System Design

(Gaurav Sen)

5 - common System Design Interview mistakes :

→ Don't go for a intro. Focus on core of the system.

→ Don't do capacity estimation without any reason.

→ Focus on Network protocols.

→ Lack on Internals knowledge on how a (redis, cassandra works internally).

A system design interview is not about building a working system. It's about measuring your aptitude.

Try to understand the internals of system instead of knowing what to use where;

→ Trade offs

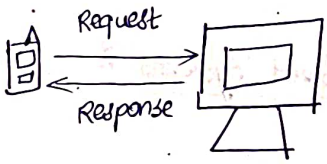
i.e. Think of trade-offs while designing a system.

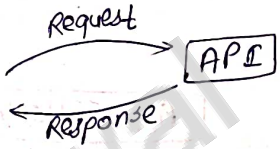
→ You need to stay with the problem that you are trying to solve.

i.e. stay close to the core problem.

System Design (Gaurav Sen)

Horizontal vs Vertical Scaling





Scalability

↓

being able to handle more requests

① Buy Bigger machine → Vertical Scaling
② Buy more machines → Horizontal Scaling

Horizontal Scaling

1 2 3 4 5

① Load Balancer required

② Resilient ie. if one of the machines fail, the request can be redirected to other machines.

③ Network calls bet^n two services
(Remote Procedure call (RPC))

④ Here, data is ~~consistency~~ complicated to maintain. So, here data consistency is a real issue here.

⑤ It scales well as users increases

Vertical Scaling

Huge Box

① No load balancer required

② single point of failure.

③ Interprocess communication
↓
fast

④ Data is consistent here. bcz there is just one system on which all the data resides.

⑤ There is a hardware limit.

→ The hybrid soln is essentially the horizontal scaling only where each machine has a big box

→ Initially, you can vertically scale as much as you like. Later on, when users start trusting you, you should probably go for horizontal scaling.

How to start with Distributed systems ?

Pizza Parlour — 1 chef + 1 backup chef

① Optimize processes and increase throughput using the same resource → Vertical scaling

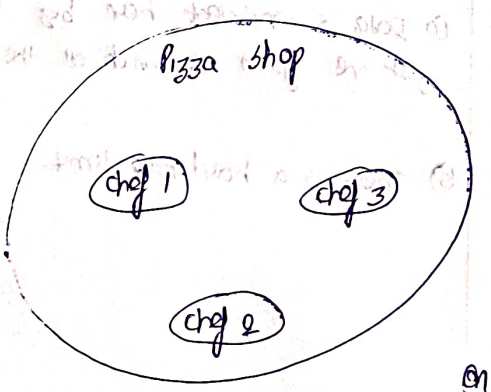
② Preparing beforehand at non-working hours → Preprocessing

Now, make the system resilient (recovering from failures quickly)

Hire a backup chef

③ i.e. keep backups and avoid single point of failure → Backups.
(just like Master slave architecture)

④ Hire more resources → Horizontal scaling
i.e. it is buying more machines of similar types to get more work done.



Here, chef 1 and 3 are experts at making pizza.

& chef 2's expertize is Garlic Bread.

On receiving an order, which chef should we assign it to ?

⑤ Microservice - based Architecture → You have all your responsibilities well - defined over here.

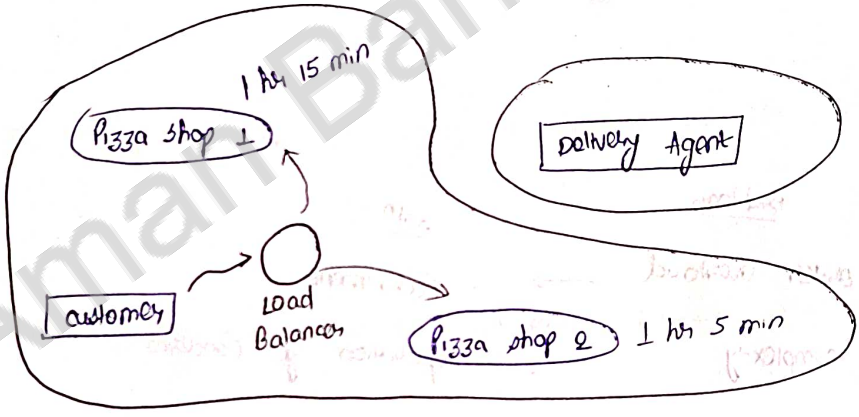
what if there is electricity outage in the Pizza shop?
what if you loose the licence for the day?

→ You have a backup pizza shop which will also prepare the food.

⑥ Distributed system
(Partitioning)

To give quick response time, you need some sort of local servers everywhere.
so, we are distributing our system so that it's more fault tolerant and also gives quicker response time.

⑦ Load Balancer



Everytime, a customer make a request, they need to either send it to one or two.

→ The system is now fault tolerant but how do you make it flexible to change?

⑧ Decoupling:

- Scalability
- Fault Tolerance
- Extensibility

→ The pizza shop whether it's the customer or delivery agent that is gonna come to pick up.
so, we are doing separation of responsibilities instead of having the same managers managing the pizza shop and delivery agents.

This called decoupling the system
ie. separating out concerns so that you can handle separate systems more efficently

⑧ Metrics :

ⓐ Analytics
ⓑ Auditing
ⓒ Reporting
ⓓ Machine Learning

You want to log everything ie you want to see at what time something happened, and what is the next event so and so forth.

Also, you want to be taking those events condensing them, finding sense out of those events

⑨ Logging and Metrics calculation

⑩ Extensibility :

Problems		soln
order overload	→	Recruitment
complexity	→	separation of concerns
Mishappening	→	Fault Tolerance

High level system design

↓

It is all about deploying our servers, figuring out how diff systems will be interacting with each other.

Low level system Design

↓

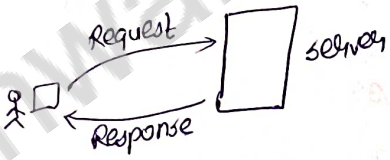
It has a lot more to do with how you're actually gonna code using classes, objects, etc.

→ System Design is the use of computer engineering principles to build large scale distributed systems. It involves converting business problems and requirements into technical solutions

Senior engineers use system design patterns to build reliable, scalable & maintainable systems.

What is Load Balancing?

Consistent Hashing:



The concept of taking actually n servers and trying to balance the load evenly on all of them is called load balancing.

The concept of consistent hashing help us do ~~that~~ to evenly distribute the weight across all servers.

→ You will get a request id in each request. When a mobile actually sends you a request, it randomly generates a number from 0 to (m-1). This request id is sent to your server.

$$\text{request id} \rightarrow 0 \text{ to } (m-1)$$

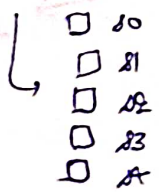
Take this request id and hash it. You will get a number (Let's say m_1)

$$m_1 = h[\text{req\_id}]$$

This number can be mapped to a particular server

Here, n = no. of servers

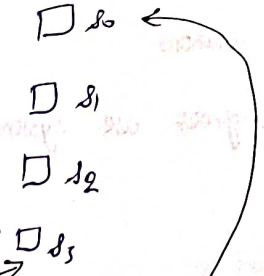
$m_1 \% n$ → whatever index you get here, you just send that request to the respective server



Eg: ① $r_1 = 10$, $n = 4$

$h[10] = 3$

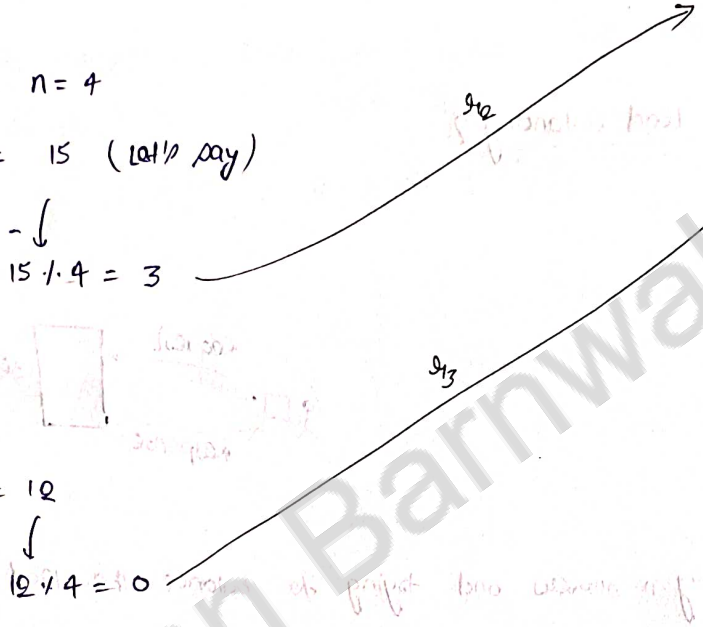
$3 \% 4 = 3$ → This is request r_1 will go to server 3



② $r_2 = 20$, $n = 4$

$h[00] = 15$ (call it say)

$15 \% 4 = 3$



③ & $r_3 = 35$

$h[35] = 12$

$12 \% 4 = 0$

→ Your hash fn is uniformly random so you can expect all of the servers to have uniform load.

→ If there are x ~~factor~~ requests will have $\frac{x}{n}$ load

& the Load factor is $\frac{1}{n}$

What happens if you need to add more servers?

→ If you are adding more servers ie. $n \uparrow$

so hashing index will change

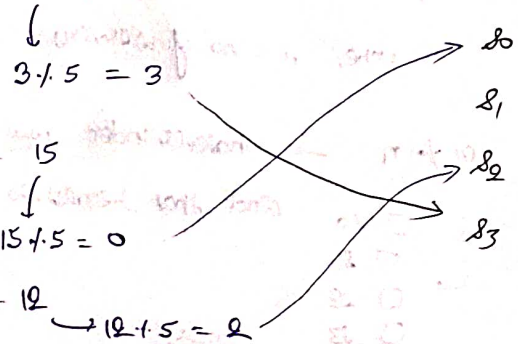
Eg: $h[10] = 3$

$3 \% 5 = 3$

$h[00] = 15$

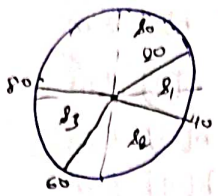
$15 \% 5 = 0$

$h[35] = 12$ → $12 \% 5 = 2$



Initially, n = 4

If n changes to 5



5 + 5 + 10 + 10 + 15 + 15 + 20 + 20

= 100

↓

i.e. entire possible space

→ Depending on the user id, we can send people to specific servers and once they are sent there, you can store relevant information in the cache for those servers

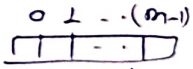
The overall change will be minimum if you are adding more servers

What is consistent hashing and where is it used?

The problem is adding & removing servers that completely changes the local data that we have in each servers and to avoid that, we will use consistent hashing.

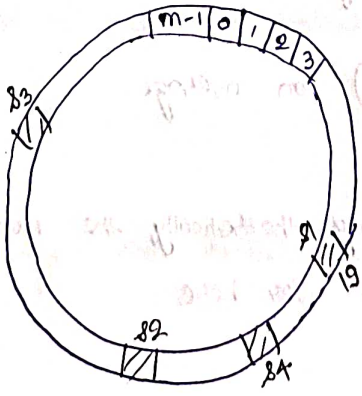
Request ID → h(key)

Instead of array



which can map this hash fn value from 0 to (n-1)

Ring of hash



We can take the servers and we actually need to send those requests to those servers.

The servers themselves have IDs from 0, 1, 2, ...

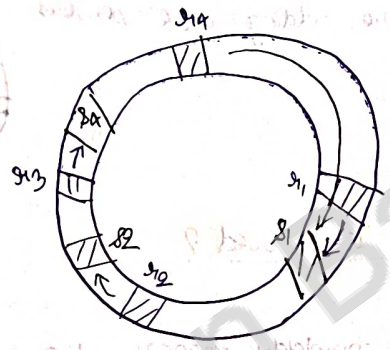
Using the same / different hash fn, we can hash this server id.

$m = 30$

server id		
0	hash →	$h[0] \% m = 49 \% 30 = 19$
1		$h[1] \% m$
2		$h[2] \% m$
3		$h[3] \% m$
4		$h[4] \% m$

ie. server 1 will be mapped to position 19

→ Whenever a request comes in this ring, we will go clockwise and find the nearest server.



This server s_1 is serving request r_1.

ie. s_1 has a load of two requests here. (r_4 and r_1)

s_2 — — — — 1 request (r_2)

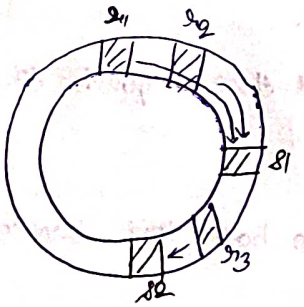
s_4 — — — 1 — — (r_3)

→ Because the hashes are uniformly random, you can expect the distance betn them to be also uniform, because of which the load is uniform.

So, the load factor turns out to be $\left(\frac{1}{n}\right)$ on average.

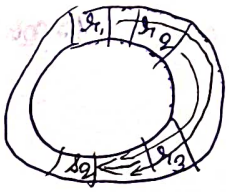
→ The problem with this architecture is although theoretically, the load should be $\left(\frac{1}{n}\right)$. Practically, we can have skewed distribution over here

Eg:



if server s_1 got removed, then all the requests r_1, r_2 and r_3 got directed to server s_2.

ie.



Here, you can start making virtual servers

↓

It does not mean that you have virtual boxes or you start buying more servers bz these are expensive.

What you can do instead is to use multiple hash fn.

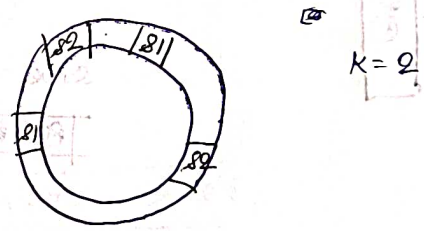
$h_1, h_2, \ldots$

→ If you have k hash fn from which you pass each of the server IDs then each server will have k points. So, here the likelihood of one server getting a lot of load is much much lesser.

→ If you choose k value appropriately, you can entirely remove the chance of a skewed load on one of the servers. ($k = \log n$)

→ Now, if a server is removed, you need to remove k points on it and clockwise assign them to the nearest server.

Eg:



→ It can be used in many many places

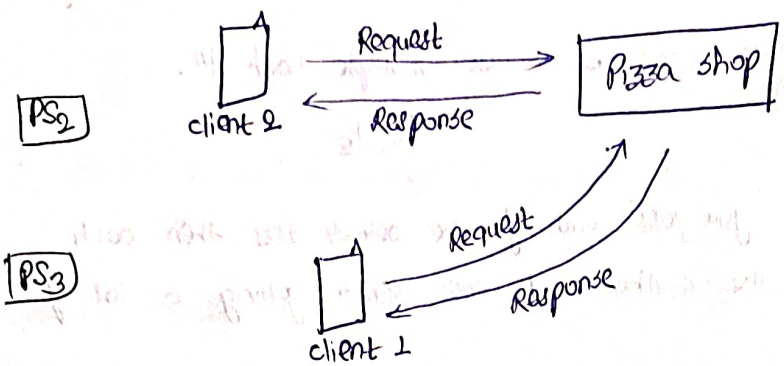
Load Balancing is used in Distributed systems extensively. This is being used by web caches, databases

→ Consistent Hashing gives you the flexibility and load balancing in a very very clear and efficient way.

What is a Message queue and where is it used?

→ PS_1

Eg:

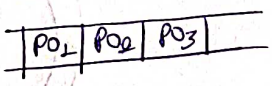


Multiple clients request their pizzas and they get the responses immediately like "Please sit down". You relieve the clients from expecting an immediate response by giving them a response which is not the pizza but a confirmation that the order has been placed.

List

You need list which contain order no.

Queue



→ Once a pizza has been made, you remove that order from queue. & you then ask the client to pay and the client send you back money.

This is asynchronous ie. you didn't make the client wait for your response for the payment or for the pizza.

→ Similarly, this allows you as the pizza maker to order your tasks a/c their priority.

Asynchronous Processing

→ Lets say, you have multiple outlets in your pizza shop. (Eg: Dominos)
Each pizza shop has multiple clients connected to them.

→ Assume that one of these pizza shops actually goes down, (bcz of power outage)
So, we need to get rid of all takeaway orders. Just, we need to dump them.
But, the delivery orders can be sent to other shops.

ie They can complete delivery orders and you can still save money

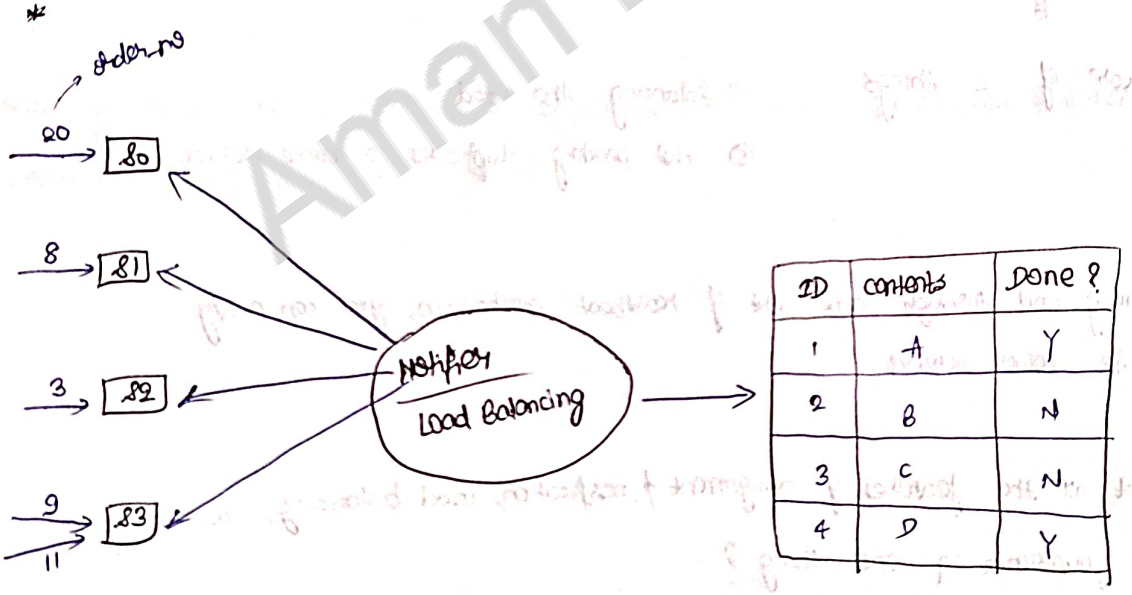
Eg:

PS_1

PS_2

PS_3 goes down

You need some persistent in your data
& you need to store this list in database.



set of servers

Database is storing the list

→ If S3 crashes, then order id 9 and 11 needs to be routed to other servers.
For that, we could have some set of notifiers, it checks for a heartbeat in each server.
ie A notifier talks to each server and ask them if they are alive every 10 or 15 sec.

→ If a server doesn't respond, the notifier assumes that the server is dead. If it is dead, it can't handle orders and then it can query the database to find all of those orders which are not done.

It picks these undone orders and distributes it among remaining live servers.

Here, problem is duplication.

Here, you can use load balancing.

↓

It will ensure that you do not have duplicates requests to the same server.

→ consistent hashing is one technique by which you can get rid of duplicates also.

↓

It will take care of 2 things: ① Balancing the load

② Not sending duplicates to same server.

→ Through load balancing and through some sort of heartbeat mechanism, you can notify all failed orders to newer servers.

What if you want all the features of assignment & notification, load balancing, a heartbeat and persistence in one thing?

→

That would be Message Queue.

↓

It takes tasks, persists them, assign them to the correct server, waits for them to complete.

If it's taking too long for the server to give an acknowledgement, it feels that server is dead and then assigns it to next servers.

→ There are multiple strategies of assigning the failed orders just like load balancing has multiple strategies but that's all encapsulated by a task queue.

→ We can use Messaging Queue or Task Queue to get work done easily so that you can encapsula all that complexity into that one thing.

An e.g of Messaging queue is
- "Rabbit MQ"
- zero MQ
- Java Messaging service (JMS)
  - Amazon also has some MQ.
- Kafka

They are really good encapsulations for complexities in server side.

→ Messaging Queues are widely used in Asynchronous systems. It provides features such as persistence, routing and task management.

→ There are multiple strategies of assigning the failed orders just like load balancing has multiple strategies but that's also encapsulated by a task queue.

→ We can use Messaging Queue or Task Queue to get work done easily so that you can encapsulate all that complexity into that one thing.

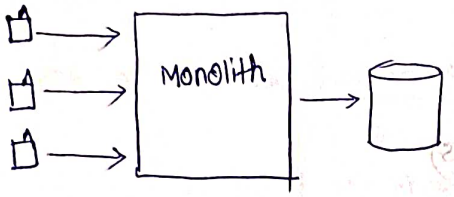
An e.g of Messaging queue is
- "Rabbit MQ"
- ZeroMQ
- Java Messaging service (JMS)
  - Amazon also has some MQ.
- Kafka

They are really good encapsulations for complexities in server side.

→ Messaging Queues are widely used in Asynchronous systems. It provides features such as persistence, routing and task management.

What is a microservice architecture and its advantages?

Monolithic vs Microservices

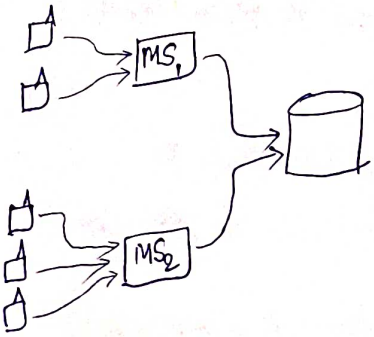


→ Monolithic does not need to be single in terms of no. of machines you are running it on.

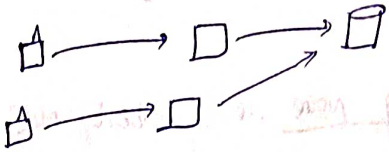
Because there might be multiple machines with the same monolith.

→ It is good for small teams.

When put under a lot of load, the monolith architecture scales out.



MS is monolithic server

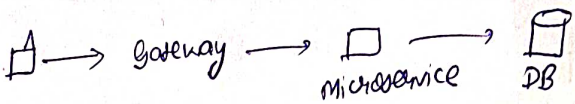


→ In microservices, you just have one of them running on one machine and they have tiny tiny databases with each of them connected.

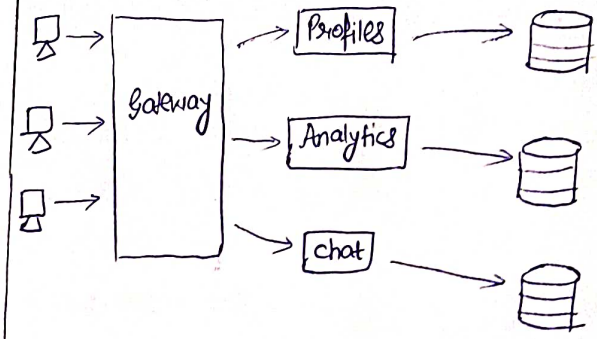
→ They usually have dedicated database connected to it.

→ The client might not be talking to microservice, it might be talking to a gateway.

& These gateway is talking to these microservices internally.



Eg:



→ Easier for new developers.

Easier to scale servers.

Monolithic

Advantages

→ Less complex

Deployment in monolithic application are easy

→ less duplication

All that code for setting up tests, connections, etc in your system need not be duplicated for every service that you create.

→ It's faster b/c you are not making calls over the network.

It's RPC call, which is faster.
(Remote Procedure call)

All logic and code are going to be in the same box so it's a lot more faster.

Disadvantages :

→ More context required if you wanna add a new member in the team

→ Complicated deployments b/c any change in the code requires a new deployment.

So, your code is going to be deployed very frequently and it has to be monitored every time it is deployed.

→ There is too much responsibility on each server.
i.e. single Point of failure.

→ Tight coupling in code as well as in developer time. i.e. more dependency of one task on another.

Microservice

Advantage

→ New services can be tested easily and individually

→ It's easier to scale.

→ Easier for newer team members

→ Parallel development is easy b/c there is lesser dependency for the chat developers on the analytics developer now. (Eg)

→ Used in a lot of companies like Google, fb, etc.

→ They are more available as a single service having a bug does not bring down the entire system.

→ Individual services can be written down in different languages

Disadvantages :

→ They are not easy to design.

→ A good indicator that you have a microservice that should not be microservice is that if it's talking only to one service.

→ It need smart architects to architect well for a service architecture.

* There is a very successful system which uses the monolith architecture which is Stackoverflow.

* Monolithics are favourable when

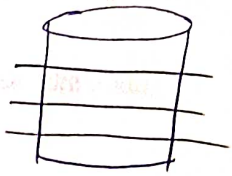
→ The technical / developer team is very small.
→ The service is simple to think of as a whole.
→ The service requires very high efficiency, where network calls are avoided as much as possible
→ All developers must have context of all services

Database sharding:

Partitioning of database:

Horizontal Partitioning: This kind of partitioning which uses some sort of a key to break the data into pieces and allocate that to different servers is called horizontal partitioning.

It involves putting different rows into different tables



Vertical Partitioning:

It involves creating tables with fewer columns and using additional tables to store the remaining columns.

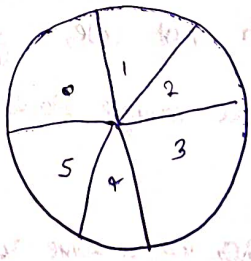
Database sharding:

It is a common scalability strategy used when designing server side systems.

The server side system architecture uses concepts like sharding to make systems more scalable, reliable and performant.

Sharding is horizontal partitioning of data a/c to a shard key. This shard key determines which database the entry to be persisted is sent to.

Some common strategies for this are reverse proxies.



Joins across shards

→ Eg: On application like Tinder, you could shard the data a/c to location

Problems in sharding that we have to take into consideration →

→ Joins across shards

The query need to go to two different shards, need to pull out their data then join the data across that network.

↓

This is going to be extremely expensive

→ shards are inflexible.

Memcached Database

Using hierarchical sharding, we can get rid of inflexibility over here.

→ Your read and write performance goes up bcz all of your queries falls on one ~~important~~ particular point.

What happens if a shard fails?

→ In that case, you can have master-slave architecture.

Here, we have multiple slaves which are copying the master.

Whereas it's a write request, it's always on master.

(The master is the most updated copy)

while, the slaves continuously pull the master and read from it.

→ If it's a read request, it can be distributed across slaves, while the write request always goes to master.

→ on case, the master fails, the slaves choose one master among themselves

Note:

Database sharding is the process of splitting the data table into multiple database instances in order to distribute the load.

However, sharding a database is an expensive operation (maintainability and overhead) and I suggest you do that only when you absolutely need to

How Netflix onboards new content : Video processing at scale

How Netflix onboards new content on their platform

→ Challenges faced in uploading new content :

we need to store it in different formats like mp4, etc. Eg different people have different connection speeds

① Different formats
- High quality
- Medium "
- Low "

codec

② Or Play with different resolutions
- 1020p
- 720p
- ⋮

A single video has multiple formats and resolutions and each is creating tuples

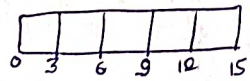
No. of videos that you end up processing = F × R

F: no. of formats
R: no. of resolutions

Netflix takes the original video, it breaks them into chunks, run those chunks into different resolutions and format

Eg: chunk A: mp4, 1020
chunk B: avi, 720

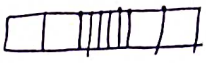
Video file



Each chunk is of 3 min each.

Breaking the chunks based on scenes rather than on timestamp

Each scene Netflix pay has a lot of tiny chunks of 4 sec each.



How Netflix onboards New content : Video processing at scale

How Netflix onboards new content on their platform

→ Challenges faced in uploading new content :

we need to store it in different formats like mp4, etc. Bcz different people have different connection speeds

① Different formats
- High quality
- Medium "
- Low "

codec

② Or Play with different resolutions → 1080p, 720p, ...

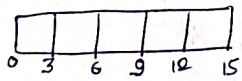
A single video has multiple formats and resolutions and each is creating tuples

No. of videos that you end up processing = F (no. of formats) × R (no. of resolutions)

Netflix takes the original video, it breaks them into chunks, run these chunks into different resolutions and format

Eg: chunk A: mp4, 1080
chunk B: avi, 720

Video file



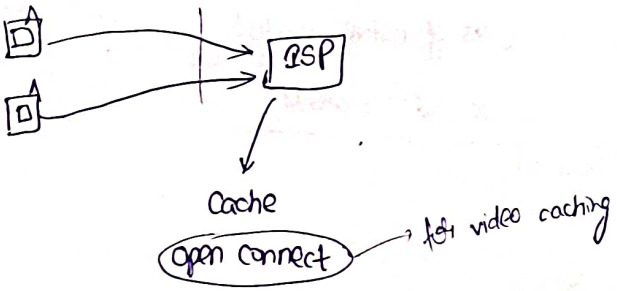
Each chunk is of 3 min each.

Breaking the chunks based on scenes rather than on timestamp

Each scene Netflix say has a lot of soft chunks of 4 sec each.

Sparse movie

→ Netflix stores all its data in Amazon S3.



→ Around 90% of Netflix traffic is taken care of by these ISP boxes that they provide. They are called open connect.

You can store all local popular movie in this open connect box.

→ Netflix provides open connect servers to ISP, which acts like a cache of movies.

Tinder as a microservice architecture:

Features:

→ store profiles (images – 5 images/user
→ Recommend matches (No. of active users)
→ Note matches × 10^{-3} matches
→ Direct messaging

How you are gonna store images?

→ as a file or Blob
(binary large object)

Objects are large in size.

clob
(character large object)

Database is giving these properties →

① Access control
② Mutability
③ Transaction ACID
④ Indexing

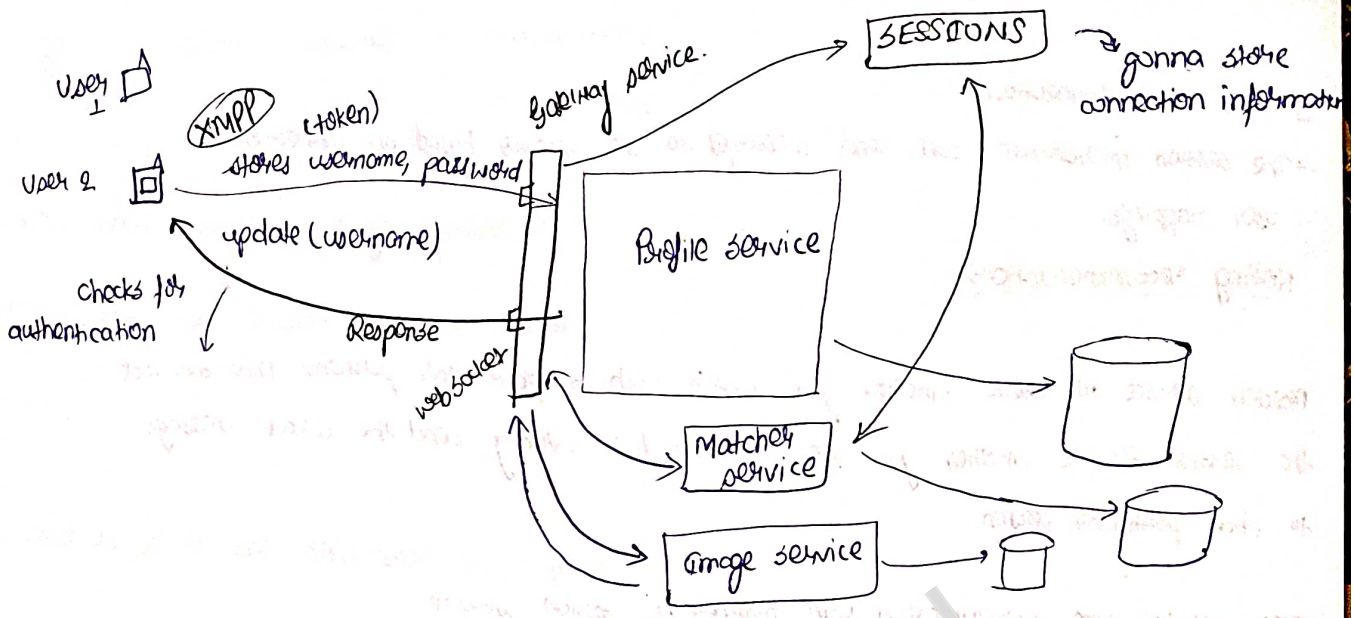
not good for images

File is having these properties →

→ cheaper
→ Faster
→ static so you can easily build CDN over it
↓
It allows fast access

profile id	image id	File url

Distributed File system → handles store profiles feature



The client always talks to the gateway. The gateway takes this request (token) and ask profile service whether this token is authenticated or not

→ Logically, you would probably want to store the images in a separate service.

Direct messaging: It can be done using xmpp protocol, which uses web sockets to have peer to peer communication bern client and server.

message to user 1 from user 2

message (UD_1, UD_2)

Each connection is built over TCP ensuring that the connection is maintained.

→ When you have client-server communication (HTTP) protocol, you cannot effectively have chat.

Bcz if there is a client and server, the only way that this user is going to get messages that are sent to them is poll to server ie. every 5 sec, it asks here is there any message for me?

↓

It is really inefficient

So, you don't want to poll the servers. You want messages to be pushed to you.

So, use peer to peer protocol. One of these protocols is XMPP.

→ This xmpp is going to take web socket connection. Instead of this, you can also use TCP with this connection, you can talk to clients

→ Web socket connection.

→ The session microservice can send messages to the receiver based on connection to user mappings.

Noting recommendations :

(24:18)

Matcher service will check whether you match with a particular person. That can tell the session service whether you are authenticated to actually send the direct message to that particular person.

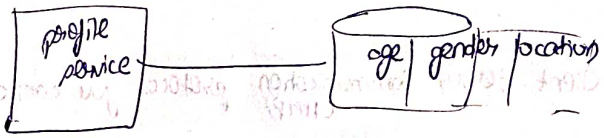
There will be some communication betn matcher & session service

The matcher service can note down all the matches that all have.

→ When you ~~let~~ uninstall the app, the only information you will lose is what all persons you swiped left or right. Biz this is not the critical info which needs to be stored.

Recommending people to you :

based on which gender you are interested in
age
location



You need to optimize on various parameters in dB.

↓

① So, you need to either use a NOSQL database like Cassandra.
or
Amazon Dynamo.

② Sharding (Horizontal Partitioning)
on Relational database

Eg: name → (A-J) → a node ①
(K-P) → node ②

what about single point of failure?

→ You can use Master slave architecture.

→ You need to shard the data based on location.

Eg: It could be chunks of a city. A person within this location is within this chunk and if it is within this chunk, they are being sharded to a particular node.

Based on that chunk also, you can pull the data out within that chunk, and then search amongst it within the age and gender.

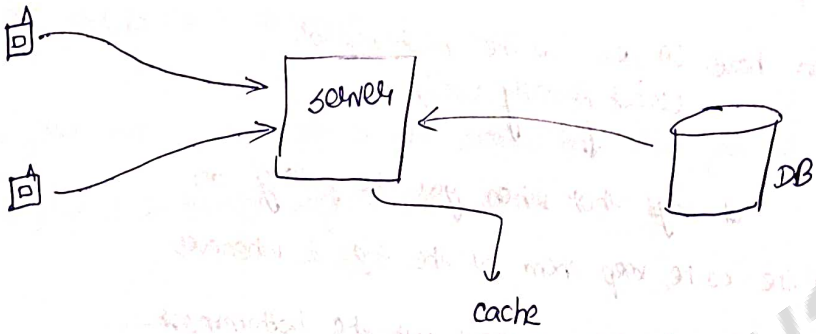
Recommendation service

It pulls out all relevant people.

What is Distributed caching ?

→ caching in distributed systems is an important aspect for designing scalable systems.

→ cache management is important bcz of its relation to cache hit ratios and performance.



Two scenarios where you might want to use the cache →

① when you are querying for some commonly used data
ie. to save network calls.

② when you want to avoid recomputation.

Eg: calculate age avg
↓
Find it once & store it in a cache with a key value pair.

③ when you want to avoid load on the database.
ie. reduce database load

All above helps us to speed up responses ~~from~~ to your clients. So, when client makes a call, you immediately gives a response if you have it in cache instead of making a query to the database.

→ The hardware on which a cache runs is much more expensive than that of a normal db.

If you store a ton of data in the cache, then the search time will increase.

→ The way in which you are loading and evicting data in the cache is called a cache policy.

↓

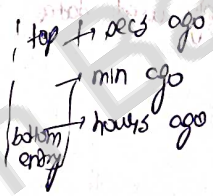
Cache Policy

* Cache performance is entirely dependant on cache policy.

There are multiple policies you can have. ⓐ LRU is the most popular. (Least Recently Used)

↓

It says that when you are pulling an entry to the cache, keep them at the top. & whenever you have to kickout data, kickout out the bottommost entry ie least recently used entries in the cache.



ⓑ LFU policy (Least Frequently used)

ⓒ Sliding Window based policy

Problems

Note: If you have a poor eviction policy, it's actually harmful to have a cache bcz you are making that call.

ie. Extra calls

→ Thrashing

→ If different servers makes a call to the database for an update, but the cache doesn't have that entry updated.

↓

Data inconsistency

Where to place the cache?

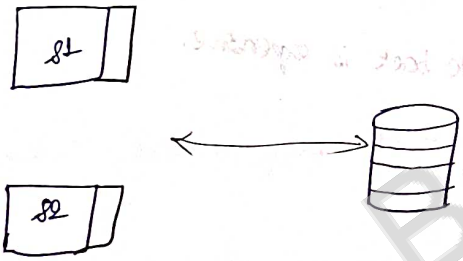
either you can place the cache close to the server

or you can place it close to the database.

→ There are benefits and drawbacks for both.

→ If you place the cache close to the server, how close you can place it?

You can place it in memory itself in the server



What if server s2 fails?

→ Then the in memory cache of server s2 also fails. And, we need to take care of that.

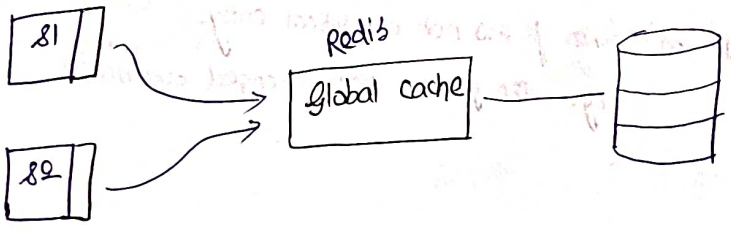
What if data on in memory cache of servers s1 and s2 are not consistent?

ⓐ benefits of having cache closer to the server →

It is faster

It is easier to implement

ⓑ



This is faster disk read.

Redis →

→ Here, even if s2 crashes, nothing will happen bcz all are anyways querying to the global cache.

↓

Higher Accuracy

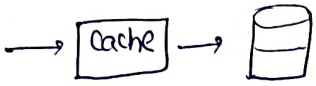
→ You can scale the Redis caches independently while keeping the services as containers running on these boxes

→ How consistent the data of cache is ?

Write Through cache

↓

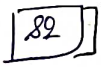
Here, I write on the cache before going through it and hitting the database.



Problems:

If you have multiple inmemory cache on different servers, then it will be a problem.

Eg:



Write back cache

↓

It is to hit the database directly once you hit the database, make sure to make an entry to the cache.

→ But, write back is expensive.

→ We can also hybrid sort of both write through and write back.

Whenever there is an entry that has to be updated in the DB, don't write back immediately. Write on cache if it's not a critical entry.

(Eg: The person has changed comment).

~~When you hit:~~

After some relevant time, take entries in bulk and persist it on the database so that's one network call and you are saving on hitting the database consistently.

→ If data is like pay passwords, you can't have write through cache, you need the write back cache.

What is an API and how do you design it ?

API Design :

↓

~~Applicatn~~

Application programming interface

→ Naming is important.

→ Don't take additional parameters unless absolutely necessary.

Eg: You want to fetch all group admins of WhatsApp.

getAdmins (string groupID)

error → group does not exist
↳ group is deleted

Response → List <Admin> admins

→ A lot of times, we need to expose our API to an HTTP endpoint.

Eg: POST www.aman.tech / chat-messaging / getAdmins / V1

- www.aman.tech → address of your website
- getAdmins → which API needs to be called
- V1 → version

Request : { "groupID": "123" }

Response:

```
{
  "admins": [{
    "id": "xyz",
    "name": "Aman"
  }]
}
```

Eg: GET www.oman.com/chat-messaging/admins?groupId=123

of API

* If response is much large, → you can break the response into pieces i.e. Pagination
                              → fragment the API.

Capacity Planning and Estimation

How much data does Youtube store daily ?

(a) Estimate Youtube's daily video storage requirements ⇒

Let's say, there are 1 billion youtube users. one out of 1000 upload a video of avg duration of 10 min.

$$\text{Total min} = \left(1B \times \frac{1}{1000} \times 10\right) \text{min} = 10^7 \text{ min}$$

Let's say, 2 hr video takes 0.4 GB (can be upto 4 GB)

1 hr — 0.2 GB

= 200 MB / hr

$= \frac{200}{60}$ MB / min

$\simeq$ 3 MB / min

$= \downarrow$

memory taken = 3×10^7 MB

= 30 Terabyte

= 30 TB

↓

This is the approximate amount of storage you need to just keep raw footage of Youtube.

I am assuming 3 copies for fault tolerance.

i.e. 30 TB × 3 = 90 TB

We also want multiple resolution of a video.

720p →	Let's say, it takes x MB
480p →	$\frac{x}{2}$ MB
360p →	$\frac{x}{4}$ MB
240p →	$\frac{x}{8}$ MB
144p →	$\frac{x}{16}$ MB

} sum $\simeq x$

If you want to store the video in all the resolution, memory taken = $x + x$

$= 2x$

∴ Total memory taken to for storing the video in all resolution = 2 × 90

= 180 TB

$\simeq$ 0.2 PB

ie. No. of users = 1B

Ratio of uploaders : users = $\frac{1}{1000}$

No. of uploaders = $1B \times \frac{1}{1000} = 1M$

Avg length of each video = 10 min

∴ Total uploaded video length = $1M \times 10 = 10M$ min

Let's say, 2 hr movie avg size = 4 GB

Optimized codec and compression savings = 90 %

∴ 2 hr Youtube video avg size = $4 \times (1 - \frac{90}{100}) = 0.4$ GB

ie: 2 hr — 400 MB

1 hr — $\frac{400}{2} = 200$ MB

ie 200 MB/hr

$= \frac{200}{60}$ MB/min

≈ 3 MB/min

Total video size = $10M \times 3$ MB

$= (30M)$ MB

$= 30$ TB

I could also think that 1 video consists of collection of images.

What's the size of 1 min long video?

1 min = 60 sec

No. of frames / min = 24

Size of an image = y (Let's say 1 MB)

∴ Total size = 60 × 24 × 1 MB

= 1500 MB

≈ 1.5 GB / min

You already calculated earlier that the storage that a video takes per min is 3 MB i.e. 3 MB / min

Huge difference, why?

You might have taken the wrong assumption of size of an image in video.

→ Youtube reports 600 hours of new content every minute

Here, we have just assumed above, 10^7 min of new content = 115 hours / min

60 min — 1 hr

1 day — 3600 min × 24

= 86400 min

∴ 1 day — 10^7 min of new content

$\frac{10^7}{86400}$ = 115 hours of new content / min

So you took a nice guess.

⑥ Estimate cache requirements for video metadata →

→ You have thumbnail and title of the video that you have to show.

↓

an image, so takes more memory than title of video

When you are showing the thumbnail to the user, you are not showing the actual image in the original dimensions, but you are showing in much much smaller size & dimension (Let's say 10th size) smaller from both side.

Let's say, image will take 1 MB in original dimensions

∴ Red Image in reduced dimensions = 1 MB × $\frac{1}{100}$ = 10 KB

↓

This is what you need in the cache for a thumbnail.

10 KB × No. of videos you upload in a day

↓

This could be let's say the popular videos in the last 90 days

(assuming 1 million videos a day)

Memory taken for thumbnail = 10 KB × 90 × 1M

$\simeq$ 10 KB × 100 × 10^6

= 10^9 KB

= 1 TB RAM → This is much more expensive than usual

↓

You can't get this 1 TB RAM in a single computer.

~~If I write~~

If I am using 16 GB RAM computer/nodes,

∴ No. of nodes required = $\frac{1 \text{ TB}}{16 \text{ GB}} = \frac{1000 \text{ GB}}{16 \text{ GB}}$

= 64 nodes in cache system.

If you are looking for any kind of redundancy, which is good to have bcz you need to serve video all over the world.

⑥ 3 copy of each node

ie. (64 x 3)

If any of those nodes crashes, then we are going to have cascading failure bcz you were at your peak capacity

If any one of them crashes, then loads on the other one will increase.

Assuming 50 % capacity

ie 64 X 3 X 2

≃ 500 nodes

ie. with 500 nodes, I will have a caching system, each with 16 GB which should be enough for Youtube metadata.

⑤ Estimate no. of processes required :

we want to know how much MB we can process per sec ?

1 million videos a day

1 video is of 10 min each.

ie. $(10^6 \text{ videos}) \times 10 \text{ min} = \left(\frac{10^7 \text{ min}}{\text{of videos}}\right)$ a day

assuming, 1 hr unprocessed video has average size of 1 GB.

$(10^7 \text{ min of videos})$ a day

$= 1000 \times 10^4$

$\times \frac{10^4}{3}$ hrs of videos

$\frac{10^4}{3}$ GB of videos to be processed in a day

$= \frac{10^4 \text{ GB}}{3 \times (\text{No. of sec in a day})}$ of videos in a sec

$= \frac{10^4 \text{ GB}}{3 \times (24 \times 60 \times 60)}$

= 40 MB of videos to be processed in a sec

When we talk about processing power, we talk about this is the amount of data that it can process in a second.

Read the video into the memory

Processing

Write it back

40 MB/sec is the raw footage

↓ if you are ~~doing raw footage~~ doing multiple formats in multiple data centres

Let's say, it ↑ to 400 MB/sec

i.e. 400 MB/sec

To read a single MB of data from a disk takes 10 msec

Writes can possibly require → Exclusive locks
→ Index updates

Let's say, write speed → 20 ms
processing → 20 ms

∴ Time taken for 1 MB of data ~~to read~~ = Read + Processing + Write
= 10 + 20 + 20
= 50 millisecond.

i.e. You need 50 millisecond to process 1 MB of data.

For 400 MB → $400 \times (50 \times 10^{-3})$ sec
= (20 sec of work to be done) per sec

↓

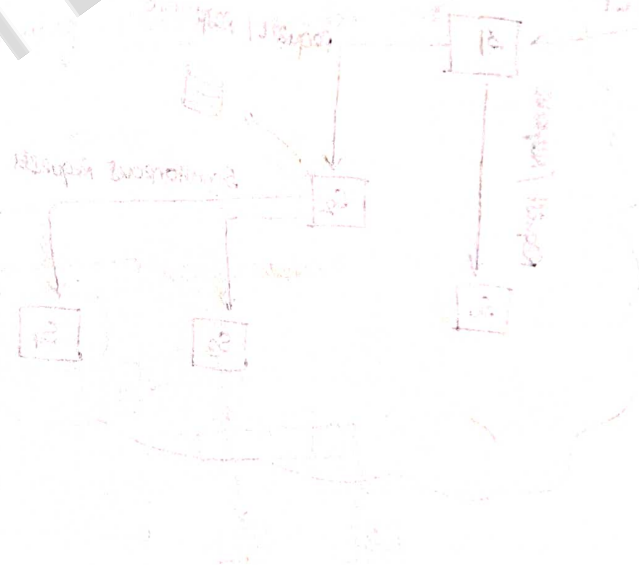
If computer can't do this work, so you have to keep computers in ||er.

i.e. 20 processors are required.
(by our assumption).

after on could be

→ Estimate the hardware requirements to set up a system like youtube.

→ Estimate the no. of petrol pumps in the city of Mumbai.



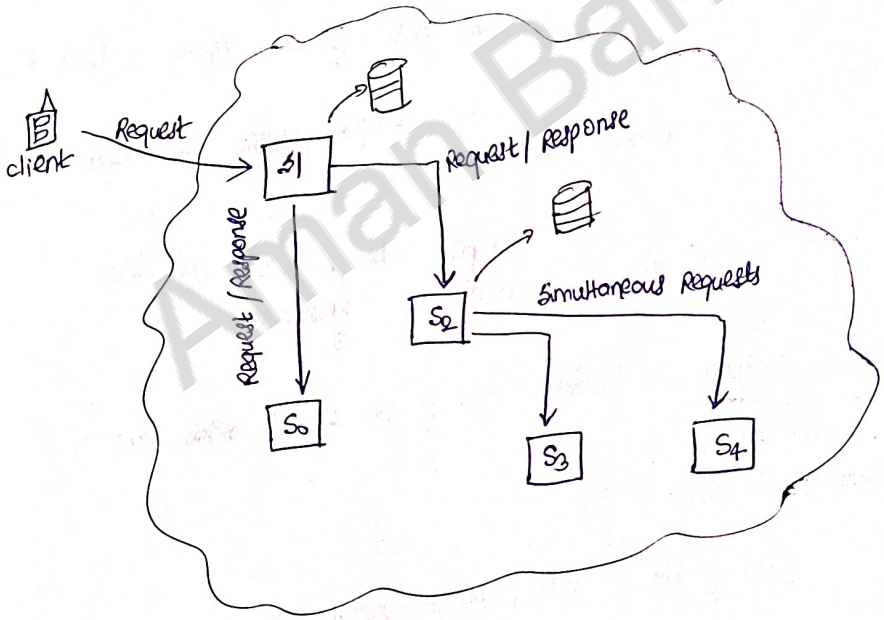
Publisher subscriber model:

It is also known as pub/sub. It is an architectural design pattern that provides a framework for exchanging messages b/w publishers and subscribers.

This pattern involve the publisher and subscriber relying on a message broker that relay messages from the publisher to the subscribers.

The host (publishers) publishes messages (events) to a channel that subscribers can then sign up to.

Event Driven system:



We have a microservice architecture in the above.

$S_0, S_1, S_2, \ldots$ are the services we have.

If you are using request-response architecture, smart thing to do is send the requests from S_2 to S_0 and S_2 asynchronously and wait for the responses.

The main drawback of this architecture is S_2 might be waiting for both of these services (S_3 and S_4) to complete even it is asynchronous

Strong coupling

if S_4 fails, S_2 is waiting for it, after a timeout it's going to send a timeout to S_1, which intuin is going to send the timeout to the client

↳ it took a long time for this request to fail.

↓

Failure Latency

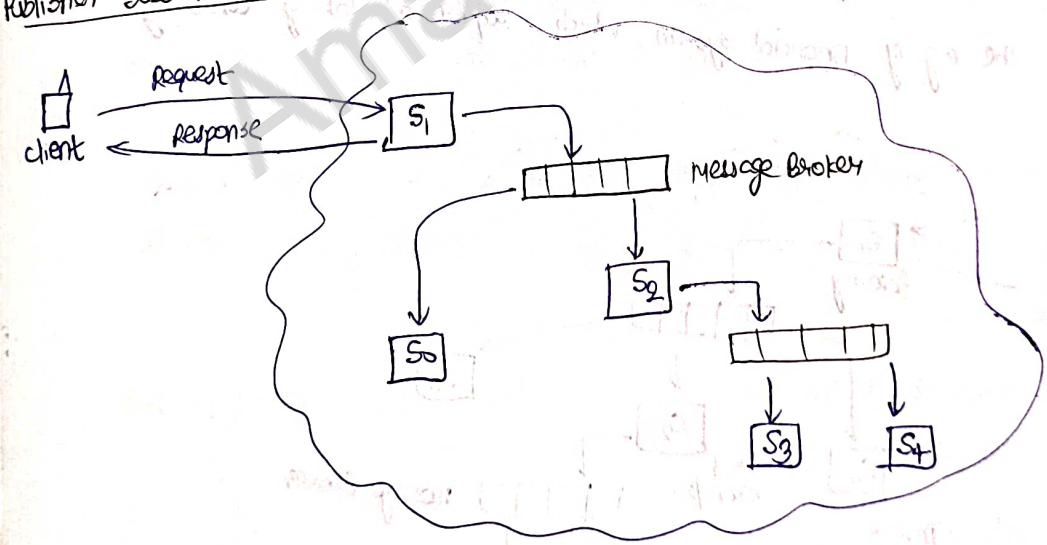
→ When the request is sent again, S_1 is going to make a change in DB again, and S_2 also is going to make a change in DB on getting the same request

↓

It leads to inconsistent data bcz there are two different changes for the same request

Better way to handle this is by using Publisher subscriber model.

Publisher subscriber model:



There are both advantages and disadvantages to this architecture.

Advantages

① → This is going to decouple a lot of responsibility that you had. Decoupling

ie. S_1 is no longer dependent on S_0 and S_2. Instead, it just publishes to the message broker. It sent a message to the client saying 'I'm successful'.

It simplies interactions. Instead of multiple point of failure, it's a single point of failure which is far more easy to deal with.

② It also provides transaction guarantee.

ie. If you are sending a message to S_1 and if it's able to persist to the message broker, that means that the message will somehow reach S_3 at some point of time in future.

This is a loose transaction guarantee of atleast once.

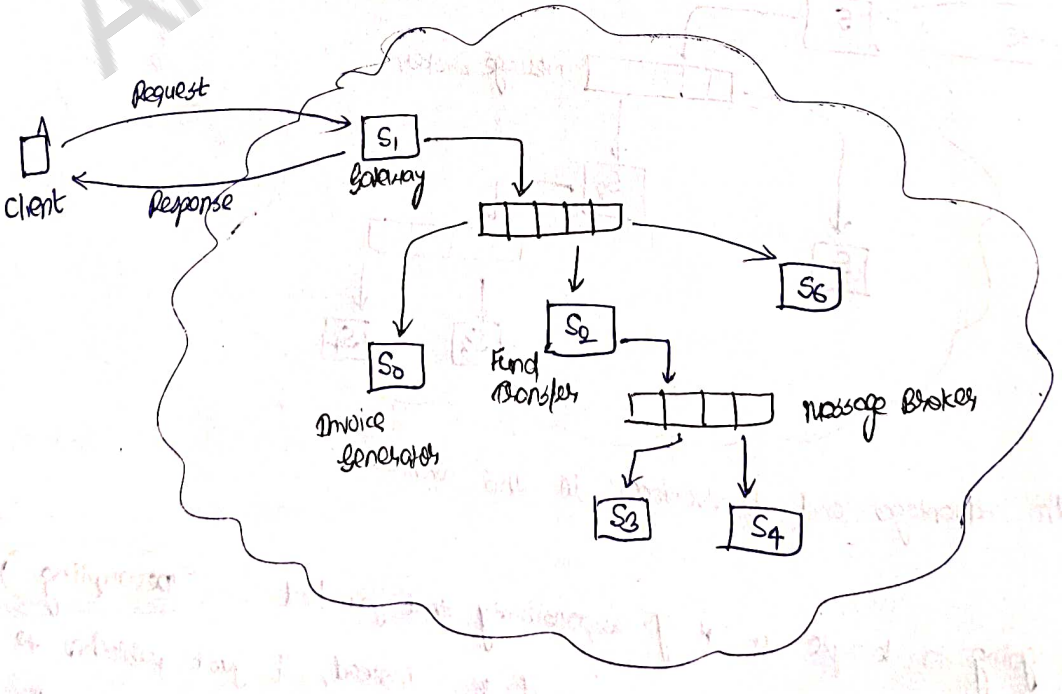
③ It's also more easily scalable.

Bcz if you have a new service S_6 which is interested in S_1's message, all you need to do is to register this service S_6 with the message broker, and then the message broker will send the messages being produced by S_1 to S_6 also.

S_1 do not need to know the subscribers of its messages.

Disadvantages: ① Poor consistency

Let's take the e.g of Financial system which require a lot of consistency.



Let's say S_0 is Invoice generator service

S_1 is Gateway (It just processes the message and send to S_0 and S_2.)

S_2 is Fund transfer service.

Let's say, you get a request from client saying that "Please transfer my funds of 950 from Account A to B.

Initial amount in A = Rs 1000

Bank charges for transfer = Rs 50

∴ After transfer, net amount in A should be = 1000 − (950 + 50)

= 0

Gateway gets the message, forwards it to SQ and so.

So says "Remove Rs 50 from the account" bcz that's my commission, and sends an email to the client saying that this is transaction amount you asked for.

1000
− 50
950

Let's say, Fund transfer service is down, which means the message broker is not able to send these messages to SQ.

↓

The client send then another request to transfer Rs 900.

So in total of 950 needs to be in your account.

(A) —900→ (B)

↓

Request come to SQ, It checks whether 950 is there in account.

It has, so it deducts Rs 50. Remaining balance now = 950 − 50

= 900

And, the Fund Transfer got two requests, { one of Rs 900, other of Rs 950 }

The first transaction of Rs 950 now fails bcz the remaining balance is just Rs 900.

The 2nd transaction of Rs 900 ~~succeedd~~ succeed leaving Rs (900 − 900) = Rs 100 in the account.

The initial expectation was there should be 0 rupee left and the 2nd transaction should have failed

↓

Instead, you have 100 remaining and 800 transferred, and Rs 100 as commission to the bank.

so, there is a lot of confusion bcz of having a transaction across services.

↓

Poor consistency

② It doesn't guarantee anything about idempotency.

Eg:

Let's say, SB is doing something which is not idempotent, let's say withdrawing Rs 50 from your account. It might fail when publishing over to message broker.

So, it sends the failure response.

After the message is replayed, it might have debit Rs 50 more from here and then sends the message again.

↓

So, here you are seeing the issue of 50 Rs removed multiple times bcz the message that the message broker is sending is not idempotent.

Instead, it should be also sending request id along with it.

Publisher subscriber model is the basis for Event driven services.

↓

It can't be used for finance services

but can be used for gaming services.

Note: Twitter uses 'Publisher subscriber model' architecture.

For that business requirement of posting a tweet and many people actually consuming that tweet, the publisher subscriber model is perfect.

You have events published and subscribers for that.

some well known messaging frameworks based on Publish-subscribe pattern —:

* Apache Kafka
* RabbitMQ
* RushPin

Why do Databases fail ? Anti Patterns to avoid

Database are often used to store info, but one case where it becomes a problem is when being used as a message broker.

The database is totally designed to deal with messaging features, and hence is a poor substitute of a specialized message queue. When designing a system, this pattern is considered an anti-pattern.

Here are possible drawbacks →

① Polling intervals have to be set correctly. Too long makes the system inefficient. Too short makes the dB undergo heavy read load

② Read and write operations heavy on DB. Usually, they are good at one of the two.

③ Manual delete procedures to be written to remove read messages.

④ Scaling is difficult conceptually and physically.

Disadvantages of message queue:

① Add more moving parts to the system.

② Cost of setting up the mq along with training is large.

③ It may be overkill for a small service.

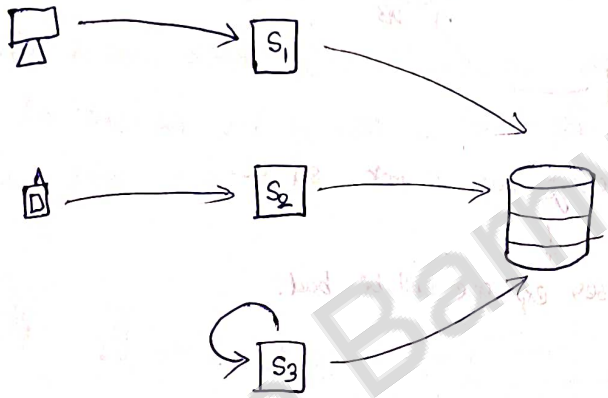
→ Using database as message queues

Anti Pattern :

so, if you are using a database to send messages from one server to another, that's considered an anti-pattern.

so, you should avoid that.

Eg:



We have S_1, S_2, S_3 servers. They are connected to desktop clients, mobile clients. And, they talk to each other using this database.

When S_2 wants to send a message to any other server, it goes to the database and then puts in an entry ie. an insert command.

Through these inserts, you are telling the database that there needs to be some communication with some other server.

How will the database actually tell it to other server?

→ The database can't talk to servers, it can only receive. So, the other server has to poll this database at a specific interval (could say 5 sec, 10 sec, etc) and ask it then — "Is there any new message for me"? If so, it will just read the message.

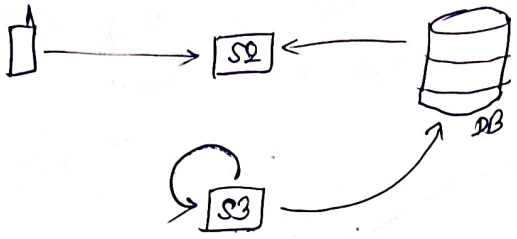
Here, dB is just being used as a queue.

Polling a database is actually a problem. Bcz when you are polling the database, you are actually doing a lot of read operations on it. And, you are polling it very frequently, you are actually putting a lot of load on dB ie. continuously asking "Is there any message for me"?

Frequent loading on dB leads to load on dB.

→ If you poll at long intervals, may be our mobile application wants the message in 2 to 3 sec, but let's say poll interval is 50 sec.

Eg:



Polling interval = 50 sec

S3 wanted to send a message, but it took 50 sec.

↓

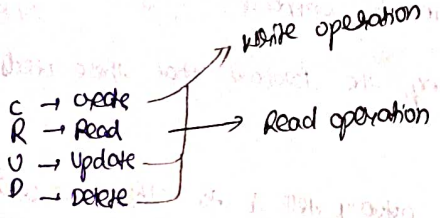
User experience will be bad.

So, long intervals are inefficient.

→ A database is optimized for either reading or writing, not both.

When you are having these both operations being performed in the same database, it's going to have issues like locking, deadlock, etc.

→ If you have CRUD model



They are loosely designed such that the read and write operation will both be fast.

→ What if your servers are talking to each other very frequently?

What if there are a lot of messages these servers are sending to each other?

↓

So, These dB will get filled up with these entries.

Either you need a lot of space to store all that data or you can start clearing that data.

if you are not going to delete that data, you need to update it and mark it then completed.

↓

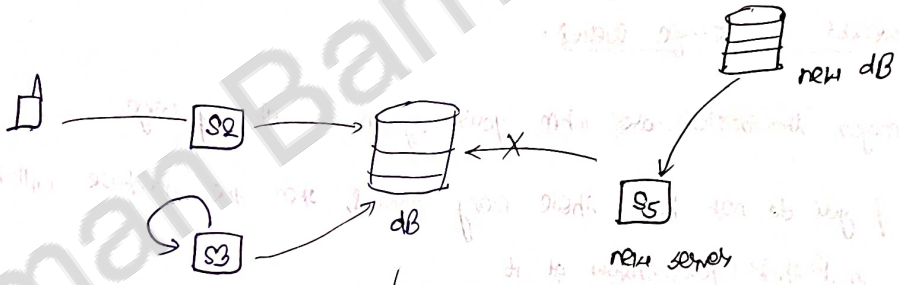
That's an expensive operation.

→ The other way is to manually delete these entries & use a cron job. In both cases, the solution is going to use scripts using complicated logic.

→ The final issue which is a very big issue for large systems is scalability.

Eg: If we have a new server S_5, and when this server starts asking the database for messages sent by other servers. This database is ~~getting~~ going to be unable to handle that load.

Eg:



↓

It cannot perform these many read operation due to addition of new server S_5.

One thing I could do is to actually add a new dB which is going to talk to new server S_5.

But how does S_5 talk to S_2?

→ You could have a broker in between and then they could talk to each other through this broker.

↓

But ~~that's~~ there's so much of complication, when you have a system design concept (a data structure) which is going to solve this and that is "message queues".

Message queues

→ So, the anti-pattern here is that we are using databases as message queues. Instead, we should use specialized message queues, when we have a very large system.

Message queue is going to avoid all these above problems.

→ The polling interval will not come into consideration bcz the message queue will push the message to the other server.

So, instead of the server asking, the message queue is going to give the message. So, it's not going to be inefficient in that way.

They are optimized in such a way that there is not so many read operations. So, writing to it is easy while reading is really the message queue responsibility.

And, the other thing about scalability is also going to be taken care of because if you need more message queues, you can just add them.

Drawbacks of message queues:

① The major drawbacks are when your system is not very large.

Bcz if you do not have those many servers, then the database will be able to handle the load that you throw at it.

② Depending on your system, may be there's not a lot of messages being sent bcz the systems. In that case, writing is going to be expensive, but it's going to be few while reading is going to be a lot, but you can optimize your database for read operations, and still work.

③ Whenever you introduce a new concept to a system, it requires training and time. And may be, this system is not worth the effort.

It's a call that you have to make, but may be this database is a perfect message queue for you.

Note:

So, have a look at the drawbacks and pros, & then decide whether you want a message queue or a DB.

→ But if you are asked in an interview, then don't start off with the database without asking what kind of scale is this system supposed to handle.

So, 8x3 90% of times in system design interview, they are asking you to scale the system or handle a lot of users.

If you're using a DB, then that's an anti-pattern.

Keep that in mind, it's not always bad, but for large-systems, the DB is

→ In a system design interview, it is important to be able to reason why or why not a system needs a message queue.

These reasons allows us to argue on the merits and demerits of the approaches.

→ In general, for a small application, databases are fine as they bring no additional moving part to the system.

For complex message sending requirements, it is useful to have an abstraction such as a message queue handle message delivery for us.

Content Delivery Networks (CDN)

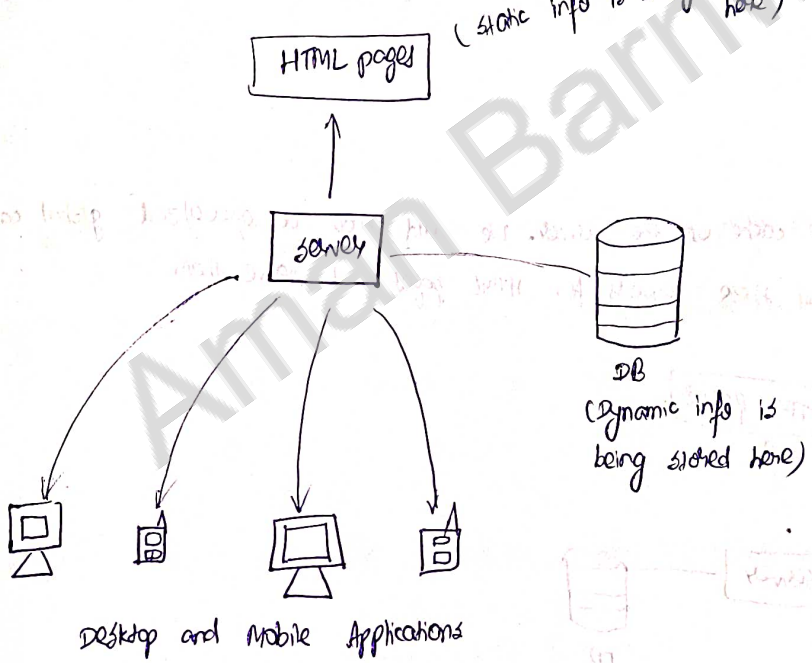
Content Delivery Networks are a bunch of servers spread across the globe to serve information. These networks are available on rent to deliver static content quickly to nearby users.

→ Some examples of CDNs are Amazon CloudFront and the Akamai CDN.

They are relatively cheap to rent and have high availability.

They also provide pluggable algorithms to invalidate and fetch data.

Eg:



① → Data Path is long i.e. there's a lot of distance that will be travelled every time a user is actually calling for a single page.

↓

So, we can cache static pages onto ~~another~~ the server so that we don't need to make this call everytime.

② You may have different kind of HTML pages that we want to send to different types of ~~services~~ devices (Desktop, mobile, tablets, etc).

So, there is customized data that you want to send depending on the device or location.

Eg:

Let's say, you get 5 combination for devices. And let's say, there are 200 countries

ie $5 \times 200 = 1000$

so, if you are serving all 200 countries with unique web pages, then you have atmost 1000 unique data points that we will be serving.

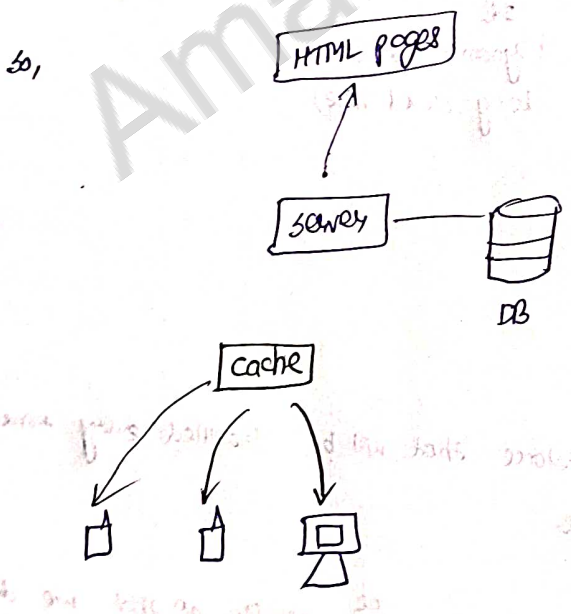
↓

1000 is a manageable no so we can keep it in cache.

③ we want our web pages to be able to serve to users quickly.

ie. Fast.

we don't need to keep the cache on the server. We could keep a specialized global cache, which is going to take all these requests for HTML pages and serve them.



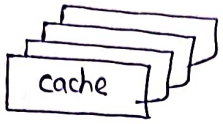
so, all the requests for getting an HTML page / static content are being now redirected to this cache, and all other requests (for dynamic data) are being sent directly to the server, which then consults the dB to retrieve dynamic info and send it back.

→ There is one possible problem with this cache.

This cache is a single point of failure. So, when this crashes, the whole system collapses.

↓

So, in order to remove this problem, we can make it a distributed cache.



Distributed cache

→ Let's assume that Distributed consensus is no longer the problem. You have Paxos / Raft to take care of that.

Even then we'll have some issues.

Eg: If you take users only from India, then let's say they are only concerned for 5 pages. ∨ Indian.

And we are storing a lot of irrelevant info for some countries in this cache.

for no reason when it comes to Indian users.

↓

So, we can horizontally shard this cache based on location / country of the user.

ie. Horizontally partitioning of a cache

Horizontal sharding based on country / location of the user

Distributed Cache

However, there is still a problem.

The problem is that you have the distributed global cache in your cloud, which is probably hosted in a single country (let's say, USA).

↓

So, the Indian users are being sent to a group of nodes (which are caches).

But it's not still serving the purpose.

Bcz, the Indians are still sending the message which is travelling all the way to us and coming back to them just to get a silly HTML page.

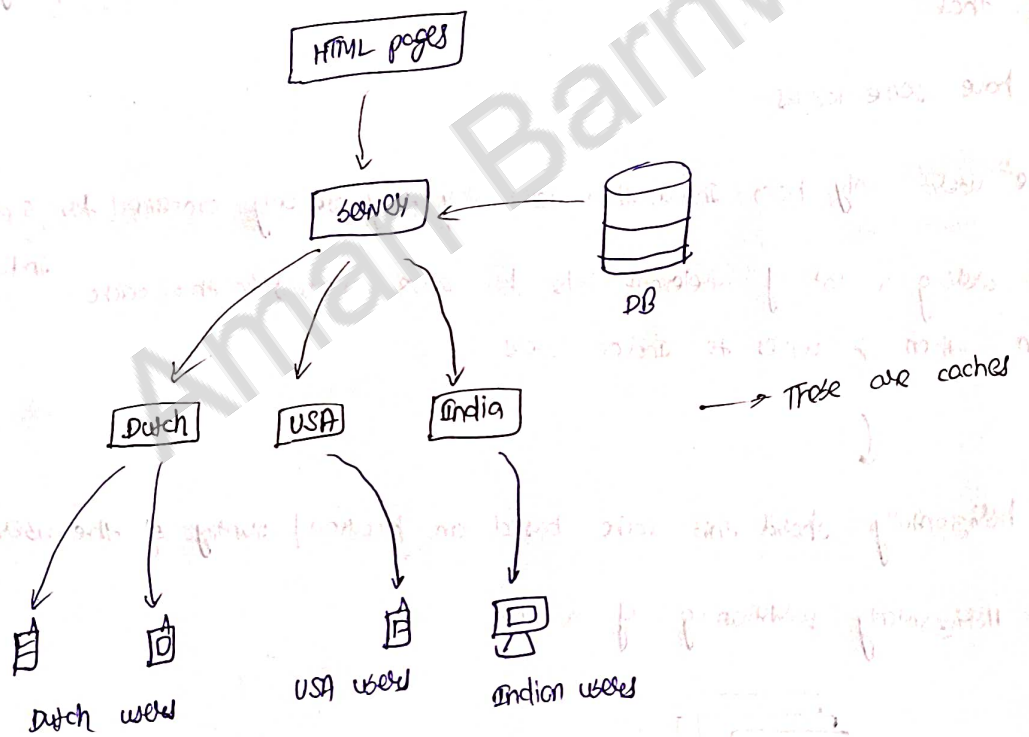
↓

so, you don't want to do that.

You want to rather construct a data centre in India, & you can create a cache here in India, through which Indian users can access the HTML page.

↓

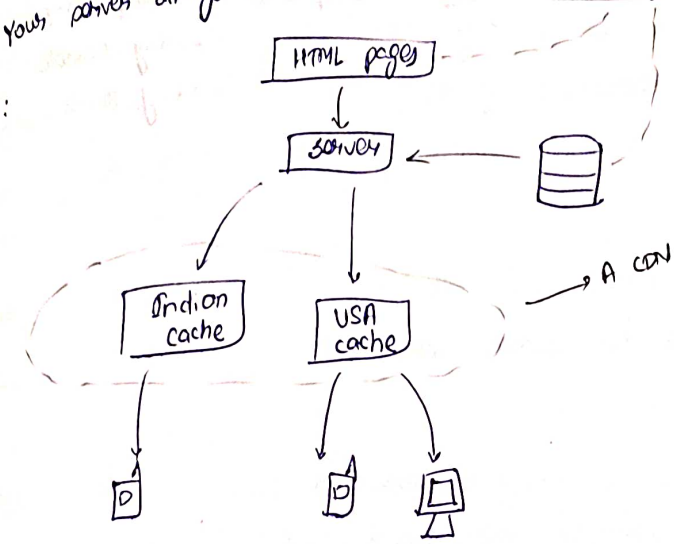
so, the request and response times are quick. The latency is less which means that the system is now fast.



Now, this customized data is being stored at relevant places a/c the location of that data

→ on any distributed system, there needs to be single source of truth.
Your servers always have access to that source of truth.

Eg:



→ A CDN takes care of this.

→ The data in these caches are copies of original data (HTML pages) of that source of truth

insert once, remove never → That's not practical bcz if you're going to change the page for a particular country, you have to remove that page.

or

Atleast, you have to version it, and then put it out of service.

→ As a small company / startup, how do you afford to build the caches in different countries?
How do you make sure that the regulations have been followed?
" " that it serves the latest content?

↓

If you have a look at certain specialized locations like Akamai, what they provide a CONTENT DELIVERY NETWORK.

CDN specializes in

① Hosting boxes close to the users.
② making sure that they follow the country's regulations
③ allow posting content in the boxes via UI.

most of the caches have the time to live also.
Eg: Sometimes, you want data to exist only for 60 sec / 30 min,
you get a nice UI to set this up

→ The most popular CDN is "cloudfront" from Amazon.

bcz cloudfront is → supercheap
→ very reliable
→ easy to use

S3 is the storage engine.

How to avoid a single point of failure in Distributed systems?

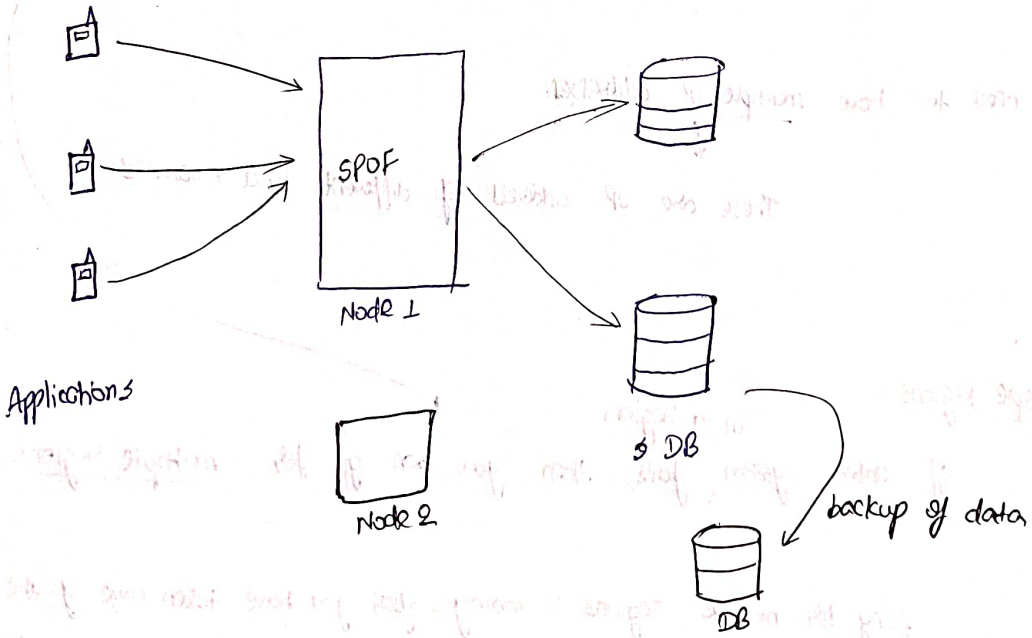
→ one way to mitigate the problem of SPOF is to use multiple instances of every component in the service. The graph of dependencies then becomes more flexible, allowing the system to resiliently switch to another service instead of failing requests.

→ Another approach is to have backups which allow a quick switchover on failure. The backups are useful in components dealing with data, like databases.

Allocating more resources, distributing the system and replication are some ways of mitigating the problem of SPOF. Hence, designs include horizontal scaling capabilities and partitioning.

→ CAP theorem doesn't allow removing SPOF if perfect consistency is required.

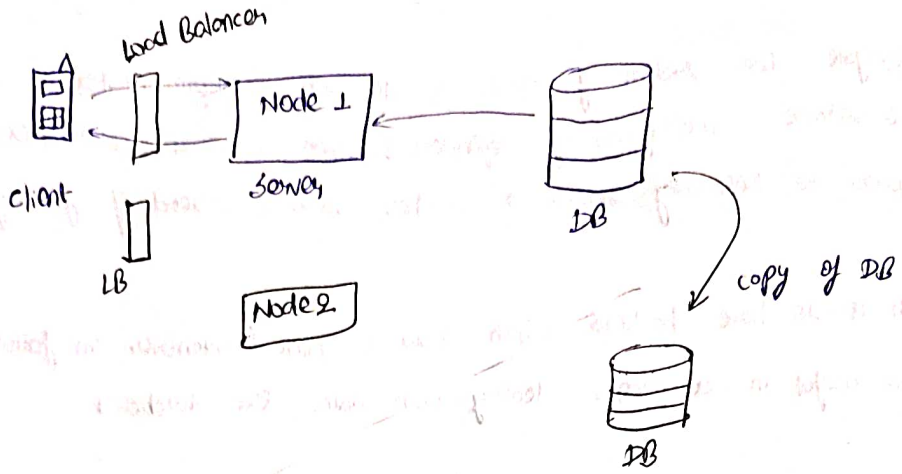
Having SPOF in your system means your architecture is not resilient.



The easiest way to mitigate SPOF is to add another node.

① More nodes

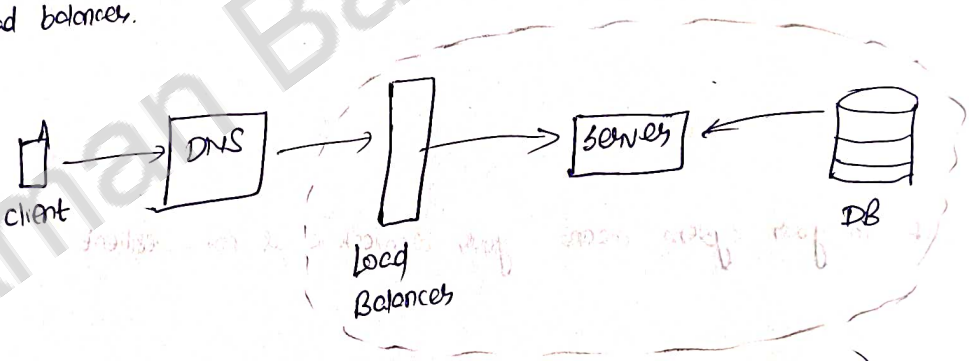
Eg:



② Master- slave Architecture

③ Multiple load balancers

The client may not know which load balancer to connect to. So, use DNS in both client and load balancer.



You need to have multiple IP addresses.

↓

These are IP address of different load balancers.

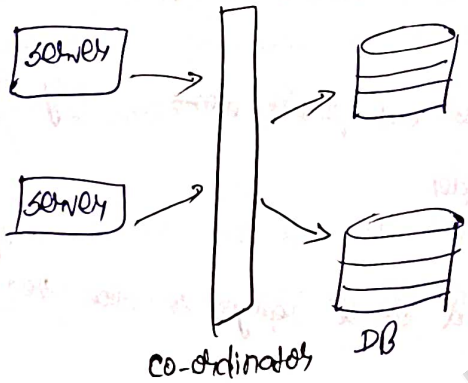
④ Multiple Regions :

in a region

If entire system fails, then you can go for multiple regions..

Going for multiple regions is mainly after you have taken care of the entire process of getting rid of single point of failure through DNS, load balancer, multiple nodes, etc.

Sometimes, when you are looking for a distributed reads or writes in the database, you can have a co-ordination bean server and dB to get rid of SPOF



Note:

Netflix is mitigates "single point of failure" really well. They do chaos monkey. It randomly goes on production and takes down one node just to make sure that your system is really resilient and distributed.

What's an Event Driven system?

or

Event Driven Architecture

Event driven systems pass and persist events. They have evolved from the Publisher - subscriber model.

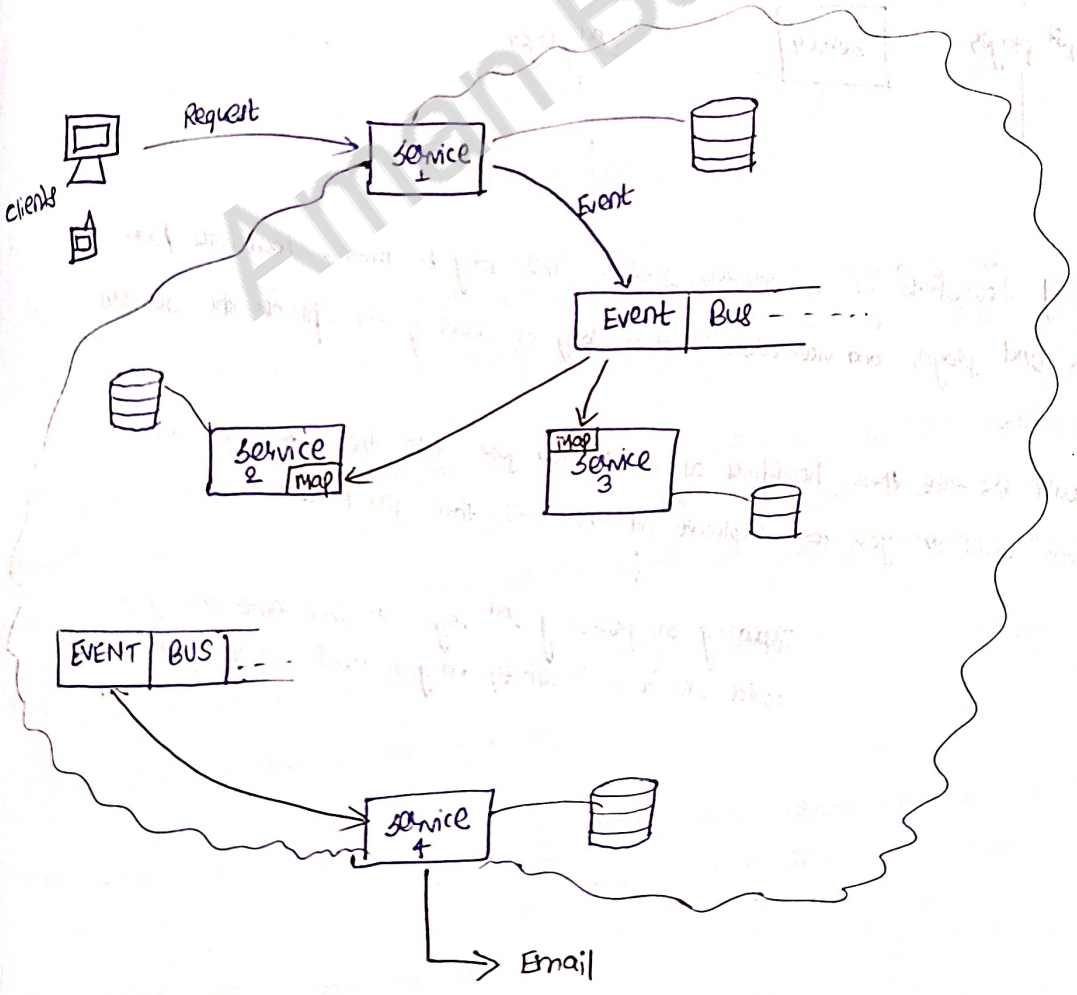
→ Events are immutable and can be replayed to allow the systems to take snapshots of its behaviour,

→ The main difference bw request-response architecture and event driven architecture is that :-

You have requests sent by clients to our gateway services. But internally, these services will never interact with each other directly. Instead, they use events to state that "Yes, something has changed".

So, if a service is sending an event to the event bus, it's just saying that 'something has happened which concerns me". And if it concerns anyone else, you can consume this event.

All subscribers and producers consume this event and see if this event is relevant or not to them. After that, they might create events themselves bcz their internal state might change.



Each of the services stores data they are getting from event bus.

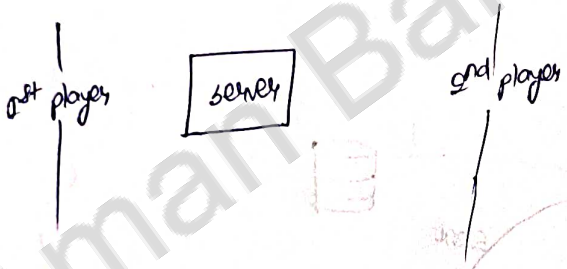
* The most popular Event-driven architecture is (Git).

Eg: Git uses events (commits) to get through its history.

① Git
② React
③ Node.js
④ Many game servers
⑤ Smalltalk

} These uses event-driven architecture.

Eg:
Use-case of Event-driven architecture :—



Take e.g of headshots in a counter strike. There may be miss in headshots from 1st player to 2nd player and vice-versa due to delay in reaching the info to the servers.

The sever could the take these headshots as events. You just take the timestamp, wherever required, you can undo or you can replicate all the ones that you had.

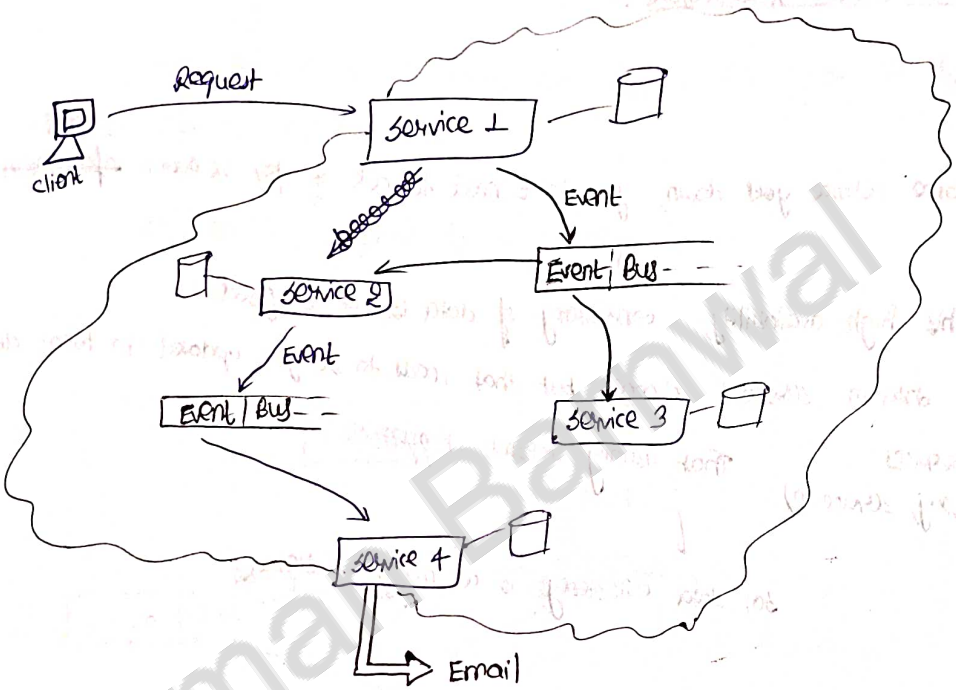
↓

Depending on position of 2nd player at that timestamp, you could call it a headshot, and you could win the game.

→ In a system design question, there are very specific areas where you might find event-driven architecture being ~~used~~ useful.

If you are not able to get this architecture working for your model quickly enough, don't pursue it.

Eg:



Let's say, service 1 gives an request and sends an event which service 2 gets. When service 2 gets this event, it actually stores it in its own local database.

Each of the services stores data they are getting from event bus. ~~The~~ It's not compulsory necessarily. The event bus can also store all the data but you usually want to pass this persistence requirements to the services which are actually consuming this event.

By then the event bus can be free ie. it can get rid of the events. It does not even need to persisting them in the right way.

The services can store these events by adding additional fields which are relevant to it
or by removing some of field " " not " " ".

But it stores them in the local db.

↓

So, even if service 1 is not working, service 2 don't have to ask relevant information everytime

(This is different from standard microservice architecture that we have where all the services store data only relevant to them).

→ Here, in event-driven architecture, you are storing data which is relevant to them but also coming from other services.

↓

so, the database is actually storing event information.

Advantages of Event-driven Architecture :

① Availability

ie. Even if some service goes down, you don't need to ask it for relevant ~~information~~ events.

But b/z of this high availability, consistency of data is not so good.

Let's say, if data in service 1 changes, but that needs to be get updated in local db of other services. (Let's say, service 2)

That usually doesn't happen.

↓

so, data consistency is a major problem.

consistency means all of your data across all your services being the same.

② Easy roll back :

If you have logging data of all of your events, you can actually move to any point in history using this event log.

so, if you know any bug which came after timestamp T, just go to that timestamp T, run event one by one and you can actually debug it even for production system.

③ Replacements :

Eg: Let's say, you have a new service S_5 which needs to replace service 2 (S_2)
All you need to do is to ask the event bus to send the new events.

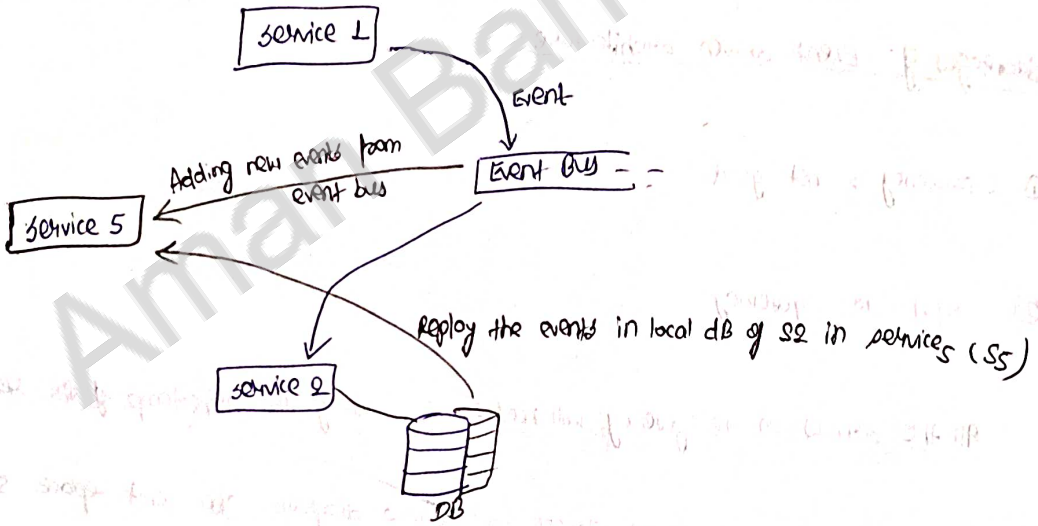
Before that, you take all events from timestamp 0 to current timestamp, replay these events in S_5.

↓

then S_5 will be consistent with S_2, & since you already accepted the new events from event bus

↓

so, it can replace S_2 very easily.



In this way, S_5 can easily replace S_2 since the data becomes consistent.

④ It gives you transactional ~~message~~ guarantee, messages when sent to any service are either at least one or at most one in this event driven architecture.

If it is atmost one, then you send the message & you don't care if it reaches or not.

Eg: A welcome email

" " atleast one, then you wanted that the message should reach definitely.

Eg: An invoice email.

↓

Here, you will be retrying event bus logic in case the service doesn't get it. Try again, try again but definitely send the email atleast once.

⑤ Stores intents :

When you are storing these events which is basically relevant data to you, you are also storing the intent of the event.
ie why did you change the data?

Bcz, let's say tomorrow if you have a new service (S_5), you can actually look at the intent of the data and make changes in your code such that you have a completely different state in your S_5.

Disadvantages of Event-driven architecture:

① consistency is not good.

② N/A to gateway

All the services on the gateway will need to be storing the timestamp of the response

S_4 is sending email to external services in previous diagram. You can't replace S_4 with any new service just in the way you replaced S_2 with S_5.

Bcz when we are sending emails, we are dependent on responses from external systems and if that response is dependent on time, then when we are trying to replace this service by replaying all these events, those events will get different responses

So, its behaviour will change without us actually wanting it to.

③ Lesser Control :

It's not giving you too much control.

ie. We can't fine tuned the handling of these events.

Eg: Whenever you are ~~saying~~ sending an event to a event bus from S_1 and then it's sending it to S_2 and S_3. We really don't have that fine-tune control of a request and a response.

→ May be, you want some services to consume your events and you want to stop some services from ever touching some events, which brings an additional layer of complexity in the event bus.

④ Compaction :

If it's storing all the events one by one, and you need to get to a particular point sometimes

There are only 3 ways :

ⓐ Replay from start

ⓑ Diff-based . In this, you take 1st event and then just store the difference.

ⓒ Use undo upto a particular point.

But some operations can or can't be undone.

Eg: Addition subtraction

Eg: Sending an email

You can squash all the events upto a particular day instead of a particular timestamp.

⑤ Hidden flow :

It's difficult to reason about the flow of the system.

Eg: Just looking at S_1 will tell you that it' publishes an event to an event bus, but that's where code stops. But you don't know, what happens with that event.

↓

For this, you need to go to subscribers of this event in the event bus.

∴ The flow of program is not easy to understand.

⑥ Migration :

Migration to some other archi is not easy.

Note :

We can think of services in event-driven architecture to be having some sort of log of their events and using a publisher subscriber model to pass and consume these events.

Event-driven Archi	Request-response archi
All the services in event-driven archi. publish events when they feel like someone needs to know something.	While in request-response archi, the services asks for something - it might be data of service.

Introduction to NOSQL Databases:

NOSQL is a popular database storage method. It keeps data as key-value pairs.

→ There are equally high no. of scenarios where NoSQL is not used.

These companies { Youtube, stackoverflow, instagram } doesn't use NOSQL databases.

↓

They use NoSQL for analytics. Their core functionalities run on RDBMS though

It's important to know when to use NoSQL db and when not to use them.

It's not true that scalability always demands NoSQL database.

SQL:

① In case of SQL database, when you are running your query, usually the pointer comes to one attribute (last_pay, id) then it has to sequentially read all the columns.

This database is also not inbuilt for denormalizing things.

↓

So, you might need a join, which is pretty expensive, considering that most of the times, you need both data.

② Even if one of the column is NULL, we still have to add a new column. (last_pay, district)

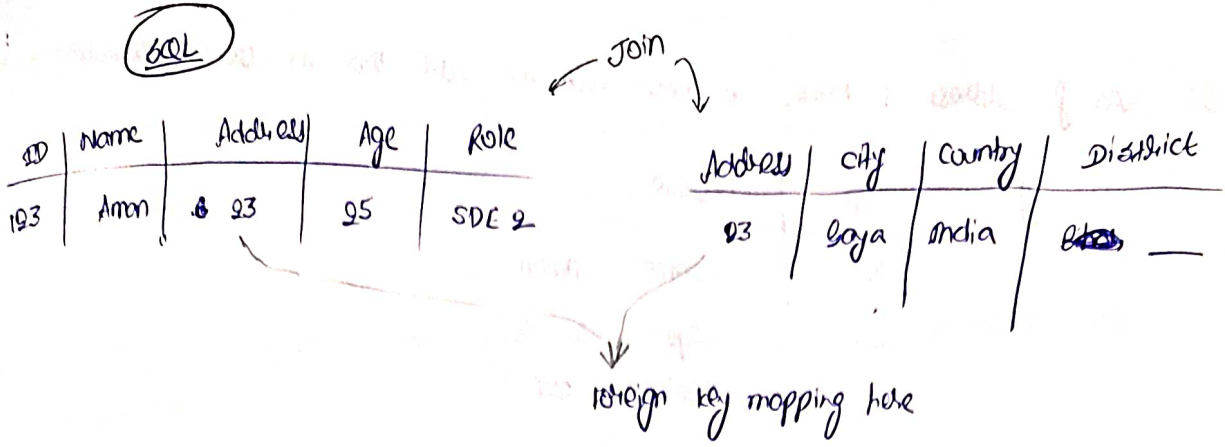
ie it is not flexible

Also, if last_pay if you need to add one more column of salary, which is a very expensive beg you need some kind of locks on the column and it's also risky to maintain consistency at this time.

③ ACID properties are guaranteed.

④ Read times are comparatively faster than NoSQL.

Eg: SQL



NoSQL

ID	value
193	{ "name": "Aman", "address": { "id": 23, "city": "Gaya", "country": "India" }, "Age": "25" "Role": "SDE-2" }

→ This is JSON data

Here, address is no longer a foreign key (compared to SQL) But it becomes another object within this object

↓

kind of JSON nesting

Advantage of NoSQL

What makes NoSQL so efficient?

→ ① When you are inserting, there is usually all the fields inserted together.

When you are pulling out any information about any user, usually you need all the info about that user, which means you need to pull whole blob.

ie. Insertion and retrieval require the whole blob of data

② → I/O cheap here in case of NoSQL database.

Bcz all the relevant data is contained together in one block, so I/O a little easier to insert and retrieve.

② Even if address is NULL, we don't need to add this in JSON structure.

Eg:

ID	value
123	"name": "Aman" "age": "25" "Role": "SDE 2"

ie. here the schema is very flexible in NoSQL database.

ie. schema is easily changeable.

③ NoSQL database have inbuilt horizontal partitioning. ie. it is built for scale.

Most of the times, they expect a lot of scale to come in.

Bcz it is allowing horizontal partitioning, so it is more focussed on availability

④ They are built for aggregations also.

i.e. They are usually expecting to be getting some important info out of that data.

Eg: Avg age,
Total salary, etc.

∴ These kind of databases are built for finding metrics and getting intelligent data.

Disadvantages of NoSQL:

① Not too many updates are inherently supported in this
ie. NoSQL is not built for too many updates.

② consistency is a problem here which means that the ACID properties are not guaranteed.

If ACID are not guaranteed, you cannot have transactions using NoSQL databases

That's why, financial systems don't use NoSQL database for their transactions.

③ NoSQL database is not read optimized.

Eg: If I asked you to find ages of all employees that we have in a company.

↓

It's going to go to the blocks and each time, it is going to read entire block, then filter out the age and then do that for every row and then return you the result.

Eg:

ID	value
123	{ "name": "Aman" "age": '25'
124	{"name": 'Ankit', "age": '23'}

while in SQL database, all you need to do is → just go to that column & then return the age of all employees

ie.

Read time for NoSQL database are comparatively slower than SQL database.

④ In NoSQL, we don't have implicit information about relations. ie. Relations are not implicit.

(Here, you can't force a constraint like foreign key constraint which in SQL says that this column 123 only exists ~~only~~ if there is corresponding column in the employee table.)

Eg: In SQL

name	address
Aman	123

employee

address	city
123	Agra

⑤ Joins are hard in NoSQL database.

↓

They are all manual here.

(But SQL database is built for joins bcz they have an inherent relations betn them)

When do we NoSQL database ?

→

It depends on if your data is a block and if you are making few updates and if you want to keep all of them together, and if there will be a lot of writes coming in.

↓

Then NoSQL database would be good to go.

→ There might be scenarios where you might want inherent redundancy or aggregation in the data.

↓

Then also NoSQL would be good to go.

→ There are still some disadvantages of NoSQL bcz of which some organization like Youtube, stackoverflow, etc still don't use NoSQL database.

SQL → also called relational db
↓
It stores info in tables.

①

② ACID properties are guaranteed.

③

④ Read time are faster than NoSQL

⑤ Relations are implicit here.

⑥ Built for joins.

⑦ DB is not flexible.

some popular SQL dB are :-

NoSQL → also called non-relational dB.

Disadvantages

① Not built for too many updates.

② ACID properties are not guaranteed
i.e. consistency is a problem here.

③ This dB is not read optimized.

④ Read time are comparatively slower than SQL

⑤ Relations are not implicit here.

⑥ Joins are hard in NoSQL

Advantages :

① So efficient

② cheap

③ Schema is very flexible in NoSQL.

④ Inbuilt horizontal partitioning.
i.e.
NoSQL is built for scale.

⑤ Built for aggregations.

⑥ Extensive control over availability.

NoSQL stores info in non-tabular form.

some popular NoSQL dB are :-
→ Apache Cassandra
→ Apache HBase
→ MongoDB
→ Amazon DynamoDB
→ CouchDB
→ Neo4J

Cassandra : → created by Facebook

cassandra is one of the most efficient and widely used NOSQL databases.

It offers → highly available service

→ No single point of failure

→ It can handle massive volume of data

→ It can effectively and efficiently handle huge amount of data across multiple servers.

→ It offers users "blazzingly fast writes" without affecting the read efficiency

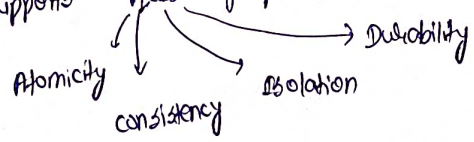
→ It is just as fast and as accurate for large volumes of data as it is for smaller volumes.

→ It is horizontal scalable ie. it allows users to simply add more hardware to accomodate additional customers and data.

→ Flexible data storage : It can handle structured / semi-structured / unstructured data, giving the users flexibility with data storage.

→ Flexible data distribution : It uses multiple data centers which allows for easy data distribution wherever or whenever required.

→ It supports Acid properties.

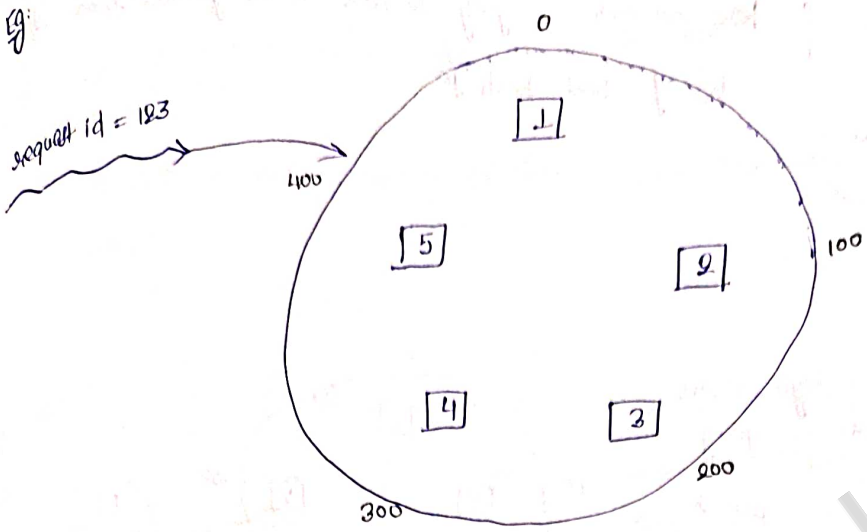


→ Cassandra stores the data in key-value pairs

but MongoDB " " " in form of document

[0-499]

Eg:



Cassandra cluster

→ Its actually very expensive to host a cassandra cluster.

→ The above cassandra cluster is having 5 nodes.

Any requests betn 0 and 100 will be falling in node 1.
" " 100 and 200 — — — 2.
— — — 200 and 300 — — — — 3.
— — — — 300 and 400 — — — — 4.
— — — — 400 and 0 — — — — 5.

Request id may not always be a numeric. It may be a UUID or a person's name, etc.

h[request_id] = n (can be any)

↓

ie. This hash is used to map this request (request_id) to a particular node in this cluster.

Eg: h[123] = 256

↓

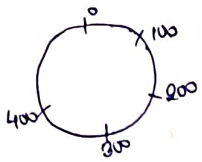
falls betn 200 and 300

* If hash fn is such that the requests are uniformly distributed, then we can assume that if there is a lot of requests which is coming in, they will be falling with equal probability on any of the nodes.

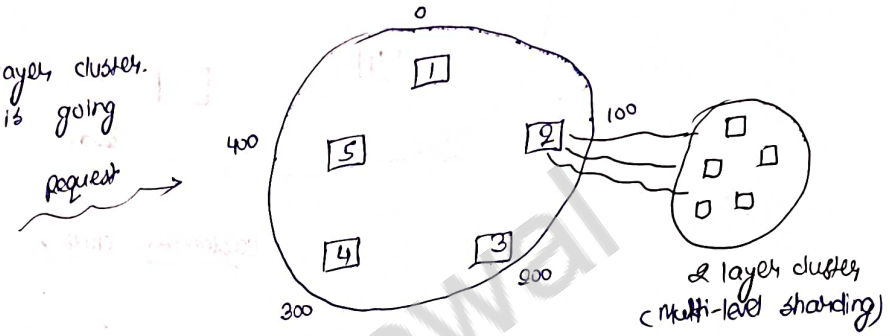
↓

All the nodes should have approximately equal distribution of the loads.

Eg: $h(<100) = 0$
$h(>100) = 1$
} here one node is going to have a lot of load than others bcz of bad hash fn.



→ You can also do a 2 layer cluster. in case a lot of requests is going on a single node.



Let's say, the request goes to 2 (let's say, it is having a lot of requests) Then it doesn't actually store it in its database.

Instead, it sends to another cluster which has 5 nodes and you run this request through a different hash fn. (let's say)

Eg: If let's say, you are user of Google maps in India. Your hash fn may be on the basis of country. Based on country ID, if you are sending at one place, it's possible that one of the country is going to have a tremendous amount of load for certain festivals (Eg. Diwali).

↓

In this case, a node (India) is having a lot of loads.

↓

So, you can go for multi-level sharding.

* If request falls on node n, the node (n+1) must have the copy so that even if node n crashes, data will not be lost.

ie. You have 2 nodes holding the data. So, the probability of you losing the data is lower.

* If let's say, your request is falling on node x based on hash fn, then it can fall into any node which is having replica data of node x.

↓

Your read queries are optimized.

* Your writes are also more guaranteed and it can also be optimized - by

Eg: If request id falls on node x. If node x misses the write, then you could write it to next node (x+1).

Note:

Cassandra gives features of

① Load Balancing

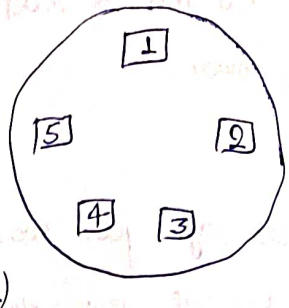
② Redundancy , Replication

↓ ↓

This gives you the speed in reading.

This gives you data guarantee.

→ one of the most important concepts when it comes to NOSQL database is the idea of Distributed consensus.



eg: let's say, there are 5 nodes
& Replication factor is 3.

↓

If a request falls on node 5, then (node 5, 1 and 2) is copying the data.

let's say, I write on node 5. concurrently, I am going to write on node 1 and 2 also. However, let's assume that node 1 and 2 are little slow. They have not got the write yet.

↓

If you are making a read operation and node 5 crashes. So, when it goes to any of its replica either 1 or 2, it returns an 404 error ie. Record not found.
(bcz writes were not completed till this time on node 1 and 2)

↓

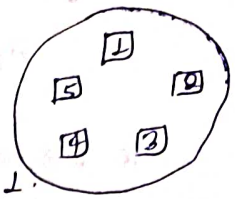
So, the user will just get confused that he made the profile and why it is not there on the database.

To avoid this issue, cassandra should be returning a database error so that the application knows that there is something wrong in the db.

↓

To do that, we need some set of distributed consensus, and one of the ways to achieve this is through ~~the~~ Quorum.

Quorum → It is a way in which multiple nodes who are related to a particular query accepts a particular value.
or vote for



Eg: Let's say, node 5 crashes the request come to node 1. Node 1 says "I don't have this data" and node 2 says "I have this data".

On that case, 1 will be picking up the data with the latest timestamp. (version id) and will return it to the user.

Let's assume that now even node 2 don't have this data. So, both nodes 1 and 2 will agree that there is no profile created and unfortunately, the user will be given a "No user profile found".

Eg: If Quorum = 2
Replication factor = 3

That means if 2 of 3 nodes accept a particular value, then we take that to be the truth.

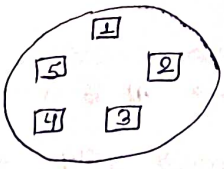
So in above case, if node 1 and 2 do not have writes replicated to them, that will result in a wrong error sent to the user.

This is really rare bcz the possibility of node 5 crashing and node 1, 2 not having the writes before they get a read operation is really rare.

This is the risk that you are willing to take when you are taking a NoSQL db and just move forward with availability instead of consistency.

Eg: If Quorum = 3

Replication factor = 3



↓

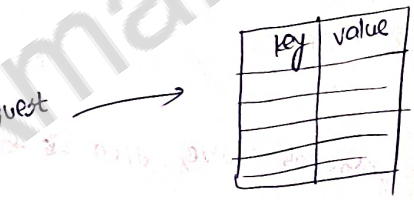
ie. 3 nodes will have to agree with the replication factor of 3.

In this case, the query will fail bcz node 5 is failed. You need 3 nodes to agree on one value. Node 1 and 2 will return a particular value but node 5 is not returning a value.

∴ The query fails.

Cassandra stands out in
- the way in which cassandra stores data and writes data.

Eg:
so, if you have a request coming into cassandra,



Assume this table to be existing in memory
bcz you need to write it somewhere

Periodically, this memory is dumped into sorted string table. (SS table)

Why is it sorted string ?
→ Bcz the key ~~string~~ is sorted. in string table.

so cassandra will be storing all these records in memory as a log file.

↓

Whenever there is a request for some write, it is going to write in a sequential fashion. In this way, you are actually storing all the data like a log.

This is efficient because all you need to do is go to the point where you have current pointer and write down the data, instead of searching anything. just

↓

This is fast

→ sorted string table is immutable ie. this data is not going to change.

key	value

→ Every time Cassandra has some data in its memory, it flushes it into a new sorted string table.

Now, bcz these requests are coming in after few days, you ~~will~~ are going to have a lot of sorted string table all over your clusters which is going to take up a lot of space.

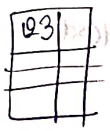
Why ?,

→ Bcz any update

(Let's say, key is 123 and 2 days later, you got an update on that key 123)

↓

some data has been changed due to update



The latest record for updated on this key is in some other sorted string table bcz that was created later on when it was flushed to SST.

Effectively, there are multiple records for same key. You can always use a timestamp for each record.

The problem is not consistency, the problem is data usage.

like you are going to use a lot of storage within duplicate keys.

Eg: If you have 10 records for the same key. Then you are using 10 times storage required.

↓

Cassandra and Elastic search both provide a feature called Compaction.

Compaction:

We take different sorted string tables and we merge them.

Time comp: O(n) operation

space comp: O(min(n, m))

where size of 2 arrays are n and m

→ We have SS Table which are immutable so they are really fast to flush into disk.

You don't need to worry about whether there are duplicate keys.

Later on, like a Batch process, you are going to be compacting these SS tables to optimize for space.

How do you get rid of deleted records?

→ You go to deleted records and cassandra place a tombstone.

You probably set a flag and tombstone will say that this record is dead.

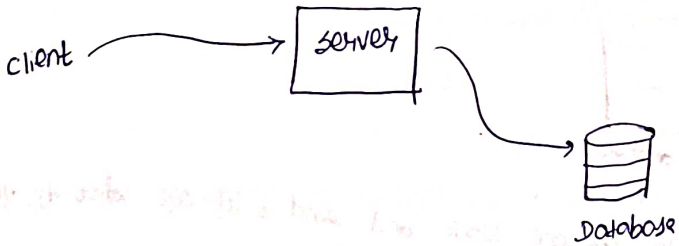
If there is read operation on 3 or 4 records and if there is a tombstone, you see tombstone on the latest timestamp, you call this record to be dead, and all 3 or 4 of them are killed.

If there is an update on the key and if you see a tombstone then you know that an update is impossible and you give an exception like record doesn't exi

therefore

How database scale writes : The power of the log

We will talk about the ways in which we can optimize writes in our database.



Here, client is sending information to the server and server has to write to the dB

→ You can think of your dB as a data structure.

To speed up queries, the traditional database uses a data structure called a B+ tree.

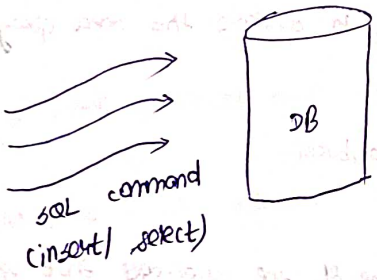
This is like Binary search tree without binary inside it.

B+ tree is preferred because it gives you good insertion and search time.

Both operations are log n.

In B+ tree, insertion → o(log n)
deletion → o(log n)

Whenever you use a SQL command of an insert or select, that maps to an insertion or a search operation in B+ tree and the underlying data structure is manipulated.



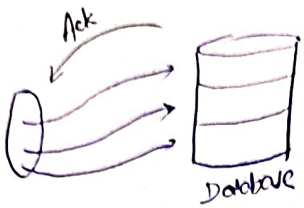
Each of these operations needs an acknowledgement from the dB that "Yes, I have successfully executed the request".

→ To scale our dB, we want to reduce the unnecessary exchange of data which are acknowledgements and headers & also we want to reduce the IO calls which will help us to free up some resources and reduce the request-response time.

Header ← Bandwidth

Per IO call ← Time & resources

→



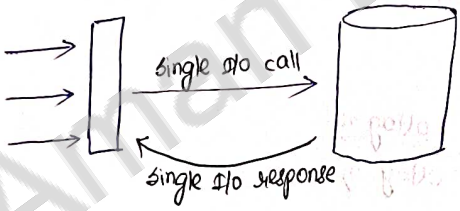
If we could condense this data into one block and send it in one shot to the database and get a one acknowledgement.

↓

that's a good idea.

That would be more efficient use of your bandwidth bcz you are not sending any extra data that you are sending around.

Also, you are going to do less work bcz there will be one IO call and one IO response.



① Condense data queries into a single query

The server has to give up some memory to condense this data query. That requires additional memory.

↓

This is one of the drawbacks.

The reason we do this is because of one of the advantages that we need lesser IO operations.

If you have a lot of write operations, what ^data structure is the fastest which can help you write operations quickly?

→ Linked List

→ All you need to do is whenever you have a write, you need to just go to the end and append something to it.

write operation in Linked list → O(1)

↓

bcz you just have to go to the end and you just add a node.

→ Log follows the philosophy of Linked List. We are going to use log to assist data, bcz it's really fast when it comes to write operations.

Disadvantages of Log:

Read operations are very slow.

ie in Log, if you have to find something, you can't jump to one point. You have to search sequentially the entire log.

→ If you can think of your database as a log, that's a huge amount of data you need to read. for every query a person will be sending.

↓

ie Fast writes.

Note:

Database with heavy write workloads need write friendly architectures.

Trees are an alternative to B+ trees, as they scale writes better.

efficient

Advantages:

→ Lesser IO operations
→ Fast writes

Disadvantages:

→ Additional memory
→ Slow reads

Log using Linked List:

→ You can't help much w.r.t additional memory bcz you need to condense the queries and keep them somewhere

→ Slow reads are something that we definitely wants to avoid.

You are not reading on the log bcz searching in linked list is O(n).
But if you could convert the data structure to either use of sorted list, it will make the searching really fast.

Instead of linked list, if you use sorted array, searching becomes O(log n)

But, you already had a B+ tree, which was giving you O(log n) search time.

So what should be take - B+ tree or Linked List ?

	B+ tree	Linked List
Write:	(log n)	great a writing O(1)
Read:	(log n)	terrible at reading O(n)

That's why, we will use (Linked List + sorted array) to get great write speeds and great read speeds

sorted array → search O(log n)

can we convert our linked list to the sorted array?

→ You can

→ we don't want to convert it in memory by that's defeats the purpose.

The whole point of having a linked list was so that you do fast insertions.

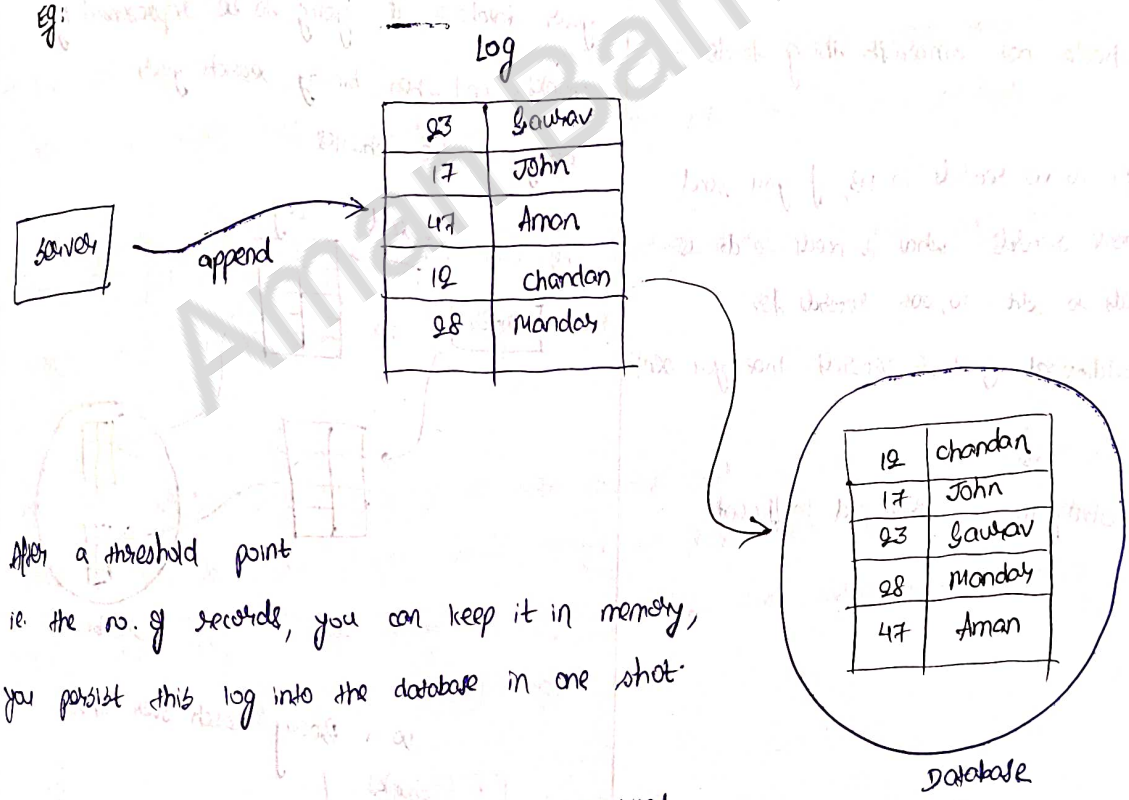
If you want sorted always, insertions are really slow.

so, you'll do the conversion somewhere in the database

↓

so, whatever information you are getting from the clients, you sort it and then you persist it so that the read operations are super fast.

Eg:



After a threshold point

ie. the no. of records, you can keep it in memory,

you persist this log into the database in one shot.

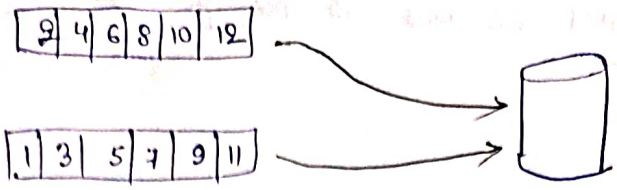
This data is going to be sorted before it is persisted.

↓

You can do here a binary search which allows you to query this data in an efficient manner.

What if after the current append, you want to append the new stuff to the dB?

Eg:



When you want to persist the new append in DB, the database can do 2 things →

It can take both records and merge them in a sorted manner.

But, that's not smartest thing to do.

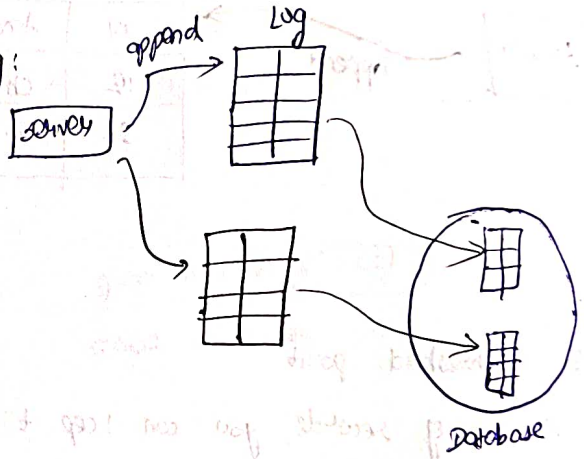
Eg:
After 10,000 records in DB, if you send 6 more records, what it needs to do is it needs to sort 10,006 records for the additional just 6 records that you sent here.

↓

O(n log n) worst and inefficient

Instead of merging always all the time, you can keep them in sorted chunks so that your database is going to be represented by chunks, and then binary search your query in the chunks

Eg:



Do a Binary search over these chunks.

↓

have only 2 chunks

So, This is better than slowing down your write operations tremendously.

Now, your write operations are reasonably fast but your read operations are slightly slower.

→ If you have N insertions,

$$\text{read time} \propto \left(\frac{n}{\text{no. of rows in a chunk}}\right)$$

↓

So, your read times are extremely slow.

Eg: Imagine the facebook. There are billions of record in the database.

Let's 1 chunk contains 6 rows.

↓

So $\frac{1 \text{ billion}}{6}$ still very very slow. So, how will you optimize this?

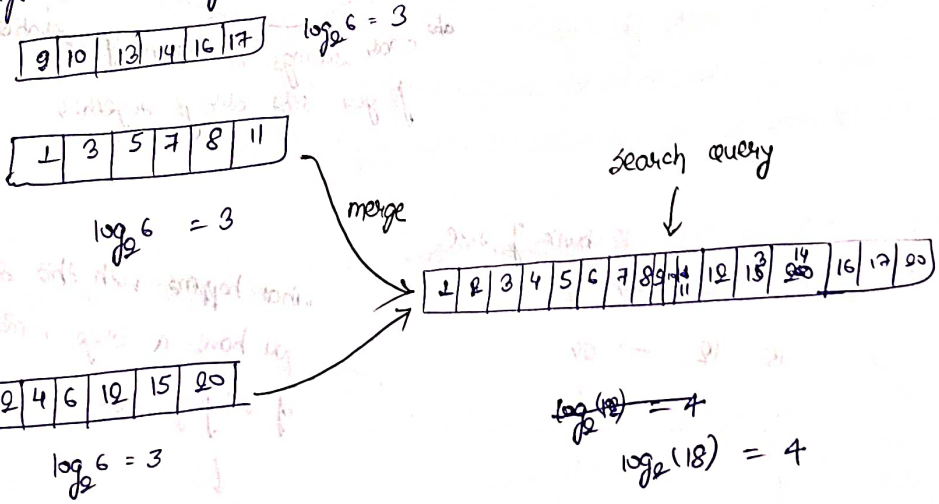
→ You can use some sort of hybrid approach which is taking some records and merging it with other records as long as the sort time is not so high.

Now, everytime when you are getting a chunk, you are deciding whether or not you should merge this chunk with the existing chunks.

↓

In the background, we are going to take chunks and we are going to merge them together to make larger sorted arrays so that when there is a search query, this large sorted array can help us reduce the overall time complexity.

Eg.



Time req to search in 1 chunk $= \log_2 6 \simeq 3$

If you have to search in all chunks for a particular record $= 3 + 3 + 3 = 9$ operations

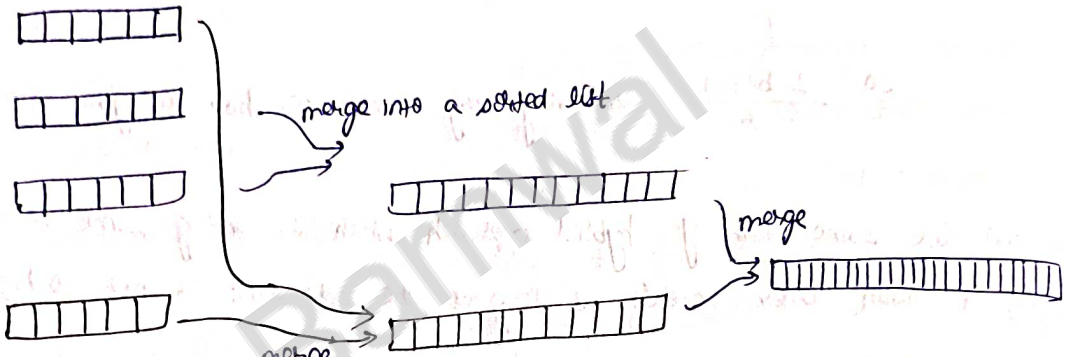
But, searching in so merged sorted array just take $\log_2(18)$ ie. 4 operations. ↓ saving

→ There is a clear saving here of the no. of operations but you need to merge sorted always and then persist it, so that has some overhead.

Let's see, what is the best approach?

When should you merge the sorted chunks?

Eg:



But, if you had them all 4 chunks unmerged,

then no. of operations = $4 * \log_2 6$

$= 4 * 3$

$= 12$ operations

$\log_2(24) = 5$

↓

if you merge all 4 into a single sorted chunk of size 24,

then no. of operation $= \log_2(24)$

$= 5$ operation

→ clear savings in the no. of operations if you sort always together.

i.e.

chunk of size

6 6 → 12

12 12 → 24

⋮ ⋮

max size n

→ what happens with this strategy is you have a large number of blocks of varying size.

↓

so, your read operation is spread across these blocks..

, And to speed up your the read operation, you can apply "Bloom filter".

Bloom Filter :

Eg: If you have a book, you have to find out whether this book has word "CAT".

If yes the book provides you the particular data structure "Bloom filter"
of 26 positions, one each for each alphabet like

If there exists any word in the book starting with letter C, then mark C block as true. ie. ⊥

For CAT → ⊥ for C block

Even for corona → ⊥ - - - (that's called a false positive)

Note:
In Bloom filter, you can have false positives, you cannot have false negative.

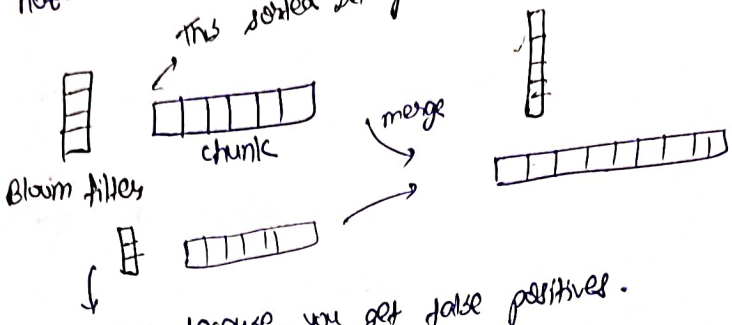
Eg: If there is no word in the book starting with C, there is no way the block C will be marked as ⊥.

You can think of here book as db and searching of word as Bloom filter query.

→ If you are looking for whether this record exists in the chunk, just create a Bloom filter on top of this chunk, which tells you that this record exists in this chunk or not.

This sorted set of records is called "sorted string table".

Eg:



This speeds up your query because you get false positives.

The rate of false positives can be managed by ↑ the no. of bits.

Instead of saying, whether there is a word with "c", I can say where there are words starting with "cA".

CAT ✓

CORONA ✗

→ Also, any time I merge two arrays, I am going to be requiring a separate bloom filter for this one merged one too.

A good idea is to create a Bloom filter of larger size. Bcz if there is a more information here, the chance of a false ⊕ve is higher.

↓

∴ I need to ↑ info in the bloom filter to reduce the chance of a false ⊕ve.

This way, when you are reducing the false ⊕ve, you are indirectly reducing the read times also.

Log-structured merged trees

Note:

You want to speed up writes for which you use a Linked List but you can't read well on Linked List.

So, in the background, you merge sorted arrays which can be binary searched to get fast read operation.

So, your writes are fast b/c you are flushing writes in a sequential fashion but your reads are also fast b/c you are reading from sorted tables.

Compaction:

compaction is taking a lot of sorted string tables and using merge process to condensing them into a single array so that we can do fast search query.

Service Discovery and Heartbeats in microservice:

Servers crash due to various reasons like hardware faults and software bugs. Service Discovery and health checks are essential for maintaining a service ecosystem's availability and reliability.

→ A heartbeat service can be used to maintain system state and help the load balancer decide where to direct requests.

Now, when a server crashes, the heartbeat service identify and restart the system immediately on the server.

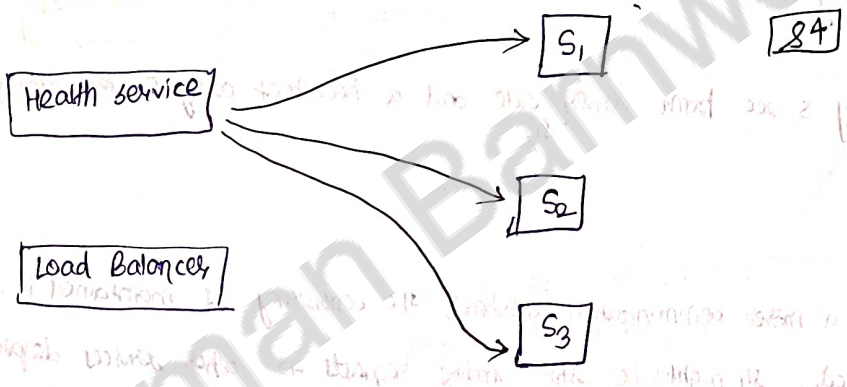
→ Service Discovery is another important part of deploying and maintaining systems. The load balancer is able to adapt request routing. Both features allow the system to report and heal issues efficiently

→ Most of the things on server side deal more with reliability and availability rather than efficiency.

A lot of people when they are starting a service, they think about how do I make this service more efficient but what you really need to do is have enough checks and balances to make sure that this service is running all the time.

When that happens, user tends to use that service more which ends up generating more revenue

Eg:



The main job of health service is to make sure that everyone is alive. and the way it does that is by talking to all these services.

Eg: Server S1 will be sent a message that "Are you alive"?
and S1 will respond with yes.

5 sec later, the health service will again S1 that "Are you alive"?
This cycle continuous.

Let's say, S1 goes down so here now S1 will not respond yes to the requests sent by health service. At this point, the health service can mark S1 as critical.

And, if S1 is missing the 2nd Health service request, then you can assume that S1 is dead.

Now, if S1 is dead, there is no point of sending requests to this server S1. And we can restart the machine (server S1) or go on some other server S4 and run the service that is running over on S1 to S4.

You actually send this information to the load balancer saying that S1 is dead.

→ This may not necessarily a new machine S4. You can run another instance on the same machine S1.

→ In some cases, where application itself is dead rather than service, the service should be telling you that "I am alive"
the health service

So, it's a 2-way heartbeat.

A heartbeat every 5 sec from server side and a heartbeat every 5 sec through (S1)
health service.

→ Although there is a more communication overhead, the consistency is maintained in the sense that if S4 is dead, S4 might be still pending requests to other services despite being dead or despite not taking new requests because it might be having cron jobs running on it.

So, when S4 does send these requests to these services S_2 and S_3 (let's say), it has stale data that's one problem. It might be manipulating the internal state of these machines.

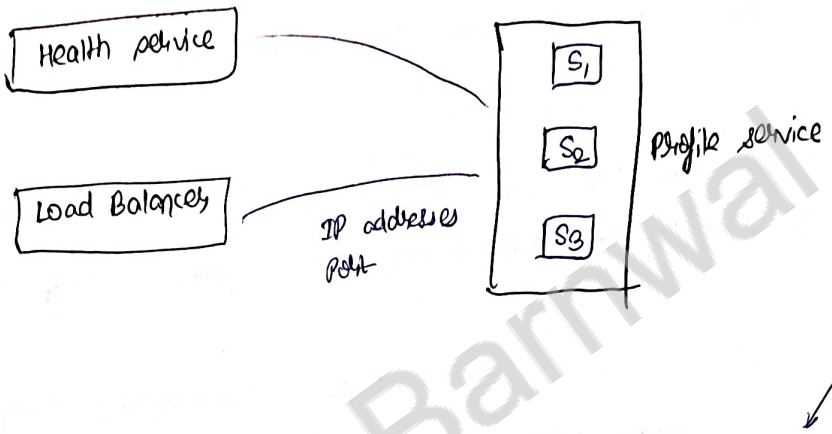
The way we can avoid this is by to like basically avoiding zombies is by having a 2-way heartbeat mechanism.

Even this does not actually solve the problem entirely but this really helps you reduce the problem to a large extent

Bcz if S4 has been marked dead by health service, then S4 will kill itself.

→ Interestingly, the health check problem is very closely tied to the service discovery problem. in the sense that if there is a new service which is coming up having these 3 boxes S_1, S_2 and S_3, all it needs to do is to tell the load balancer that "I have 3 boxes on which I am running my service (These are IP add, these are ports on which I am running my Http client)." and " " needs to persist data somewhere.

Eg:



Based on this, the Load Balancer changes the snapshot that it has, so the LB need to persist data somewhere.

This data is basically a snapshot of entire universe of the services that we have

When the profile service comes alive, you need a new snapshot (that's a new version). You need to persist that in the dB saying S_1, S_2 and S_3 have these ports and IP addresses for which the profile service is running.

Once you have this snapshot, you can then tell every other service that hey this is the snapshot. If you need to send a message to profile service on S_3 then this is IP address

In that way, what happens is that services themselves don't need to keep that info. Everytime you need info, you come and ask the load balancer.

or

you can even cache the info on load balancer.

→ Everytime there is change in snapshot, the health service can see the difference in snapshot and based on that, it can open up connection with these 3 services on their HTTP paths, always making sure that they are alive and the health of system is good.

Distributed concensus and Data Replication strategies on servers:

we talk about the master slave replication strategies for reliability and data backups. This database concept is often asked in system design interviews with discussions on consistency and availability tradeoffs.

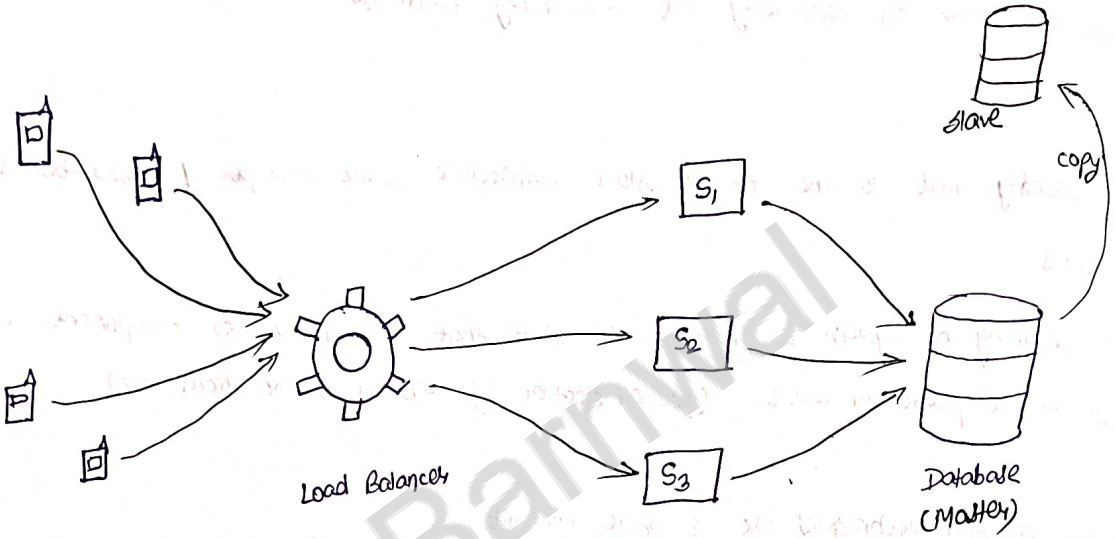
→ very closely tied to the master slave architecture is the concept of Distributed consensus.

When designing a system, we must make sure that the individual components can agree on a particular value. (Leader election, distributed transactions, etc).

some popular techniques are 2 phase commit,
Multi version concurrency control
SAGAS
& Quorum.

Master - Slave Architecture :

Eg:



4 cell phones

Here, the load balancer is redirecting the requests coming from 4 cellphones to our servers.

Here, we have a single load balancer & one db → leads to single point of failure

To avoid this, the simplest way is just to replicate the data that you have in the database.

Now, that you have this copy of your db, how are you going to pull data from original database?

2 ways

- Asynchronously
- Synchronously

Adv:

If you do it this way, then there is not too much load on the original db.

There is not somebody continuously asking for an update everytime there is an update on the original db.

Here, when there is an update on original db, it's going to send a message to its replica db that "Hey, I got an update" and you can make this change in your transaction log.

Eg: ADD 100 users → command that master got

Asynchronously

Synchronously

↓

The master gets the command and it sends it to the slave to be copied

Eg: ADD 100 USERS (for master)
ADD 100 USERS (for slave)

Pitfall:

It's going to be out of sync.

Let's say, if there is a transaction in the original dB, it may not reflect in the replica dB before the original dB crashes.

↓

In that case, you are going to have some inconsistent data.

What if one of the servers want to update this slave?

Eg: S3 wants to send this command to slave dB.

→ In this case, the slave needs a way to tell the master that there's a change that needs to be propagated to the master.

↓ 2 ways

First way is to ignore the problem or just, allow that problem to happen never

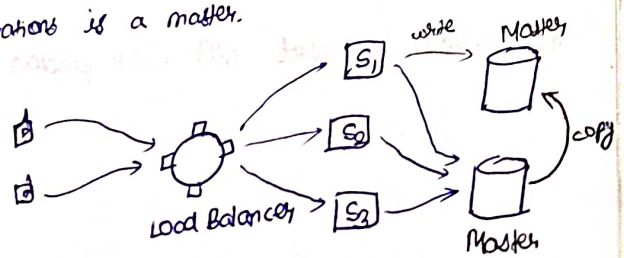
One of way to do this is to never allow write operations to come directly to slave dB from any servers.

Allow the command, and the slave will propagate this to the master.

In this case, the slave has actually taken the role of master. It's no longer a master-slave relation, but now it's a peer to peer relation, in which case both of them are masters.

Any database copy / replica which is taking write operations is a master.

Eg:



Master-Master Architecture

Master - Master Architecture :

Here, there are write operations happening on both original and its replica database, so there is some load balancing on write operations also.

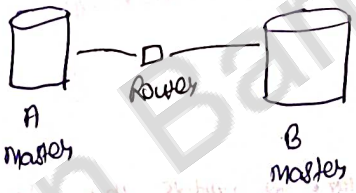
So, if any of 2 dB crashes, every request can be redirected to the other master dB.

↓

which results in a very resilient and strong system.

However, there's a problem, if the router bein the two masters fails. 8

Eg:



Here if router fails, both masters will individually assume that the other master fails so now, I am the only master who is going to take read and write operations from now

↓

A problem will arise

Eg:

Let's assume here, the router bein A and B fails.

And a user has an initial balance of Rs 120. And that user sends a message to A to deduct Rs 100 from its account and then another message to B to deduct Rs 50.

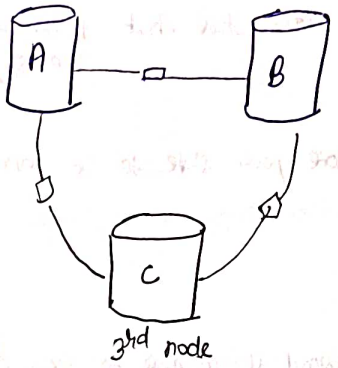
→ 120 − 100 − 50 ⇒ -ve balance

This problem is called split-brain problem.

Split-Brain problem:

Interestingly, the split-brain problem can be solved by adding a 3rd node to the original architecture

Eg:



Here, A and B has a split-brain problem.

↓

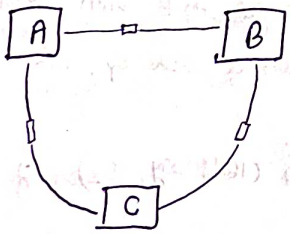
So, you add C to solve this problem.

One assumption we made here is that the chance of a node (C) crashing and the router bein other two nodes crashing is highly highly unlikely. (bein A and B here)

→ If C crashes, A and B are still be in sync bein of router bein them & when C come again, it will read from either A or B to sync data.

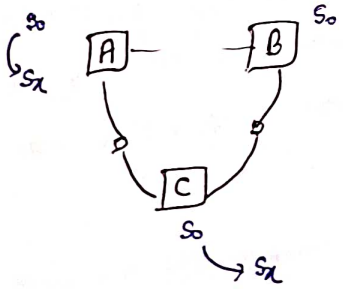
→ But when the router bein A and B goes down,

Eg:



Initially, assume state of all A, B and C are S_0.

Now, router bein A and B goes down. And there is a write operation on A due to which it's state changes to S_x. This change propagated to C.



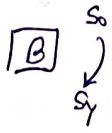
Now, again there is a write operation on B due to which its state changes to S_Y.

B now tries to propogate this state S_Y to C.

↓

But that doesn't happen bcz C says that what's previous state to B.

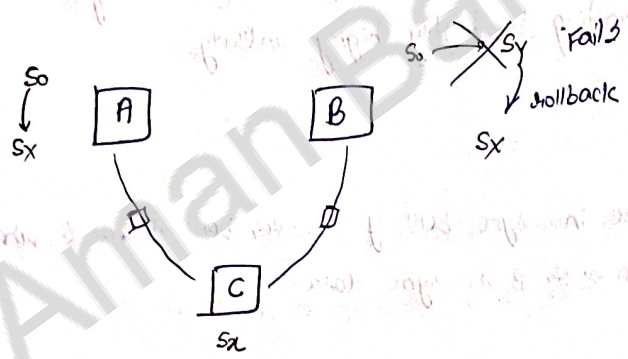
C then says that S_0 is the not the same state that I am in. (i.e. S_X)



It will say to B that you need to update your state to S_X and then you can have a successful transaction to S_Y.

↓

So, B's transaction will fail and instead of moving to S_Y, it'll rollback on the transaction and it will synch up with S_X.



Now, the user can send the request again to B with newer state S_X.

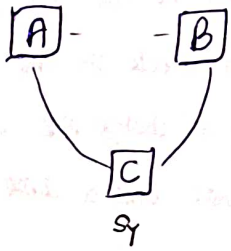
B can then run the transaction, reach a state S_Y, ask C to go there.

Now, when A gets a new transaction (let's say, S_Z), it can have some problem.

↓

Here, A can never make a transaction which doesn't have the updated state from B also bcz that is being propogated by C.

Distributed consensus: It is a way in which the multiple nodes agree on the same value.



All A, B and C are agreeing on final state Sy.

Note:

→ There are many protocols that you have for distributed consensus.

① → If you have a lot of nodes then 2pc is a very popular protocol, although it's really slow. (2-phase commits)

② → 3PC (3 phase commits)

③ → MVCC (Multi version concurrency control) → It is used by Postgres.

It keeps the multiple versions on same data

ie. if you send multiple update on the same data, it keeps multiple copies.

→ If you are okay with dirty reads then it's going to keep an older version of data and someone can go and read it.

→ But if you say that I want really nice serializable reads, then it's going to make it slow but it will make sure that data is consistent.

④ **SAGA** : used by Microsoft

Eg ① Let's say, you have a food ordering app. You makes an order, restaurant has not yet agreed on order. The bank locks the fund but restaurant tells the bank that until I get a confirmation, I'm actually not going to withdraw that money.

If transaction fails, then I'm just going to keep the money. You don't need to go through the whole transaction process and settlements at end of day.

* SAGA is really a long transaction which at any point can fail and you may need to roll it back.

Eg: (ii) If you have a phone app and a person makes a call. You need to charge the person but it might last for 30 min, so you can break it into smaller transactions of 1 min each, in which you are not going to charge the customer, but you are going to lock funds every min a little more and a little more.

At the end, you are going to decide whether call was a failure or success. Based on that, you are going to charge the customer.

Advantages of Master - Slave Architecture:

① → Nice replica of master.

② → You can scale out your read operations.

③ → You can add as many slaves as you want and just do read operations on all of them.

Eg: If data is not critical, let's say, you are adding a comment on facebook, it does not really matter if it does not reflect immediately. It reflects on the master, but the slave will take sometime to get the data.

→

How to avoid cascading failures in a Distributed system?

Thundering herd problem → This problem occurs when there are a large number of requests on the server, and this results in the server crashing due to overloading.

One of the solutions to this problem is rate limiting.

On a distributed environment, the rate limiting problem is a complex problem to solve.

→ When designing systems on the server side, we often need to predict the capacity of a server and apply limits on the number of requests it can receive per second.

This is mentioned in the service QPS document.

QPS stands for Queries per second

→ The system design discussion helps us understand how to deal with requests based on priorities, how to deal with cascading failures, how to handle a large no. of requests, vital posts or videos and their requirement.

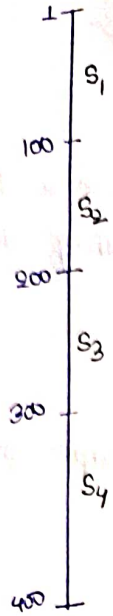
→ some other scenarios like Job scheduling on cron jobs being fired in batches is also discussed. We use batch processing and approximation to reduce server load.

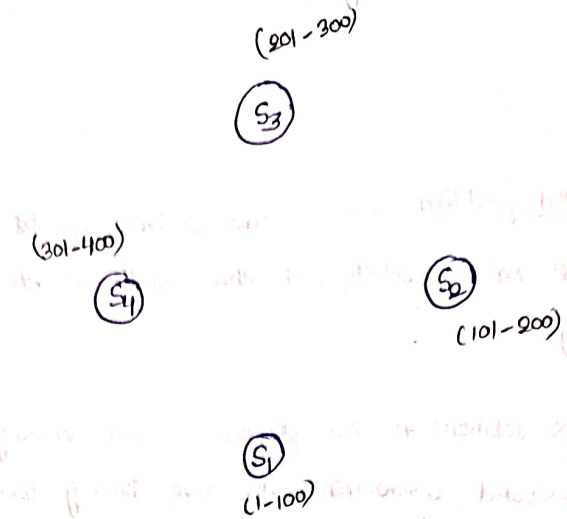
→ The last few approaches include gradual deployments and using caches to store responses for common requests. This helps us improve ~~inform~~ performance. Improving QPS and performance help us handle more requests which means more users and more money for the product.

→ caching and coupling system can also help improve performance. However, they must have time outs and appropriate cache eviction policies set while designing the system.

① Cascading Failure : SOM is (Scale limiting)

Eg:





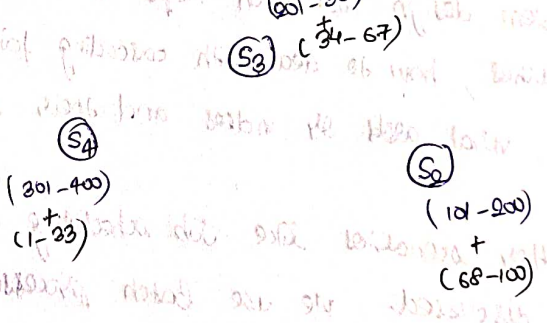
Let's say, you have 4 servers and the requests ranging from 1 to 400. So, every server here is load balanced to serve a range of 100 requests.

Now, let's assume S_1 crashes

↓

which results in additional load on remaining servers S_2, S_3 and S_4.

Let's say



Here, we are assuming that each of the remaining servers can handle the new load.

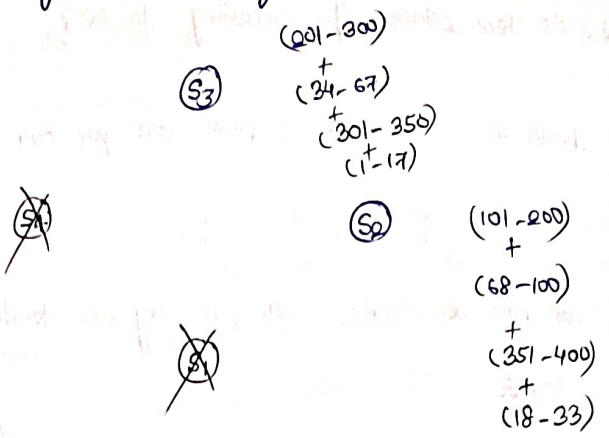
Let's assume that S_4 don't have that much compute power to handle the additional increased load due to S1 crashing

↓

which results in crashing of S_4 (Let's assume)

Now, S_1 and S_4 both are dead.

so, now the load of S_4 will go to remaining servers S_2 and S_3.



There is good chance that S_3 also crashes bcz of 200% additional load.

$\downarrow$

$\therefore$ S_3 will become dead.

$\downarrow$

Due to this, all load went to S_2, due to additional 300% of load of its original capacity, S_2 also crashes and become dead.

It results in the crashing of whole system, and so all the users are upset.

$\Downarrow$

This problem is called cascading failure problem.

It is actually a race against time. When S_1 has crashed, that small time gap that you have for bringing in the new server before S_4 take that much load and crashes.

One of the things that you could do is to have a really smart load balancer or have seamless port of new server bringing in.

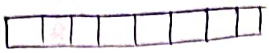
One workaround for this problem is to just stop serving requests of to all users having request id $\perp$ to 100.

If you see that some of the services can't take in more load, then it is better to be available to some users than to be available to none of the users.

Note: We often use message queues for rate limiting requests.

But what's the real solution of cascading failure?

→ What we should do is to take a queue and put our requests in this queue



Every server can have a request server and they can decide on answering or not answering a request.

So, what I will do is give these servers a particular compute capacity (Here, 1 unit of compute capacity means that it can handle 1 request per second).

Eg:

This is the compute capacity.

Let's say, now S_1 crashes.

↓

If it went to S_4 then it sees that the maxm requests it can handle per sec is ~~400~~ 300.

So this queue is going to keep expanding till it hits 300.

↓

301st request on S_4 fails ie we are going to ignore that request

So when we return a failed response to the client, atleast this server S_4 will not get overloaded.

Also, the client is now aware that "ok, this request has failed, maybe after 5 min I should try again". The user which made this 301st requests (which failed) has a bad user experience.

↓

But serving some users is better than serving no users.

→ There are some kind of errors that you can send the client.

Temporary

↓

It means you should try in sometime, may be there's some internal server issue going on, or may be db is too slow, or may be there's so much load.

Try after sometime and client can display messages accordingly.

Permanent

↓

It means that there is some serious mistake in the request that you sent and there is a logical error.

→ But the general idea is to limit the no. of requests you can take on the server side so that you can handle the load till the scaling bit comes in, i.e. till you can bring in the new service.

② Viral ~~Black Friday~~ → soln is prescaling

③ Predictable load increase: → soln is auto prescaling

(Eg: Black Friday, Diwali, etc)

The second problem is - If you go viral or if there is some sort of event (Republic Day, Black Friday), which increases the load on the servers at times, then that would be the issue.

↓

Prescale your area system beforehand so that there would be no failure of system in case of additional loads.

If you are not very sure about the no. of servers you will need during event, you could autoscale.

↓

It is being provided by cloud services.

ⓐ Prescale
ⓑ Autoscale
ⓒ Rate limiting

How about if you go viral?

→ If you go viral, you can just fallback to the old way of rate limiting.

ie. If you do rate limiting, you will be stopping the max no. of users that you can actually serve.

Going viral is not something that you predict so pre-scaling is not really a solution for this.

Autoscaling and rate limiting both can be the soln if you go viral

③ Bulk Job scheduling: → soln is batch processing

The 3rd problem is a real server side problem.

We often write cron jobs which run at some point of time

↓

used for scheduling the tasks to run on the server

Eg: Imagine a cron job which is sending email notifications to all users wishing them a Happy New Year on 1st Jan.

→ one way is to send all the email together, when clock becomes 00:00.

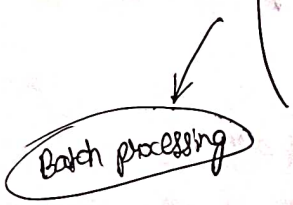
↓

That's a bad idea when the no. of users is very very large (Let's say, 1 million)

The way you avoid this is by breaking the job into smaller pieces.

such that out of 1 million users, 1st 1000 users will get email in 1st min

and — — — — — 2nd min



and so on.

→ If your users will not get the auto-generated email notifications of 1st new year at 00:00, they don't really care.

↓

So, it's something that you can do.

ie. Batch processing is something that you can do here.

④ ~~When someone~~ Popular post : → soln is (Approximate statistics)

When a post becomes really popular,

Eg: Let's say, a user like ~~Sidhu~~ T series post something on Youtube then you need to send it to all of their subscribers.

↓

If you do it in naive way, the same issue of Job scheduling will come here too. (ie. Too many users and a very small time 🙂).

So, you could do the Batch processing over here.

↓

Eg: sending to users in chunks of 1000.

But, Youtube does it really smartly by adding (Jitter).

If you have a lot of subscribers, the notifications will go to them in a Batch-processing way but if they start hitting the video page, then the video content is come to Youtube but there are a lot of data which doesn't matter.

Eg: No. of views
-- likes
-- comments.

→ If you have a very popular user (let's say, T-Series) posting the video, the no. of views are going to change dramatically so what you could do is to faithfully display them or you could do it in a smart way

ie. in 1st hour, we get 1000 views.

2nd hour, ---- 1000 × 1.5

= (1500) views

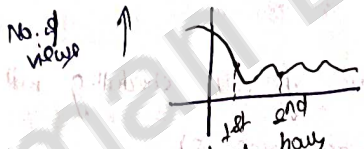
(but let's say the reality is (1700) views)

mismatch in no of views

but we don't really care.

Bcz this is metadata (which is something that is not core to the video).

→ Youtube could figure out how the views changes over time and then figures out the no. of views at a specific time from the graph



Youtube must be much much smarter than this, but I am just giving the general idea of approximation.

Instead of showing people the truth, do the approximation and save a lot of load on your server.

→ Potentially, this could save a lot of database queries that you are making to get the metadata of a post.

Soln is (approximate statistics).

Soln :

Problems :

Problems	Soln
① cascading failure	① Rate Limiting
② Going viral	② Prescale
③ Predictable load increase	③ Autoscale
④ Bulk Job scheduling	④ Batch processing
⑤ Popular posts	⑤ Approximative statistics

Apart from these soln, there are some good practices in the server side to avoid a Thundering Herd.

⑥ Caching :

ie. If you are getting a lot of common requests, the response is going to be same and you can just cache the responses for those requests.

→ These are key-value pairs and this is going to save a lot of queries that you will be making on the database.

↓

which will improve the performance of the system and also you can then handle more loads.

⑦ Gradual Deployments :

most of the issues that people get on the server side is when they are deploying their service.

Site reliability Engineers who are fighting deployments and the developers who want to deploy more, bcz they want to get more features out

↳ ie. they want to stop the deployments as much as they can because that makes the system most stable.

↓

so, its an interesting tug of war ☺

→ Here, you don't deploy all the servers together.

You deploy the first 10, then have a look at what's going on - and then you deploy the next 10 and so on and so forth.

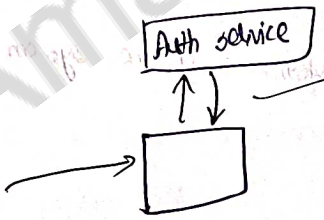
↓

This won't be possible in certain scenarios when there is breaking change.

⑧ Coupling :

To improve performance sometimes, you need to store data.
(i.e. very similar to caching)

Eg:
Let's assume, you have a service. For every request that it gets ask the authentication service to authenticate the user first to authenticate this request and then serves this request.



Let's say, this network call is too much for you (may be it's an external service)

⇓

You could cache the user's username and token or password whatever you like.

Instead of talking to authentication service and verifying whether it's true or not, what we could do is → You can see that if username, password worked once in the past one hour, then may be the password does not change.

& we are going to assume that this user is authenticated to call this service.

→ It's good in a way that it is not querying the authentication service all the time which results in reducing the load on authentication service, improving performance and user experience.

→ Except that it is a financial system and the password has changed and there is a person who has hacked into his account and then he is using this password then you are in big trouble

↓

So, you should only couple systems which is actually keeping data in the cache.

~~sensitive~~ sensitive data in the cache should be avoided.

→ However, if you have some data like profile picture, you can keep that in cache for 1 hour or 2 hours.

↓

ie. You want to take this case-by-case and understand keeping some data for an external service in your own service whether that's a good idea or not.

↓

That will improve performance which in turn help you handle more requests, so the problem of Thundering herd will be slightly mitigated.

Containers and Virtualization in cloud computing:

Virtualization of hardware and compute resources is a key value idea when operating on the cloud.

→ containers allows us to scale applications faster and easier, while avoiding large initial investments and reducing maintainance costs.

More importantly, containers allow DevOps team to run programs on heterogenous hardware with small startup time and better configuration capabilities than virtual machines.

→ Initially what used to happen is when you used to write code, you would have DevOps team or a Finance team, which asks you which box would you want to buy for this code so, you would do capacity planning, which is a bunch of guesses like this is the compute I need, this is the kind of memory and storage I'll be needing for the application.

What if your business scale?

→ You don't want to buy a small computer, bcz that investment doesn't make sense.

You want to buy a large computer or a reasonably safe computer, so that you don't need to do this again and again.

But the problem is with this is the initial hardware investment is very large.

→ If you want it to horizontally scale just on the basis of hardware, you would need to buy as many computers as possible

→ So, one of the approaches that the organisation took was to let their employee to use that single computer.

But if you have multiple people using the same resource, there is going to be some contention.

↓

Shared compute

so, you want to isolate resource usage as much as possible.

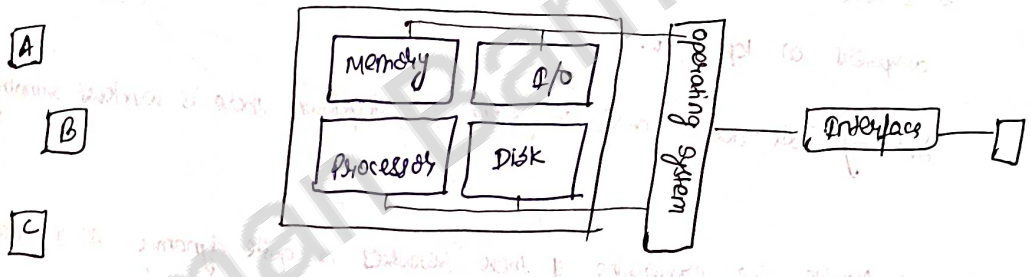
The qn is "Can we do this for application?"

Let's say, you have a program which requires a no. of resources (memory, IO, processing, and disk).

Obviously, the more programs you have, the more resources you need.

But you also want to do some sort of boundary management which means that A does not interfere with B's memory, and C doesn't interfere with A's I/O.

And that responsibility is going to be taken by operating system



When you write a program, you talk to OS saying that "Through your interface, I would like to book x amount of memory, y amount of IO and so on".

Similarly, you can have multiple programs which are going to be taking up slices of resources that the operating systems can provide and remaining is going to be unused.

The problem with this is you can still have the same problems of shared compute.

So, we prefer to have a very strong boundary which is provided by virtual machines.

↓

It is like an OS in itself.

So, its an OS running on top of OS.

→ You can interact with this fake world that you have without concerning yourself of what other programs are running in the same hardware

→ You're now mainly concerned with virtual machine that you have been assigned which has been given a set of resources

→ The basic idea of cloud ~~computer~~ computing is huge companies like Amazon and Google have a lot of hardware. What they can do is take all the spare hardware and rent it out to small businesses.

↓

which will save big investment of maintenance and upfront cost of buying a computer.

ie. The rent cost of placing and maintaining the computer is taken care of.

→ The other good thing is my code doesn't need to be platform dependent.

Eg: I can take a Windows computer and run Linux on top of it.

so I have a 64 GB windows computer, I can run 10/10 day, 4 to 5 Linux computers on top of it.

And, my code doesn't need to know that deepdown there is windows running.

→ This is very flexible. The provisioning of these resources is quite dynamic. All I need to do is shut down my virtual machine and restart a new machine.

I don't need to go to the shop and buy a different computer.

① App Isolation

② Platform Independence

③ cost saving

Eg: Java became popular bcz you no longer had to take OS considerations. You could push that onto the ops team. You just needed the .class files, and you could run the program on any operating systems.

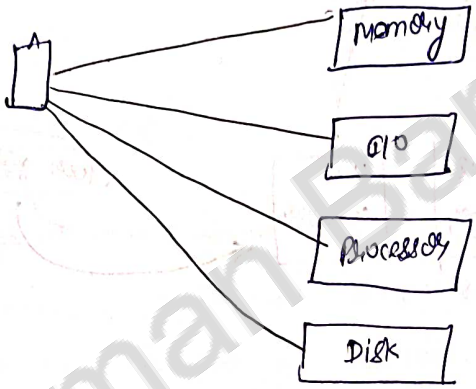
→ People could go to these cloud providers and ask them for a virtual machine and they would get it very cheap.

But one of the problems with the virtual machines is that when you are running a program, you don't want to start the entire virtual machine.

It's like booting up your computer. It takes some time (a few sec).

And then the idea came up that " You just need processing power, memory, disk and I/O.

I just need my memory fix. & I don't care where it comes from.



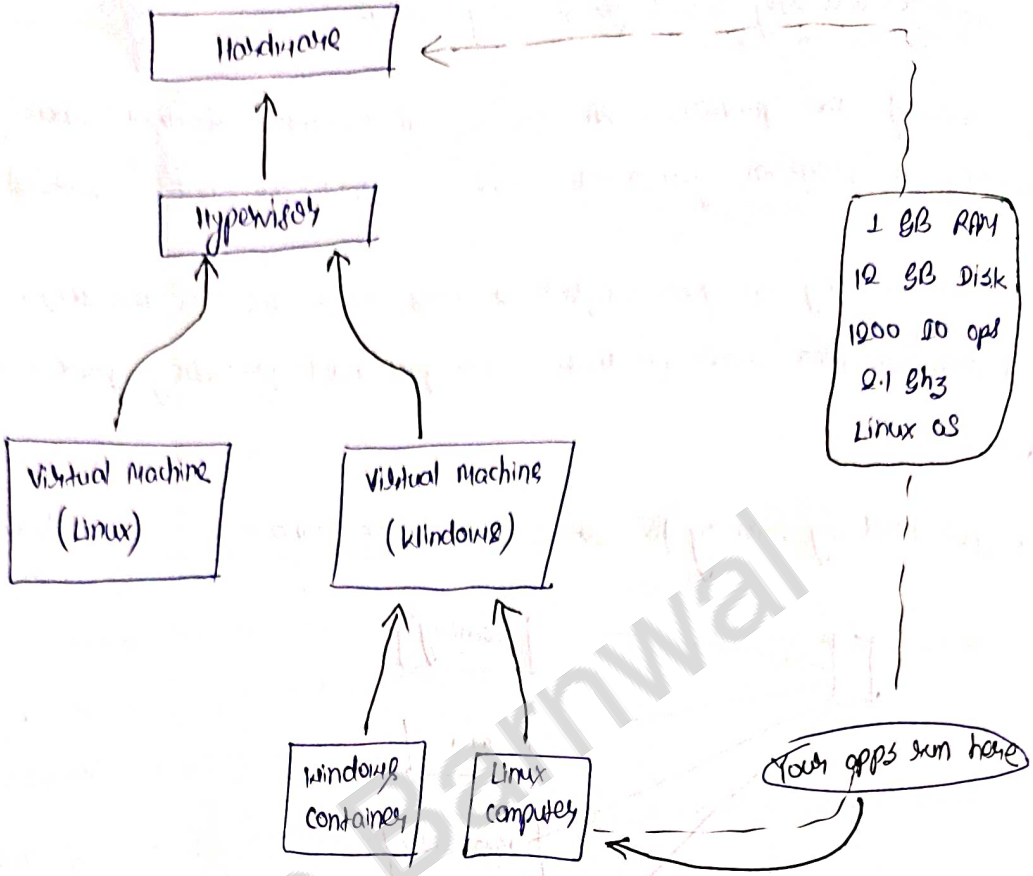
Taking all this into consideration, there came something like Lightweight OSs.
↓
containers

containers :

containers are effectively a form of virtualization. They let you do the app isolation.

You are not going to be having all features of as isolation, but you don't (virtual machines)
really need that.

And the benefit is you just need to tear down and build this lightweight containers. So, that's faster when it comes to boot times.



→ Interestingly, this process of building and tearing down is called mounting and unmounting a disk.

→ In a container, you specify the os and your disc requirements. And then the mounting ~~system~~ process figures out the underlying file system, which you will be using through interface of the container.

ie. Requirements are defined by 'images', which are mounted on containers.

Docker:

→ Technologies like Docker did the same thing that Java did for programming language.

It's a way for us to move all these considerations back to the developer.

The developer can actually specify that these are the resources I need. This is the OS I will be running on.

And you don't need to worry about the hardware so much. Docker will handle that. It will create an interface which will be interacting with the hardware of the virtual machine, whatever be the case.

It will create a container on top of it and let your program interact with the interface.

Disadvantage of containers :

① As compared to bare-metal apps, it is considered slow.

② There is also possible firewall issues.
Although it's much lesser in containers than a virtual machine.

③ The overhead of container management is not really worth it for simple applications.

(If your app is very simple and you don't want cloud, run it directly over the OS).

Data consistency and Tradeoffs in Distributed systems:

consistency

→ You have heard this word in CAP Theorem & ACID properties

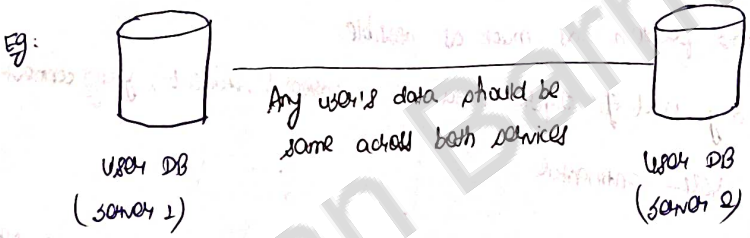
CAP Theorem: Consistency, Availability, Partition tolerance

ACID properties: Atomicity, consistency, Isolation, Durability

What is consistency?

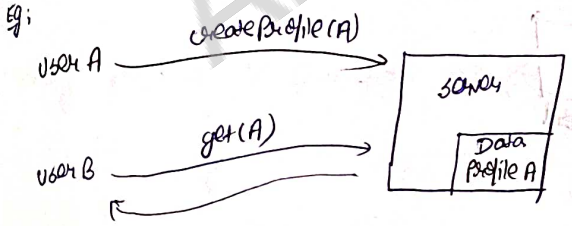
→ If you have multiple copies of data, those two data should match each other.

Eg:



Eg:

Let's say, user A creates a profile in the server.



of Facebook.

Problems:

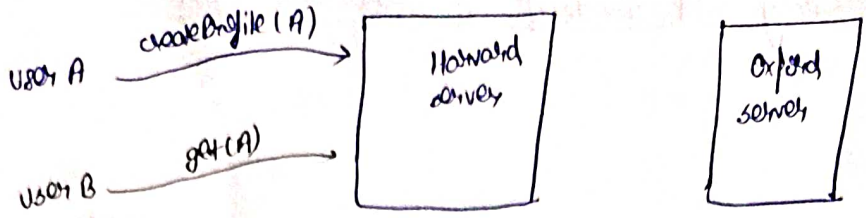
① single point of failure

② cost of vertical scaling is going to be high.
(ie there is a vertical scaling limit)

③ Latency is high.

We probably need to add more servers

Let's have 2 servers → one in Harvard
→ other at Oxford



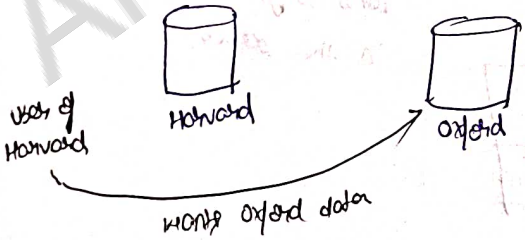
There is no any communication betn these 2 servers. They just had data copies relevant to the students here.

So, there is no data sharing betn the servers.
(here, universities)

Let's say, we try to avoid this problem as much as possible.

We say that if you need any kind of oxford data as a Harvard student, you connect to oxford server which is close- continental.

You connect to it. We are not going to keep any data belonging to oxford in any other region



But the older problem of
- single point of failure is still there.
  (if oxford server goes down)
- High latency (going from US to UK)

→ (High cost of vertical scaling) problem is solved by spreading out data according to where it belongs.

But we still have problems with high latency and single point of failure.

→ To reduce latency, we can cache some of the information (i.e. some of the information in the Harvard servers).
Oxford

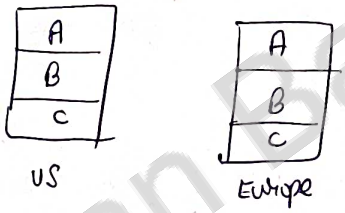
You fetch the data from Oxford once, you cache it on Harvard servers and then you keep giving the response back

↓

It will reduce latency but we will not get rid of latency. There is still a chance that you are going to have a cache miss.
(i.e. if profile didn't find in cache, we still need to go a long way to Oxford servers).

→ If you have multiple places where student data have been stored.

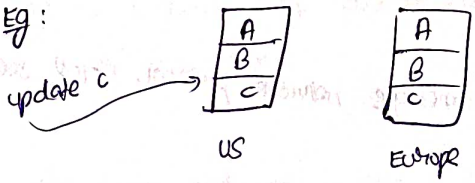
Eg:



i.e. No single of failure anymore bcz of multiple copies of data.

Latency problems are also fixed bcz you have quick responses

Eg:



The moment you say "update c", once it is updated in all copies of data, it actually sends back that response and users are happy.

↓

i.e. Latency is low.

How do we ensure consistency in multiple copies of data when you do an update in one copy?

→ When you update on one copy, how do you propagate this update to other copy.

You can do it ~~manual~~ manually.

If you have very infrequent updates in a product, you can make the decision by yourself.

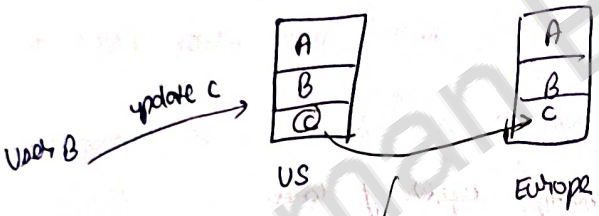
Eg: This often happens in Bank accounts

You actually make a change to the ledger. You just note down and eventually you make that system consistent by spreading information all around the world

They can notified that will going to be take a few days to reflect in your bank statement in worst case

→ On a realtime response system, you can't afford that.

Eg: If you have something like Facebook, you can't afford that



We want to propate this change / update in C quickly.

We can use some sort of electronic message network protocol. TCP seems like a good idea.

Why TCP?

→ Bcz TCP is reliable delivery.

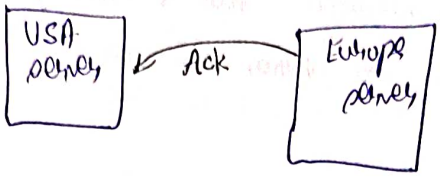
How do TCP work in our case?

→ When you send an update on C to C++ on US server, it needs to propogate from USA server to Europe server.

When Europe gets this message (C → C++) internally in their database, and then update it.

what happens if the message transmission fails?
if there's a network issue or may be the Europe server is down

→ You will never get to know unless you get an acknowledgement.
ie. the European server should send back " " .



What happens if you don't get an acknowledgement?

→ You can keep trying as US server to make this update to Europe server till it succeeds.

How many times do you retry?
↓
Infinite
which means that every 5 sec, you are going to be retrying.
or
you can just retry 5 times & give up or you can afford these two data to be inconsistent.

This actually bring us a fundamental problem with Distributed systems, which depend on acknowledgement for consistency.

The idea is that when you send an update to one of these servers then it has 2 options :

Either it locally commits that update ie. makes that change

or it wait for its peer to give acknowledgement back.

→ If it make the change locally then it's already committed and it can't depend on the peer to actually make the commit to.
bcz the message might fail bcz of unreliable network.

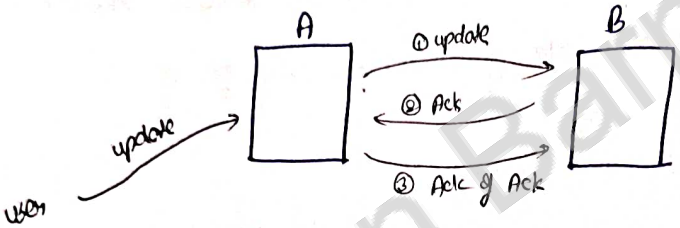
Let's assume that this is an unreliable network and B may not be able to make that commit.

If you want for both to make the commit. Let's say, if B makes the commit then B has no idea whether the acknowledgement is ever going to reach you

You can use here special acknowledgment which is Ack of a Ack then when A makes a commit then it has no idea whether this special acknowledgment is ever going to reach B.

↓

The problem is called "Two General problem".



infinite process

You can never be sure if you are making a commit based on the hope that the acknowledgement actually goes through.

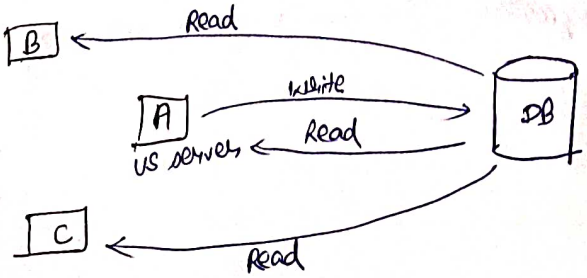
So, we can't have this for our distributed systems.

If you are hoping for consistency, what can we do?

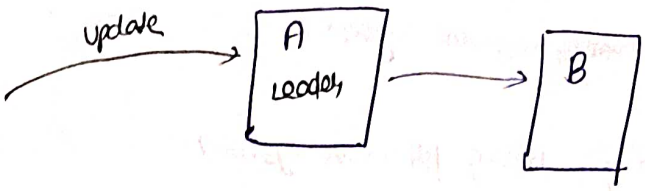
→ we can give them a special role ie, leader role.

Eg:

Let's make us server a leader. It sees the only person who can commit ie. actually do writes on a database



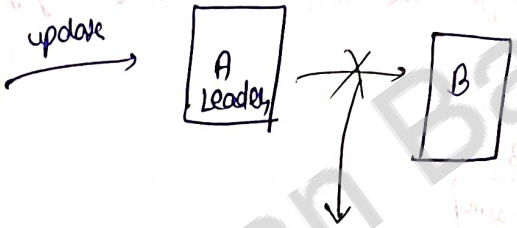
If you send an update, you can't send it to B. It can send only to A.



Now, B gets regular updates from A bcz A is the only place which can write info on a dB and B is the place where you can read info from.

↓

so here B is a follower of A and two read operation on both A and B will be going to give you the same data.



If this data doesn't goes through, what do you do ?

→ If you read the data without change from B, you have inconsistency. i.e. you will have dirty reads.

If you wait for the update to come through before you give a response from B, at that time, the system is actually not available.

↓

If you hit the system, you are not getting a response. It's as good as system is dead.

↓

That's the fundamental problem with the Distributed systems

① If you have pure consistency, then your system is not available.

② If you have some sort of availability, then your system have to suffer through the inconsistency.

That's the reason, why consistency is such a big deal.

ie. Hey can you have a totally consistent system?

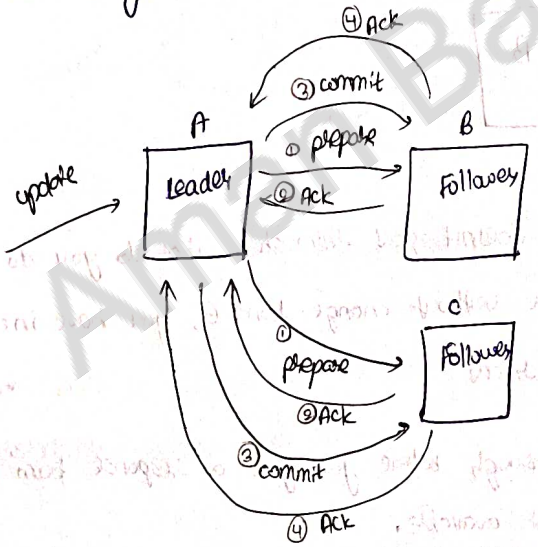
or

How much consistency are you looking for in the system?

2 phase commit protocol: (2PC protocol)

Here, the idea is you have a leader now and you have multiple followers (lots of followers).

Eg:



→ If you want to ensure that a message or the event is pushed into multiple systems, then you can go for something as strict as 2-phase commit

→ The general idea here is if you are a leader and get an update, you send a prepare request to your followers. Your followers will give you the acknowledgement that "Hey, I got the prepare request".

And when you get these acknowledgements, you ask your followers to commit. So, this is effectively committing a transaction.

Once you ask them to commit, I'm assuming that they are giving you the acknowledgement and the system works perfectly.

→ However, there is a lot of edge cases in this.

If you send a prepare statement, what actually happens ?

→ Let's say, you have a bunch of statements in this update. So, in dB what's going to happen is you will begin the transaction and write down these statements but there is commit missing here.

When you commit, you actually tell the dB to reflect these changes to everyone who is trying to get this data.

1st phase is prepare

When you send Ack, you say that I'm done with executing those statements and I'm ready for commit.

→ If the leader does not get all the acknowledgement back, it assumes that some of the system which it was sending prepare request to has failed.

There might be a timeout or you might give a -ve response also.

Whatever the case, the leader fails the transaction.

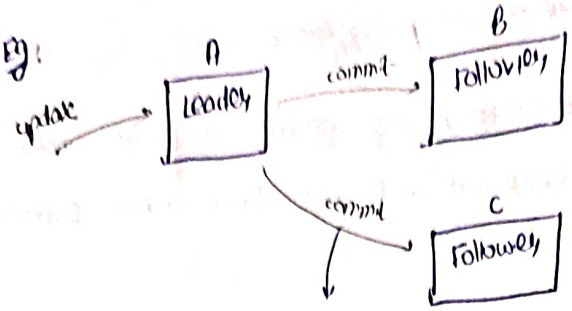
↓

So, you get a rollback

A follower could keep a time limit. i.e. if in 5 or 6 min, the leader message should have reached. If it doesn't, you just do a rollback and the system is consistent.

Let's assume one of these commits actually fail, what happens then?

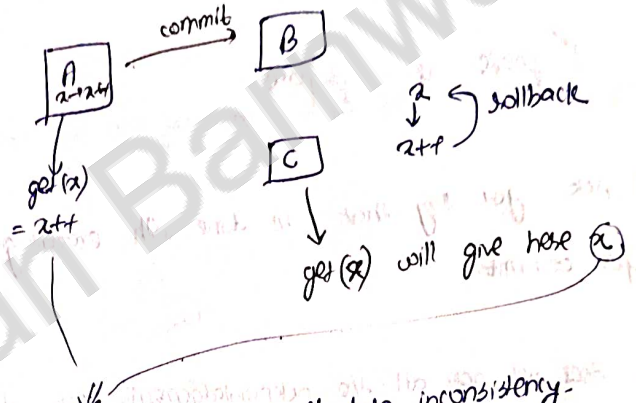
Eg:



Let's say, this commit failed.

So, after a certain time, C will assume that the transaction has failed and it's going to roll things back

Eg: update x → x+1



We are in trouble bcz of data inconsistency.

The problem is rollback. You can't rollback by yourself based on time.

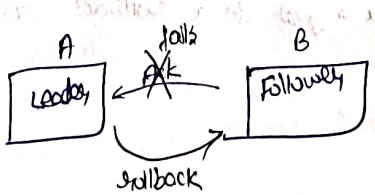
→ You have to wait for the master or leader to actually send messages of rollback.

If someone sends a Ove response or if there is a timeout, the leader actually tells them to rollback the transaction, if ack has failed to reach.

This statement of rollback is not going to be done by followers themselves.

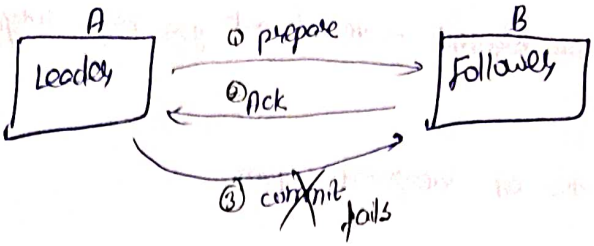
They are effectively brain dead.

i.e.



→ If your ack goes well and if after that, your commit fails. What happens?

Eg:



You do a retry. You keep retrying commit.

So, it might be that the followers actually get the commit message, but it's not able to send you an acknowledgement bcz of which you keep on retrying.

↓

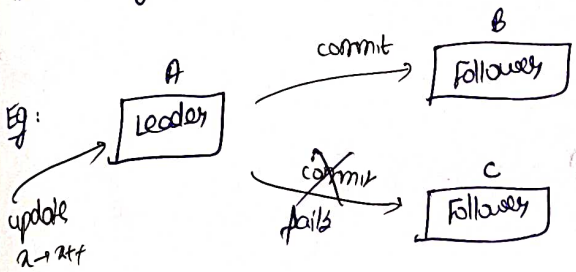
That's OK.

For same transaction ID, if you get another commit message → That's okay.

→ As long as ack keeps failing, the leader keeps retrying and the commit messages keeps going through.

Once a commit message is actually acknowledged, the leader knows is is acked ie. it knows that the data it's showing is consistent with the data that you are showing.

Eg:



Bcz of rollback here, you are going to get inconsistent data.

What you can do is take the lock and block all the read operations here.

If you take a lock on A also, you block the read and write operations

So, if anyone does a get request on A, they get nothing. They are waiting

The problem is that your system is not available.

But it is consistent.

Now, your system is totally consistent;

When there is a change in x, this change is also get reflected to all of its followers.

Bcz you wait for the acknowledgement in the end and you keep retrying with no side effects.

Transaction id makes this an idempotent system.

↓

This is a good idea if you need absolute consistency.

Eg: In financial systems

But the cost is that your system is down. Any get request is failing bcz of locking.

⇓

The general idea is to see what happens when you have a perfectly consistent distributed system.

The costs are usually high not just for the availability. There is also performance which is another factor here.

In certain cases the cost of system ↑.

→ Instead, you might want to go for "Eventual consistency"

Detecting Anomalies using Isolation Trees : Practical Machine Learning

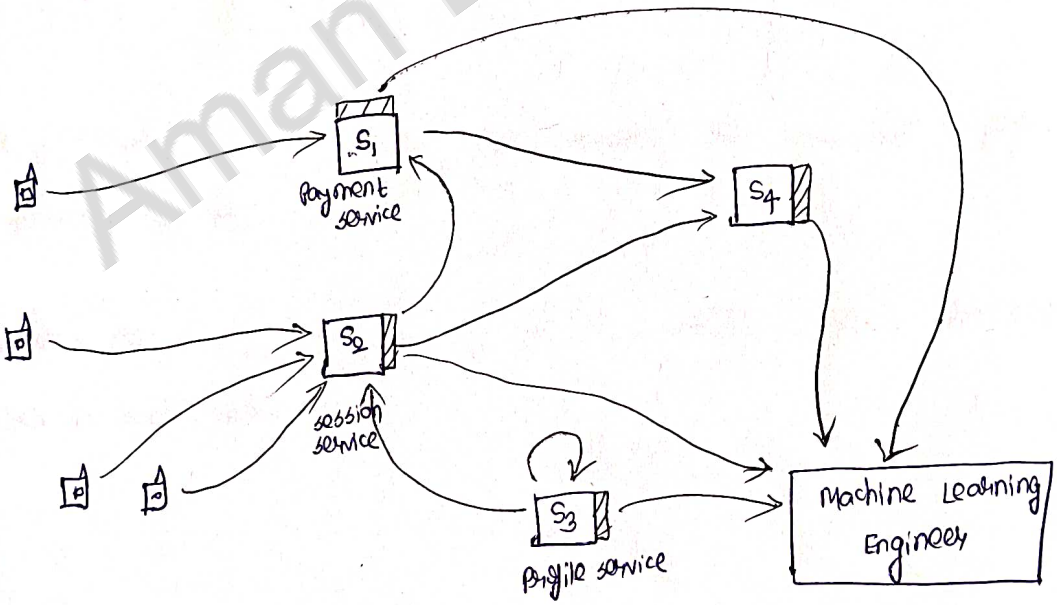
Production alerts are an important way in which engineers monitor the health of their services. The alerts are fired when important service metrics behave irregularly.

An e.g would be a sudden spike in the no.of errors or a crash in no.of running processes.

→ Here, we design a system to allow important server metrics to be published, analyzed and acted upon. The metrics are sent using sidecars in a service mesh.

The metrics are then capped and run through ML algorithms like isolation trees to search for anomalies.

Eg:



Today, we will talk about Anomaly Detection on the server side.

What that means is we will be managing the health of the system, trying to figure out if at any point of time if the system is in trouble.

The way we do that is by having all these servers publish the metrics which are important to us.

Eg metrics:

1. No. of processes
2. Request queue size
3. No. of requests
4. Error rate

Instead of the server figuring out whether it is good or bad, what it does is that it publishes these metrics to a particular engine. The purpose of this engine is to make sure that the health of the system is good by detecting anomalies.

At any point of time if you see an anomaly, what you should be doing is flagging that and sending an alert to a system engineer.

One of the keypoint of anomaly detection on the server side is to allow false positives a lot more than allowing false -ve.

The cost of a engineer investigating a false alert is much cheaper than the cost of a real alert being missed out by the engine.

If we are managing the infrastructure here, the 1st thing we need to do is to allow some sort of uniformness and standard way in which all of these services can publish their metrics to this engine.

One way you can achieve this is an infrastructure change by implementing a service mesh, which involve a sidecar.

A sidecar is a microservice architecture that handles a lot of common functionalities.

Eg: Routing, Parsing, etc

Eg: If S2 is sending message to S4, it figures out IP address of the box of S4 where it needs to send the message in a particular way.

S3 to S1 is in the same way. So, you can actually code that logic in sidecar.

→ Similarly, all of these services are going to send messages to each other probably over HTTP, or over some protocol which you have written over TCP.

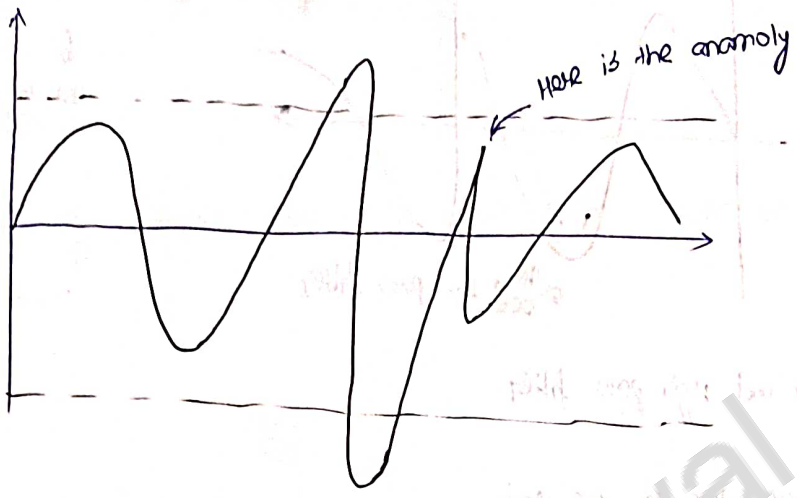
You can just parse the message in the same way on all services but you don't want to write that code in all microservices. (Don't Repeat Yourself)

↓

So, you can shift that onto a library and that library is part of big library called a sidecar.

→ If the sidecar exists, then it's going to be taking these metrics from every application and publishing that to this engine without any dependency on any team to actually get into this.

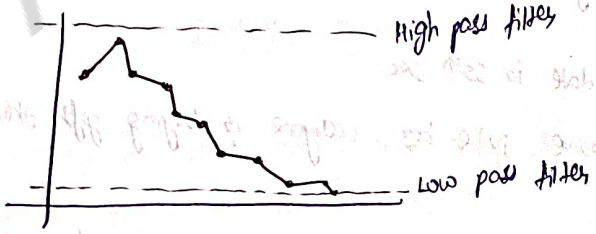
Now, we are going to find anomalies in the below time-series data.



→ Here are some extremes that you want to top off ie the limits. If we can say anything beyond these limits will be the anamoly.

This is very prone to errors. One thing it doesn't take into consideration is overall trend of a graph.

Eg:



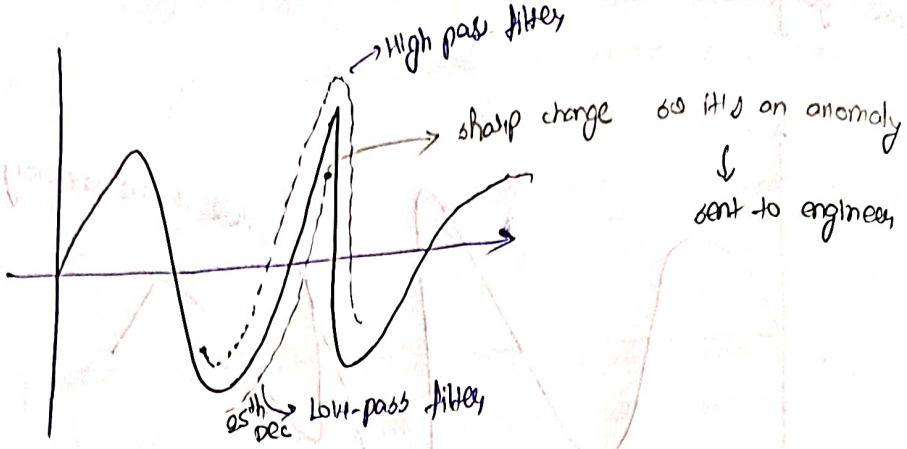
If things are going down and this is happening for months. And when you have a low-pass filter, its going to hit that low-pass filter and you immediately think its an anomaly.

But, this has been going on for months

↓

To avoid that kind of problem, we can keep a Dynamic low-pass and High pass filter

Eg:



Dynamic Low and High pass filter

Here, this sticks close to the graph.

→ If there is a very sharp change in the metric, then it's going to be flagged as an anomaly.

↓

This can be sent to the engineer who is looking into it.

The only drawback here is that you are not taken the time into consideration.

Eg: Let's say, date is 25th Dec.

So, there is massive spike bcz everyone is buying gift through your website.

Once it's over, there is a crash and may be there is new year after that

↓

What you can do is " You can take previous year data and see that the change was dramatic during 25th Dec, so this is not an anomaly. It's normal.

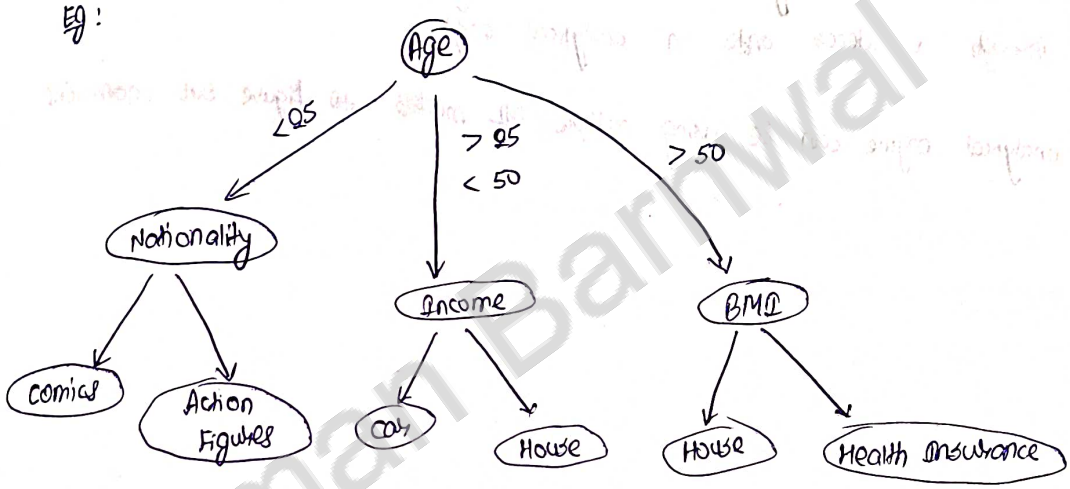
→ There are a lot of algorithms which find anomalies in a time-series data.

One of the algorithm is **Isolation Tree.**

Isolation Tree:

It is a basically a decision tree. It makes decisions based on the inputs that you have.

Eg:



Isolation Tree

Here, we are partitioning the data based on certain features. It is trying to make decisions based on the partitioning data into pieces.

With this idea, we can try to partition the data such that in the minm no. of cuts, we are ~~going~~ able to get something which is an outlier.

↓

That's what an anomaly is.

Eg: If your age < 10 → You probably buy nothing.

But if you end up buying a house, I am assuming that no. of partitions required on the data that set you apart is going to be really really low.

↓

That also means that you are an anomaly.

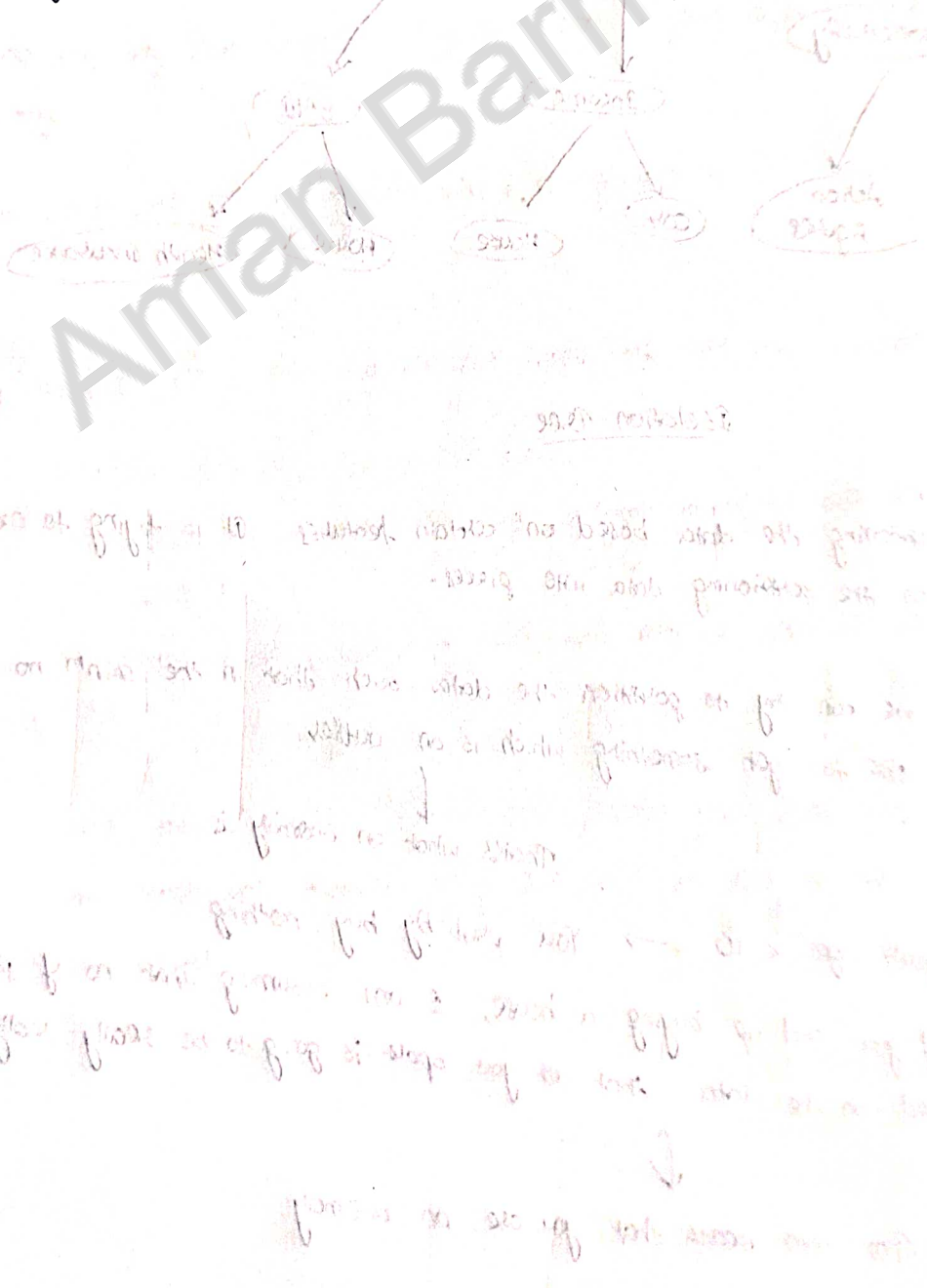
→ If you run a time-series data through the isolation tree, you should be able to find such anomalies using this kind of pattern.

Note:

→ We are more forgiving for false, alerts rather than false ove.

→ We want some uniformity in the sampling rate, so all those metrics are sent through a sidecar onto an analytical engine.

This analytical engine can be using multiple ML models to figure out anomalies



Moving from Monoliths to Microservices:

Today, we are talking about how to move from Monoliths to microservices.

Monoliths:

They are rather large code bases, which contains all the logic required for you to run your application anywhere you like.

Monolith = single code repository

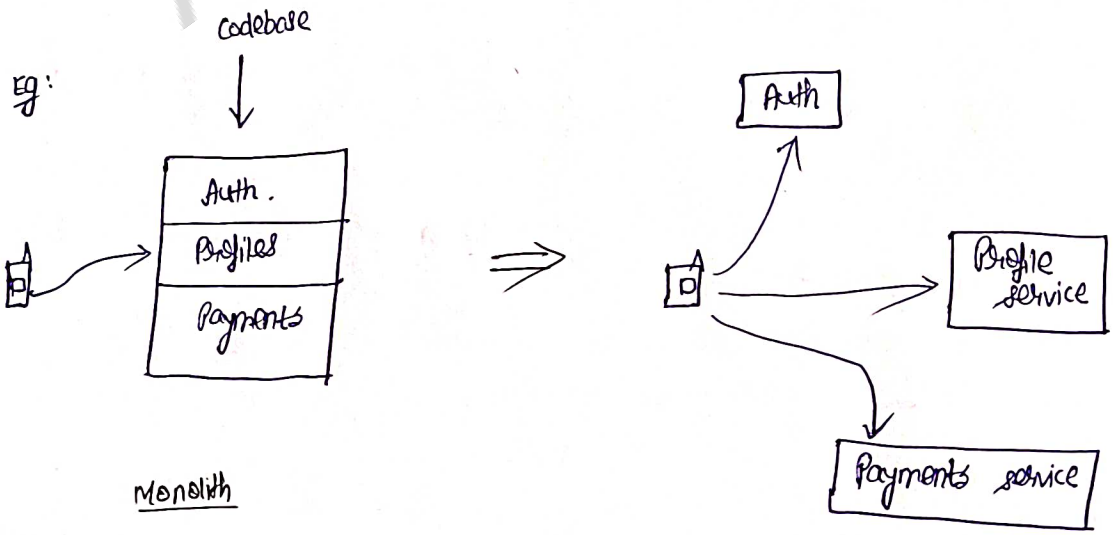
All of the relevant code in this project is kept in a single repository.

Microservices:

→ While in microservice architecture, you have cellphones which connect to different services based on what the requirement is.

Here, the code has been broken into pieces and converted into services themselves.

Eg:



Monolith

Microservice Architecture

→ ~~There are~~

Advantages of Microservices over monoliths :

① If you need to make any feature change (Let's say, your payment now also needs to accept Paypal payments).

↓

All you need to do is just go to Payment service and make the change.

ie. There is a good separation of concerns.

② Coding in microservices is sometimes easier.

Bcz all you need to worry about when you are making a feature change is just the code in that particular microservice.

so, your expectations are well-defined (in the payment service, here).

Your responses are well-defined and that's all, you need to care about.

Expectations = API + SLA contracts

Your assumption is that the engineers in rest of the services will respect the contracts that you are making with them.

Engineering these services is actually easier.

③ The 3rd critical advantage is here is that you can make deployments much easier with microservices.

Eg: When there is a change in authenticate service, you can just deploy the authentication service separately.

There might be a service which is not critical.

Eg: You have some ranking service which ranks your restaurants.

So, if it stops for 1 sec, that doesn't make much difference. So, the deployments are easier.

Monoliths also have certain advantages.

Note:

The time when you move to a microservice architecture is not when you have a lot of scale (ie not when all of your users are coming in).

But you move to a microservice architecture when your team has scaled.

→ When you have a large team and all of them want to move about individually, you have different product teams working on different products.

You want deployment to be smooth. That's why, you move from monoliths to microservices.

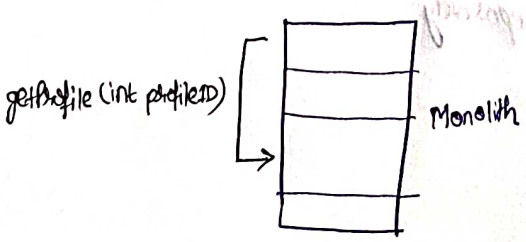
→ For small teams, monoliths are good.

For a large team, microservice architecture is probably the way to go.

(Small teams benefit from consolidated code)

→ Because there is more clear communication happening in the monolith (module A to B), you are doing a single fn call.

The parameters that you are passing into that fn are explicit. They are clear.



so, when you are making that fn call, if you make a wrong fn call, the compile is going to say no.

while in a microservice, what could happen is 'ID is earlier int'. Now, you ran out of integers, so you made it a string.

i.e. getProfile((int) profileID) — ①

This is called breaking change.

getProfile((string) profileID) — ②

so, all the other services who connect to service ① are going to use the profile with ID as an int. Bcz they are not aware that ID has turned into a string, which means all communication with this service will break.

Earlier you have something and now you have something else which is not compatible. That's a breaking change.

→ so, in microservices, it's much easier to have breaking change, which is bad but in monolith, bcz the code and expectations are in one place, it's much harder to have these kind of problems.

Advantages of Monoliths:

① Duplication is lesser

② Good for small teams

③ Contracts are in one place

↓

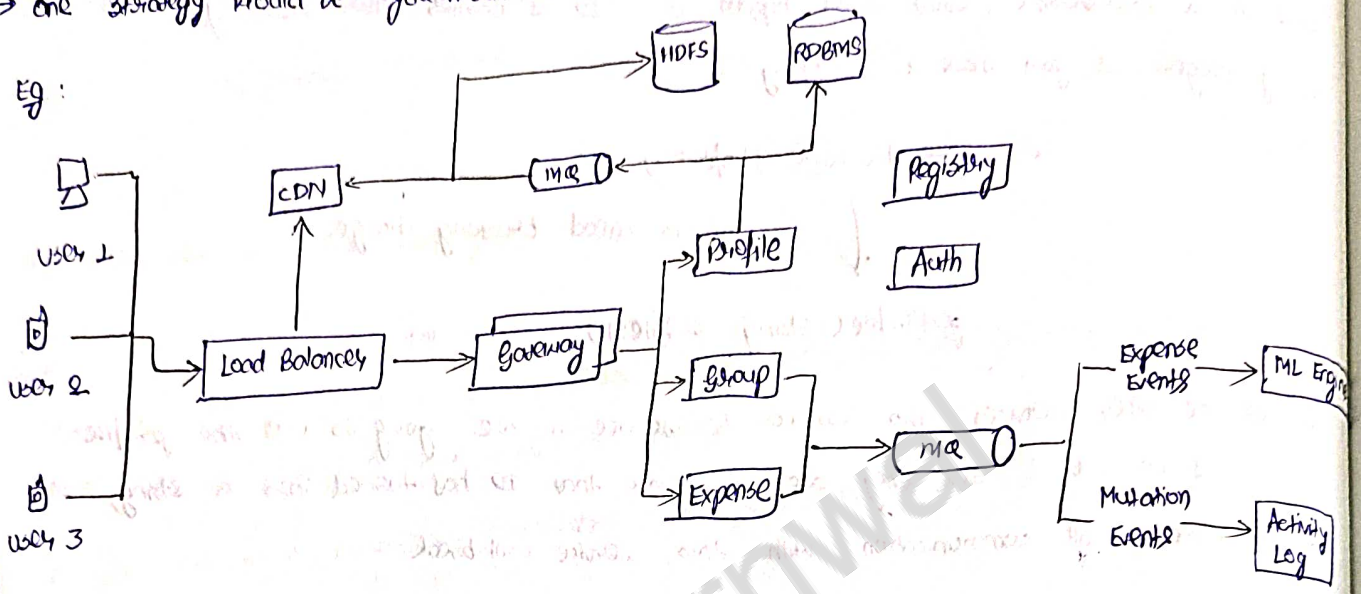
These are service contracts.

Steps for migration from Monoliths to Microservices :

①

→ one strategy would be you made this microservice architecture,

Eg :



let's say, some part of users are being redirected here, if they have no issues ie. (10 %)

these services are working perfectly along with their databases then you can make the switch from monoliths to microservices.

→ But the problem with this is that the engineering challenge is huge bcz you need to write down all these microservices. Also, make sure that the databases are correctly configured.

Then you have to redirect these users. So, the effort for doing this is massive.

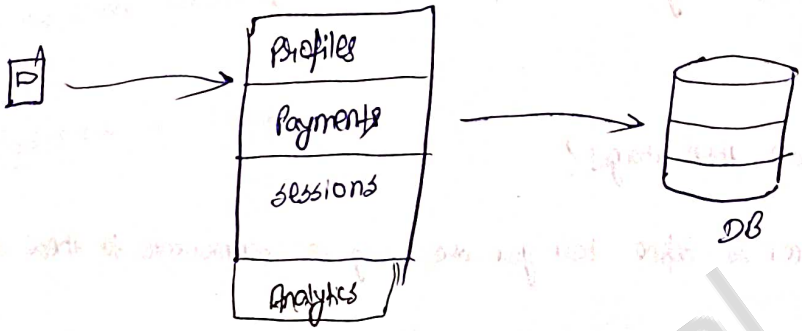
ie. Engineering invest is huge.

~~bcz you need to writ~~

② A much better approach is not to invest so much in the start.

When you have a new feature in the monolith architecture, you can try to separate that out into a single service.

Eg:



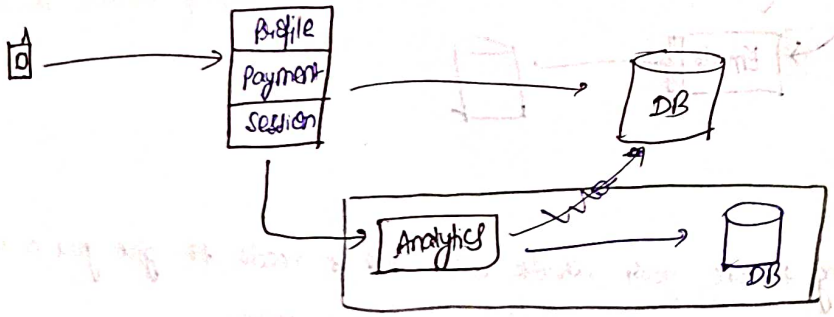
Let's say, you might need a new analytics feature that you need. Whenever any of the existing modules produce an event, you need analytics module to process the event ~~and then~~ to give dashboards, graphs, reports, etc.

Instead of putting it in the monolith architecture, we separate it out. We make this a separate service.

Earlier you just need to make a fn call in monolith, now you are going to make the call over the network in microservice architecture.

(either HTTP or gRPC)

Now,



You cannot access any of its functionalities through a fn call and the analytics service will also have its own database.

So, the analytics service will have its own repository, while the existing 3 modules has its separate code repository.

→ With this, we have opened the ~~response~~ possibility for all future services being sent to their relevant services.

So, ~~there~~

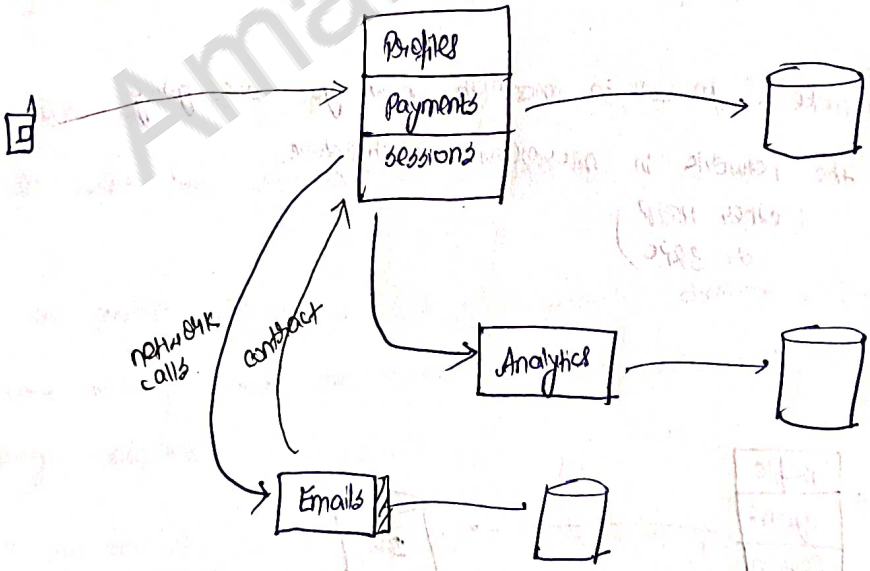
Also, we need to now bring in some infrastructure changes to make addition of these services easy.

And, what are these changes?

Obviously, you need to define how you are going to communicate to these services

→ You might find the functionality of sending emails which is being used by Payments, Profiles, Analytics, etc. So, you separate that out into separate service.

Eg:



Email service is going to take your network calls, but it needs to give you a contract that for the profiles that I have, ~~this is the~~ I have id - as string

This contract is to be maintained by Emails service. And whenever an external service wants to call a fn over here, it has to read this contract, construct an object as per the requirement and then send the request over network.

@ Contract:

So, we need to define contracts for each microservice.

These are also called clients.

→ Whenever a service in session service wants to call an email service, it doesn't really read the contract.

Each service has their own client.

~~We will~~

If I'm an engineer of Analytics service, I say, if well you are going to read my contract, construct the object in my way. That's pretty bad.

Instead, I will write some piece of code, make a library out of it. And, anybody who wants to talk to Analytics, you use this library to make those fn calls.

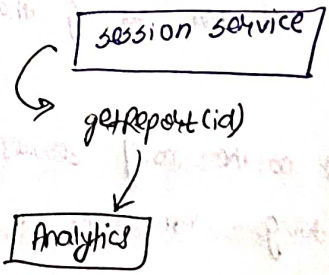
So, if session service wants to talk to Analytics, all it needs to do is download the library

What happens if my library is not upto date?

→ The code will break most likely.

So, you need to continuously update the libraries.

Eg:



Routers :

(b) We need to have a mechanism for routing requests from any service to another service.

We need some sort of router. most likely, its going to be load balancer, the service registry which have requests based on service capabilities.

(c) Simplifying Deployments →

The simplest way to deploy any service is to do a scp.

which is copy of the code that you have in GitHub repos onto an AWS box or GCP box and then going onto that box and running the service.

~~onto an AWS box or GCP box~~

→ scp repo _ dest cloud
→ ssh box
→ run repo

Here, you are copying the code from one place, then keep it in the cloud as service and then running that service by hitting the command.

But as the no. of services in your organisation increases, that's not you want to be doing. You want a more seamless way.

May be on every (git push) to master, you want the deployment to be dead. Bcz your tests have run through and you are sure that everything is fine.

* Automated Deployments

→ You want to simplify deployments using tools like

① Jenkins

② using containers instead of deploying it on the box that makes your DevOps team happier.

③ service Dashboards

④ Communication:

Communication betn services can be varied.

Eg: Let's say, the session service talks to Analytics service using a message queue. If the event doesn't reach Analytics service, what's the big deal?

↓

Only the reports will be wrong. It's not like users is going to drop off.

→ While in payment service, you might need a much more immediate response. So when profile service is talking to payment service, you might want a request-response architecture so that you know exactly what happened immediately after a request has been sent to payments.

⑤ Logging:

Eg: If a request is sent from the user to the session service, then it goes to analytics service which triggers something in profile service. & then it went to payment service.

↓

There is no way that you can track all this, by logging into each service and checking the logs.

→ Instead you want to take all the logs in all the services and push it into a single repository. (i.e. a single database). You could use the Elastic stack.

Few things to keep in mind for moving from Monolith to microservice :

① Single source of truth :

* Every microservice needs to encapsulate the data that it is responsible for.

Every component in your architecture has to have a single source of truth, when it comes to microservices.

↓

It only happens when you have a dedicated data store for each service

② Condense business responsibilities into a single place :

It's very tempting to take services and break them down into simpler and simpler components.

But the important thing to remember is that what is the responsibility of this service ? And, when we are breaking it down ?, are we also separating out the responsibilities ?

Eg : Let's say, the profile service needs to take info from various external services like Google, Facebook, Linkedin, etc.

Here, we might create a new service which acts as an intermediary bet^n info from these external services and profile service.

That seems like a good idea.

But the question to ask is →

→ Is data being used by any service except the profile service. ? → Here, it's NO

ie. APIs being used by multiple services ?

(b) When there is a change in the profile service requirement, is there a change in the service requirement ?

↳ Here, it is most likely Yes

Decoupling

(c) Is the business requirement of new service separate from profile service " " ?

↳ Here, it's NO

ie. separation of concerns

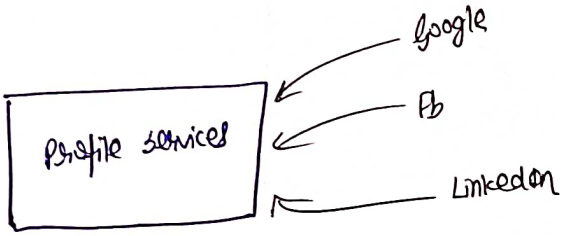
Eg:

So, what should ideally happen is that this should be a component of the profile service. You can make a library out of it

You can put it in the utils package.

The codebase should not be separate.

So,



* → Don't break it down just because your engineering will get much simpler.

Really do consider, whether you need to separate it further and further.

③ Initial infrastructure cost is high!

Can we afford the infrastructure requirements?
for building a microservice architecture

The cost of logging, the cost of building a deployment infrastructure → All of this is really not worth it for a small team.

* For a startup, going for a microservice architecture is fine if the total no. of services that you are handling per person is less or equal to 2.

ie. one person is handling ≤ 2 microservices.

It also depends on user scale.

* For a medium org, 1 person should be handling ≤ 1 service.

* For a large org, 2 to 4 persons per service.