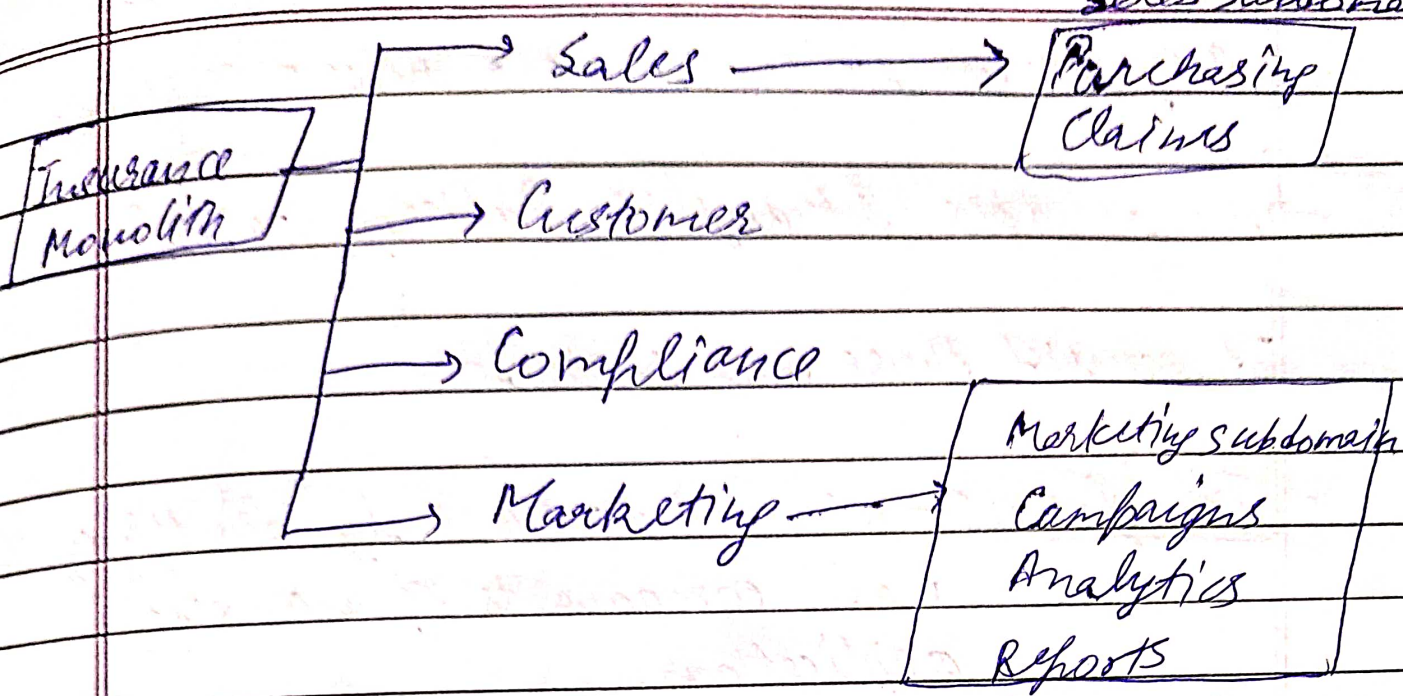


2) Strangler Pattern :-

- Many organizations start building new applications with microservices or convert existing monoliths to microservices
- So, Transforming a large scale monolithic application to microservices from scratch is a challenging task.

- So, Strangler Pattern is a software design pattern used to refactor monolithic applications to microservices gradually

- It acts as a fail-safe allowing the organization to rollback to roll the ~~new~~ monolith if there

any issues with the new microservice.

→ How does Strangler Pattern works?

It consist three main steps :-

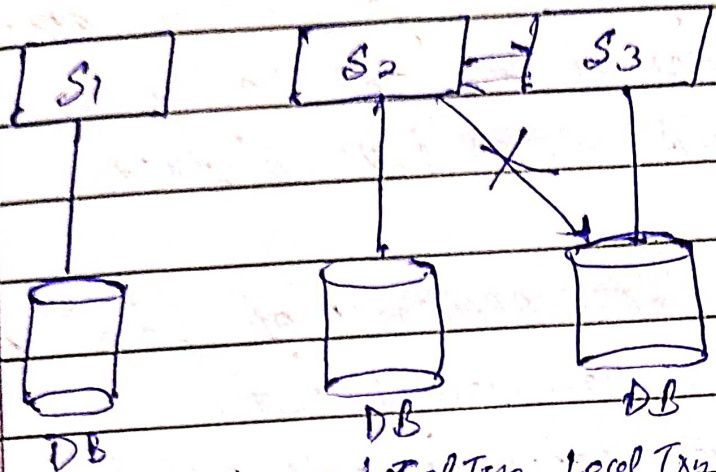
- Transform :- we need to start by identifying the main components of monolithic application.
It involves identifying the boundaries between the existing application & new components being developed
- Coexist :- Build a wrapper around the monolith to allow the new components to coexist with existing application.
- Eliminate :- Finally eliminate the monolith by replacing parts with new components.
We must ensure that each microservice works as expected before integrating it into system.

⇒ Database Management in Microservices

- Database for each Individual Service
- Shared Database

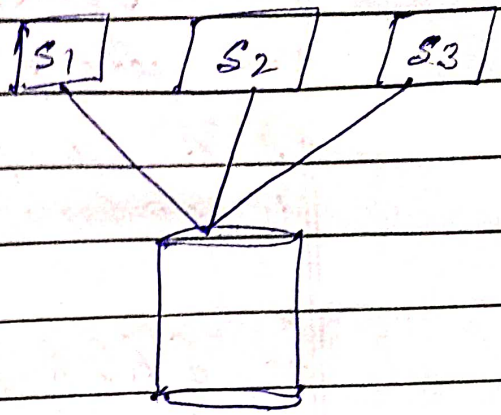
Page No. _____
Date

Database for each individual services



Local Transaction Local Txn Local Txn

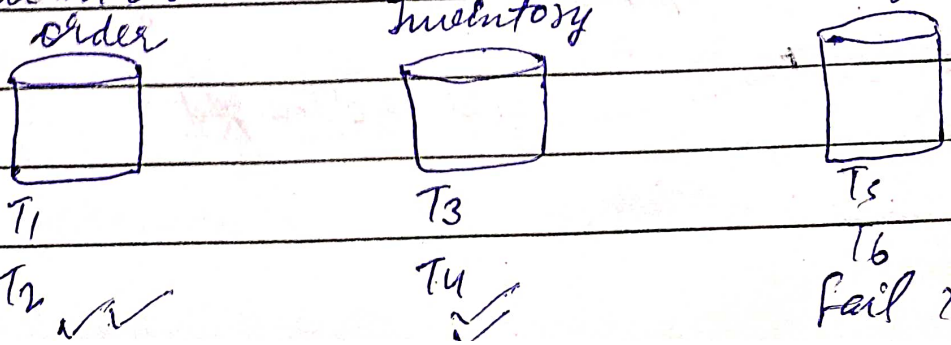
Shared Database



Let say if service S2 needs some data from S3's database, then no service can directly access other's database they need to communicate with the specific service & that service will fetch that data from its database & give the data to the respective service through API

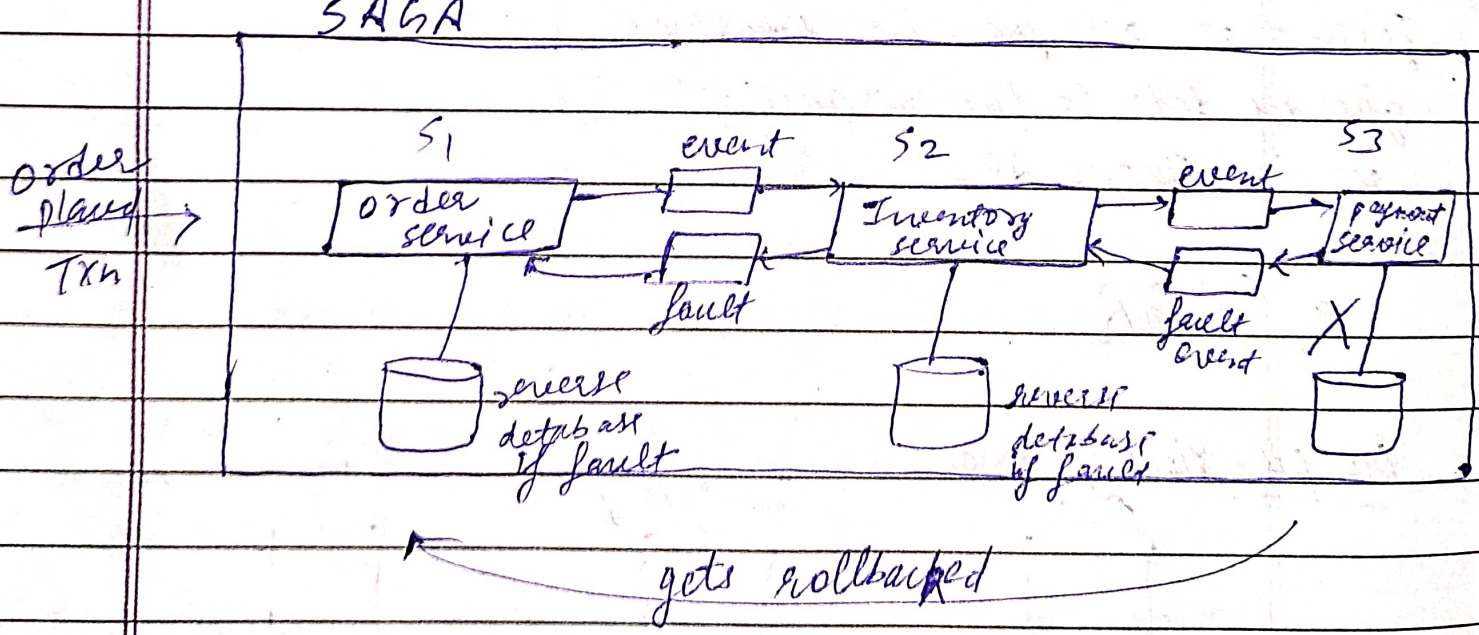
→ SAGA

Example - Place an order

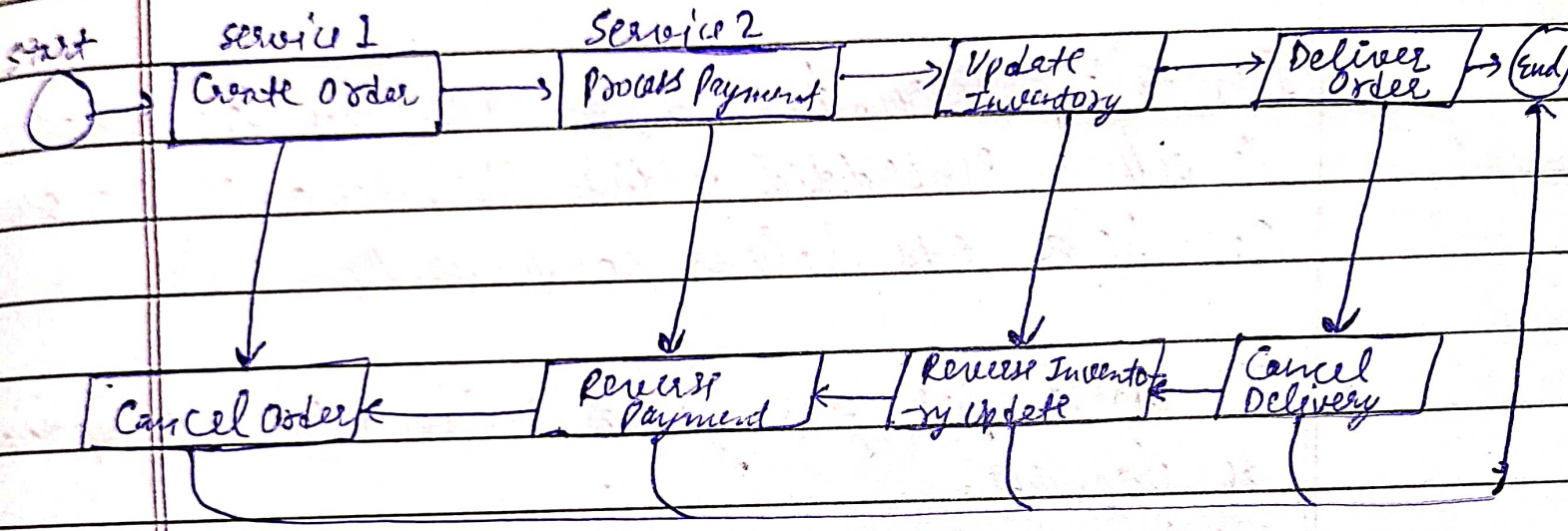


Here, let's say to place an order, from order DB we select the data then added it to inventory but when we were doing payment it fails. Then according to ACID properties it must be rollbacked then it is where SAGA comes into picture.

So, SAGA stands for sequence of Local Transactions which provide transaction Management.
So, here, each service will generate event which will tell the transaction type which is either successfully happen or fail. it will send that event to other service where that service needs the process value to change in its database & it then sends back the event of fail or success if any event fails then the respective process will gets rollbacked.



- The SAGA pattern guarantees that either all operations complete successfully or the corresponding compensation transactions are run to undo the work previously completed.



- SAGA Execution Coordinator - (SEC)
It is the central component to implement a Saga flow.
It contains a saga log that captures the sequence of events of a distributed transaction.
It helps to identify the transaction successfully rolled back, which one are pending and can take appropriate actions.

★ There are two types of SAGA pattern :-
1) Choreography
2) Orchestration

Page No.
Date

(i) → Saga Choreography Pattern :-

- Each microservice that is a part of the transaction publishes an event that is processed by the next microservice.
- In this pattern, the Saga Execution Coordinator is either embedded within the microservice or can be a standalone component.
- It works for greenfield microservice application development.
  It is suitable when there are fewer participants in transaction.
- Axon Saga, Eclipse Microprofile LRA, Seata are all framework available to implement this pattern.

(ii) → Saga Orchestration Pattern :-

- A single orchestrator is responsible for managing the overall transaction status

Page No.
Date

- If any of microservices encounters a failure, the orchestrator is responsible for invoking the necessary compensating transactions
- It is useful for brownfield microservice application development architecture.
- Camunda and Apache Camel are few framework to implement this pattern.

Event Sourcing

- Event Sourcing gives us a new way of persisting application state as an ordered sequence of events.

We can selectively query these events and reconstruct the state of the application at any point in time.

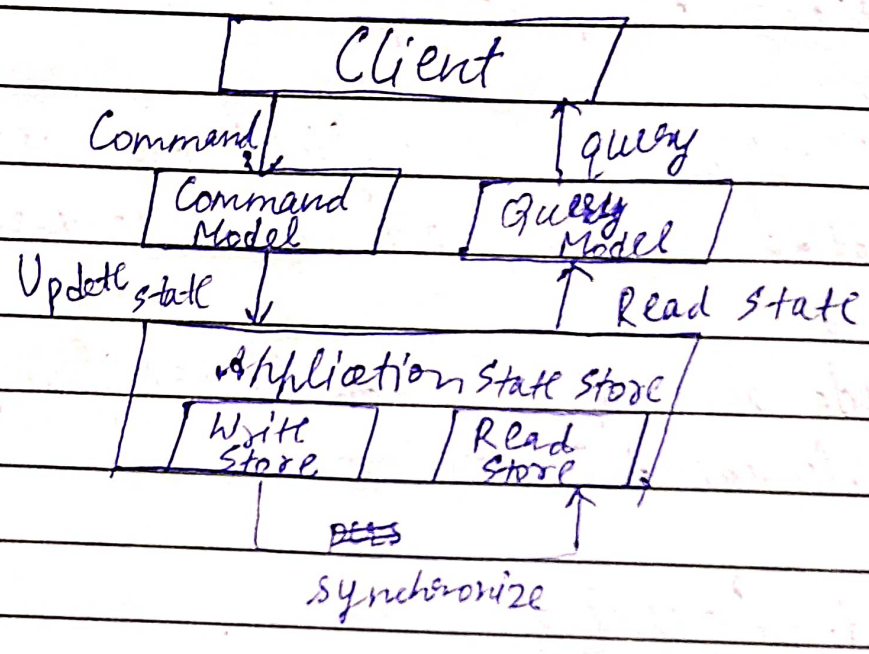
CQRS (Command Query Responsibility Segregation)

- CQRS is about segregating the command and query side of application architecture.

It is based on Command query separation principle which suggests that we divide the operations on

Page No.
Date

domain objects into distinct categories :-
Queries and commands



→ queries return a result and do not change the observation state of a system.
→ Command change the state of the system but do not necessarily return a value.

Scaling from Zero to Millions of Users

DNS :- (Domain name System)

DNS is the Internet's naming service that maps human-friendly domain names to machine-readable IP addresses.
It is transparent to users

Page No.
Date

→ Vertical or Horizontal Scaling

Vertical scaling referred as "scale-up" means the process of adding more power (CPU, RAM etc.) to your servers.

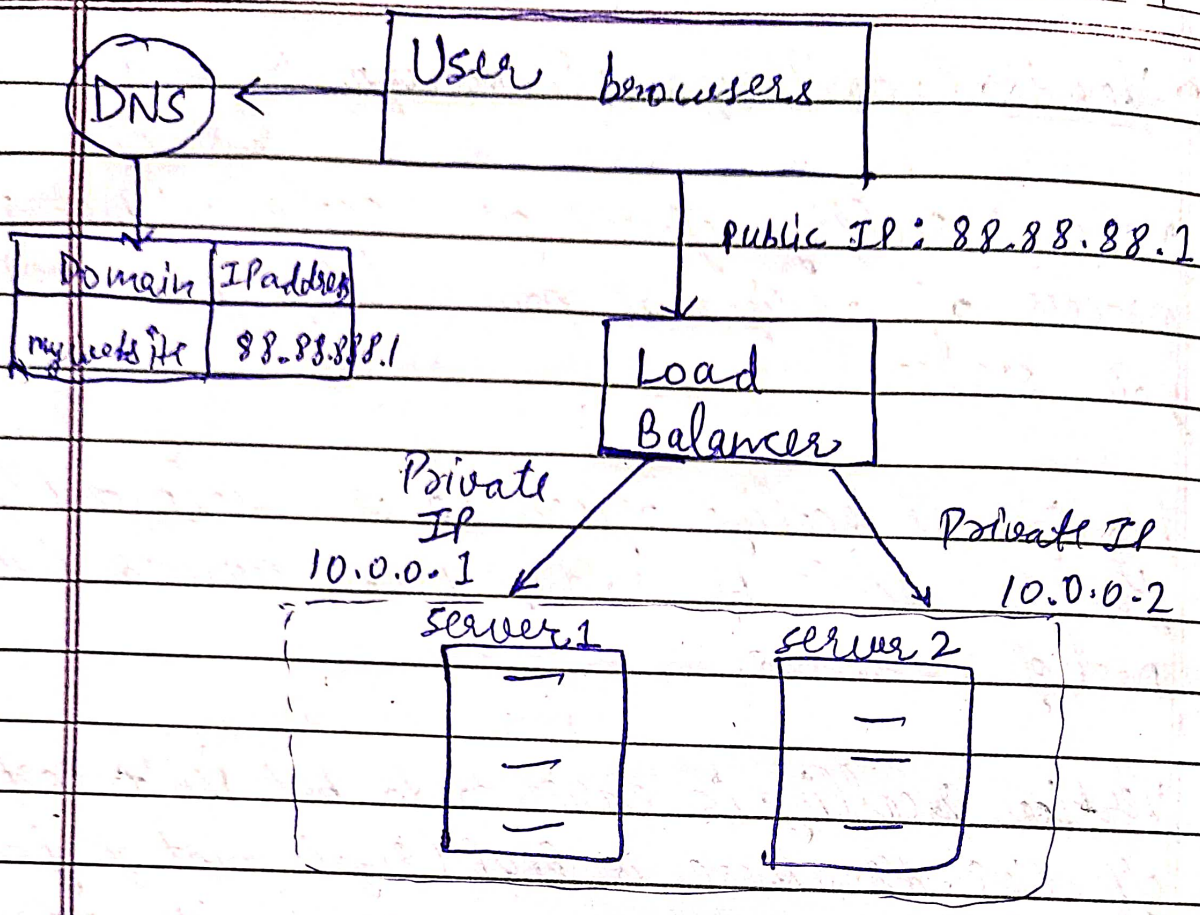
Horizontal scaling referred to as "scale-out", allows you to scale by adding more servers into your pool of resources.

Vertical scaling is easier to do but isn't cost-effective. Also, there is SPOF (single point of failure) as there is only one server present with very high power. If it goes down whole application goes down.

→ Load Balancer →

A Load Balancer evenly distributes incoming traffic among web servers that are defined in a load-balanced set.

User first connects to public IP of the load balancer which further connects with private IP of our defined servers.



- If one server goes offline, all traffic will routed to server 2. It helps the website from not going offline.

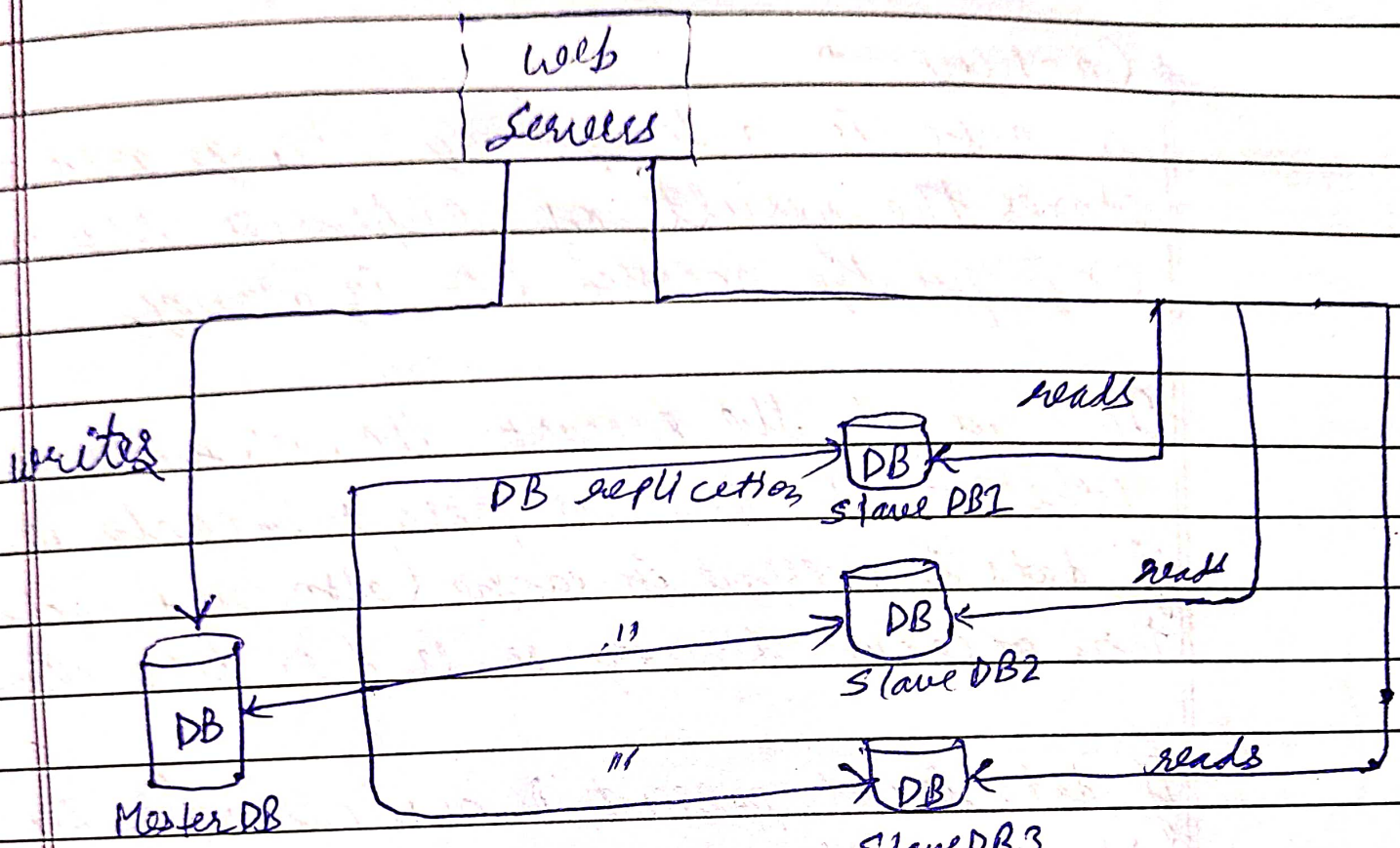
Data Replication :-

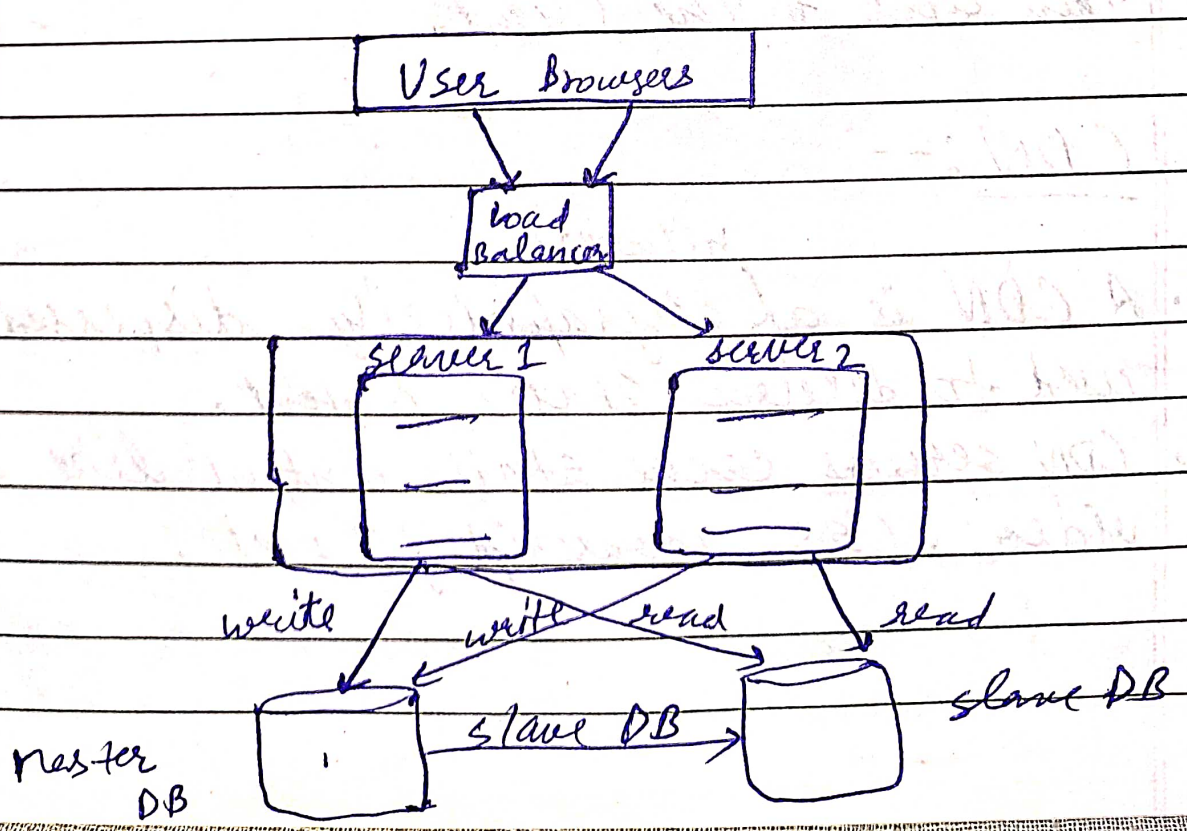
- Database Replication can be used in many database management system, usually with a master/slave relationship between original (master) & the copies (slaves).

- master database only supports write operation &

Page No. _____
Date

Slave database gets copies of data from master database and only supports read operation.





Page No.
Date

Now, to improve the load/response time, we use caching and CDN for this —

Caching →

A cache is a temporary storage area that stores the result of expensive responses or frequently accessed data in memory.

Once we get the request, the web server checks whether the data is present in cache or not. If data is present in cache (also called cache-hit), then cache directly sends data to the client.

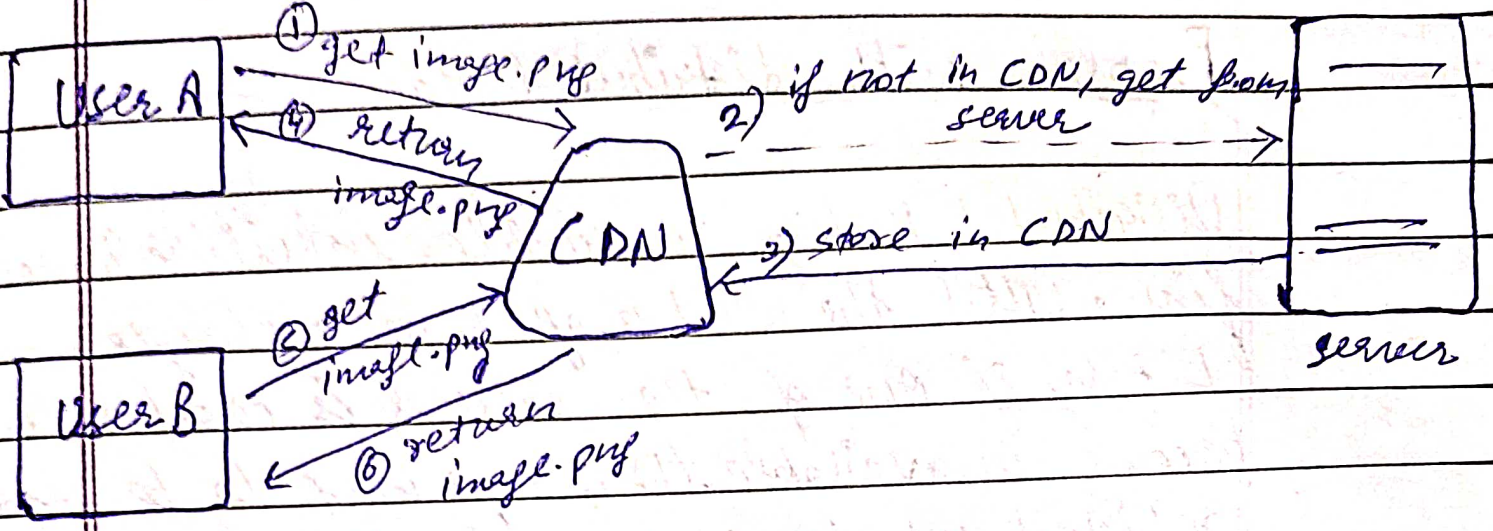
If data is not present in our cache (cache miss), it asks to the database, then store it to cache then send to the client.

CDN :—

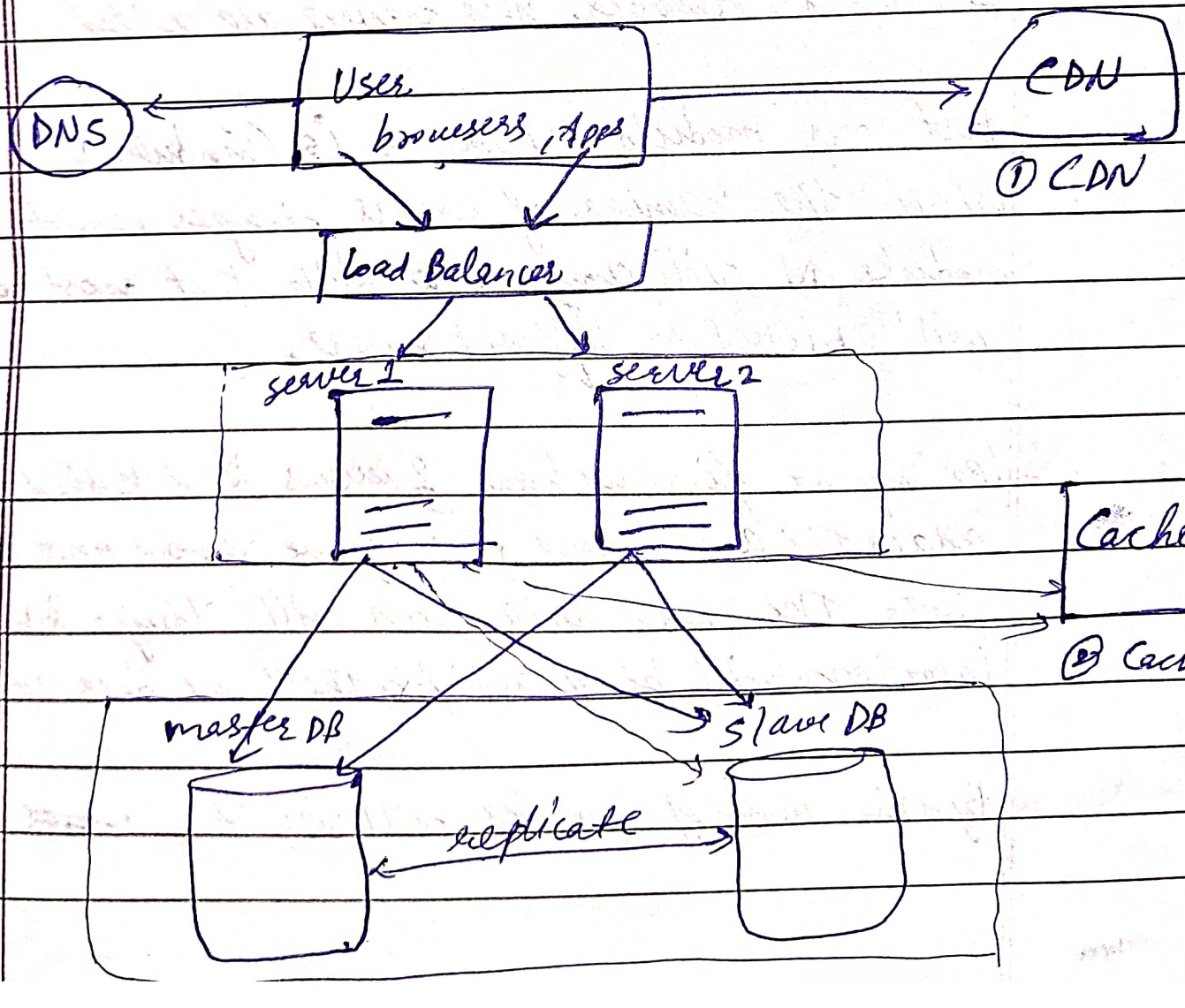
- A CDN is a network of geographically dispersed servers used to deliver static content.
- CDN servers cache static content like images, videos, CSS, Javascript files etc.

Page No. _____
Date

Workflow of CDN



So, our system design now will be —



Page No.
Date

Consistent Hashing :-

Problem with distributed Hashing -

Distributed Hashing is simple, intuitive & works fine, until the number of servers changes. Let say if one of the servers is crashes or become unavailable then, keys need to be ~~redistributed~~ redistributed to account for the missing server.

Also, keys also need to redistributed to include new servers, whenever new servers are added.

but our modulo distribution is (modulo N). That when the number of servers changes, most hash module N will change, ~~so~~ with that most keys will moved to different servers.

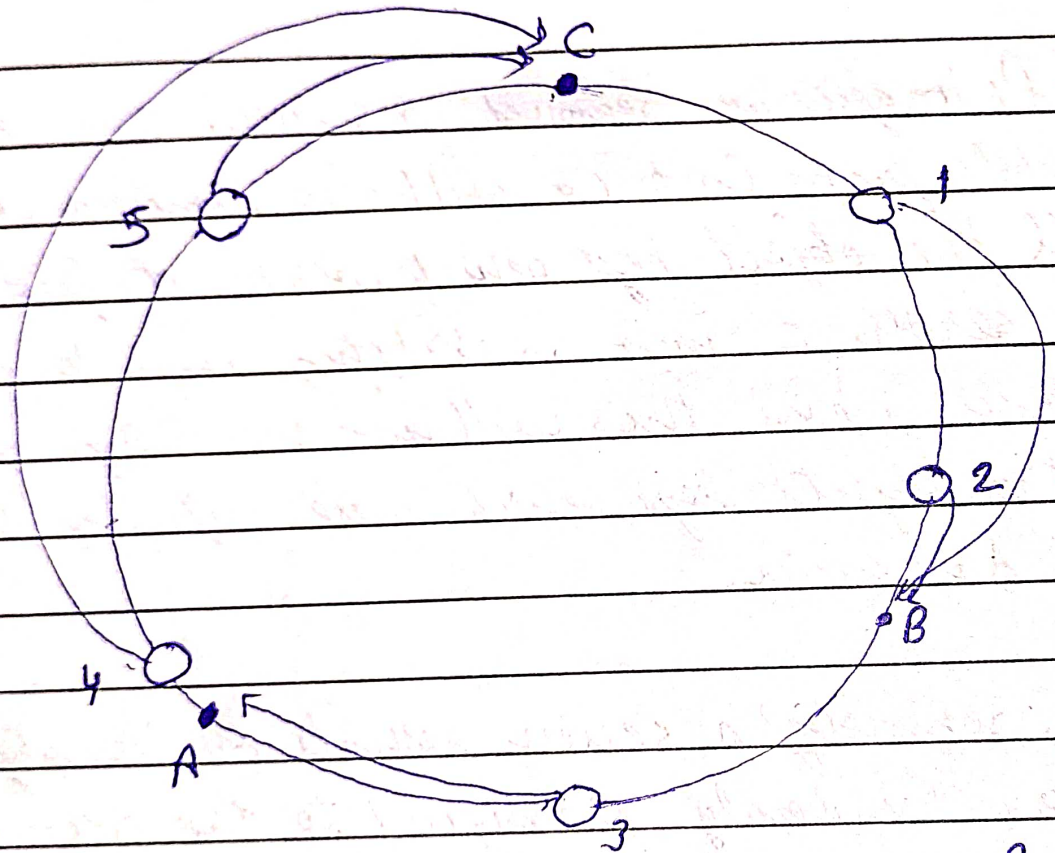
let say at first we have 3 servers & we hashed it & added the data., now, let say we remove one server now, the hash data fetch will change & new location will be added for the same data now.

By this most of queries will result is ~~misses~~ misses

Page No.
Date

So, Its where consistent hashing comes into play :—

Consistent hashing is a distributed hashing scheme operates independently of the number of servers in a distributed hash table by assigning them a position on a hash ring



Here, A, B & C are the servers & let 1, 2, 3, 4 & 5 are the objects & data.

Simple rule is that :—

- Each object key will belong in the server whose key is closest, in a counterclockwise dir^{n}.

- Another thing is that if any server has more weight to add data then it can hold as many objects as per its weight.

for example — if server B is twice as powerful as the rest, then it could be assigned twice as more labels.

- let's imagine we removed a server. then all its labels from C0 to C9 will also removed, by this all the object keys which were at first assigning to server C now reassigning them to other servers, but this will not make any impact on other objects keys which were originally present in A & B server.

* So, removing a server results in its object keys being randomly reassigned to the rest of the servers, leaving all other keys untouched

* Similar thing will happen when new server gets added.