantages of Sharding

1. Query Optimization
2. Facilitates storing large volumes of data
3. No single point of failure.
4. Lower latency, as the shards can be located in different geographical locations.

Disadvantages of Sharding

1. Celebrity Problem (Hotspot Key Problem) - Even if we divide our data as evenly as possible, excessive load to a specific shard is still possible, which could cause server overload.
   How?
   Some records of the database will be accessed more heavily than others, and if multiple such records cluster in a single shard then that shard could be overwhelmed.

2. Once the database has been sharded across multiple servers, it is hard to perform join operations across database shards, it will be a very expensive operation.

Methods of Database Sharding

https://aws.amazon.com/what-is/database-sharding/#:~:text=Database%20sharding%20is%20the%20process,a%20limited%20amount%20of%20data.

There are multiple methods of database sharding, which differ in how they define the shard key hence how the dataset is partitioned.

Do note, if we have n physical shards, then
Shard key $\in$ [0, n - 1] (0-based indexing)
Shard Key $\in$ [1, n] (1-based indexing)

1. Key Based or Hashed Sharding: A shard key is assigned to each row of the database by using a mathematical formula called hash function.
   The hash function takes some information from the row, and produces a hash value.
   The application uses the hash value as the shard key and stores the information in the corresponding physical shard.

   Shard key = hash(information from the row),

   The information from the row, is in the form of values of some of the columns for example say columns ci, cj, ck, then
   shard key = hash(ci, cj, ck)

   This strategy distributes the data evenly across the physical shards.

   It does not separate the database based on the meaning of the information. With Hashed Sharding it is difficult to add or remove physical shards, as if we change the number of physical shards in the system then we'll need to change the hash function and thus a lot of data will need to be remapped and shifted. Suffers from Celebrity (Hotspot key) problem.

   The shard key should be static in nature.

2. Range Based Sharding: Range Based Sharding also known as Dynamic Sharding splits the database rows based on a range of values. Then a shard key is assigned to the respective range, and the information is stored in the corresponding physical shard.

   Depending on the data, it is possible for Range Based Sharding to *overload* data to a single physical shard, and overwhelm that shard with read and write requests, i.e. uneven distribution of data.

   For example, if we have a database containing usernames and we partition it using Range Based sharding where
   Shard 1: Customers with names beginning A to H
   Shard 2: Customers with names I to P
   Shard 3: Customers with name Q to Z

   Obviously Shard 1 will have many more entries than Shard 3

   Since there is no hash function involved hence Range Based Sharding is more flexible.

   Use Range Based Sharding when you have range based queries, for example: “get the names of all the users who made a purchase, in the period January - February”, or select all users whose age is in the range 32 - 40.

3. Directory Based Sharding: Directory Based Sharding is an example of dynamic sharding, it uses a lookup table to map database information to the corresponding physical shard.
   The lookup table links a database column to a shard key.

   For example, the following diagram shows a lookup table for clothing colors.

Colour	Shard key
Blue	A
Red	B

Yellow	C
Black	D

When an application stores clothing information in the database, it refers to the lookup table. If a dress is blue, the application stores the information in the corresponding shard.

Advantages: It is flexible, and each shard is a meaningful representation of the database.

4. Geo Sharding
   Geo Sharding (or Geographical Sharding) splits the database information according to geographical location.

   For example an application could use the cityname as the shard key, and store the data in the physical shards that are geographically located in the respective cities.

   Geo Sharding makes information retrieval faster, as users can retrieve information from the physical shard geographically closest to them, thus reducing latency.

   However Geo Sharding can result in uneven distribution of data and lead to Celebrity (or Hotspot Key) problem.

   Use Geographical Sharding when data access patterns are predominantly based on Geography.

Normalization

Normalization is technique in DBMS using which we can organize the data in the database tables so that:

1. There is no data redundancy / duplication of data.
2. A large set of data can be structured into multiple smaller tables.
3. Remove insertion anomalies, update anomalies and Delete Anomalies.

In other words, Normalization is the process of decomposing (breaking down) tables to eliminate Data Redundancy (repetition), ensure data integrity, and remove undesirable characteristics like the ones mentioned above.

Normalization helps in:

1. Eliminating redundant data thereby improving data integrity, because if data is repeated it increases the chances of inconsistent data.
2. Storage Optimization - As we are not storing redundant data.
3. Breaks down large tables into smaller tables with relationships, so it makes database structure more scalable.

Recognizing dependency: A dependency of attribute B on attribute A, indicating that B depends on A implies for every occurrence of a value x $\in$ A, the corresponding value y $\in$ B is fixed and deterministic, if for a particular value in A, B has multiple corresponding values then the B does not depend on A.

Insertion, Updation and Deletion Anomaly
https://www.studytonight.com/dbms/database-normalization.php

Types of DBMS Normal Forms

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Fifth Normal Form

First Normal Form (1NF)
For a table to be in 1NF it should follow the following rules:

1. Each column / attribute should have a single (atomic) value for each row, i.e. multi-valued attributes are not allowed.
2. Values stored in a column should be of the same domain, i.e all the values must have the same datatype.
3. All the columns should have a unique name.
4. The order in which the data is stored should not matter.
5. The table must have a primary key to uniquely identify each row, a table without a primary key violates 1NF.

Second Normal Form (2NF)
Each non-key attribute must depend on the entire primary key, i.e. there should be no partial dependencies.

For example if we have a table consisting of

Roll Number	Subject	Marks	Teacher

Here Roll Number + Subject is the primary key. However the attribute - Teacher depends only on Subject and not on the entire primary key (since it doesn't depend on Roll Number). Hence, this table is violating 2NF. To fix this remove the 'Teacher' column from the table and create a separate table to store information regarding subjects and teachers.

Subject	Teacher

Roll Number	Subject	Marks

Third Normal Form (3NF)
To be in 3NF a table must first of all be in 2NF and additionally have no transitive dependency.

A non-key attribute should never depend on another non-key attribute, more specifically a non-key attribute should depend on the Primary Key, the whole primary key and nothing but the primary key.

For example, in a gaming system we store for a player his / her skill level.

Player	Skill Level

Assume skill level is INT with values b/w 1 - 5, now we want to add another column ‘Skill Level in words’ such that corresponding to each possible value of skill level we have a unique descriptive string.
1 - newbie
2 - learning
3 - good
4 - Expert
5 - Boss mode

We can add another column to this table directly.

Player	Skill Level	Skill Description

However, this is in violation of 3NF as skill descriptions depend on the Skill Level, hence there is a transitive dependency. The correct solution would be to have 2 tables, one storing the Player and Skill Level information and the other storing the mapping between Skill Level and Skill Description.

Player	Skill Level

Skill Level	Skill Description

Now the tables are in 3NF.

What could be the problem here if violate 3NF, suppose a player’s skill level changes from 2 to 3, in this case we’ll need to update 2 things in the table:

1. Player skill level from 2 - 3
2. Player skill description from Learning to Good

Assume the first updation goes through, however for some reasons the second updation couldn't go through, this will lead to a data integrity problem, as the skill level of 3 would suggest the player to be “good”, however the skill description is still showing ‘Learning’, i.e. the data is

disagreeing with itself, that is why we need to eliminate transitive dependencies.

Boyce Codd Normal Form (BCNF)
BCNF is a stronger form of 3NF, it is also called 3.5NF. To be in BCNF a table must obey the following rules:

1. The table should be in 3NF
2. For every dependency A -> B, A should be a super key.

2NF and 3NF place constraints on non-key attributes. 2NF prohibits any partial dependency i.e. each non-key attribute must depend on the entire primary key, 3NF prohibits transitive dependencies i.e. each non-key attribute should depend on the primary key, the entire primary key and nothing but the primary key.

BCNF goes further. For example consider a relation with 3 attributes A, B and C. Where the primary key is {A, B} and C is a non-key attribute. 2NF and 3NF normalizations would have ensured that C depends on the entire primary key. However, what if an attribute which is part of the primary key depends on a non-key attribute, such dependencies haven't been covered by 2NF or 3NF.

This is where BCNF steps in and prohibits any dependencies A -> B, where A is not a super key.

For example in our 3NF relation consider that the attribute B (part of the primary key) depends on the attribute C (non-key attribute), i.e C -> B. This table is not in BCNF because for BCNF compliance attribute C must be a super key which it is not in this case.

Fourth Normal Form (4NF)

Q. What are Multivalued Dependencies?
1NF, 2NF and 3NF deal with functional dependencies, 4NF deals with Multivalued dependencies. We can think of Multivalued dependencies as a more general version of function dependencies. A ->-> B, implies B has a multivalued dependency on A and corresponding to a value of A, there are multiple values in B.

To be in 4NF a table needs to be in BCNF and Non-Trivial multivalued dependencies on a non-key attribute are not allowed. A multivalued dependency is considered trivial if the attributes involved in the dependency, together make up all the attributes in the table.

Fifth Normal Form (5NF)

A table is in 5NF if it is in 4NF and does not contain any join dependency. 5NF is satisfied when the table has been broken down into as many smaller tables as possible in order to avoid redundancy (i.e. it cannot be further decomposed) and the joining should be lossless.

5NF is the most complex form of Normalization and is also known as Project-Join normal form (PJNF).

How do Databases store data / BTrees / Indexing cont.

B-Trees
B-Tree or Balanced tree is a data structure that can handle massive amounts of data, i.e. they can store and search large amounts of data efficiently. Traditional data structures like BST struggle in these situations due to poor performance and high memory usage.

The idea behind a B-Tree is that each node can contain multiple keys (in a BST, each node has only one key). By storing a large number of keys in a single node the height of the tree can be kept relatively small.

Properties of B-Trees

- All the leaf nodes of the B-Tree should be at the same level, to ensure the B-Tree is balanced.
- In an M-order B-Tree each node has at most M children nodes.
- Every node in the B-Tree except the root node and leaf nodes contains at least m / 2 children.
- Number of keys in any node = Number of children - 1.

B-Trees are widely used for disk access.
Use of B-Trees in Databases: Since disk access is a time consuming process, hence we use B-Trees to index the data in order to provide faster access to the required data.

Time complexity of B-Tree search, insert, delete operations: $log_m n$

Where n = total number of elements in B-Tree
M = order of B-Tree.

Data Storage
As mentioned before, the data in the database is actually stored in the disk, across multiple disk blocks. Disk I/O is slow, and if we have millions or billions of records it becomes very impractical and non-performant to do a full database scan for each query, remember the (age == 23 filter queries) from the ‘Indexing’ section.

An index acts as a quick reference guide for the database, it helps the database to search data in the table more rapidly. Indexes are essential for improving database query performance.

Types of Index

1. Clustered Index (Primary Key) -
    a. Determines the physical location of the database records on the secondary storage.
    b. Each table can only have one clustered index.
    c. The clustered index can consist of one or more attributes / columns.
    d. When we create a clustered index the database records are rearranged in the order of the indexed value, which involves changes in the physical locations (blocks) of individual records. Hence Clustered indexes determine the physical location and order of the data.
    e. In case of B-Trees, the actual records are stored in the leaf nodes.
2. Non-Clustered Index (Secondary Index)
    a. Non-Clustered indexes do not alter or affect how the data is physically stored.
    b. We can have multiple Non-clustered indexes for a single table.
    c. The index is stored in a separate data structure, as mentioned in the previous Indexing section that the index is basically a 2 column table containing the Indexed value and the corresponding row reference (the row containing the indexed value). The index is stored on the disc as well i.e across disk blocks. Like the main database table / collection we can store the Index using a B-Tree as well.

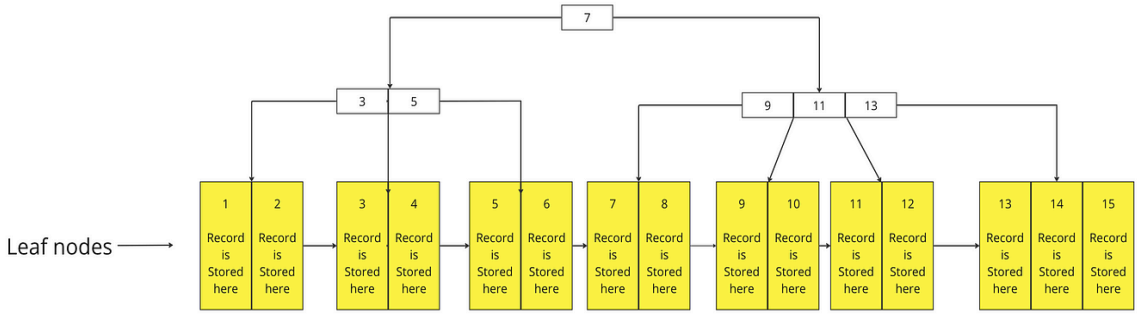
B-Tree is just one data structure, there are other data structures available too like Hash indexes, Bitmap indexes which help in storing and retrieving data on the disk.

Examining in greater detail: How are Indexes stored using B-Trees

Clustered Index:

As mentioned before, Clustered index determine the physical location and order of the database records, also remember that the data is ordered by the clustered index. The actual database records are stored in the leaf nodes of the B-Tree.

The database builds a B-Tree using the Clustered index values as the keys. For example a database using the ID (INTEGER) attribute as the clustered index will use the ID values of the records as keys.

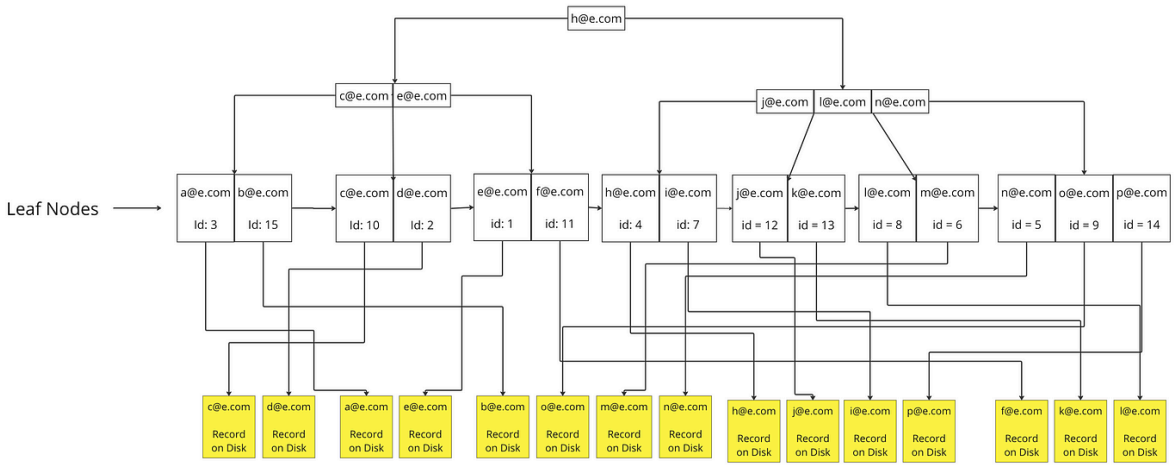


What happens when a record is inserted - The database engine will need to perform a series of operations to add a record to the database. It begins by retrieving the relevant index blocks, then it adds the new record to the B-Tree, it'll check if the B-Tree is balanced and if all of its rules are being obeyed if not, it will update the B-Tree structure to ensure that it remains in an optimized state. Once the restructuring is completed the engine will write the updated blocks back to the disk (storage).

Some Database Engines:
MySQL - InnoDB, Blackhole etc.
MongoDB - WiredTiger

Non Clustered Index:
As mentioned before Non Clustered indexes do not alter / affect how the database records are physically stored. Instead the index is stored in a separate data structure (B-Tree).
Here the leaf nodes instead of containing the actual database records, will just contain the row reference [As mentioned before an Index maps the indexed value to the row reference]. The row reference is generally just the primary key (ID) of the row, however PostgreSQL stores the actual disk address of the row as the row reference.

Example of a B-Tree for a Non Clustered Index (email)



This is similar to the diagram above (in case of clustered index) except for a couple of major differences.

1. This B-Tree uses the values of the Non Clustered Index (herein - email) as the keys.
2. Instead of storing the actual database records in the leaf nodes, herein the database engine will store the row references (Primary Key or the actual disk location) in the leaf nodes. This row reference is basically a pointer to the row which contains the indexed value.

Database Replication

Master Slave Architecture, also known as Active Passive Architecture
With the master slave architecture, the master database is the only one which takes writes, i.e. add data modifying operations like create / read / update / delete are sent to the Master DB.

Further we create multiple slave databases (also read-replicas) to service read requests. The slave db will get copies of the data from the master database (this is called db replication).

Since most applications need a higher number of reads than writes, it makes sense to have multiple slave dbs.

Advantages:

1. This architecture improves performance as the read requests are distributed across multiple slave databases, hence we can scale the reads, this improves the performance as multiple queries can be processed in parallel, this is specially useful for read-intensive applications, i.e. this architecture scales reads.
2. Reliability and High Availability: As our data is replicated across multiple dbs (probably across multiple regions), we do not need to worry about data loss, even if a database is offline the website can access the data from another database server.

Failover:

1. If we only have one slave database, and it goes down then the Master database will need to temporarily serve read as well as write requests. If we have multiple slave databases and one of them goes down the load balancer will forward the read requests to the other healthy slave databases. Additionally when a slave db goes down another database server will be added into the pool to replace it.
2. If the master db goes down, then one of the slaves will be promoted to master, since replication b/w master and slave is asynchronous hence the slave database might not have the latest data. To get the missing data we'll need to run some data recovery script.
3. Alternatively for disaster recovery we can have a standby master database in another data center. There is synchronous replication b/w the master and the standby. Thus if the master goes down we

can quickly failover to the standby instance and it will become the new master.

Refer AWS Aurora section in ‘AWS - Certified Developer Associate notes’, as Aurora provides excellent database replication features.

Distributed Transactions - 2PC, 3PC and SAGA

A transaction is a group of instructions which modifies some data, a transaction must follow the ACID properties. In case of distributed transactions the data to be modified is located on different nodes.

The challenge with distributed transactions is coordination, suppose a transaction affects tables from multiple databases. Say the transaction consists of two local transactions, the first one affecting table T and second one affecting table Y. The local transaction on T executes successfully and say we commit the result on T. Now we proceed with performing the local transaction on Y, however for some reason this transaction fails now we'll need to rollback the entire transaction, rolling back the local transaction on Y is straightforward, as we can simply rollback the local transaction on Y, however rolling back the committed transaction on X is difficult, because it is on a different node.

A transaction is local to a particular database, and each database will have its own transaction manager.

Techniques used to perform distributed transactions:

1. 2PC (2 Phase Committing)
2. 3PC (3 Phase Committing)
3. SAGA pattern

2PC (Two-Phase Commit): There are 2 phases in this protocol:

1. Prepare for Voting Phase
2. Decision or Commit phase

2PC uses a transaction coordinator, which interacts with all the participants, say we want to update the Orders and Inventory services as part of the transaction, these services are called the participants or the Resource Managers, as they store the actual data.

How 2 PC works:

=> The transaction coordinator starts the transaction.

=> The coordinator will talk to all the participants, and will pass the information regarding the operations (the transaction set of operations) which need to be performed, to each of the respective participants. When the participants receive the update queries, it'll acquire locks on the resources (row or the entire table) being modified, and make the changes, but it won't commit the changes.
=> In the first phase, (prepare phase or Voting phase): The coordinator inquires each of the participants to check if they are prepared to commit, if the participant was able to successfully perform an update it'll tell the coordinator that it is ready to commit.
=> In the second phase, (Decision or Commit / Rollback phase) - Once the coordinator receives responses from all the participants, it will make a decision. If all the participants respond that they are ready to commit, then the coordinator will send the COMMIT message to all the participants, if even one participant is not prepared, the coordinator will send an Abort message to all the participants.
=> If the coordinator responds with a COMMIT message, then all the participants will commit the changes / updates they made previously, and hence transaction would be completed.
=> If the coordinator sends an ABORT message, then all the nodes which were able to perform the update will need to revert (rollback) those changes.

Dealing with Node failures: Transaction Coordinator and Participant failures.

Each of the nodes will maintain a log file (stored in non-volatile storage). The Transaction coordinator when sending an update message, will update its log to reflect the message sent along with the timestamp, so it will update the log file like, the participants will also update their log files in a similar fashion.

In phase 1 before sending the prepare message, the coordinator will update its log adding an entry to reflect that it has sent the prepare message. For example

Sent Prepare Message at time t

Each participant will maintain their own log files as well, in which it will log the messages it is sending, for example if the participant responds with an OK or NO message to the prepare message of the coordinator, then it'll update its log with this information.

In phase 2: Post making the decision (COMMIT / ABORT) the coordinator will update its log file and then send the message to the participants, i.e. it'll first write its decision to the log file and then send the decision.

The participants also write the decision sent by the coordinator to their log files.

So the goal is that the coordinator and participants before sending any message, will update their respective log files (permanent storage). So if any of the nodes fails, when it comes back it can see its log file and check the state of the transaction, i.e. they can see what happened last.

Some Scenarios

Prepare Message Lost

All the participants have acquired the locks and updated the resources locally, however at this point the coordinator goes down without sending a prepare message. In this case the participants are just holding some locks and waiting for the coordinator's prepare message. In such a situation, after a certain period of time of waiting for the prepare message the participants will abort the transaction and rollback the changes made locally.

What if after a participant aborts a transaction, the coordinator comes back and sends the prepare message. In this case the participant can simply reply with a No to the prepare message, forcing the coordinator to abort the transaction.

So if the prepare message is lost, the transaction will be aborted.

Participant's OK reply message to the coordinator lost

The transaction coordinator will wait for some time to receive the response to the prepare message, after the timeout happens it'll abort the transaction.

What happens when the participant finally comes up, the participant will need to check what happened to the transaction, for which it had acquired the locks on its resources, it needs to know whether it should commit or abort. In this case it'll ask the coordinator what action it should perform wrt this transaction. The translation coordinator will check its log, and find

what decision was made for that transaction and tell that to the participant. The participant will act accordingly.

Transaction Coordinator's Commit message is lost

All the participants have returned an OK message to the coordinator, at this point they are holding locks on the resources and waiting for the coordinator's decision. What happens if the transaction coordinator goes down before it informs the decision to the nodes.

Further it is possible that there are a lot of participants, and some of them actually received the commit message before the coordinator goes down.

In such cases the nodes which haven't received a reply are in the blocked state, and need to wait for the coordinator to come up again, and only then can it inquire the coordinator regarding the status of the transaction, the coordinator will check what decision was made regarding the transaction - Abort or Commit, in the log, and relay the information to the participant. The participant will act accordingly. However the participant will remain blocked until the coordinator comes back up. This is a problem with 2 Phase Commit.

2PC is a blocking protocol

3PC (Three-Phase Commit)

This protocol has three phases:

1. Prepare Phase
2. Pre-Commit Phase
3. Commit Phase

This prepare phase is exactly the same as 2PC, however the decision / Commit phase is divided into 2 phases: Pre-Commit phase and Commit phase.

In the pre-commit phase, the coordinator will inform all the participants what decision it has taken, i.e. it will share the decision with the participants, however the participants don't act on the command just yet. This is not a command, and the participants don't need to take any action based on the pre-commit phase, i.e. the participants don't need to commit or abort the transaction. The coordinator is just sharing the decision, so that in case it goes down, the participants don't get blocked,

instead if it goes down the participants can act according to the decision shared in the pre-commit phase.

3PC is a non-blocking protocol.

In the pre-commit stage we are just sharing the information, we are not telling the participants to commit or abort. The participants will also store this info in their non-volatile log.

In the commit phase, the coordinator will send the COMMIT or ABORT message to the participants and this time the participants need to perform the required action, i.e. the participants will need to either commit or abort based on the decision taken.

Why is this protocol called non-blocking?
Suppose just before the commit phase, the coordinator goes down and it cannot relay the decision taken to the participants, in this case the participants won't need to boundlessly just keep waiting, after a certain period of time waiting for the message from the coordinator, the participants will just see the decision which had been shared during the pre-commit stage and act on it, i.e. they will either commit or abort the transaction, hence they wouldn't be blocked.

What if the coordinator fails before the pre-commit stage:
The different participants can talk to each other and check if the other participant has received the pre-commit message, if none of the participants have received it then after a certain time the participants will consider that the coordinator has failed, and will rollback the local changes. Participants can talk to each other both in case of 2PC and 3PC, if none of the participants has received the pre-commit message, then the transaction can be safely aborted, and the participants can unblock themselves.

The 3PC protocol is not very popular because of its complexity.

SAGA Transaction

2 PC and 3 PC are synchronous in nature, SAGA is asynchronous and this pattern is used to perform long lasting transactions, i.e. transactions involving many participants. SAGA Transaction is a sequential transaction.

2PC and 3PC protocols are not suitable for such transactions because it is not feasible to put locks on all the participants until the entire transaction is completed, because it is a long transaction.

So SAGA is used to perform long trailing transactions in an asynchronous manner, where we need to perform transactions across different participants in a sequential manner, i.e. only after T1 is committed can T2 start, only after T2 is committed can T3 start and so on.

Microservices and Microservice Architecture

- **Introduction to Microservices**
- **Microservices Design Patterns**

Introduction to Microservices

Microservices architecture is an architectural style that structures an application as a collection of services.
Services are designed to do one thing well and interact with other services.

Microservice architecture Principles:

1. High Cohesion and Low Coupling
2. Business Domain Modeling
3. Independent Deployability
4. Independent Scaling
5. Observability
6. Reliance
7. Team Organization

High Cohesion and Low Coupling

=> What is Cohesion
Cohesion is a measure of how closely elements within a single module work together to achieve a single well defined purpose.
For example if we have a service for Orders, then all the functionality, data and business logic related to Orders should reside within the order service or very close to the order service.

If cohesion is high it means the elements within a module are closely related and focussed on a single purpose.
If cohesion is low it means the elements within a module are loosely related and serve multiple purposes.

The services should be cohesive in nature, Higher the cohesion the better and hence we want our services to be highly cohesive.

=> What is Coupling
Coupling measures the interdependence between different software modules.

High Coupling means the modules are closely related, hence changes in one module affect other modules.
Low Coupling means the modules are loosely related or independent and the changes in one module don’t affect other modules.

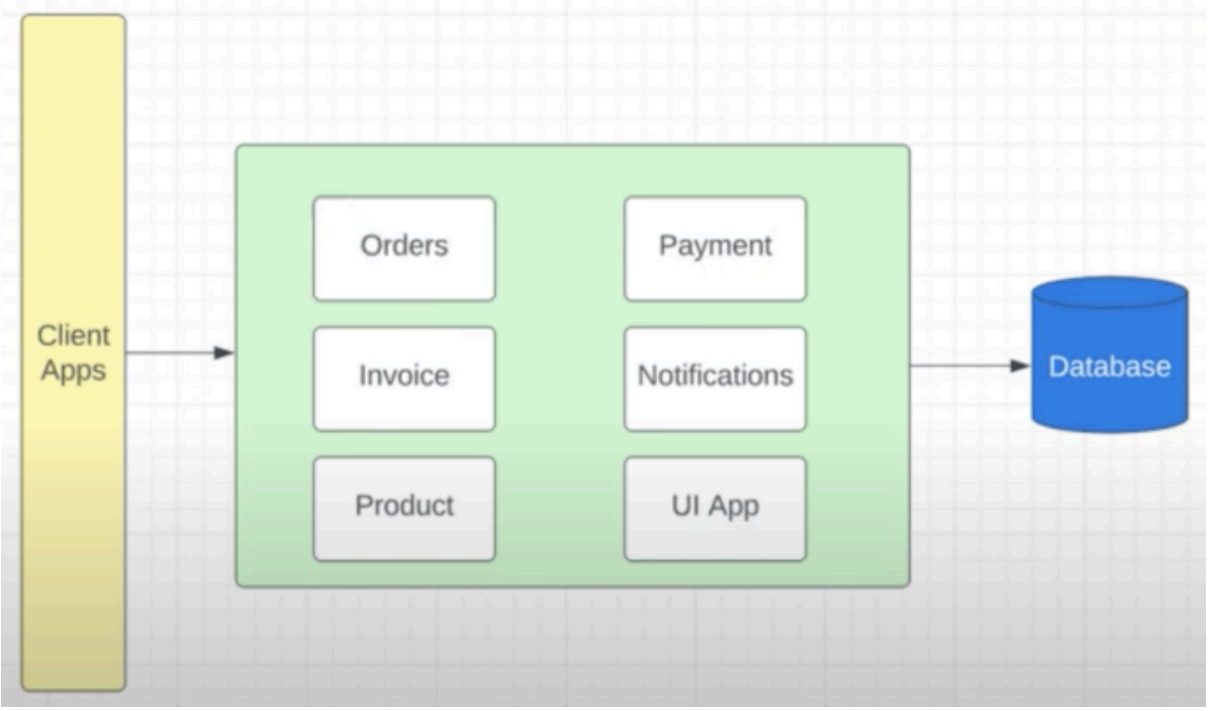
Lower the coupling the better, i.e. services shouldn’t be strongly coupled. The services should be loosely coupled.

Business Domain Modeling

Services are modeled around the business domain or business features.

Independent Deployability

The services can be deployed independently of each other. In a monolithic application if we make any change then we need to redeploy the entire application.



Consider the above system, here the code is modularized however it is still packaged as one unit. Suppose we make a change in a single module, for example Orders, we will still need to build the entire application and re-deploy it, hence the code cannot be independently deployed. In this case even though we have modularized the application it still suffers from coupling as a change in one module requires all the modules to be deployed again.

Independent Scaling

We can scale each service independently without touching the other services. For example if the orders service is not able to handle the load we can scale it horizontally, this doesn't affect other services. In the case of a monolithic architecture independent scaling is not possible.

Observability

Since our system has been divided into multiple services, we can more easily monitor them individually. For example we can check the health of all the services, record metrics etc, this enhances debuggability.

Resilience

In the case of microservices if one of the services goes down the other services are not affected. We need to build Fault Tolerance into the system so that there are no cascading failures. This ensures that even if a part of the system is failing, the whole system will not go down

Team Organization

Each service is owned and managed by a single small team, this increases service ownership and since the engineers need to focus on only a small part of the system (their service) hence they can ramp-up quickly and start contributing hence speeding up the delivery of features.

Advantages of Microservice Architecture

1. Autonomy of Tech Stack - Each service can be written in a different language, in a different tech stack which is more suitable for the use-case of that service. This allows us a lot of flexibility as we are not bound to a single Programming Language for example, this includes possibly having different databases (SQL / NoSQL) for different services.

2. Robustness (Reliability) - As mentioned before in the case of microservice architecture if one service goes down the other services are not affected, hence even if a part of the system is failing, the whole system will not go down. For example: if the invoice generation service goes down in an e-commerce application the customer can still browse through the products, make payments and place orders. Hence even when the invoice service was down

other services like the Catalog service, payment service, Notification services etc. continued to function as usual.

3. Every service has its own CI / CD pipeline.

Disadvantages of Microservice Architecture

1. Increased complexity and setup cost
2. Advanced monitoring is needed.
3. Distributed state management is a big problem, data might become inconsistent and Distributed Transactions are hard to perform.

Communication between Microservices

1. Synchronous (Blocking) Communication - A microserving calls another microservice and blocks its own execution waiting for the response, i.e. the service is waiting for the other service to send it a response.
2. Non-Synchronous (Non-Blocking) Communication - The microservice making the call does not wait for a response from the other microservice, it can carry on execution, there is no waiting.

Types of Synchronous Communication:

Request Response Model - A service sends a request to another request and waits for a response synchronously, when a response is received the service will process it and then carry on with execution.

Types of Asynchronous Communication:

1. Request Response Model: This is asynchronous so the service sends a request but doesn't wait for a response and can carry on with its execution and performing other tasks. However when a response is received the service will need to process it.

2. Event Driven Communication Model: Uses a messaging system like a Message Queue or Topic based pub / sub architecture. In this case the service sending the messages (the publisher) is not aware of the consumer services. Hence the publisher doesn't wait for a response, so it will not need to process any response.

3. Shared Database Model: Uses a shared database or a shared file system as an intermediary where services write their messages

which can be read by others. Not commonly used pattern, however it can be useful for implementing data lakes and analytics for the data lake.

Microservices Design Patterns

1. Distributed Transactions and the SAGA pattern
2. Database per service pattern
3. API Gateway pattern
4. Aggregator Pattern
5. Decomposition Patterns
6. Strangler Pattern
7. Circuit Breaker Pattern
8. Command Query Responsibility Segregation (CQRS) Pattern

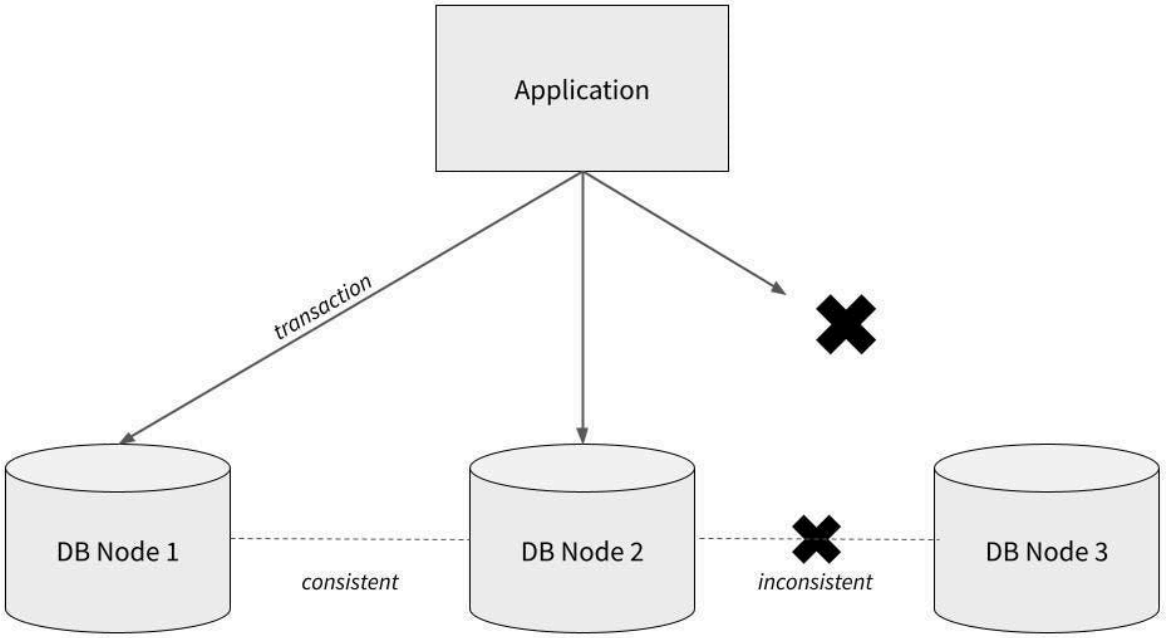
Distributed Transactions and the SAGA pattern

A transaction is a set of operations which modifies our data, a transaction should either execute completely or not at all, i.e. there is no intermediate state possible. In case of distributed transactions the data can reside on different nodes (i.e, different physical hardware), these nodes can be in different geographical locations and connected via a network.

A Transaction gone wrong: For example if we have an Orders service and Notification service where each service stores some state regarding a particular Order (identifier by the o_id), which tracks whether the order has been delivered or not. Suppose the order gets delivered and we wish to update the state in both the services, however for some issue (node failure) we could only update the data in the orders service. This leads to a data inconsistency problem as Order service says the order has been delivered while the Notification service says it hasn't been, in such a scenario we should roll back the transaction as a transaction should be atomic.

Distributed Transactions are more challenging because ensuring the data remains consistent on each node is complex.

Specific challenges in Distributed Transactions:
Imagine an application wants to commit a transaction to three separate database nodes (Perhaps the data is replicated on all three nodes, or the transaction affects multiple rows and the rows are stored on different nodes, the data could be sharded for example).



The above diagram depicts the transaction as can be seen the transaction is written successfully to DB Node 1 and DB Node 2 however it couldn't be written to DB Node 3 - perhaps due to a Network Disconnection or Node failure. This results in data inconsistency in the system as the first two nodes and the third node disagree about the state of the data, implying data integrity has been breached.

We need to ensure the data remains consistent even when it has been partitioned across multiple nodes.

How to perform Distributed Transaction

Two Phase Commit Pattern (2PC): In the 2PC pattern one node acts as the coordinator (also called the Transaction manager) while the other nodes are the participants (also called Resource Managers). The Resource Manager nodes are the ones which actually handle the data.

2PC divides the Distributed Transaction into 2 phase:

1. (Phase - I, Prepare) The coordinator asks all the participants to prepare to execute the proposed transaction and lock all the required resources (tables). When a participant is ready, it will report to the Coordinator that it is ready.

2. (Phase - II, Commit / Rollback) Once the Coordinator confirms that all the participants are ready it sends a commit command to all participants.

The 2-PC pattern is a blocking protocol as it requires all the participants to simultaneously acquire locks hence it hampers availability. If all the nodes confirm to the coordinator that they are ready, but before the coordinator can send a commit command it dies in such a situation all the participants will keep on waiting for the decision of the coordinator, this hampers availability.

Also we might encounter a situation like above diagram, where all nodes responded positively to the Coordinator however just before receiving the commit command one of the node dies or the network connectivity is lost in such a case the transaction will be written to nodes A and B but not to C, obviously resulting in data inconsistency.
https://stackoverflow.com/questions/50042776/in-2pc-what-happens-in-case-of-failure-to-commit

SAGA pattern

A SAGA is a series of local transactions, the SAGA pattern can help maintain data consistency during distributed transactions.

The idea is to write a long-lived transaction as a series of multiple short local transactions. If all sub transactions of a SAGA are complete, then the SAGA completes successfully. If one sub-transaction fails, the compensating transactions will be invoked one at a time in the reverse order to rollback the sub-transactions (This is called Backward Recovery).

To implement a SAGA pattern we have two common approaches:

1. Choreography - Using the Choreography approach a service will perform a transaction and then publish an event. Some other service will respond to that published event and perform its own local transaction. This secondary service will in turn publish an event which will be a signal to some other service to start performing its local transaction. In other words when a service completes a local transaction it publishes an event that triggers a local transaction in another service.

2. Orchestration - In the case of Orchestration we designate a service as the orchestrator. The Orchestrator will trigger other services to respond once their local transaction is completed, the orchestrator tells the participants what local transactions to execute. In other words, the orchestrator service has all the command and control and tells the other services what to do.

Database per service pattern

Database organization affects the efficiency and complexity of the application. When designing an application, the developer needs to make a choice with regards to organizing the database and where it should reside.

a. Dedicated database for each service: In this approach each service has its own dedicated database. A service cannot access another service's dedicated database.
   This approach allows for a lot of flexibility as each service can pick a database suitable for its use case, for example SQL or NoSQL or what kind of Non-Relational database.
   This makes the system easier to scale, as each service can scale independently of each other, also it reduces coupling as one service cannot tie itself to the data of another service, instead the Services are forced to communicate via APIs.

b. Single database shared by all services: This approach is not commonly used in Microservice architecture, as it does not provide flexibility or independent scalability. However if for some use-case a single database is shared by all the services then it's very important to enforce boundaries, and ensure that a service can't interfere with another service's data or tables.

API Gateway pattern

An API Gateway acts as a single entry point for all calls to any microservice behind it, and the API gateway will route the request to the concerned microservice.

It is beneficial for the client as it doesn't need to know where to connect to each of the microservices, it can simply connect to the API gateway and send requests to it, which will in turn route them to the correct backend service, this is crucial as the microservices in the backend keep on changing and evolving.

Furthermore it boosts system security as the client cannot directly connect to the microservice, as the API endpoints aren't directly exposed and the client instead needs to connect to the API gateway, which takes care of authentication, rate limiting and SSL Termination and even protocol translation (eg. AMQP to HTTP).

API gateway is basically a layer of abstraction, which abstracts complex connectivity details from the clients.

Aggregator Pattern

In a microservice architecture the functionality is divided across multiple services, hence it might be possible that we need to aggregate data from different services and then send the final response to the consumer. This can be done in following ways:

a. Composite Microservice - A dedicated microservice responsible for aggregating the data. This composite microservice will make calls to all the required microservices, consolidate the data returned by them and transform the data as needed before sending it back to the consumer.

b. API Gateway - An API Gateway can also perform the role of aggregator, i.e. it will partition the request to multiple microservices and aggregate the data before sending it back to the consumer.

Decomposition Pattern

Decomposition design patterns are used to break a monolithic application into smaller microservices. A developer can achieve this in the following ways:

1. Decomposition by Business Capability: A business capability is something a business does to generate value. For example the

capabilities of an e-commerce company include managing product catalog and inventory, Orders, and delivery. Each business capability can be modeled as a separate service.

2. Decomposition by Subdomain: This is well suited for exceptionally large and complex applications. The application is broken down into subdomains which makes managing the codebase easier, Each subdomain can be modeled as a separate service.

3. Decomposition by Transactions: This is an appropriate pattern for many transactions that involve multiple components or services. For example if a transaction involving multiple components occurs frequently then we can model these components together as a single service.

Strangler Pattern

The Strangler pattern is used to incrementally transform a monolithic application to the microservices architecture. This is accomplished by gradually replacing the old functionality with the new services, once the new service is ready to take over, the older monolithic functionality is strangled so that the new service can take over.

Strangler Pattern is useful for transforming large scale monolithic applications to microservices architecture (example legacy codebases). The Strangler pattern offers gradual migration from the monolithic architecture to the microservices architecture, and during the migration process both these architectures can co-exist, allowing for uninterrupted system functionality.

In conclusion the Stranger Pattern incrementally replaces components or modules of the Monolithic application with microservices, leading to the eventual replacement (strangulation) of the legacy system.

Circuit Breaker Pattern

The Circuit breaker pattern is used to deal with Non-Transient failures in microservices.

Non-Transient Failures: Any failure which can render the system unavailable for a certain period of time or the recovery can take longer than a few seconds is known as a Non-transient failure. Examples: Databases going down, APIs timing out due to CPU / memory issues.

The circuit breaker pattern is usually applied between services that are communicating synchronously, this pattern helps to avoid cascading failures in the system.

When a service is down we stop the calls to the failing service by applying the Circuit Breaker Pattern, we stop the calls for some time until the failing service recovers. Therefore calls won't be piling up and using the system resources, which could cause significant delays and even result in a string of service failures.

How does the Circuit Breaker Pattern work?

The Circuit Breaker Module monitors failure conditions, when a failure condition is detected the circuit breaker will trip (i.e. transition to open state), once it has tripped all the calls to the circuit breaker will be rejected or a default error message will be returned.

Suppose a circuit breaker module sits b/w A and B, and A is sending requests to B. The circuit breaker module will determine if B's health is good enough to respond to requests from A. For example if the database connected to B goes down, the circuit breaker will determine that the service is not in a healthy state and hence cannot take requests from A. In other words, the circuit breaker logic will decide if the requests from A to B will be passed through, rejected or returned with an error. So if there are some underlying issues with B, then the circuit breaker will prevent A from sending requests to it, so that B does not degrade further.

States of a Circuit Breaker

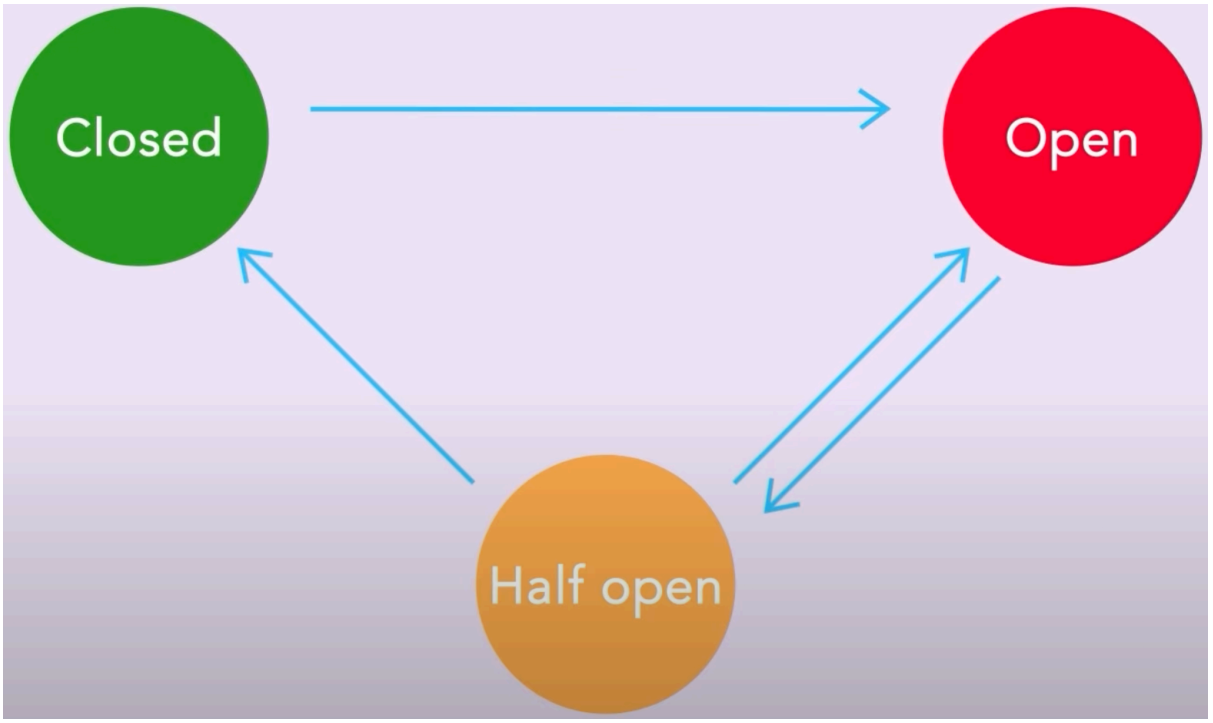
1. Closed State: By default a circuit breaker is in the closed state, that means traffic from A to B will be allowed to pass through and the response will also be returned from service B to A.

2. Half Open State: When a circuit breaker is checking for underlying problems, it remains in the half open state, some traffic can go through for testing purposes to verify if the calling service can

actually connect to the downstream service, and if there are any errors.

3. Open State: A circuit breaker is open when the number of failures has exceeded the threshold, in this case no traffic from A to B will be allowed to go through, the calling service will be returned an error informing that the service they are trying to connect to is down.

A timeout or delay is defined in association with the open state, i.e if the circuit breaker is in open state then it will transition to half-open state every time this timeout expires (for example once every 50 ms) to check if the service has recovered from the failure or not, if it has the circuit breaker transitions to the closed state else it moves back to open state. However what is the criteria based on which we transition from half-open to closed state? In the half open state there is a success counter and we try to send a call to the remote service if there are no errors we increment the success counter, and if this success counter exceeds a predefined threshold then we can transition the circuit breaker to closed state, however if errors are still encountered then the circuit breaker moves back to the open state.



Hence, the Circuit Breaker pattern is an effective way to add resilience into the system, however this pattern should not be used for transient failures.

Command Query Responsibility Segregation (CQRS) Pattern

CQRS pattern suggests splitting the application into two parts - the command side and the query side. The command side handles Create, Update and Delete requests. The query side handles the query part by using the materialized views, multiple views can be provided for query purposes. This allows for the read and write side of the system to be scaled independently.

In summary the command side handles the Create, Update and Delete Requests and persistence of data, and acts as a data source for the query side.

The query side of the system generates projections of the data, which are highly denormalized views.

System Design Case Studies

- **Back Of the Envelope Estimation OR T-Shirt Scoping Estimates**
- **URL Shortener**
- **Designing a chat application**
- **Unique ID Generator in a distributed system**
- **Rate Limiter**
- **Design a Notification System**
- **Newsfeed**
- **Design a key value store (Concepts)**
- **Design a Search Autocomplete System**
- **Design a Web Crawler**
- **Ola / Uber System Design**
- **Design a hotel booking service like Airbnb, Booking.com**
- **Tinder System Design**

Back Of the Envelope Estimation OR T-Shirt Scoping Estimates

Get an idea of what we are dealing with.
Get an estimate of:

1. Traffic, compute the requests per second (read) and requests per second (write)
2. Storage Requirements
3. Finer estimations: Number of servers needed, CAP tradeoffs, RAM etc.

Give high level design first and then drill down on specifics.
So here is the order:

1. Back of the envelope estimates discussion
2. High level design buy in
3. Drill down on specific components of the design
4. Wrap Up: Potential Improvements / Bottlenecks and feedback

Size of Data Types

Data Type	Size
int	4 bytes
char	1 byte
long	8 bytes
short	2 bytes
double	8 bytes

URL Shortener

Designing a URL shortener service like tinyurl.

Clarification questions:-

1. Can you give an example that shows how the URL shortener works?
2. What is the traffic volume?
3. Which characters can the shortened URL contain?
4. Once a user has generated a short URL, then for long will it be valid also can a user delete a short URL.

System Design:
https://drive.google.com/file/d/1pEqPVhAo3d2r-8a0PMuWFJSikN81jcT2/view?usp=sharing

Points to Consider:

A note on Redirection Status Codes

Terminology:
301 - Moved Permanently
302 - Found

301 - A 301 redirect indicates that the requested page has been moved permanently to the long URL, since it has been moved permanently the browser caches the response and subsequent requests for the same short URL, will not be sent to the URL Shortening service, instead the requests are redirected to the long URL directly, by the browser using the cached data.

302 - A 302 redirect indicates that the requested page has been moved temporarily to the long URL, meaning that subsequent requests for the same URL, will be sent to the URL shortening Service first. Then the requests are redirected to the long URL by the URL Shortening service. Hence in case of a 302 redirect, the browser doesn’t cache the response.

Designing a Chat Application

Design a chat application like whatsapp or FB Messenger

Clarification Questions:-

1. What kind of chat app are we talking about? Does it support 1 to 1 messaging or group chats or both?
2. What is the traffic volume?
3. How long do we need to save the chats?
4. What can a chat message consist of, and what is the text limit?
5. How many people can be there in a group, max?
6. Is it a web application or mobile app?

System Design:
https://drive.google.com/file/d/1dNz3fC9rIagATkXVY3omTX2KDiwP63Ou/view?usp=drive_link

Points to Consider:

1. Polling / Long Polling / Web Socket
2. Key Value Store
3. Messaging Queue - SQS / SNS
4. Local Storage
5. Network Protocols - TCP, HTTP (keep-alive)
6. DynamoDB: Partition Key and Sort Key
7. Pub Sub Model

Additions, Online Presence: How do we design a feature showing whether the user is logged in and if he's not then show the time when the user was last active. (this information could be presented as: Last Active 30 mins ago, for example).

We need to have a database, which records the time when the user was last active. Whenever we want to get the status of a user we can just query this database.

How do we populate this db:

1. Whenever a user performs any activity, like post a message or fetch messages from the server, or login then we update the last active

database entry for that user. When the user has been inactive for some time, he’s marked offline.

2. Heartbeat Mechanism - An online client sends a heartbeat event to the presence servers periodically, if the presence server receives a heartbeat within a certain time, the client is considered online, otherwise it is offline.

Design a Unique ID Generator in a Distributed System

In a non distributed environment, we can use the primary key with the auto increment attribute to generate unique IDs, however as the data volumes grow bigger and bigger one single database will not be able to handle all of it, and we'll to perform horizontal Scaling (Sharding), generating globally unique IDs in a sharded distributed environment is complex, additionally it is to be noted that No-SQL databases like Cassandra have no built it method to generate unique IDs.

Requirements from IDs:

IDs must be unique.
- IDs are numerical values only.
- IDs fit into 64-bit.
- IDs are ordered by date.
- Ability to generate over 10,000 unique IDs per second.

Solutions:

1. Increment by K - If we are using a Relational database, like MySQL then we can get access to the primary key with the auto increment feature. However as the data volumes grow larger we will need to horizontally scale this database into multiple smaller shards.
   Suppose we have divided our database into K shards.
   In such a scenario we can generate global unique IDs as follows:

   Instead of increasing the next Id by 1 (As auto increment does), we increase the next Id by K, i.e. next Id = previous Id + K.

   Additionally we need to configure where the database servers start generating primary keys from. In a normal situation, the database starts generating primary keys from 1.
   However in our case we will need to change this behaviour as follows:

   Server 1 start generating Primary Keys from 1
   Server 2 start generating Primary Keys from 2
   …..
   Server k start generating Primary keys from k

Hence the id's generate by the first server will be 1, 1 + k, 1 + 2*k ….
Second sever: 2, 2 + k, 2 + 2*k …..

These values will be globally unique.

Disadvantages: This approach is not flexible, it does not scale well when a server is added or removed, i.e. k changes.

2. Using UUID (Universally Unique Identifier) - UUID is a 128 bit number used to identify information in computer systems, UUIDs generated have a very low chance of collision. If we generate 1 billion UUIDs per second for about 86 years, then the chance of collision will be about 50%. Hence as we can see the chances of collision are very low. Additionally UUIDs can be generated independently without coordination between servers. (UUIDs are also called GUID - Globally Unique Identifier).
   Advantages:
   a. Simple to generate, No coordination between the servers is needed
   b. ID Generation is easy to scale, as each server is responsible for generating IDs they need, hence the ID Generator can scale easily with the number of servers.

   Disadvantages:
   a. In our use case, we need the ID's to increase with time however this requirement is not fulfilled by using UUIDs.
   b. We have a 64 bit ID requirement, however UUID is 128 bits.
3. Ticket Server - We can use a ticket server to generate and distribute globally unique IDs. This system was developed by Flicker.
   The idea is to use a centralized Ticket server, each web server which needs an ID will communicate with the ticket server. The ticket server runs a relational database like MySQL, which supports the auto increment feature.

   So, whenever a web server needs a unique ID it sends a request to the Ticket Server, the ticket server will insert a dummy item into the relational database corresponding to each request, and return the primary key of this newly inserted item.

Question - If we keep inserting these dummy items wouldn't the database run out of storage.
We do insert a dummy item corresponding to each request however, the total number of records in the database at any point is 1.

https://code.flickr.net/2010/02/08/ticket-servers-distributed-unique-primary-keys-on-the-cheap/

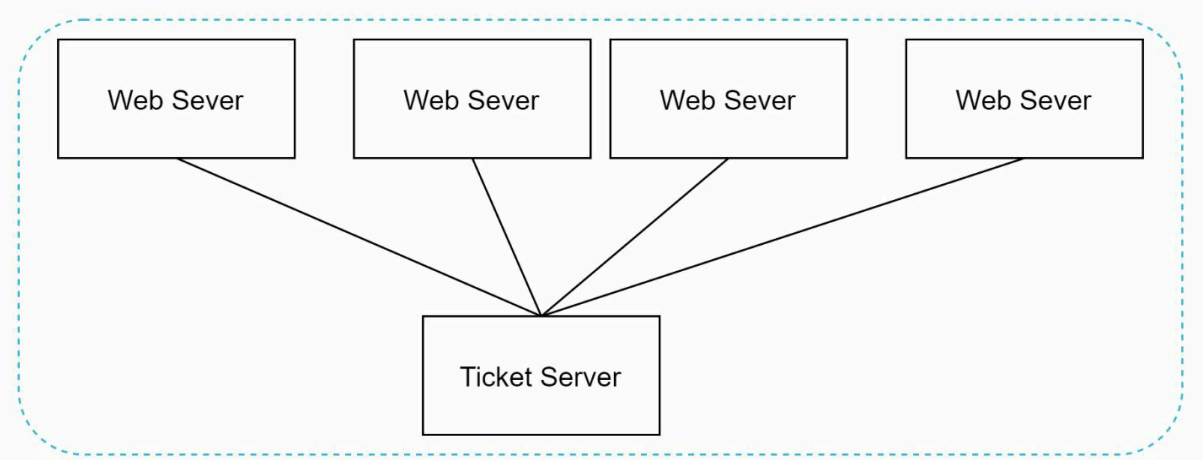


Figure 7-4

Advantage:

a. Easy to implement, works well for small to mid scale applications.
b. Numeric IDs, additionally IDs increase with time.

Disadvantages:

Single Point of Failure - What if the ticket server goes down, all the components of the systems that depend on it will face issues. To avoid this single point of failure, we can set up multiple ticket servers. However this will introduce new challenges like Data Synchronisation, refer the above link to see how Flicker achieves this.

4. Twitter Snowflake Approach - Snowflake is twitter's unique ID generation system. Snowflake generates a 64 bit long ID, the generated ID consists of the following sections:
    a. Sign Bit (1 bit) - It is always 0, this is reserved for future use cases
    b. Timestamp (41 bits) - Milliseconds since the epoch (Jan 1, 1970) or custom epoch. Snowflake uses a custom epoch of November 4th, 2010.

c. Datacenter ID (5 bits) - Which gives us 2^5 = 32 datacenters
d. Machine ID (5 bits) - Which gives us 2^5 = 32 machines per datacenter.
e. Sequence Number (12 bits) - For every ID generated on a machine the sequence number is incremented by 1, the number is reset to 0 every millisecond.

The timestamp field, stores the milliseconds since the epoch or custom epoch, it is 41 bit long hence the maximum value it can store is 2^41 - 1 ~ 70 years, i.e. post 70 years this approach will not be able to store the timestamp. Hence using a custom epoch closer to today's date delays the overflow time.

For example 70 years from 1970 is 2040
However 70 years from 2010 is 2080, hence this significantly delays the overflow time.

Sequence number is 12 bits long, for every ID generated on a machine the sequence number is incremented by 1, the sequence number is reset to 0 every millisecond, hence a machine can support up to 4096 (2^12) new IDs per millisecond.

Design a Rate Limiter

Clarification questions:-

1. What kind of rate limiter are we designing? Is this a client - side rate limiter or server - side rate limiter?
2. What is the scale of the system, and what is the traffic volume?
3. Does the rate limiter throttle requests based on IP addresses, user id or some other properties.
4. Will the system work in a distributed environment?
5. Do we need to inform the users who are throttled?

Note: The Rate Limiter should work in a distributed environment, hence multiple servers can share the same rate limiter.

Why to use a Rate Limiter?

1. Prevent resource starvation caused due to Denial of Service Attacks (DOS), almost all APIs published by larger tech companies use some form of rate limiting. A rate limiter can prevent DOS attacks by blocking excessive calls.
2. Reduce excessive Cost - Rate Limiting is useful for reducing costs when third party APIs are being used, which are charged on a per-call basis. Limiting excessive calls helps to reduce costs.
3. Prevent Servers from being overloaded - Excessive requests can overwhelm the servers, a rate limiter helps by discarding excess traffic, and thus the servers aren’t overloaded.

Where to put the Rate Limiter?

1. Client-Side: The rate limiter can be put on the client side, so that excessive requests will be dropped at client-side only and thus will never reach the server. However this is unreliable as malicious actors can forge client requests. Moreover we might not have control over the client implementation.

2. Server - Side: A more common approach is to place the rate limiter on the server side.
   We can place the rate limiter in the API servers, or we can create a rate limiter middleware which will throttle requests to the APIs.

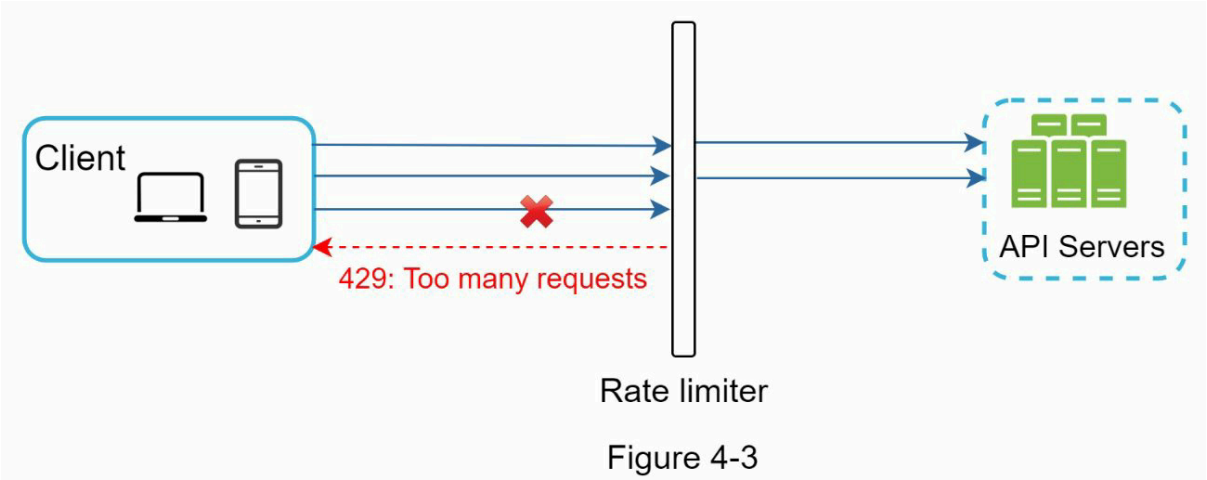


Figure 4-3

How to inform the client that the request was throttled?
HTTP status code: 429, indicates that a user has sent too many requests.

Note: API Gateway supports Rate Limiting natively. API Gateway (for example Amazon API Gateway) is a fully managed service that supports rate limiting, SSL termination, authentication and IP whitelisting.

In case of Rate Limiting we define a threshold which indicates the number of requests per unit time which the system can accept.

Algorithms for rate limiting

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

Token Bucket Algorithm

- A token bucket is a container that has a predefined capacity. Tokens are put into the bucket at preset rates periodically.
- For example the bucket capacity could be 4 tokens, and the refill rate could be 2 tokens per minute.
- Once the bucket is full, and extra tokens are added then the bucket will overflow, i.e. the extra tokens won't be added to the bucket.
- Each request consumes one token, whenever a request comes in we check if there are tokens available, if there are we remove one token from the bucket and the request is allowed to go through by the rate limiter, if sufficient tokens are not available, then the request is dropped.

- The token bucket algorithm has 2 parameters: Bucket size (maximum number of tokens which the bucket can hold) and refill rate (Number of tokens put into the bucket every second).
- Advantages:
    a. Easy to implement
    b. Memory efficient
    c. Token bucket allows a burst of traffic for short durations.
- Disadvantages:
    - It might be challenging to tune both the parameters properly.

Leaking Bucket Algorithm:

- Leacking Bucket Algorithm is similar to Token Bucket Algorithm, however in case of Leaking Bucket Algorithm the requests are processed at a fixed rate. For example 3 requests per second. It is implemented by using a FIFO queue. It works as follows:
    - When a request arrives, the system checks if the queue is full, if it is not full the request is added to the queue.
    - If the queue is full, the request is dropped.
    - Requests are polled from the queue and processed at regular intervals.
- Parameters of Leaking Bucket Algorithm
    - Bucket Size (or Queue Size), how many requests can be accommodated in the queue.
    - Outflow Rate - Rate at which the requests are processed, for example 5 requests per second.
- Advantages:
    - Memory efficient as queue size is limited
    - Requests are processed at a fixed rate, therefore it is suitable when a steady outflow rate is needed.
- Disadvantages:
    - If a burst of requests come in it will fill up the queue, and if they are not processed in time then recent requests will be rate limited.
    - Again tuning both the parameters properly is challenging.

Fixed Window Counter Algorithm:

- The Fixed Window Counter Algorithm divides the timeline into fixed-sized time windows, and assigns a counter for each window. (initialized to 0)

- Each incoming request increments the counter of the current window by 1.
- Once the counter reaches a predefined threshold, then new requests will be dropped until a new time window starts.
- For example, say we divide the timeline into 1 minute windows, and define the threshold as 4 requests per minute. So once the system has received 4 requests, the counter will be 4 (equal to threshold) and thus, all the new incoming requests will be dropped until a new time window starts.
- Advantages
    - Memory Efficient
- Disadvantages
    - Spike in traffic at the edges of the windows can cause more requests than the threshold to go through.

Sliding Window Log Algorithm:

- A major disadvantage with the fixed window counter algorithm is that a spike in traffic at the edges of the windows can cause more requests than the threshold to go through.
- To solve this problem we can use the Sliding Window Log Algorithm.
- This algorithm keeps track of the timestamps of the requests. Timestamp data is generally stored in a cache, for example Redis sorted sets.
- When a new request comes in
    - Remove all the outdated timestamps from the log. An outdated timestamp is a timestamp which is older than the start of the current window, i.e. if ts < cur_timestamp - window_len, then it is considered to be an outdated timestamp.
    - Add the timestamp of the incoming request to the log
    - If the log size is smaller than or equal to the threshold then the request is allowed to go through or else it is rejected.
- We can define the length of the sliding window, for example 1 min.
- Advantages:
    - This algorithm implements rate limiting very accurately, in any rolling window, excess requests will not go through.
- Disadvantages:
    - As this algorithm stores the timestamps of all the requests even the ones which have been rejected, hence it consumes a lot of memory.

Sliding Window Counter Algorithm

Q. How does a client know that it has been rate limited?
HTTP Response headers and HTTP status code are used for this.
The Rate Limiter returns the following HTTP header to the client -

a. X-Ratelimit-Remaining: Number of remaining requests allowed within the current window.
b. X-Ratelimit-Limit: Number of requests the client can make per window.
c. X-Ratelimit-Retry-After (seems deprecated, use X-Ratelimit-Reset instead): If the request from the client is throttled, then this header is returned to the client, it indicates the number of seconds the client needs to wait before making further requests without getting throttled.

If a client request is throttled, a HTTP Response code of 429 is returned, indicating that the client has made too many requests, along with it the X-Ratelimit-Retry-After header is also returned to the user.

https://drive.google.com/file/d/1K87CG-1Q4WgJf2BDxNfxDKW7MvYB9jun/view?usp=sharing

Designing a Notification System

Clarification Questions -

1. What types of notification does the system support?
2. How many notifications will we send out in a day?
3. What are the supported devices?
4. Will the users be able to opt-out?
5. What triggers the notifications?

Different Types of Notifications:

1. IOS Push Notifications: To send a an iOS notification we need the following components:
    a. Provider - The provider is the component which builds and sends the notification to the APNS (Apple Push Notification Service). A notifications will consist of:
        i. Device Token - A unique identifier used for sending push notifications.
        ii. Payload: The notification's payload is encoded as a JSON dictionary, containing information like Notification title, body etc.

    b. APNS (Apple push Notification Service) - APNS is a remote service provided by Apple to propagate push notifications to iOS devices.

    c. iOS Device - It is the end client which receives the push notification.

2. Android Push Notifications - Android has a similar flow for sending push notifications as iOS. However, instead of APNS, Firebase Cloud Messaging (FCM) is commonly used to propagate push notifications to Android Devices.

3. SMS Notifications - For sending SMS messages, third party commercial SMS services like Twillio or Nexmo etc. are commonly used.

4. Email Notifications - Companies can either set up their own email servers or they can opt for third party commercial Email services,

for example MailChimp, SendGrid are commonly used as they offer a high delivery rate and data analytics.

A Note on Device Tokens
Device Token, also called Push token, is a unique key for the app-device combination which is issued by APNS or FCM. It allows notification providers to route the messages and ensures that the message is delivered only to the unique app-device combination for which it is intended.

Let’s take the example of FCM, when an application is registered with FCM, a new device token (or registration ID) is issued by FCM, the application should save this token in some database for future use, when we wish to send a notification to a user. As mentioned before when sending a notification the provider must specify the device token.

A note on FCM device tokens:
FCM device token are automatically renewed when:

a. The app is restored on a new device
b. The user uninstalls / reinstalls an app
c. The user clears app data

To detect these changes we need to implement the FCM onNewToken handler or the OnTokenRefresh handler, and update the token stored in our application, and remove any stale tokens.

When a token has been inactive for 270 days, then FCM will consider it as expired. Once a token has expired FCM marks it as invalid and rejects sends to it.

Designing a Newsfeed

Clarification Questions -

1. What are the important features of a Newsfeed?
2. Does the Newsfeed contain images, videos or just texts.
3. How many friends can a user have?
4. What is the traffic volume on our application
5. Is it a mobile app or a web app or both?

Important Components of the system

1. Feed Publishing - A user can publish a post, a post can contain text, images or videos. The post data should be stored in cache and database. The post should be populated to the news feed of the user's friends.
2. News Feed Building - News feed is built by aggregating posts from user's friends. In What order should the posts be aggregated? Reverse Chronological makes sense, where the most recent post will be rendered on the top, or we can sort them by likes.

API Design: We need to design some APIs so that the client can interact with our service. We need to design APIs for the following interactions:

1. User publishes a post
2. User retrieves his / her post
3. For friend requests
4. Authentication

API design for User publishing a post

REQUEST METHOD: POST
API PATH: /api/v1/feed/publish
Content is stored in request's body
=> This API will extract the post's content and add it to the database / cache.
=> In addition some sort of auth_token will be used to authenticate requests.

API design for User retrieving posts

REQUEST METHOD: GET
API PATH: /api/v1/feed/get
=> Retrieves the user's news feed which contains posts published by the user's friends.

Different services for our design

1. Posts Service: This service will handle the post publishing component and add the posts to the posts database and cache.
2. User Feed Service: When the user wants to retrieve his / her news feed they'll send a GET request to /feed/get, which will be handled by the User Feed Service.
3. Notification Service: When a user makes a post, this service will send a notification (push notification) to all the friends of the user.
4. Follows Service: The follow service maintains the follower / followee relationships

Posts Service:

The post service will interact with the cache and the database layer, to make the service resilient and scale for higher workloads we will have multiple servers running the same service. To balance the load across them we can use a load balancer. The load can be balanced across the servers by using consistent hashing, by hashing on the user_id.

Schema of the posts database:

user_id USER,
content TEXT,
created_at TIMESTAMP

User Feed Service:

When a user wants to retrieve his / her news feed, they'll send a GET request to the /api/v1/feed/get API discussed above, requests for this API will be handled by the User Feed Service. This service interacts with the Posts service and the following service.

Follows Service:

It stores the follower / followee mappings. It can help to answer questions like: Who are the followers of a given user.

The Follows service will store this data in a database,
follower_id USER,
followee_id USER

The mapping b/w follower_id and followeee_id can be one way (for example Instagram) or two way (for example Facebook).

There are other services as well like:
=> Authentication service [Handle user authentication, sign in and sign up, forget password etc.]
=> Users service [Store the user data]
=> Media Service - Store the images / videos from the posts

Flow for retrieving the News Feed

We can generate the news feed in 2 ways:

1. Precomputation [Fanount on write] - The news feed is precomputed during write time. Whenever a user makes a post the post will be added to the news feed of all the followers of the user.

   => Since the news feed is precomputed, fetching the news feed is faster.

   => However if a user has a lot of followers (celebrity) then pushing the post to the news feed of all the possibly millions of users is a very expensive operation
   => For inactive or rarely active users, pre-computing the entire news feed does not make sense.

2. On Demand [Fanout on read] - The news feed is generated during read time. This is an on-demand model, whenever a user logs in the most recent x posts by all of the user's friends are pulled in.

   => More suitable approach when dealing with inactive users, as resources are not wasted for these users.

   => Fetching the news feed is slower, since it is not precomputed.

Description of the On-Demand model:

We send a request to the user feed service, with the user_id. User feed service will identify all the users followed by the user with the specified user_id, by sending a request to the follows service.

API:

getUsersFollowedBy(user_id)

Return:
Set<User> following;

Next we can get all the posts made by these users, by querying the Posts service.

API:
API getPostsByUser(user_id)
Return:
Set<Post> users;

Or a bulk request API
API:
getPostByUserBulkRequest(Set<User> users_id, int limit)
Return:
Set<Post> posts;

Description of the Precomputation model:

When a user makes a post, the Post's service notifies the user feed service. The user feed service quereis the follows service to get the list of all friends of this user.

API:
getUsersFollowing(user_id)
Return:
Set<User> followers;

Then the user feed service will update the post to the newsfeed for each of these users (followers). So the news feed of all the followers will be updated in an incremental manner, we can limit the size of the news feed to k, so that if we try to add the (k + 1)th post then the oldest post will be deleted from the newsfeed.

Where to store these news feeds: We can represent the newsfeed as a list of posts. In this case when a user posts we simply add another entry to each of the lists representing the news feed of the followers. Size of the newsfeed is limited to k.

Where to store the newsfeed?

We can store the newsfeed in Database or Cache.
A cache is more suitable, as it will allow faster access. Also if we lose some data we can compute it again from the database, i.e. the cache can store per-user news feed.

As the cache needs to be memory efficient, instead of storing the actual posts or the actual user objects, we will just store the user and post ID to get the actual results, some joins will need to be performed.

So this is how the data inside the cache will look like

post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id

Additionally since memory is limited in a cache, we will limit the news feed to a particular size and use an algorithm like LRU to only keep the most recent posts in the cache.

Notification Service: The Notification service will be useful in context of the precomputation model, whenever a user publishes a post, the User Feed service will call the Notification service to send a push notification to the user, that they have unread posts.

API

sendNotificationToUser(user_id)

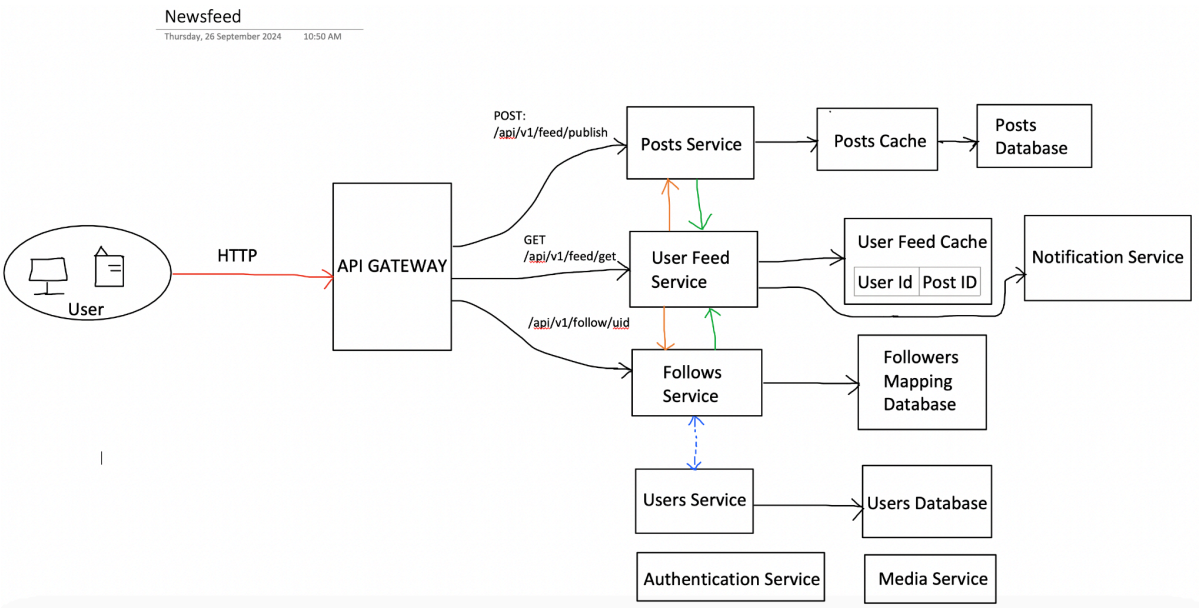
RETURN
    void

However if we have to send these notifications to many users - we can make use of the pub / sub model or services like AWS SNS. If the number of notifications to be sent out is very large, we can process them in a batched manner to prevent the system from getting overwhelmed, and rate limiting.

However in the case of celebrities, the number of followers is in the millions and adding the post to the news feed of millions of users is a very expensive operation. In such cases the On-Demand [Fanount on read model] might be more suitable.

Also for inactive users, or those who use the app very infrequently the precomputation (Fanout on Write) model will not be efficient, and will lead to wastage of memory. For such users it's better to use the On-Demand model. The News feed cache has limited size, and we will need to use some cache eviction algorithm like LRU to evict older entries from the cache. Hence as a result the more active users will have their user feeds served by the cache, however for people logging in infrequently there is no need to waste memory for them.

So in conclusion for News feed building we need to use a hybrid model, for a celebrity we use the pull model, and for other normal users we use the push model which offers a seamless experience.



Implementation Notes:

=> The user feed service in case of precomputation model will continuously send updates to the News feed cache and the Notification service, to prevent overwhelming them, we decouple the communication b/w these services by using a Message Queue (for example AWS SQS queue) which can store the messages and the end services can process them at their own rate.

=> Also we can implement some sort of rate limiting mechanism for the notification service.

=> To prevent overwhelming the posts service we can enforce a limit on the number of posts a user can make per day, this also helps prevent spam.

=> The following service will need to use a database to maintain relationships, between followee and followee id.

A simple Relational Database like Mysql with the following schema can be used:

```
followee_id INT,
follower_id INT,
FOREIGN KEY(followee_id) REFERENCES users(id),
FOREIGN KEY(follower_id) REFERENCES users(id)
```

However we can use a graph database like AWS Neptune as well. Graph databases are well suited for storing friend relationships, and help in enabling features like Friend Recommendations, this is because the friends network can be visualized as a graph.

=> The system will certainly store a lot of info for the user, like name, images, etc. It should in addition also store user settings. These user settings will help the system to filter the posts and render only the ones the user is interested in. For example in facebook Alice and Bob can be friends, but Alice can block Bob or Bob can mute Alice, in either case if one party has blocked or muted the other they still remain friends, but the first party should not see any post from the other one, vice versa is not true.

=> Joining data

When the user wants to retrieve his / her news feed, the user feed service will get this data from the news feed cache. However as mentioned before the news feed cache doesn’t contain the entire data, it just contains <user id, post id> mappings. To get the actual news feed the User Feed service will need to join the posts id’s with with the Posts Database (or Posts cache), to get the full posts data like content, image URLs to be rendered, etc to construct a fully hydrated newsfeed, which will be returned to the client in JSON format, and the client (Mobile App or Web browser) will render the content.

Design a key value store

A key value store also known as key value database, stores the data in key value mappings. The key can be anything int, string etc. They can be a plain value or a hashed value. The keys are associated with a value.

We need to design a system which supports the following operation:

put(key, value)
get(key)

Starting simple, what if we had to design a key value store located and running entirely within a single server. This would be straightforward as we can use a HashMap for example and store the key, value mappings. However it would not be possible to fit everything inside the RAM, as the memory requirements for the key-value store is much higher. A single server running the key value store will hit the capacity very quickly, additionally it suffers from a single point of failure.

To create a real world key-value store we'll need to use multiple servers, and distribute the key value pairs across many servers. Hence we need to develop a distributed system, so we'll next discuss how to design such a system, with an emphasis on designing distributed systems.

Debrief: The CAP theorem (Consistency, Availability and Partition tolerance) algorithm states that it is impossible for a distributed system to simultaneously provide all three guarantees: Consistency, Availability and partition tolerance.

Consistency - Consistency means all the nodes have the same view of data at any moment, from a user / client's perspective this means that all clients see the same data at the same time, no matter which node / server they connect to.

Availability - Availability means the client which requests data will get a response even if some of the nodes are down.

Partition Tolerance - Partition tolerance means the ability of the system to continue operating even in the presence of network partitions.

A partition indicates a breakdown of communication b/w two nodes.

A key value store can be categorized using the CAP theorem into 3 categories, depending on which of the properties they support.

CP (Consistency and Partition tolerance) Systems - As the name suggests these systems prioritize consistency and partition tolerance over Availability.

AP (Availability and Partition tolerance) Systems - These systems prioritize Availability and Partition tolerance over Consistency.

CA (Consistency and Availability) Systems - These systems support Consistency and Availability over Partition tolerance. However since in a distributed system, network failures are unavoidable, hence the system must tolerate network partitions. Thus a CA system cannot exist in real-world applications.

An example

In a distributed system data is replicated across multiple nodes, assuming the data is replicated on 3 nodes - n1, n2 and n3.

n1, n2, n3 communicate with each other and synchronize the data among them. In an ideal situation where network failures never occur, any of them can accept read / writes, and the data will be replicated and consistent across all nodes.

However in the real world, network partitions are bound to happen and in that case we need to choose one of availability or consistency. For example suppose some data is written to n3, and before it can be propagated to n1 and n2, n3 goes down, now n3 cannot communicate with n1 and n2, hence the latter two nodes have stale data.

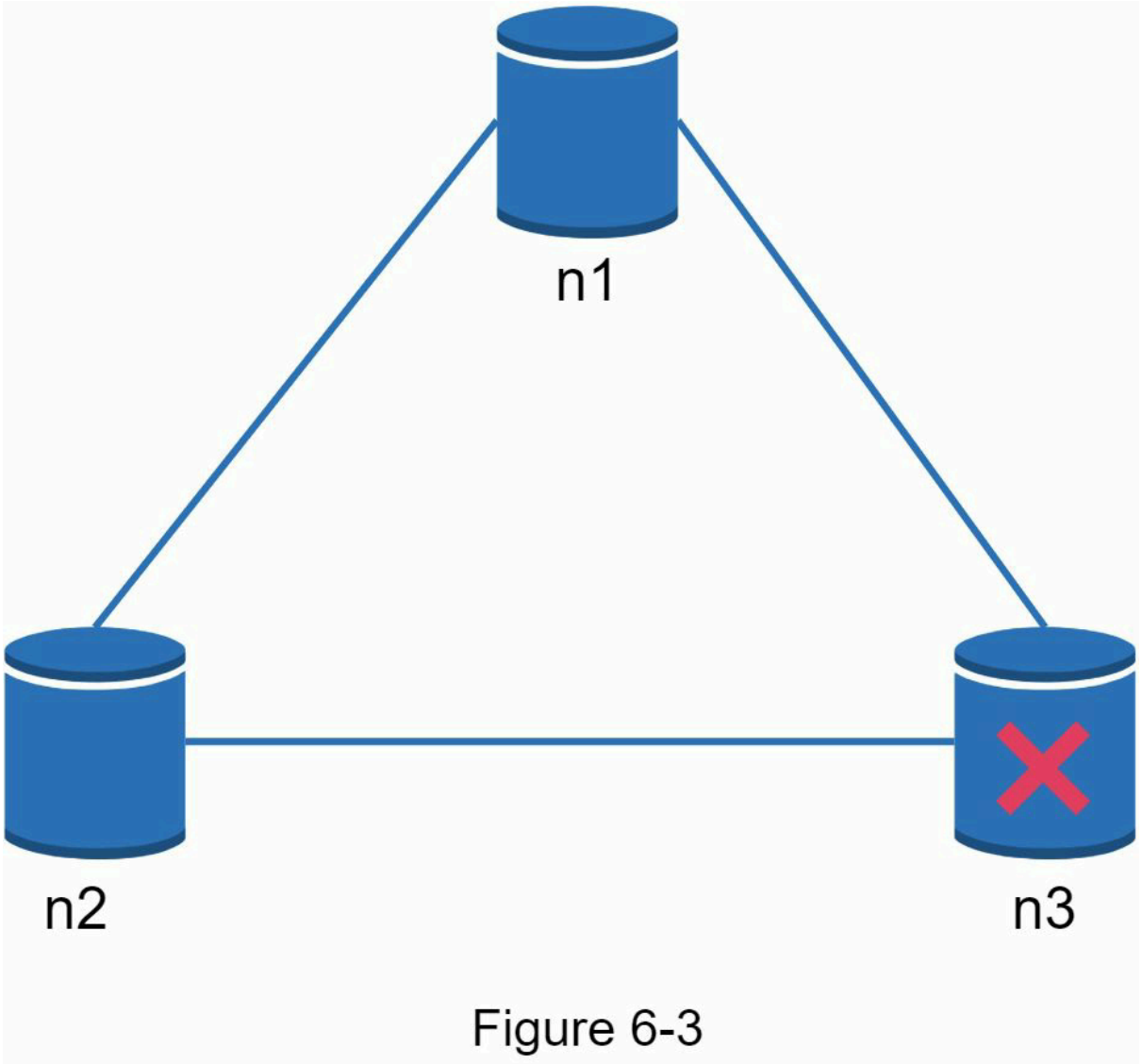


Figure 6-3

If we choose to prioritize consistency over availability (CP system), then we'll need to block all write operations to n1 and n2 to prevent data consistency among the three servers, which makes the system unavailable. Bank systems have high consistency requirements. For example it is important for the bank to display the most up to date balance information. If inconsistency occurs due to a network partition then the bank system returns an error, i.e. becomes unavailable until the inconsistency is resolved.

If we prioritize availability (AP), then the system keeps accepting reads even though it may return stale data. For writes, n1 and n2 will keep accepting writes, once the network partition is resolved the data will be synced to n3.

Data Partition

Obviously for large applications we cannot store all of the data in a single server. The approach taken is to split this data into smaller partitions and store them across multiple servers.

Objectives:
=> Distribute the evenly
=> Minimize data movement / remapping in case of server node addition / removal.

We can use consistent hashing to achieve this.

1. We first hash the server nodes onto the hash ring
2. Next we hash the key onto the same ring, and it will be stored on the first server encountered while moving in a clockwise direction starting from that key.

We can use virtual nodes / virtual servers to distribute the load as evenly as possible, further the number of virtual servers created for a server is proportional to its capacity.

Data Replication

To achieve high reliability and availability the data is replicated across N nodes, where N is a configurable parameter. How to choose these N serves? Walk clockwise from the position where the key has been mapped on the hash ring, the first N servers encountered in this traversal will be chosen and the data will be replicated onto these N nodes.

With virtual servers in place the first N servers might not necessarily be all unique, i.e. they might be corresponding to less than N physical servers. To avoid this issue we pick only unique servers while performing the traversal.

For better reliability the nodes should be placed across different data centers, so that they are isolated from each other and wouldn't be affected in case of a disaster damaging a data center for example. Data centers are connected via high bandwidth dedicated connections.

Consistency

Since data is replicated across multiple nodes, hence the data must be synchronized between the different replicas. Quorum consensus algorithm can guarantee consistency for both read and write operations.

=> N: Number of replicas
=> W: Write quorum of size W - For a write operation to be considered as successful, the write operation must be acknowledged from W replicas.
=> R: Read quorum of size R - For a read operation to be considered as successful, the read operation must get responses from at least R replicas.

In the quorum consensus algorithm we have one node acting as the coordinator, which acts as a proxy between the client and the nodes.

Protocols for distributed Consensus: 2 PC, 3 PC, MVCC, SAG

Distributed Consensus is a way by which multiple nodes agree on a particular value.

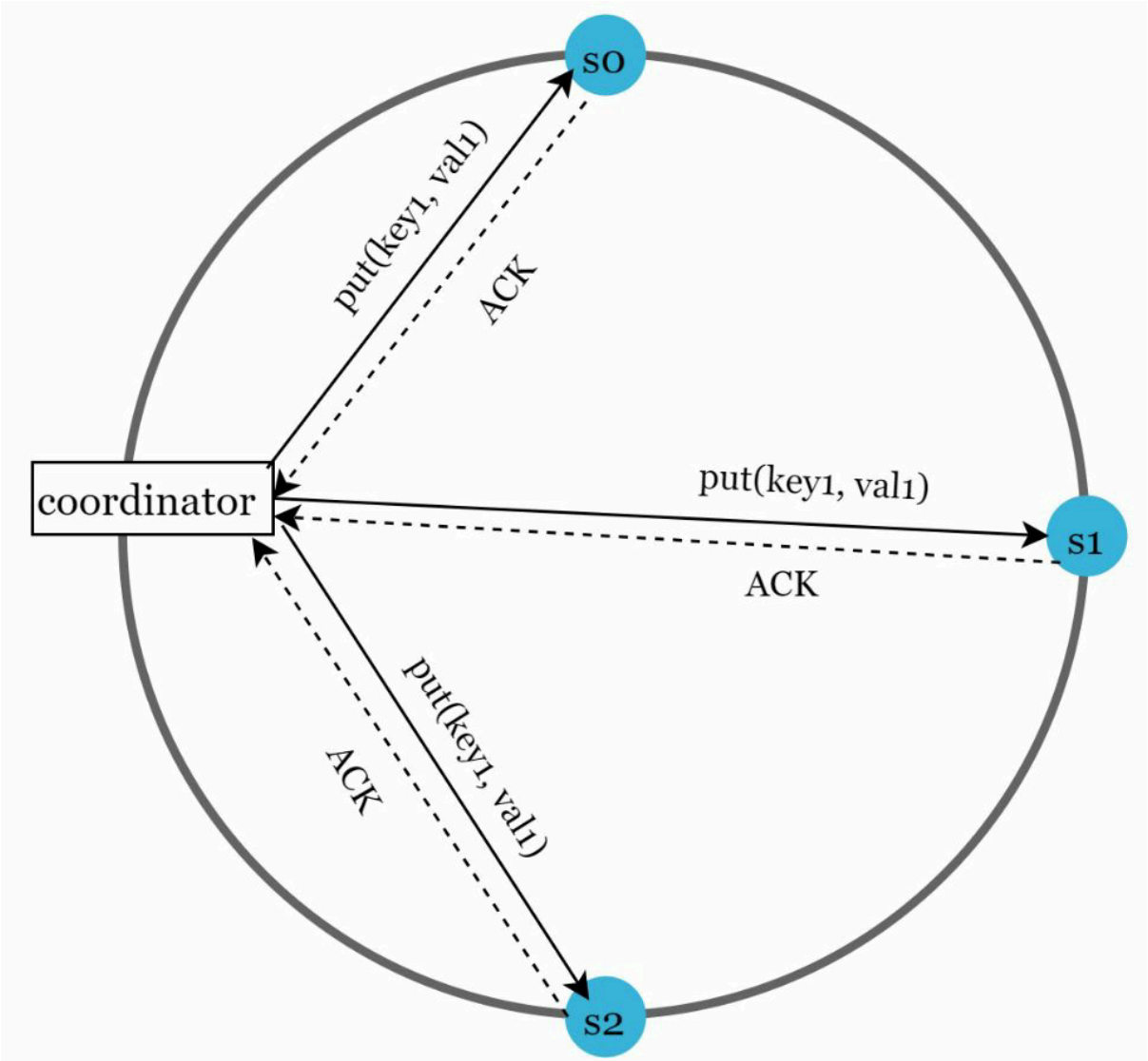


Figure 6-6 (ACK = acknowledgement)

The choice of W, R is a tradeoff between consistency and latency. If W = 1 or R = 1, the write / read operations will complete quickly as the coordinator only needs to wait for acknowledgement / response from one replica. If W or R > 1, the system will offer better consistency at the cost of increased latency, as the coordinator needs to wait for acknowledgement / response from multiple nodes.

If R = 1 and W = N: System is optimized for fast read
If R = N and W = 1: System is optimized for fast write
If W + R > N: Strong consistency is guaranteed.
If W + R <= N: Strong consistency is not guaranteed.

Consistency Models / Levels of Consistency

What degree of consistency do we need?

1. Strong consistency - Any read after the latest write should return the value corresponding to the latest write. The client should never see any stale data.
2. Weak consistency - Subsequent read operations might not return the most recently updated values.
3. Eventual consistency - It is a special form of weak consistency, given enough time all the updates are propagated and all the replicas are consistent.

Strong consistency is usually achieved by forcing every replica to not accept any new reads / writes, until every replica has agreed on the latest write. This approach is not ideal for highly available systems. DynamoDB and Cassandra use the model of Eventual consistency.

Eventual consistency allows inconsistent values to enter into the system, and the client needs to reconcile the inconsistent data.

Inconsistency detection and Resolution

Replication helps to improve availability of data, but it can lead to data inconsistencies among replicas, i.e. replicas have a different / conflicting view of the data. Versioning and Vector clocks help to solve the problem of inconsistency.

How can inconsistency occur:
Suppose we have two servers s1, s2.
s1 talks with replica node n1, while s2 talks with replica node n2.

s1, and s2 perform a get operation to get the value corresponding to a key, they'll both get the same value.

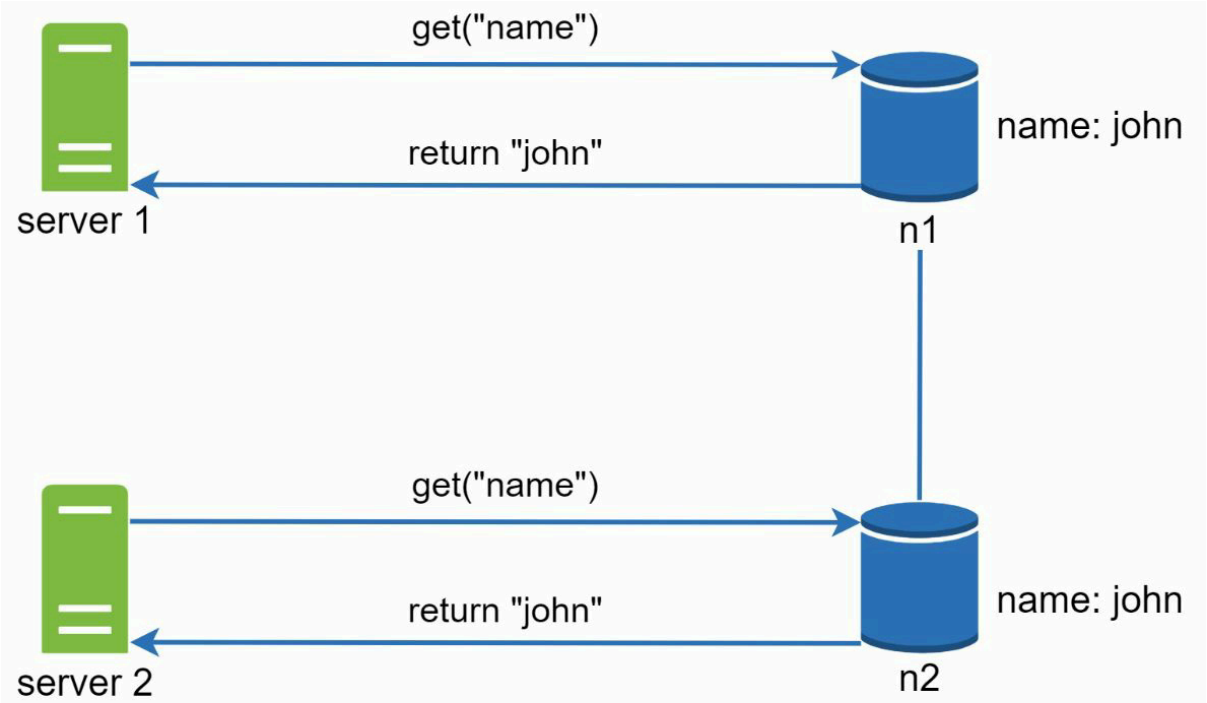


Figure 6-7

Now suppose s1 changes the value corresponding to the key “name” to “johnNewYork” while s2 changes the value corresponding to the key “name” to “johnSanFrancisco”. These updates go in simultaneously, and we now have a conflict where there are conflicting values for the key, let’s call these values v1 and v2.

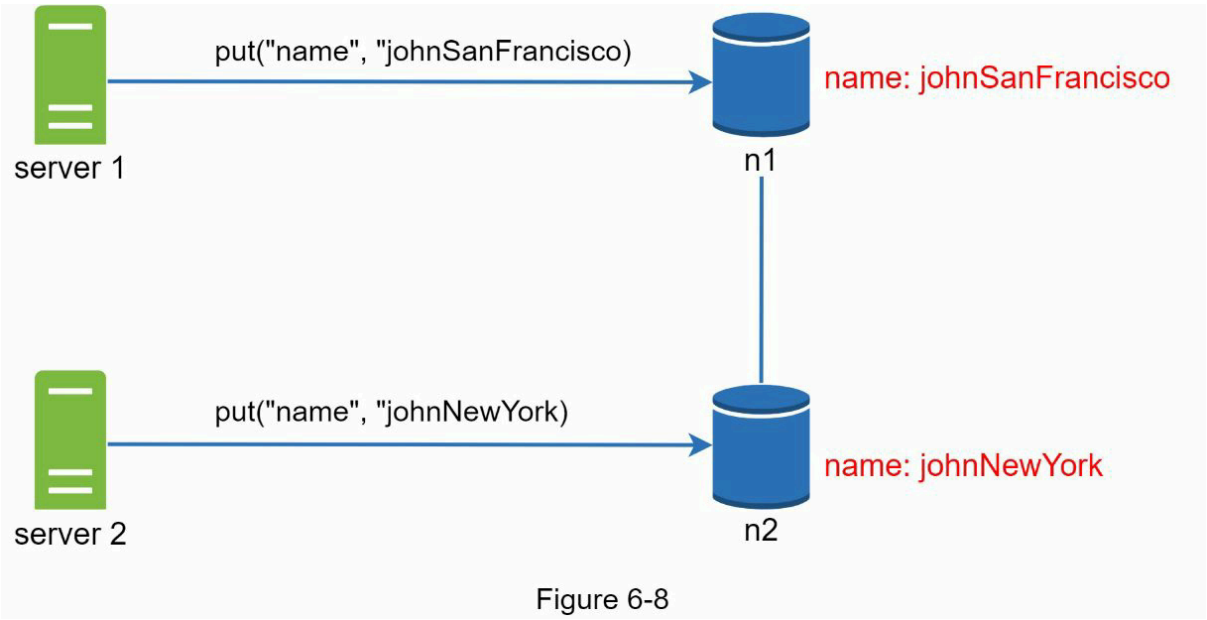


Figure 6-8

There is no obvious way to resolve this conflict. We need a versioning system that can detect conflicts and reconcile conflicts. A vector clock is a commonly used technique to solve this problem.

A vector clock associates with each data item a list of [server, version] pairs. It can be used to check if a version precedes, succeeded or is in conflict with another.

The vector clock for a data item D can be represented as:
D([s1, v1], [s2, v2],....[sn, vn])
=> Where si is the server id, i.e. the id of the server to which the data item is written.
=> vi is the version id.

Rules for updating the vector clock: If the data item D is written to server si, the system must perform one of the following tasks:
=> Increment vi if [si, vi] already exists in the vector clock
=> Otherwise create a new entry [si, 1]

Example of vector clock creation:

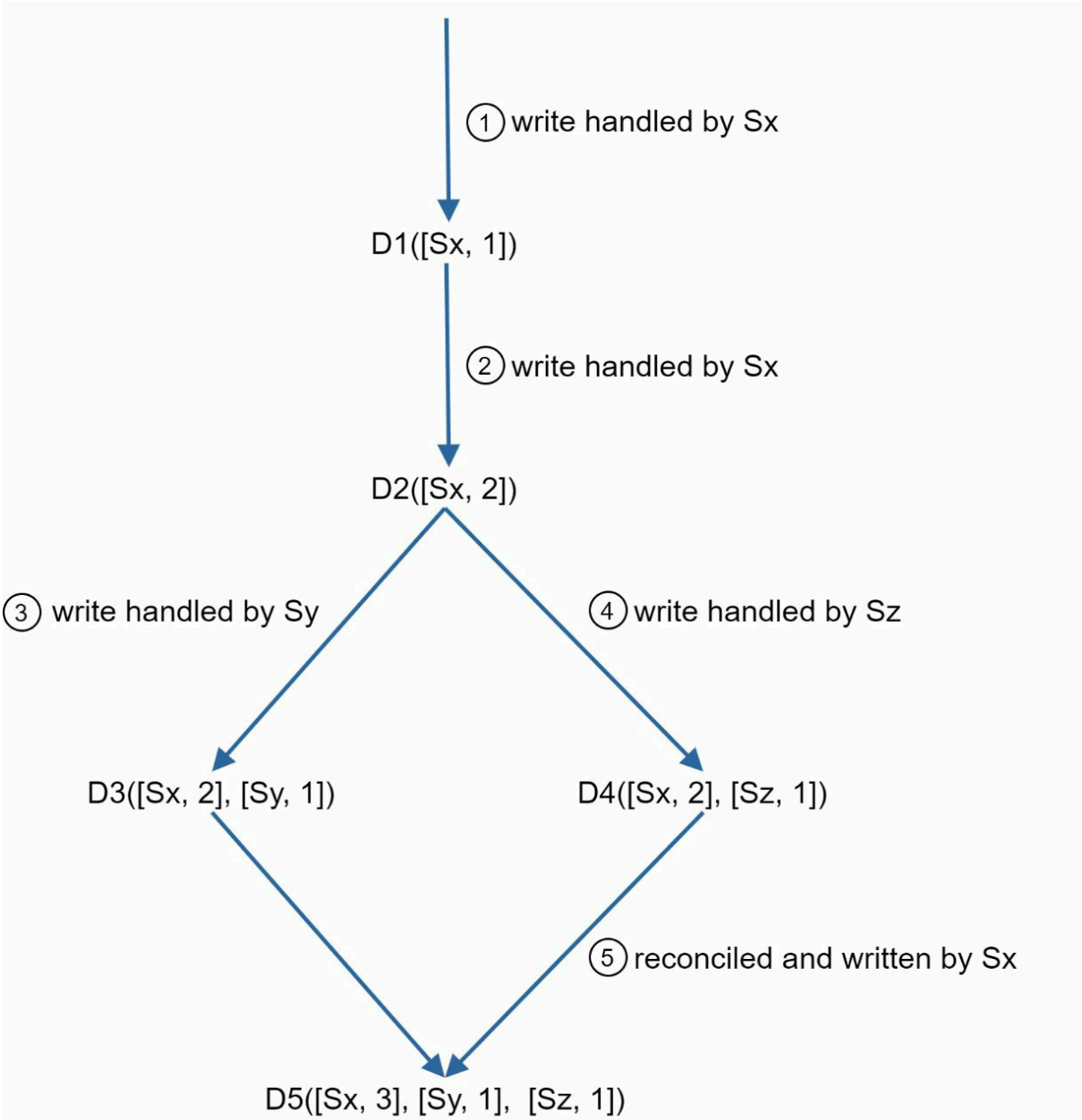


Figure 6-9 (source: [4])

1. A client writes a data item D1 to the system, and the write is handled by server Sx, which now has the vector clock D1[(Sx, 1)].
2. Another client reads the latest D1, updates it to D2, and writes it back. D2 descends from D1 so it overwrites D1. Assume the write is handled by the same server Sx, which now has vector clock D2([Sx, 2]).
3. Another client reads the latest D2, updates it to D3, and writes it back. Assume the write is handled by server Sy, which now has vector clock D3([Sx, 2], [Sy, 1])).

4. Another client reads the latest D2, updates it to D4, and writes it back. Assume the write is handled by server Sz, which now has D4([Sx, 2], [Sz, 1])).

Now the data items / version D3 and D4 are in conflict with each other. When a client reads D3 and D4, it will discover the conflict (which was caused due to the concurrent modification of data item D2 by both Sy and Sz). The client will need to resolve the conflict and write the updated data to the server. This will resolve the conflict, and the updated data will be propagated to all nods.

How to identify the relationship b/w two versions using vector clocks

=> A version X is an ancestor of version Y, if the version counter for each of the participants in the vector clock of Y is greater than or equal to the ones in version X. In this case X is an ancestor of Y, and there is no conflict.

Eg. version D[(s0, 1), (s1, 2)] is an ancestor of the version D[(s0, 1), (s1, 3)].

In other words, X is an ancestor of Y if every element of X is <= that of Y

=> A version X is a sibling of version Y (Conflict exists) if there is any participant in the vector clock of Y with a version counter that is less than its corresponding version counter in X.

Eg. version D[(s0, 1), (s1, 2)] and D[(s0, 2), (s1, 1)] are in conflict with each other, hence X and Y are siblings.

In other words, X is a sibling of Y, if there exists an element such that X>Y and an element such that X<Y.

Thus vector clocks help to resolve conflict, however they have few disadvantages:

=> The vector clock adds complexity to the client, because the client needs to implement conflict resolution logic.
=> The [server, version] pairs in the vector clock can grow rapidly. To fix this issue, we set a threshold for the length of the vector clock, beyond which the oldest pairs are removed.

Detecting failures

How do we detect if a node / server is down in a distributed system?

=> One approach could be to use all-to-all multicasting, however this is inefficient when there are many servers in the system.

A better solution is to use decentralized failure detection methods like gossip protocol. Gossip protocol is a general method by which nodes can communicate in a distributed system.

Herein each node maintains a membership list, which contains the member ID and the heartbeat counter.
=> Each node periodically increments its heartbeat counter.
=> Each node periodically sends heartbeats to a set of random nodes, which in turn propagate the info to another set of nodes.
=> Once a node receives a heartbeat, it will update its membership list with the latest information.
=> If the heartbeat counter for a particular node has not increased for a predefined period of time, then that member is considered offline.

=> In other words each node is gossiping with the other nodes regarding the heartbeat it receives from the other nodes, along with the timestamp at which the heartbeat was received.
=> Other nodes upon receiving a heartbeat, compare the timestamps to keep only the most recent heartbeat counter for each of the nodes, and then pass on this membership information to other nodes.

Nodes will use the most recent heartbeat timestamp, to determine if any of the other nodes in the cluster are down.

If the heartbeat counter or the most recent heartbeat timestamp for a node has not been updated for a long time, then that node is marked down, and this information is propagated to all the other nodes.

Handling failures

Temporary Failures

We have seen how to detect failures using the Gossip protocol, next we discuss how the system can handle failures.

When node failures happen, the system will need to use certain mechanisms to ensure availability.

=> In the strict quorum approach, read and write operations could be blocked to ensure strong consistency.
=> Another approach called sloppy quorum can be used to improve availability. In this approach the system will choose the first W healthy servers for writes, and first R healthy servers for reads on the hash ring. Offline servers are ignored.

If a server is unavailable due to network or server failures, then another server will process the requests temporarily. When the down server is back up, changes will be pushed to it / synced with it to ensure data consistency. This process is called hinted handoff. For example if a server si is unavailable, it can be temporarily replaced with a server sj. When si is back, sj will hand the data back to si.

Permanent Failures

Hinted handoff can be used to handle temporary failures,what if the replica is permanently down. To handle such a situation, we need to keep the replicas in sync.

In other words, since in a distributed system the same data exists in multiple locations, hence if a piece of data is changed in one location it’s important to change the data everywhere. Hence data synchronization is the process of ensuring the data is the same everywhere.

A Merkle tree is used for data synchronization across replicas, while minimizing the amount of data transferred.

=> A Merkle tree (also called hash tree) is a tree in which every non-leaf node is labeled with the hash of the labels of its child nodes, in case of leaf nodes the label is the hash of the value of the node.
=> Merkle trees provide efficient verification of the contents of large data structures.

How to build a Merkle tree -

=> To build the merkle tree, the key space is divided into some number of (k) buckets.
=> Once the buckets are created, hash each key in all the buckets using a hash function.
=> Create a single hash node per bucket.
=> Build the tree upwards till root by calculating the hashes of the children.

To compare two merkle trees, we start at the root. If the root hashes match then both the servers have the same data. If the root hashes disagree, then the left child hashes are compared followed by the right child hashes.

The advantage of using a Merkle tree is that we can traverse the tree to find the buckets which are not synchronized and synchronize only these buckets. So with Merke trees the amount of data needed to be synchronized is proportional to the differences between the two replicas, and not the amount of data they contain.

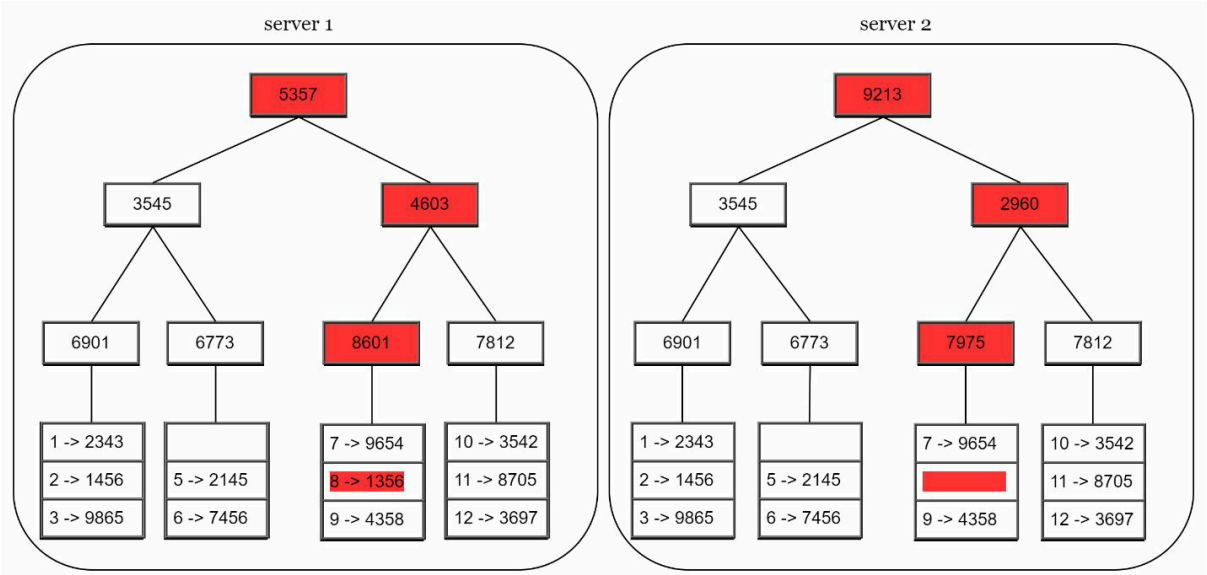


Figure 6-16

Time complexity for Synchronization:

Average TC: O(logN)
Worst Cast TC: O(N)

The worst case will be hit when there are no nodes in common between the two trees, in this case synchronization is akin to making a complete copy of the replica, which is an O(N) operation.

Handling data center failures / Outages

Data center outages could happen due to a variety of reasons like network failures, natural disasters, power outage etc. To build a system which is resilient to data center outages, it is important to replicate the data across multiple data centers. Even if a data center is fully offline, users can still access the data through the other data centers.

A Possible design

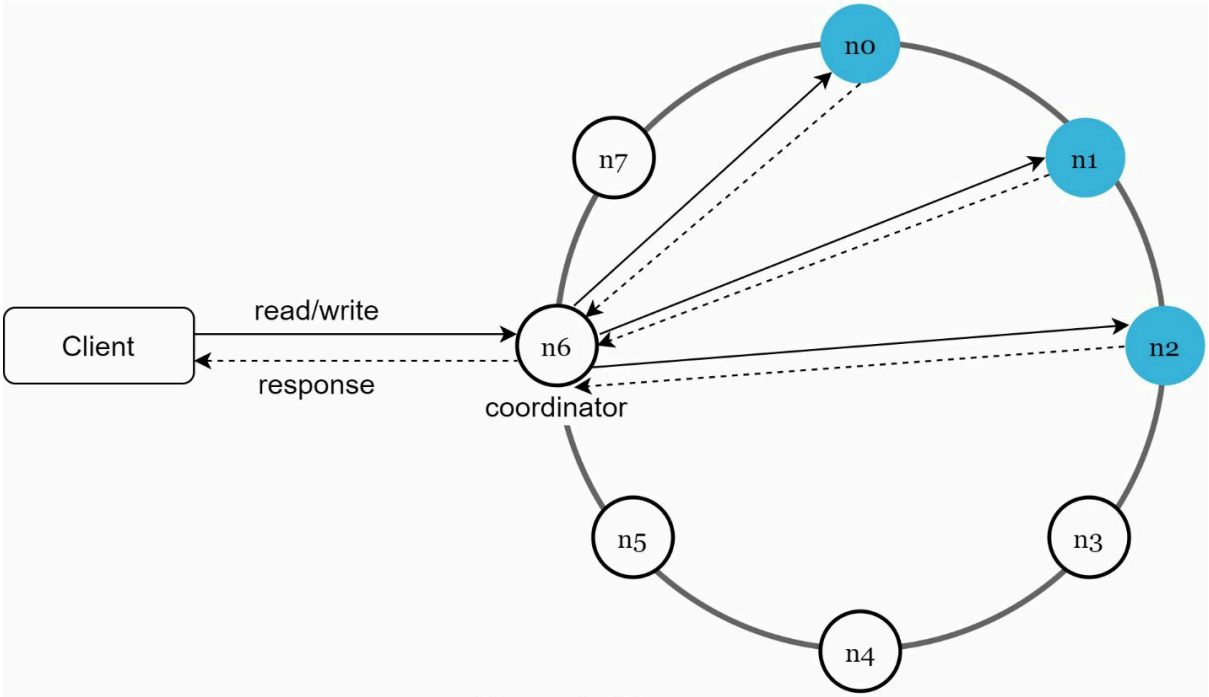


Figure 6-17

1. Client communicates with the key-value store through simple APIs get(key) and put(key, value).
2. A coordinator is a node which acts as a proxy between the client and the key value store.
3. Nodes are distributed on a hash ring using consistent hashing.
4. The system is decentralized (don't confuse the coordinator for a centralized design, the coordinator in this case is just an entry point for data into the system), nodes can be added and removed dynamically, consistent hashing will help to minimize the data movement needed.

5. Data is replicated across multiple nodes / servers.
6. There is no single point of failure, each node will have the same set of responsibilities.

Since the design is decentralized (i.e. no centralized coordinator) hence each node will need to perform many tasks, which include:

1. Data storage / Storage engine
2. Replication
3. Conflict Resolution
4. Client API (Interface through which the client requests come in)
5. Failure detection

etc.

Write Path

Let's see the flow after a write request is directed to a particular node. The design for the write path is based on the architecture of Cassandra.

=> When a write request comes in, it is persisted on a commit log file.
=> The data is stored in the memory cache.
=> Once the cache is full or reaches a predefined threshold the data is flushed to the disk (in case of Cassandra data is stored in SSTables, a SSTable [Sorted-String Table] is a sorted list of key, value pairs).

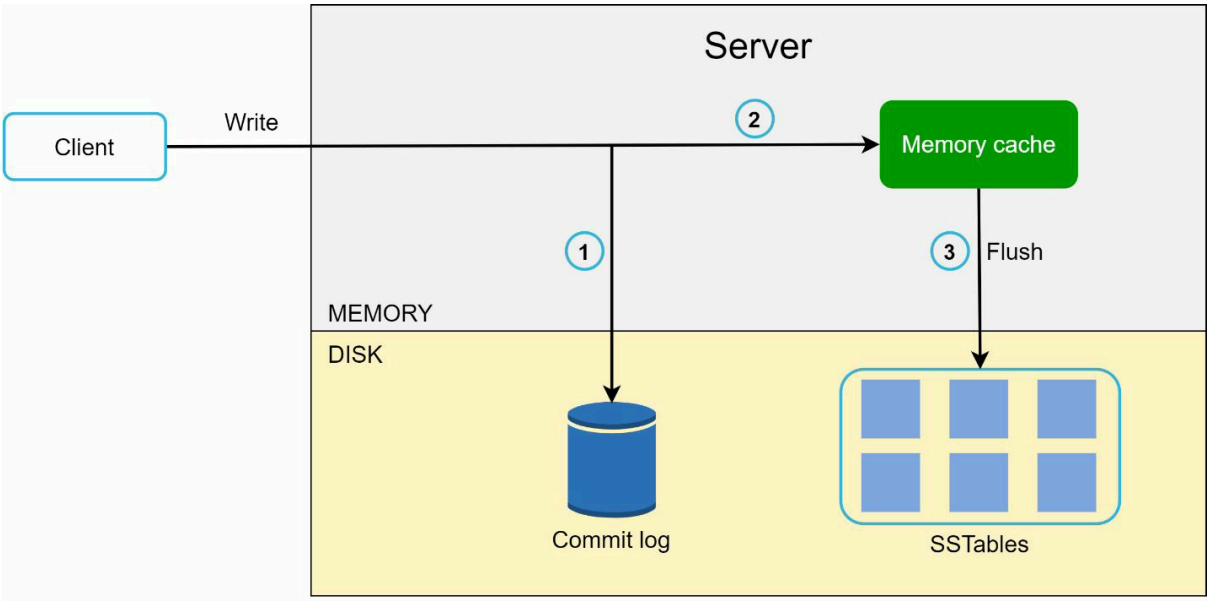


Figure 6-19

Read Path

Let’s see the flow when a read request is directed to a particular node.

=> The node will first check if the data is present in the memory cache, if it is the data will be returned to the client.

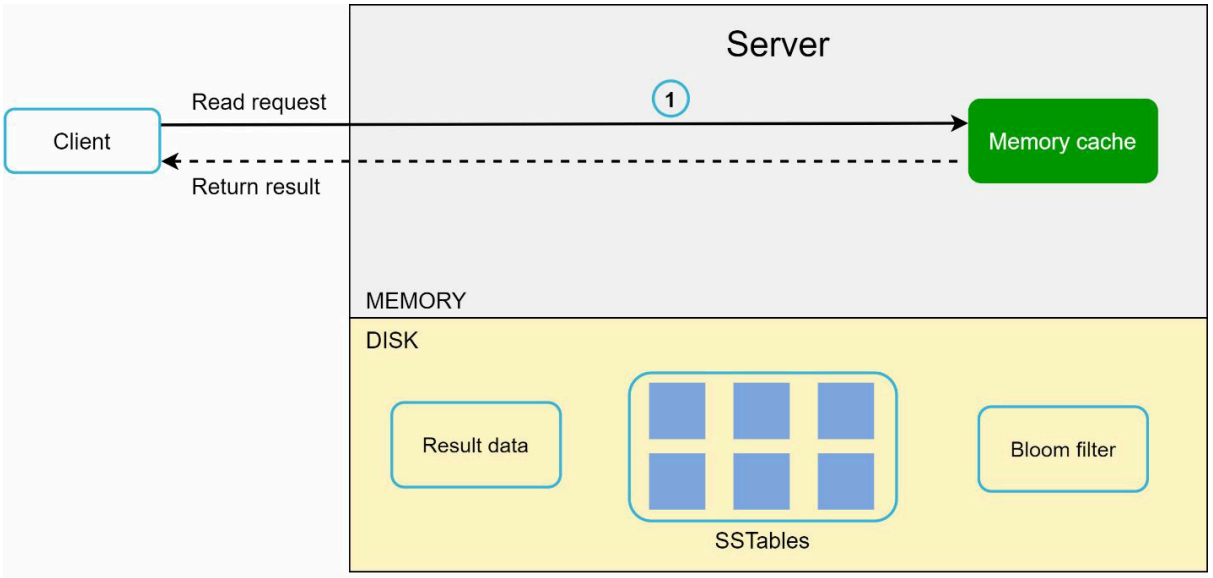


Figure 6-20

=> if the data is not present in the memory cache, then we’ll need to retrieve the data from the disk instead.

For example in the case of Cassnadra, data is stored in SSTables on the disk, we need an efficient technique to determine which SSTable contains the key. Bloom filter is commonly used for this purpose.

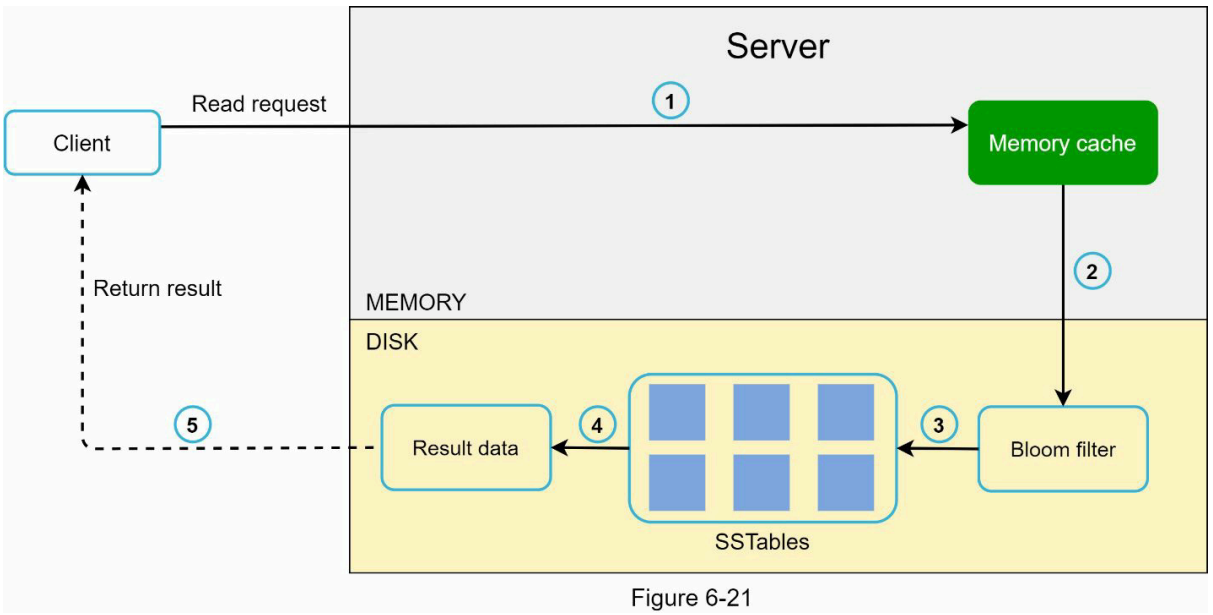


Figure 6-21

Summary for this topic:

Goal/Problems	**Technique**
Ability to store big data	Use consistent hashing to spread the load across servers
High availability reads	Data replication Multi-data center setup
Highly available writes	Versioning and conflict resolution with vector clocks
Dataset partition	Consistent Hashing
Incremental scalability	Consistent Hashing
Heterogeneity	Consistent Hashing
Tunable consistency	Quorum consensus
Handling temporary failures	Sloppy quorum and hinted handoff
Handling permanent failures	Merkle tree
Handling data center outage	Cross-data center replication

Table 6-2

Search Autocomplete System

Also known as typeahead system, search as you type or incremental search.

Clarification Questions

1. How many autocomplete suggestions should our system return?
2. Are we talking about prefix matches?
3. Among the possibly many suggestions, how will the system identify which k to return.
4. Are the search queries in English?
5. Will the queries be case sensitive?
6. What is the traffic our application is expecting?

=> The system should be fast and highly responsive.
=> The results returned must be sorted by popularity or query frequency.
=> For every character inserted in the search box, a set of suggestions must be returned.

We can break down the system into 2 components:

=> Data Aggregation Service: This component will collect user input queries, and track the query frequencies. This data will help us to find the k most popular search result suggestions.

=> Query Service: Given a user query (prefix), return the k most frequently searched suggestions.

Design Approach - I

We create a database, a simple relational database which stores a frequency table.

This frequency table has 2 attributes:
Query String: String
Frequency: Integer

Query String is the user query input, while the frequency counts how often this query string has been searched.

Suppose initially the frequency table is empty, and then the user makes the following searches:
Twitter
Twilio
Twitter
Twilight

The frequency table will look like:

Query String	Frequency
Twitter	2
Twilio	1
Twilight	1

At this point if a user is performing some search, for a query beginning with “tw” then our autocomplete system can use the data in the frequency table to return the strings with the matching prefix.

For example we can perform the following SQL query to get the auto complete suggestions:

```
SELECT query_string FROM frequency_table
WHERE query_string LIKE ‘tw%’
ORDER BY frequency DESC
LIMIT 5;
```

This is an acceptable solution only if the dataset is small, if the dataset is large then database access will become the bottleneck.

Design Approach - II

In the first approach we used a database to store <Query String, frequency> mappings and to get the most popular auto complete suggestions we performed an ORDER BY on the entire table, which is a very expensive operation. This approach is only suitable when the dataset is small. So we need a more optimal approach.

We will use Tries for this purpose.
As tries are designed to store strings, and retrieve strings efficiently.

What info to store in the trie node
boolean endOfWord;
int frequencyCount
TreeNode [] children = new TrieNode[26];

We store the frequency count in the node, as we'll need it to support sorting by frequency.

Example of what the trie will look like:

Query	**Frequency**
tree	10
try	29
true	35
toy	14
wish	25
win	50

Table 13-2

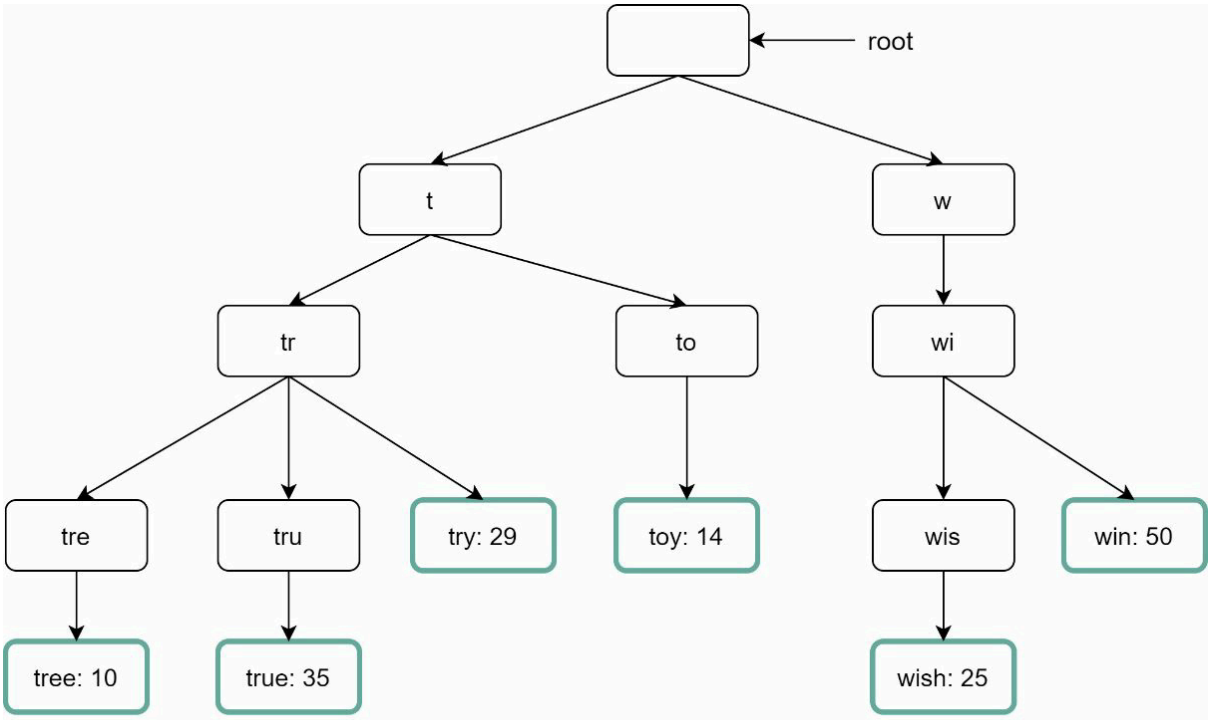


Figure 13-6

Time complexity of getting search autocomplete suggestions by trie:

p - The length of the prefix, where prefix is the string the user has currently typed into the search bar.
n - Number of nodes in the trie.
c - Number of children of a given node.

Finding the prefix: O(p)
Traverse the tree rooted at the prefix node, and get all the valid children (endOfWord = true), and add them to some sort of container: O(c)
Sort this container by frequency of the nodes, and get the k children with the highest frequency: O(clogc)

Overall time complexity of this algorithm:
O(p) + O(c) + O(clogc)

This algorithm is however too slow, in the worst case we'll need to traverse the entire tree to get the top k results.

Further optimizations:
Limit the length of prefix: Search queries are generally in most cases quite short, thus we can cap the size of the query string to say 50

characters. In this case the time complexity for finding the prefix will be O(p) = O(50) ~ O(1) [Constant time]

Storing the k queries with the highest frequency at each node in the trie: As mentioned above we are interested in the top k search auto-complete suggestions. To avoid traversing the whole trie to get these k queries, we can store the k most frequent queries at each node of the trie. k is generally a relatively small number, 5 - 10.

By storing the top queries at each node we can significantly decrease the time complexity, at the cost of increasing the space complexity, as additional space will be needed for each node to store the k top queries.

Which queries will a node store: It will store the queries, in which the node is acting as a prefix, more specifically the top k queries.

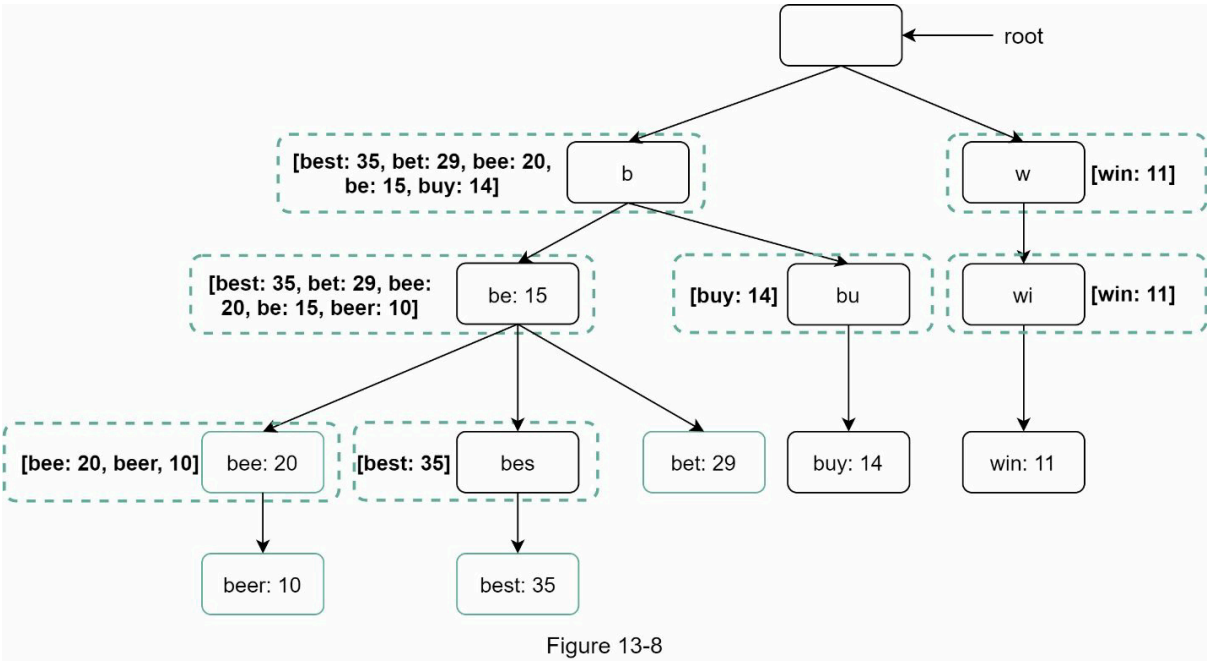


Figure 13-8

Improved time complexity of retrieving the top k autocomplete suggestions:

Find the prefix Node: O(1)
Return the top k queries: Since each node is natively storing the top k queries, hence we can just return these k queries without any further

traversal or sorting, hence since the top k queries are cached so this operation is also O(1).

So overall time complexity of fetching top k queries: O(1) [Constant]

Data Gathering Service - In the first approach, whenever a user typed a query, the data is updated in real time. However this is not an optimal approach.

=> Billions of queries come in each day, updating the trie after every query will be very expensive, and will slow down the query service as well.
=> Top suggestions may not change much once the trie is built, hence updating the trie frequently isn't necessary.

What we can do is periodically update the trie, this period can be a few hours, one day or maybe one week. Instead of feeding the data directly from each query, we can feed it from an analytics or logging service.

Step 1: So when a query comes in we store the raw query in the logging service, the size of the log would be quite large. These analytics logs contain data in a raw format, and this data needs to be processed and converted to a form which can be used by our system.

Step 2: Periodically the data aggregator worker nodes will process the logs and aggregate the data, this time period depends on our use-case, if we want the application to be real time then we'll need to perform aggregation in short time intervals, as real-time results are important, for other user-cases aggregating the data less frequently might be a good fit.

Step 3: Once the data is aggregated, worker nodes will use the data and build the trie data structure and store it in a trie DB.

So if our period of aggregation is say 1 week, then a new trie will be built each week.

How do we store the trie?
What choice of database is a good fit for trie DB?
We can use a NO-SQL key-value store for storing the trie.
Where the prefix will act as the key, and the data on each node is the value in the table.

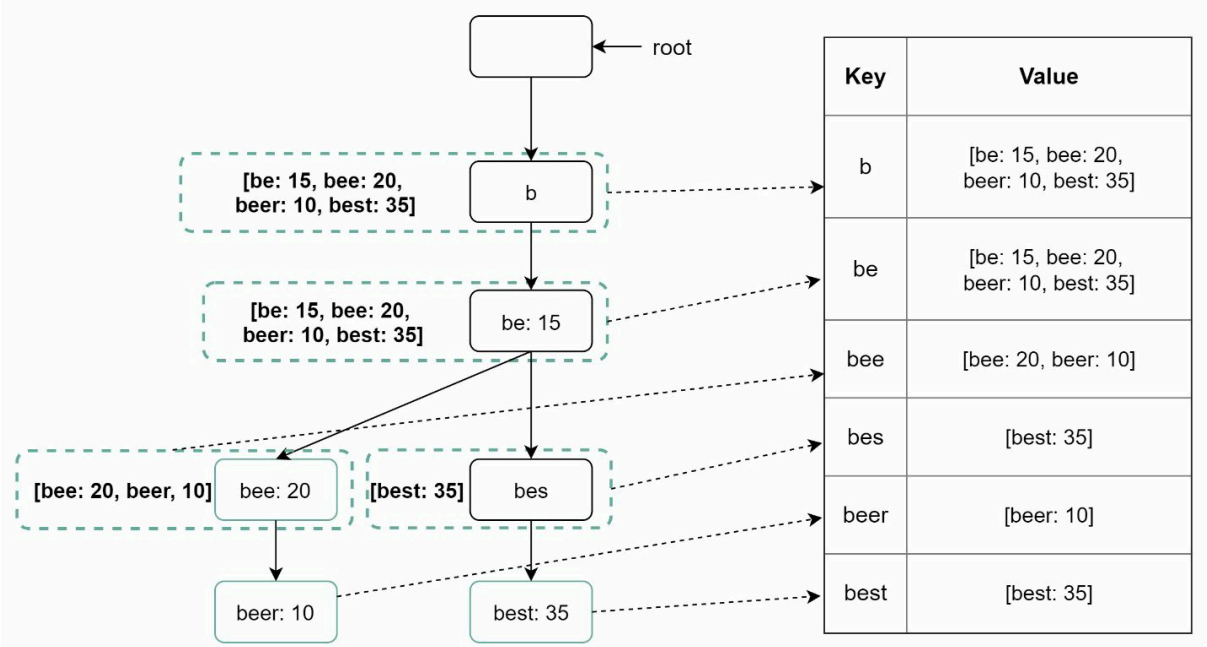


Figure 13-10

Flow of the query search

1. A search query is sent to the load balancer.
2. The load balancer routes the request to one of the API servers.
3. The API server will check in the trie cache to get the data corresponding to that prefix, if found it'll construct the auto complete suggestions and send them to the client.
4. If the data for the required prefix is not present in the cache, then the API server will get the data from the trie DB and before returning the suggestions to the client it'll write the data to the cache, so that any future requests for the same prefix will be handled by the cache itself.

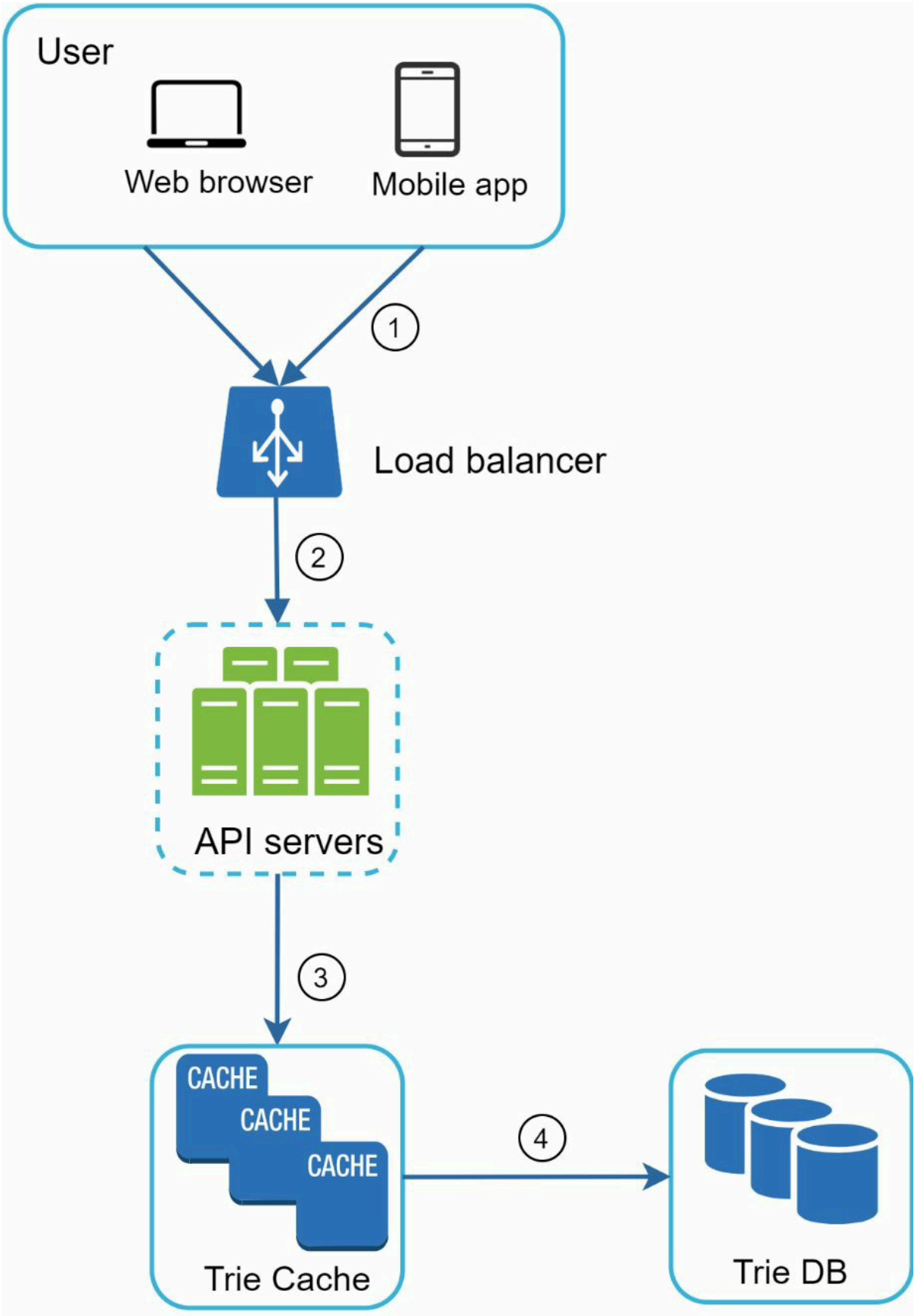

User
Web browser
Mobile app
1
Load balancer
2
API servers
3
CACHE
CACHE
CACHE
Trie Cache
4
Trie DB

Figure 13-11

As mentioned before the requests are logged in a logging service, if we store each query in the log, then that will require a lot of processing power and storage. Data sampling is important in such cases, where instead of logging each request we only log 1 out of every N request.

Trie Operations

Create: The trie is created by the worker nodes using the aggregated data. The source of this is the Analytics log / DB.

Update: The trie is updated periodically, as mentioned before this period could be anything which suits the use case. Hence periodically we create a new trie which replaces the old one.

As mentioned before updating the trie in real time, every time a search query is performed is not an efficient approach however it could be an acceptable approach if we are working with a small database.

Delete: Once a new trie is created, it will replace the old one, i.e the old trie is implicitly deleted. If we don't' want certain queries to be returned in the autocomplete suggestions (for whatever reasons), then we can add a filter layer which will filter the data returned by the Trie cache / DB corresponding to the specified prefix, and remove all such queries from the suggestions, before passing the data on to the API servers.

In addition some sort of list of such queries will need to be maintained by the system, to ensure they don't get programmed (or inserted) to the trie.

Scaling the storage

As the amount of data increases, it is possible that the trie cannot be stored on a single node / server. Hence we need to split the data across multiple servers, i.e. sharding.

We can use range based sharding based on the first character of the prefix / queries.

If we need 2 servers for storage, then we can store queries starting with 'a' - 'm' on server 1, and queries starting from 'n' to 'z' on the second server.

Similarly we can partition for any number of server nodes. With this strategy we can partition the data across 26 shards, if we want further sharding then we can do it based on the second character of the query.

This approach is reasonable, however there is a major problem here as for example the number of words starting with ‘a’ is way more than the number of words starting with ‘x’, ‘y’, ‘z’ combined. This results in an uneven distribution of data, and could result in the hotspot key problem.

To mitigate the data imbalance problem, instead of depending on these static rules we create another service, the Shard Map Manager. This service will analyze the historical distribution pattern and apply smarter sharding logic. The shard manager will maintain a lookup database to identify the shard on which a particular record is stored.

What do we mean by historical data distribution: It refers to the historical data regarding the number of queries that start with a particular character.

What if we want to support queries in multiple languages?

Currently our system only supports the English language, as we are using ASCII characters. To support other languages as well we need to use Unicode characters, so in each trie node we’ll store Unicode characters. Unicode covers all the characters from all the writing systems in the world, modern or ancient.

What if the top search queries differ across counters?

For example In India, if the user enters the prefix tw, then our system returns twitter as the top suggestion for auto-completion. But in the USA it returns Twitch

This is based on the search pattern, i.e. popularity of the query (in terms of query frequency) can differ from one region to another.

In such cases we can build a separate trie for each country. To decrease latency we can store the tries in CDNs.

Design a Web Crawler

What is a web crawler?

A web crawler also known as a Robot or Spider, is widely used by Search engines to discover content on the web. Content can be HTML files, images, pdf's etc.

The Web crawler starts by collecting a few web pages, extracting content from those pages, including the hyperlinks from those pages and then follows these links to collect new content.

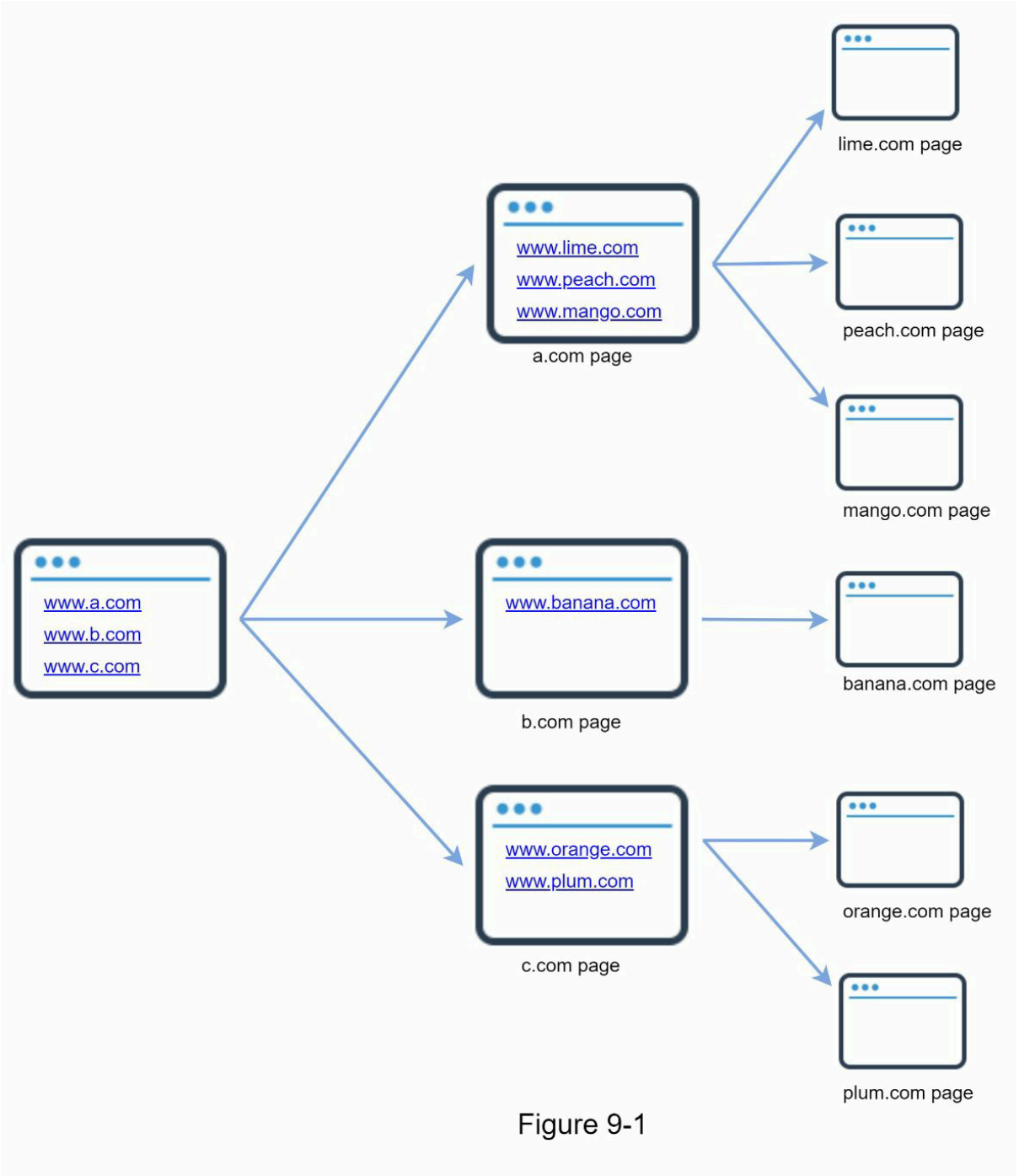


Figure 9-1

Uses of Web Crawler:

1. Search Engine Indexing: This is the most common use case of web crawlers. Search engines use Web Crawlers to collect web pages. For example Google Search Engine uses the Googlebot web crawler.
2. Web Archiving
3. Web (Data) Mining
4. Web monitoring - Monitor Copyright and Trademark infringements for example.

Clarification Questions:

1. What is the scale of the crawler, how many pages does the crawler collect per month?
2. What content types are included? HTML pages only? Or do we support images, PDF files too?
3. Do we need to store collected web pages, if so then for how long?
4. Pages can have duplicated content, how to handle such pages.

Characteristics of a good Web Crawler:

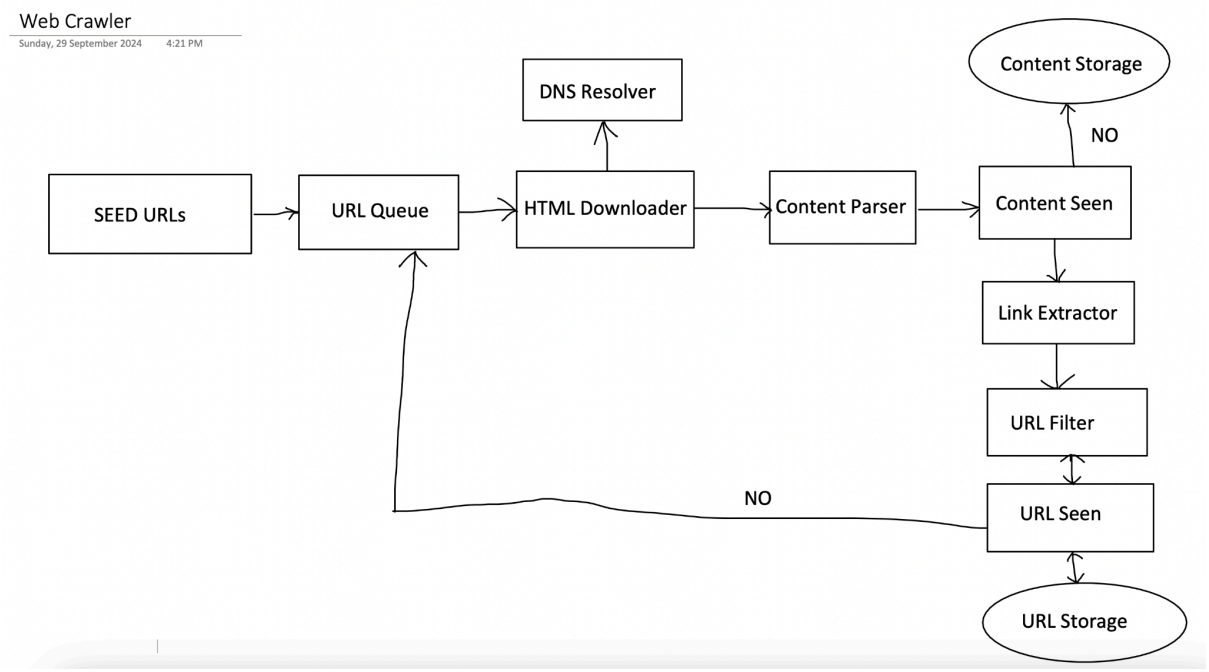
1. Scalability: There are billions of web pages out there, web crawling should be scaling and make use of parallelization.
2. Robustness: The web is full of traps, malicious links, unresponsive servers, etc. are all common. The crawler must handle these edge cases.
3. Politeness: The crawler must not make too many requests to a website in a short time interval, this will hamper the website's performance.
4. Extensibility: The system should be flexible enough, so that it can support new features in the future without having to redesign the system. For example if our system has been HTML content type, and we want to crawl image files in the future, then we shouldn't need to redesign the system.

Basic flow of the Web Crawler:

1. Given an initial set of URLs, download all the web pages corresponding to these URLs.
2. Extract URLs from these web pages.
3. Add these extracted URLs to the list of URLs which need to be downloaded.

LOOP

High Level Design of the Web Crawler



Seed URLs: The web crawler uses a list of URLs as the seed URLs, which will be the starting point for the crawl process. A good seed URL will allow the crawler to traverse a large number of links.

URL Queue: The URL Queue will store the URLs which need to be downloaded, this is a FIFO queue.

HTML Downloader: The URL queue gives the URLs to the HTML Downloader, which will download the web pages for the internet.

DNS Resolver: Used to resolve URLs to IP addresses.

Content Parser: The Content parser will parse the downloaded web page. For example tools like BeautifulSoup in python help to parse html pages, this also helps to detect any malformed pages, it’s important to remove such malformed pages as they can create problems and waste storage space.

Content Seen: This component will interact with the Content storage content. Its role is to check if the content in the web page already exists in the system, i.e. it ensures there is no data redundancy. How to match 2

html documents - Match the two documents character by character (slow, time-consuming) OR Compute hashes of the two web pages and compare them instead (efficient approach).

Content Storage: This is the storage for the html documents, most of the content is stored on the disk, popular content can be kept in memory (caching) to reduce latency.

Link Extractor: The web pages downloaded by the HTML downloader, can contain further URLs (via hyperlinks), the URL extractor will extract these links from the web pages.

URL Filter: This component scans the URLs provided by the Link Extractor and removes any blacklisted or malformed URLs.

URL Seen: This component interacts with the URL storage, to check if the URL has already been visited by the crawler, or if the URL is present in the Frontier. We don't want to traverse the same URL again.

URL Storage: This URL Storage stores the URLs that have already been visited.

Finer details about some of the components:

Traversal technique BFS vs DFS

We can visualize the web as a directed graph where the web pages are the nodes and the hyperlinks (URLs) are the edges.

Similar to any normal graph there are 2 traversal techniques available BFS, DFS.

DFS is generally not preferred as the depth of the recursion tree can be too deep.

BFS is commonly used, however it too has some disadvantages:

1. A normal web page contains many hyperlinks, however most of them actually are linked back to the same host. In a BFS traversal we'll add all these URLs in a queue and try to download the web pages, however since most of them are from the same host, this

will cause a major increase in traffic at the host website., as it will be flooded with requests. This violates the goal of politeness.

2. BFS doesn't take into account the priority of a URL, certainly some pages have higher priority, in terms of quality and importance. Therefore it could be a good option to rank the URLs by their page rank, traffic etc.

Component design enhancements:

URL Queue:
The URL Queue is a component which basically maintains data structures to store which URLs need to be downloaded by the HTML Downloader. The HTML downloader will get the URLs from the URL queue.

The URL Queue is an important component to ensure politeness and URL prioritization.

Politeness:
What exactly is politeness?
Politeness basically means we shouldn't be sending too many requests to the same hosting server in a short period. Sending too many requests will flood the application, or worse the traffic can be treated as a denial of service attack.

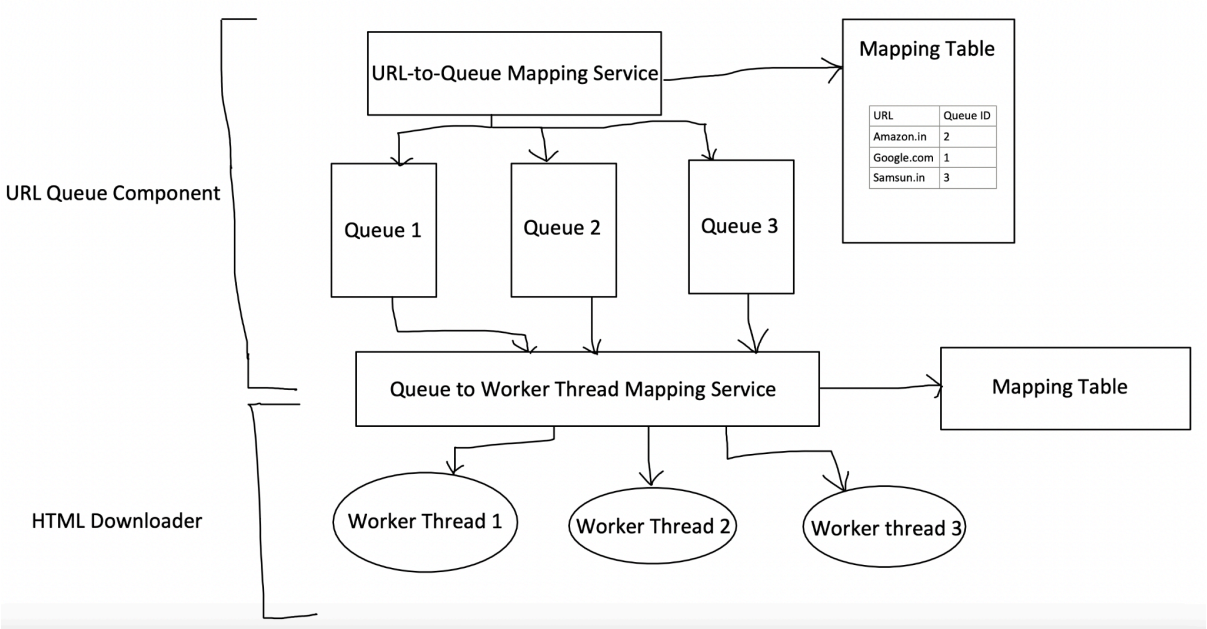
A method to ensure politeness is by ensuring that only one page at a time is downloaded from the same host, i.e. we shouldn't be sending multiple requests parallelly to the same host. In addition we can add a delay b/w consecutive request to the same host.

How can we achieve this:
The HTML Downloader component will be multi-threaded, we need to ensure that each thread is performing downloads on URLs belonging to one specific host only.

For example
T1 - Download urls from the host amazon.in
T2 - Download urls from the host google.com
T3 - Download urls from samsung.in

This way no single thread is performing downloads across multiple hosts, which ensures that we will download only at a time from a particular host. Further we can add a delay b/w consecutive downloads performed by any thread thereby ensuring politeness.

Design:



URL-TO-Queue Mapping Service ensures that each URL from the same host is mapped to a single queue, i.e. each queue only contains URLs belonging to a particular host only, to perform the mapping it makes use of the Mapping table, which contains the information regarding which queue a URL is mapped to.

Queue 1, 2, 3 … n: SQS Queues which contain the URLs to download, as mentioned above each queue contains URLs belonging to the same host.

Queue To Worker Thread Mapping Service: Maps the queue to worker threads, each worker thread is mapped to a single FIFO queue.

Worker threads 1, 2, .. n: Perform the actual downloading of web pages, each thread is mapped to a single queue and will download the URLs one by one from that queue, a delay will be added b/w consecutive downloads tasks performed by a worked thread.

URL Prioritization:

As mentioned before not all web pages have the same priority, to determine the priority of a URL we can use the Page Rank, traffic on the web page, update history etc.

We will design a Prioritizer Module which runs inside the URL Queue service, which will help to provide URL prioritization.

=> The Prioritzer takes URLs as input and computes their priorities.
=> It maintains a number of queues q1, q2 … qn, each queue has an assigned priority. The Prioritizer will compute the priority of the URL and put them in one of the queues.
=> Periodically it will return a queue, with a bias towards the higher priority queues.

The Prioritizer will return a list of URLs, which are of high priority and should be downloaded. We can feed it as input to the URL-to-Queue Mapping service which will download the content from the web while maintaining politeness.

Complete design of the URL Queue component:

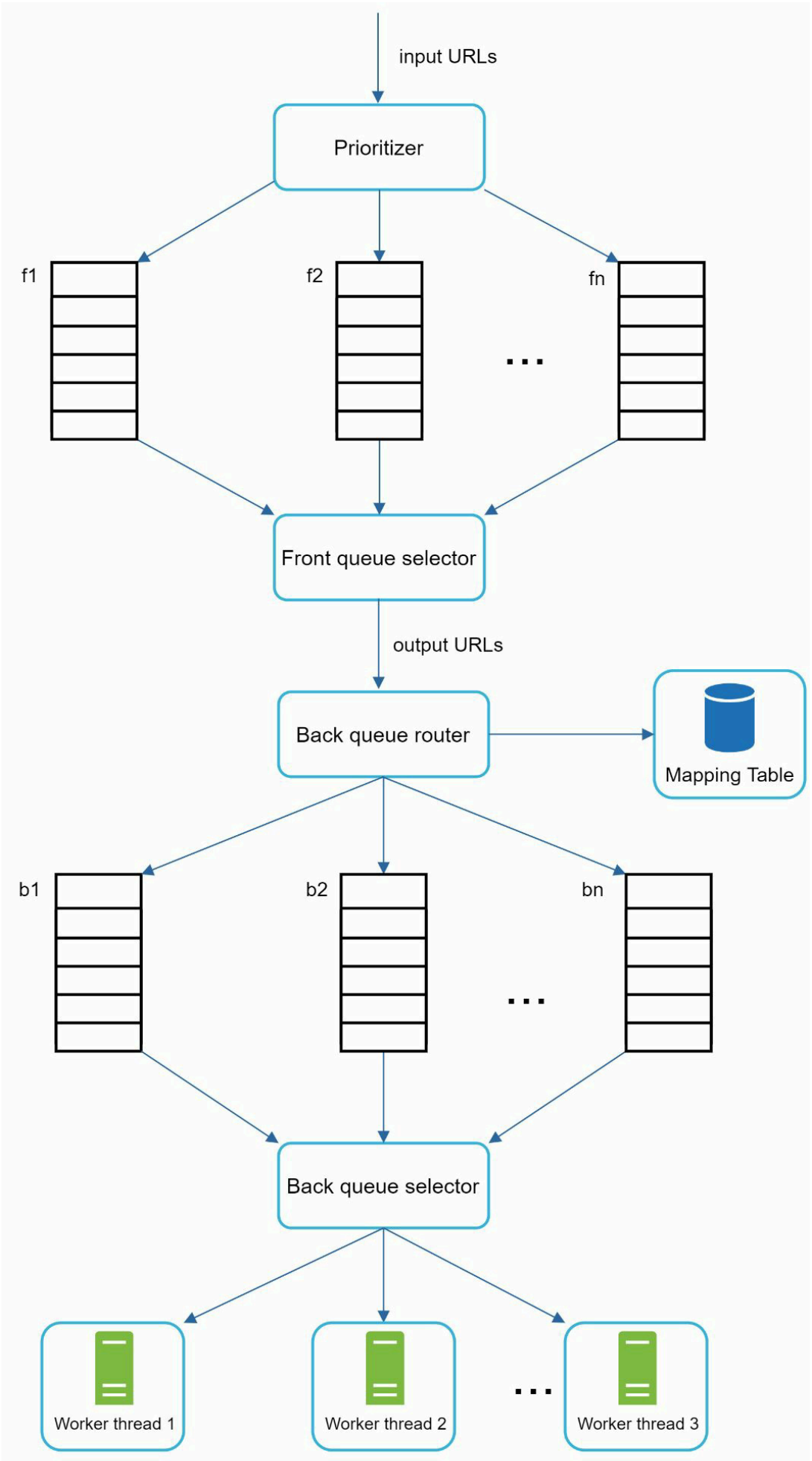

input URLs
Prioritizer
f1
f2
fn
Front queue selector
output URLs
Back queue router
Mapping Table
b1
b2
bn
Back queue selector
Worker thread 1
Worker thread 2
Worker thread 3

Figure 9-8

Storage for the URL Queue component:
We will use a combination of disk storage and in-memory storage, where the disk will obviously store most of the data, however the URLs which are being worked on by the Prioritizer module will be kept in the memory, in one of the queues. Do note as more and more new URLs keep coming in some of the URLs in the lower priority queues might need to be evicted form the main memory and swapped back to the disk.

How can we ensure Freshness of data
Web pages are constantly getting updated, deleted and created so how can the Web crawler keep the data fresh. We need to periodically recrawl the downloaded pages to ensure freshness, recrawling all the downloading pages is expensive and inefficient. A better strategy could be to:

1. Recrawl based on the priority of the web pages, i.e. recrawl more important pages first and more frequently.
2. Instead of blindly recrawling web pages, first check their update history, and compare the timestamp of the latest update with the timestamp at which the webcrawler fetched that page.

HTML Downloader: The HTML downloader downloads the web pages from the internet using the HTTP protocol.

A note about the Robots Exclusion Protocol:
The Robots Exclusion protocol is a standard used by websites to communicate with crawlers. The Robots Exclusion protocol maintains a file Robots.txt, which specifies what pages can be downloaded by the crawler, before attempting to crawl a website, the crawler must check the Robots.txt file to check which pages can be crawled.

To avoid repeated downloads of the Robots.txt file, we can cache it, and periodically redownload it.

Performance Optimization of HTML downloader: Here are some steps to improve the performance of the HTML downloader:

1. Horizontal Scaling: The HTML downloader can be scaled to run on multiple servers, each server running multiple threads. The URL space is partitioned into smaller groups and each server is assigned a subset of the URLs.

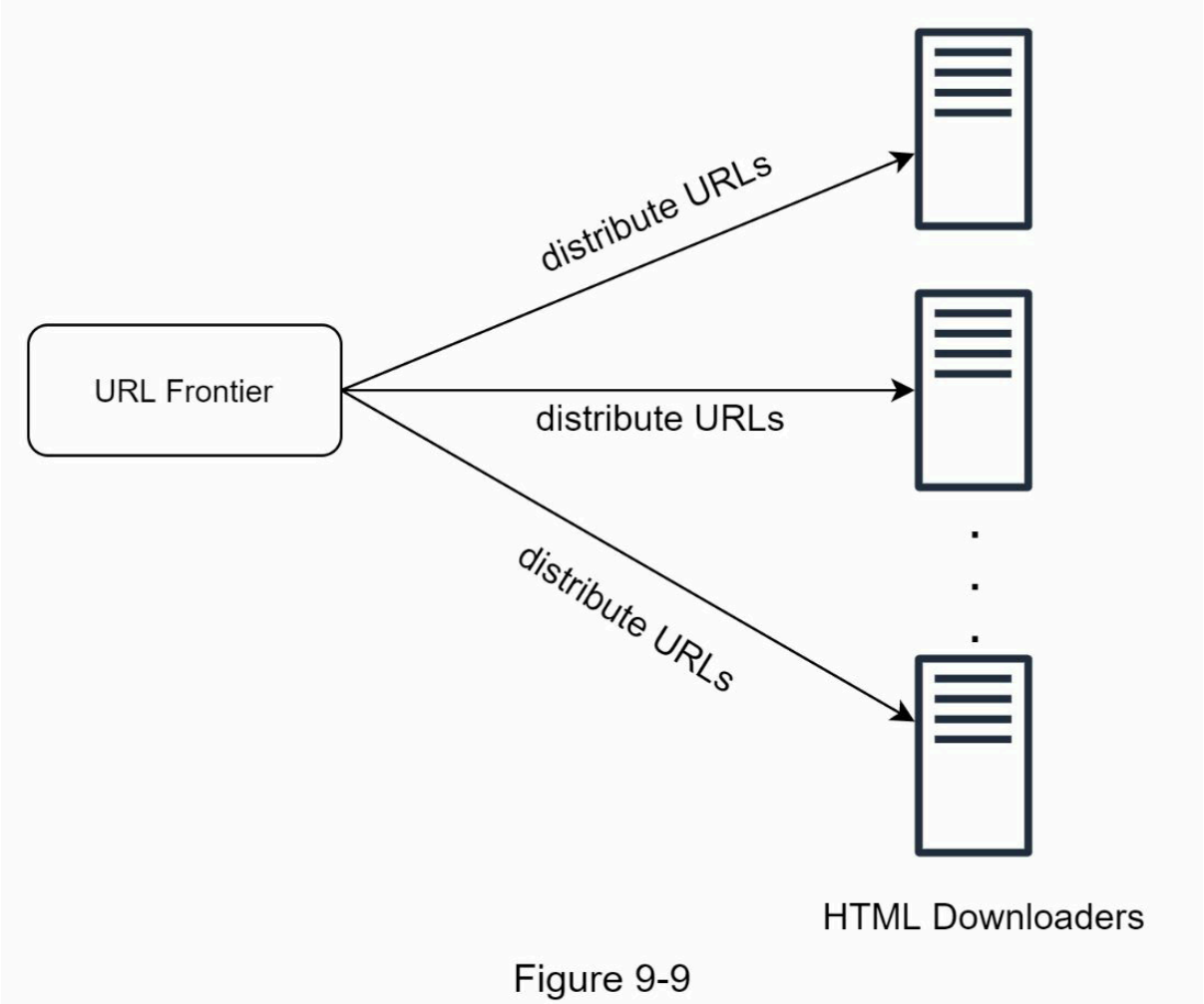


Figure 9-9

2. Cache the results of DNS queries

3. Short timeout - Some web servers respond slowly and some may not respond at all, to avoid waiting for a response for a long time, we can set a maximum threshold of waiting time, if we do not get a response in this predefined time the crawler will stop the job and move on to some other pages

Robustness: As the web is large, hence the HTML downloader will need to be robust enough to deal with certain errors and exceptions without crashing the system.

Also it is important to note since we are horizontally scaling the HTML downloader service, so we need to distribute the load evenly across the downloaders, this can be achieved by using consistent hashing, further a

new downloader server can be added or removed using Consistent hashing depending on the load.

Final points for improving the Crawler:

- Avoiding Spider Traps: A spider trap is a web page that causes a crawler to go into an infinite loop. Such spider traps can be avoided by placing a maximum limit on the length of the URLs. We need to exclude these URLs from the crawler.

- Avoiding Data Noise: Some of the content has little or no value, hence should be excluded from the Crawler, such contents includes advertisements, spam, code snippets etc.

Ola / Uber / Lyft System Design

Clarification Questions

1. Expectations: A user opens the app and selects the boarding and drop points, the app using some logic determines one cab and that cab is reserved for the user?
2. What is the daily traffic for the application?
3. Does the service support different types of cabs - Premium (SUVs), hatchbacks, mini etc.

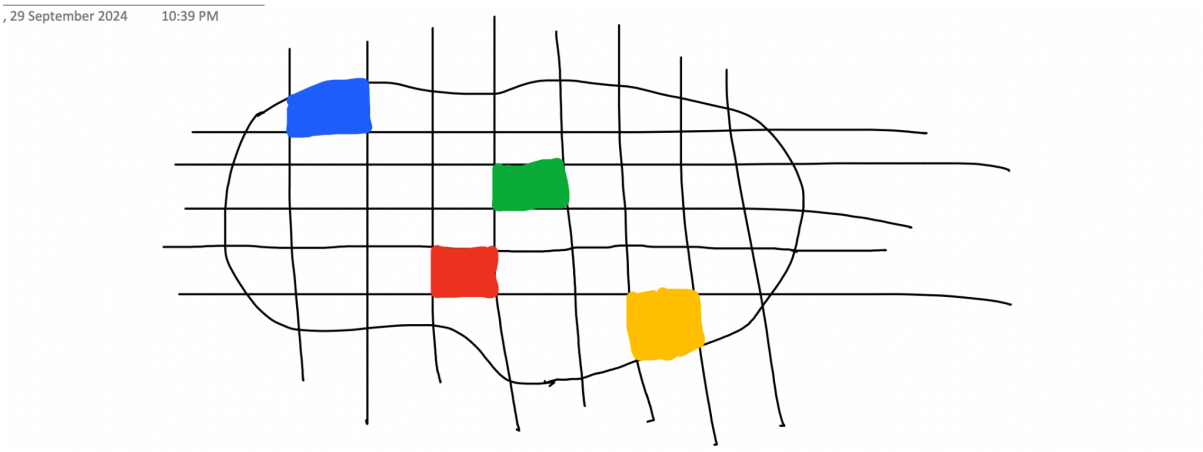
Requirements:

1. When the user opens the app, he / she should be able to see the cabs nearby.
2. Book cab
3. Display ETA
4. Display Price
5. Live Tracking

Design

The major requirement for a cab booking service is to find a cab near to the user's location, to achieve this what we can do is divide the city into a number of segments, we model the city in the XY coordinate system, and divide it into multiple segments / pieces.

Each segment can have an ID, given the current coordinates of a cab we can identify which segment it is in currently, however as the cab is moving its position will change, i.e. its segment will also change. To track this movement of the cab we need to send pings from the cab to the servers continually, to track the location of the cab.



Dividing a City / Town into segments

So as a cab moves it'll change segments.

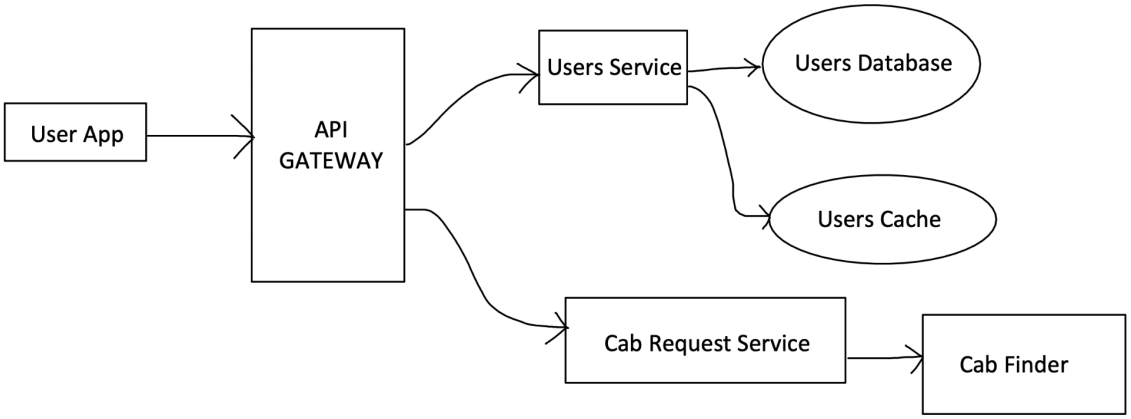
Here are the services we'll use in the design:

Maps Service: The Maps service will be responsible for dividing the city into segments and managing the segments.

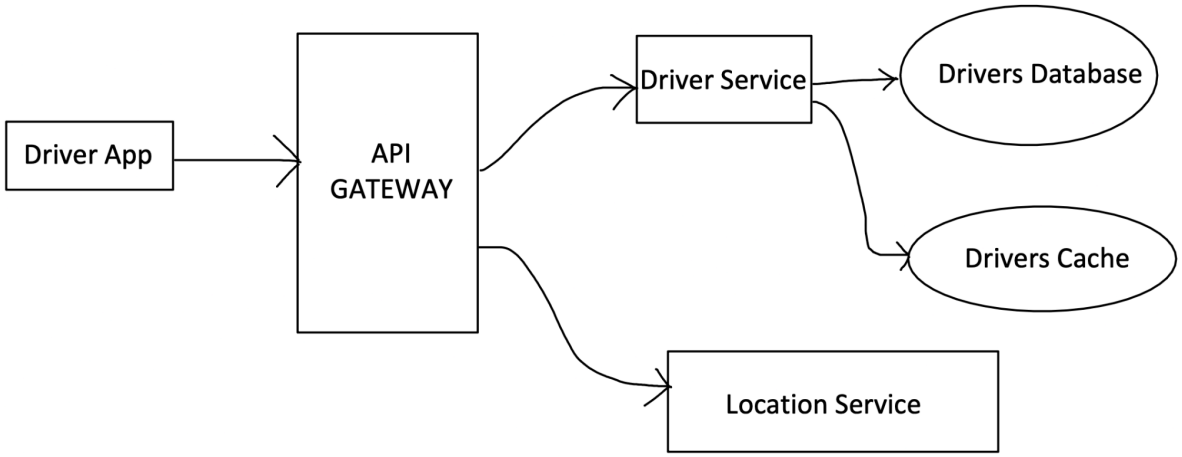
=> Given the coordinates of the cab (lat, long) determine which segment the cab is currently located in.
=> Similarly given the coordinates of a user, determine which segment he / she is currency located in.
=> Compute the distance from point A to B, the route and the ETA (duration of travel)
=> In addition it'll perform splitting and merging of segments, if the traffic is very high in a segment, we can split it into some smaller segments, on the other hand if the traffic is very low in some neighboring segments we can coalesce them into a single one. The goal is to have sufficient and small number of drivers in each segment

We can use the Google Maps service as our Maps service.

User Flow:



Driver Flow:



Users Service: The users service will manage all information regarding the user, it will connect to other services to get any information the user wants, for example information regarding past trips, this service will contain APIs to update profile, i.e. it will contain the CRUD APIs for the users data.

We can store the users data in a MySQL data, as the data is quite structured for example it contains:
=> username
=> Name
=> email
=> Contact, etc.

Drivers Service: The Drivers service will manage all the driver information, it will provide CRUD APIs for the driver information.

Additionally the Drive service will connect with the other services like the Trip service to get for example all the trips of the driver, or it can connect with the Payment service to get the payment history for the driver.

Again we can use a MySQL database for this purpose, as the data is quite structured.

In addition we can have a cache layer on top of the Users and Drivers databases(for example Redis) to quickly serve the queried information.

Cab Request Service: This is the service through which the user can book the cab by specifying the pickup and drop locations. In addition this service will display all the nearby cabs to the user. It will interact with the Cab Finder service, which will actually find and reserve the cab for the user.

The Cab Request Service responds to the user with the ride information.

Location Service: The drive will continuously send pings to the location service (every 5 - 10 seconds) as the cab moves around the city, location service will interact with maps service to identify the segment in which the drive is currently located.

Full Design:

The drivers will connect via the driver app to the application, via WebSocket connections. We use WebSocket since the connection is bidirectional. The drive app will periodically send pings to the location service over this connection, while the server should be able to send messages to the app as well, if for example a trip has been assigned to that particular diver.

But there could be millions of drivers connected to our application, how do we know which WebSocket each of them is connected to?
We create another service WebSocket manager, which stores the mapping b/w the driver and the socket, i.e which socket each driver is connected to.
<driver, web_socket>

So for example if the application wants to send a trip assignment to a particular user, then it'll identify the web socket handler for that particular

driver by checking with the WebSocket Manager service, once identified it can send the message.

The WebSocket Manager Service: As mentioned before, this service stores the mappings b/w the driver and the WebSocket handler over which the driver is connected to the application.

The WebSocket Manager service maintains a Redis database to store this information, this is because Redis is a key-value store and our data is fit to be stored in that format.

When the location service receives a ping it'll store the information in some database, for example Cassandra as there are millions of drivers, who are continuously sending location update pings, hence the amount of data is large. Hence Cassandra is a suitable database for this use case.

The location service will interact with the maps service, it will query it to identify the segment the driver is located in given the lat, long of the user. Once it identifies the segment in which the driver is located, the Location service will store the segment information in a Redis database.
The data will look like: s1: d1, d2, d3, this data changes when the drivers move from one segment to another. The use of storing this information is that given a geographical location we can quickly identify the nearby drivers.

Trip Service: The Trip service is the source of trip for all the trips ongoing, upcoming, the completed trips are stored as an archive and for auditing purposes. The trips service will expose all the APIs regarding trips, for example, get all trips, get trips by ID, get all the trips by a user or by a driver. etc.

User Interactions:
When the user wants to reserve a cab, a request will be sent to the Cab Request Service:

API DESIGN
PATH: /api/v1/book_cab
METHOD: POST
REQUEST BODY: {
    params: {
        "src_lat": _,

```
            "src_long": _,
            "dst_lat": _,
            "dst_long": _
        }
}
```

Cab Request Service passes on the information to the Cab Finder service, which'll find a cab for the user. Once a ride is found, the Cab Finder will return the detail to the Cab Request Service, this response will contain details like:

Vehicle details:
{}
Driver details:
{}
Trip ID:
{}
price: _

The Cab request service will return the information to the user app.

For analytics purposes, we can store information regarding the trip reservation to another service. This includes information like could the trip be booked? How long did it take to find a cab? etc.

How does the Cab Finder Service work?

The Cab Finder service communicates with the Location service, to retrieve a list of drivers who are nearest to the customer.

How does this work?

1. The Cab Finder service passes the user lat, long to the location service.
2. The location service interacts with the Maps service to determine which segment the user is located in.
3. The maps service returns a list of the segment in which the user is located, along with the neighboring segments.
4. The location service uses the Redis database to get a list of drivers, who are currently in any of these segments.
5. The location service sends this list of drivers to the maps service.
6. The maps service goes over this list and finds the k closest drivers.
7. The Maps service returns the list to the location service.

8. The location service returns this list of nearest drivers to the Cab Finder service.

How will the Cab Finder service choose a drive from the list?

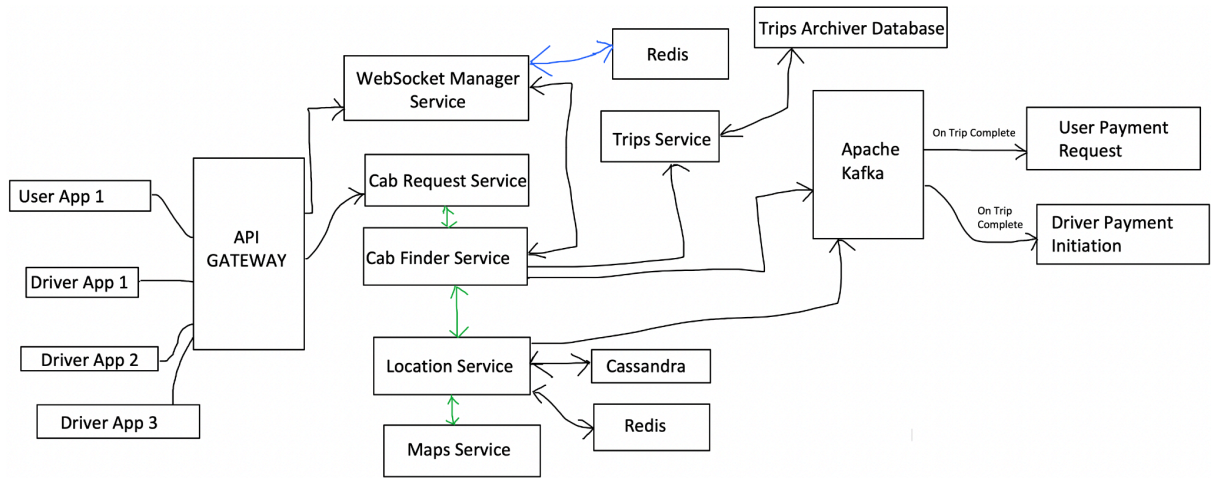
This depends on the mode, which is basically the type of the customer.
=> For premium customers we might want to choose the driver with the highest rating from this list.
=> For regular customers, we can just broadcast a ride request to all these drivers, and whoever accepts the request, will get the ride.

Once determined, the Cab Finder service will need to inform the driver that he has been assigned the ride, for this the Cab finder service will check with the WebSocket Manager service to find the WebSocket handler for that particular driver, and then send the message over that socket.

Additionally the Cab Finder service will inform the Cab Request service with the full ride information, including driver details. Finally the Cab Request service will pass on this information to the user app.

Once the rider is assigned, we'll create a new entry in the trips database.

User	Timestamp	Assigned Driver	Status	From	To



Design a hotel booking service like Airbnb, Booking.com

Requirements:

User:

A. Book a room
B. Search hotels
C. Check previous bookings

Hotel Owners:

A. Onboarding
B. Updating hotel details
C. Check bookings

Design:

Services:

Hotel Service: This service will store and manipulate all the hotel data, i.e. provide CRUD apis for hotels. The hotel owners will send requests to this service to add a new hotel to the system, or modify an existing one, or for example add photos.

The hotel service will store the hotel data in some database, since data is structured we can make use of a Relational database like MySQL.

The images and other media files can be stored in an object storage like AWS S3, and can be delivered by a CDN to achieve a lower latency.

For every creation / modification of a hotel in the hotel service, an event will be generated and stored in Kafka. The Kafka will have multiple consumers which will poll for these events, like the Notification service which can send a notification to the users regarding a new hotel being added, or an existing one being modified.

Hotel Search Service: The hotel search service will be used by the user to search hotels, based on parameters like date range, hotels, and other search filters like price, stars (4 star, 5 stars), rating etc. To build this search functionality the Hotel Search Service will use an ECS (Elastic Search) cluster. Elasticsearch is a database which provides fuzzy search. As an alternative to Elasticsearch we can also use Solar.

How do we populate the ElasticSearch cluster?
We create a Search Consumer on top of the Kafka, all the data of each individual hotel flows through Kafka, which the search consumer will poll and populate the ECS cluster with it.

Hotel Booking Service: The hotel booking service will handle the user requests for booking room(s) in a particular hotel. This happens once the user has searched and found a hotel matching theri parameters. Hotel Booking service handles the booking logic, and will store the booking details in a MySQL cluster, as the data is quite structured. The Booking service will interact with the Payment service, to process the payments.

The hotel booking service will additionally stream all the booking events to the Kafka, the Kafka search consumer will fetch this data and update the ECS cluster with the current hotel booking status. Additionally the Notification consumer will poll for booking events from Kafka, and send notifications (emails / sms) to the users, this includes details like invoice. It'll also inform the hotel owner that a particular room has been booked and by whom.

What to update in the ecs cluster?
Say for a date range, a hotel is fully booked so remove it from a cluster.

As mentioned before the booking data will be stored in a relational database, however this is not a suitable approach for storing all the bookings, as MySQL will not be able to handle such large amounts of data.

So we design a 2-tier storage system, where data for all the live / upcoming bookings is stored in the MySQL cluster, while data for completed / canceled bookings can be stored in a database which can handle very large volumes of data like Cassandra.

Booking Management Service: A separate service called the booking management service will be used by both the hotel owners and the users to get the booking data.

From a user perspective get a list of all the bookings, live, upcoming or completed.

From a hotel owner perspective, again get all the bookings against their hotel, ongoing / upcoming or completed.

This service will need to communicate with our 2-tier data storage system, to get the complete bookings data.
=> It will fetch the upcoming / ongoing booking requests from the MySQL cluster.
=> It will fetch the completed / canceled booking requests from the Cassandra cluster.

Finally it'll combine this data and hand it off to the requester.

To improve the performance of fetching from the bookings data, we can use a cache like Redis.

Analytics support - Since we are streaming all the events to Kafka, we can put a Spark Streaming consumer on top of it, which'll read the data from the Kafka and put it into a Hadoop cluster, on which we can perform queries.

APIs and data Modeling

Hotels Service:

APIs:

POST
/api/vi/hotels

PUT
/api/vi1/hotels/id

DELETE
/api/v1/hotels/id

Get
/api/vi/hotels/id

Database design:

Hotel	ID, Name, Locality_id (REFERENCES locality(id)), description, imageURL, is_active,
Locality	city_name, state_name, country, pin code
Hotel facilities	id, hotel_id (REFERENCES hotel (id)), facility_id (REFERENCES facilities(id)), is_active
Room Facilities	id, room_id, facility_id (REFERENCES facilities(id)), is_active
Rooms	id, hotel_id (REFERENCES hotel (id)), type: , is_active, quantity, price

Bookings Service:

Handles the logic to booking a room(s) in a particular hotel for a particular date range, it will communicate with the Hotels service to get the hotel details, like how many rooms are available.

Data Modelling:

Available Rooms: [room_id, initial_quantity (comes from the hotel service), available quantity (> 0 Constraint)

Booking: [id, room_id, user_id, num_of_rooms_booked, start_date, end_date, status_of_booking, invoice_id]

Status: [“Reserved”, “booked” “canceled”, “completed”]

APIs

POST /book
Request Body: {
    user_id, room_id, quantity, start_date, end_date
}

=> Check in available rooms
=> Insert in the bookings database and reduce the quantity in the available rooms.
=> Put the booking in Redis with TTL
=> Put in Kafka
=> Redirect the payment (Payment service)

Booking flow:

When a user sends a request to reserve a room, say by pressing the Book now button on the mobile app, the first thing the booking service will do is check the avaialble_rooms table to check whether the number of rooms requested by the user are available or not for that particular date range. If not, we can just error out.
If there are sufficient rooms available, then we’ll go ahead with the blocking of the rooms, temporarily, until the payment is completed, we will also add a booking entry in the bookings table with all the information provided by the user, however the booking status will be Reserved as of now, finally we’ll also decrease the available rooms quantity in the available_rooms table, this requires transaction support hence using a Relational database is a better choice.

If the payment is completed successfully the room will be booked, and the record in the booking table corresponding to this booking will be updated with Status = “Booked”, however if the payment could not go through, the room(s) will be unblocked.

How long do we wait for the payment to complete, i.e. how long to keep the room in the “Reserved” state. We can define a fixed period of time,

say 5 mins, within which payment has to be completed if not the room is unblocked, so that someone else can book it.

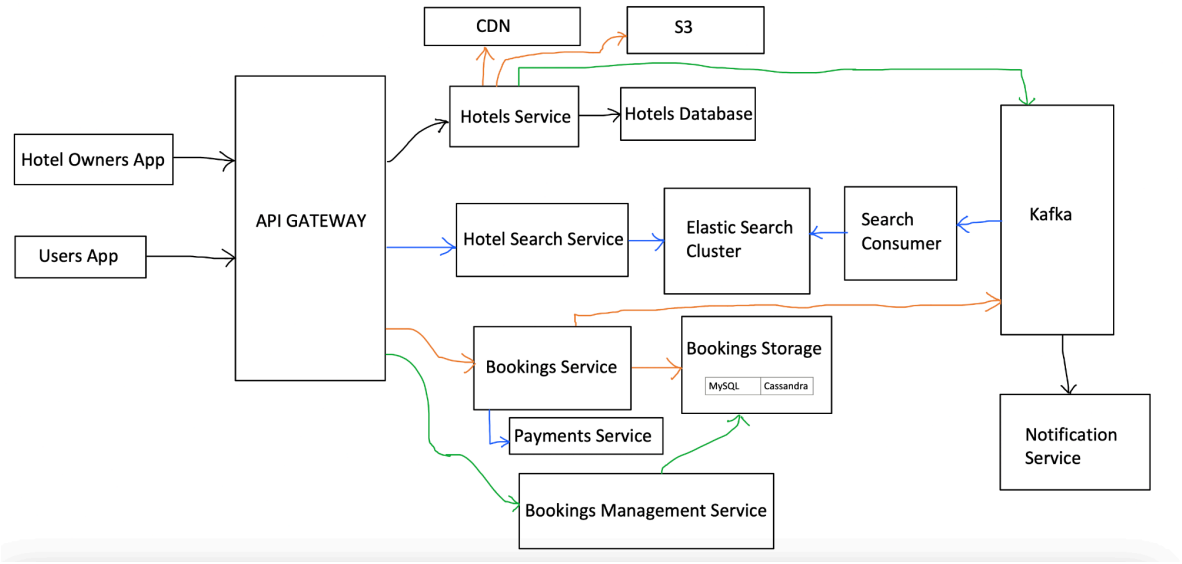
If we get a payment success notification from the payment service, then we'll change the status to Booked, however if the payment failed or couldn't happen in the 5 min window, then we can change the booking status to canceled.

If the payment does not go through, we'll need to rollback the entire transaction, i.e. reset the available rooms count in the availalbe_rooms table to what it was before the transaction.

We use Optimistic Concurrency control in our design.

Additional points to consider:

=> Monitoring, for example by using AWS CloudFront
=> Auto Scaling based on these metrics



Tinder System Design