Date: __/__/____

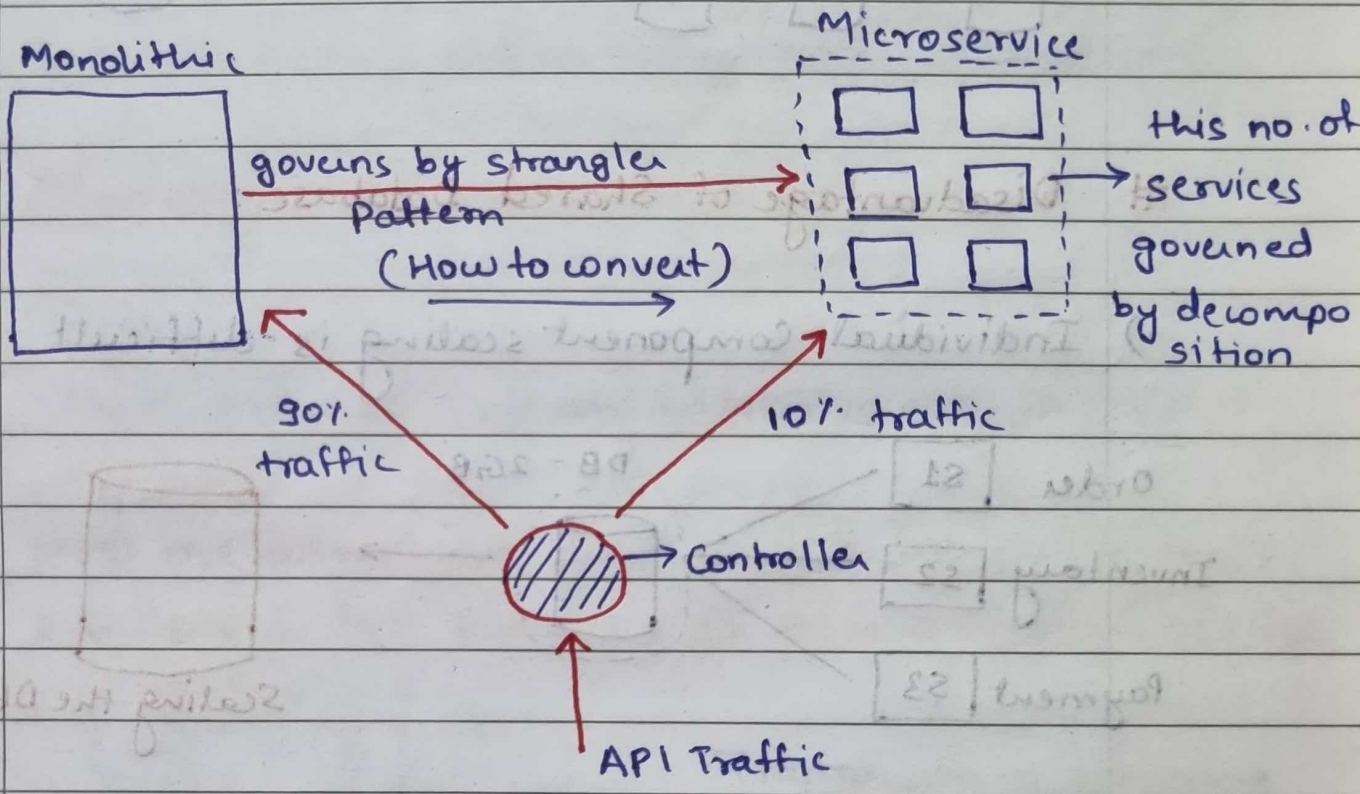
#4: Microservices Design Patterns :

a) ** HLD SAGA Pattern

b) ** Strangler Pattern

c) CQRS

1) Strangler Pattern : Used when legacy application refactor to Microservice



→ Suppose we only have 1 feature of monolithic, into microservice. Now controller can decide how much traffic can be redirected to monolithic and microservice.

→ If Microservice is down/fail, then 100% traffic will be shifted to monolithic.

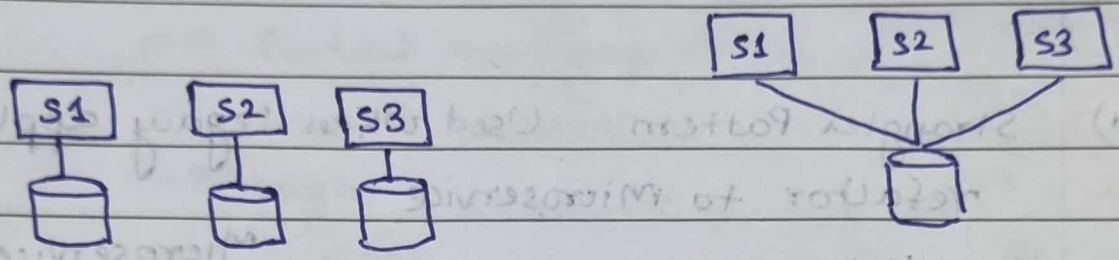
→ Once success, we can have 10%, 20%, 50%, 100% traffic on microservice.

Date: __/__/____

2) Data Management in Microservice

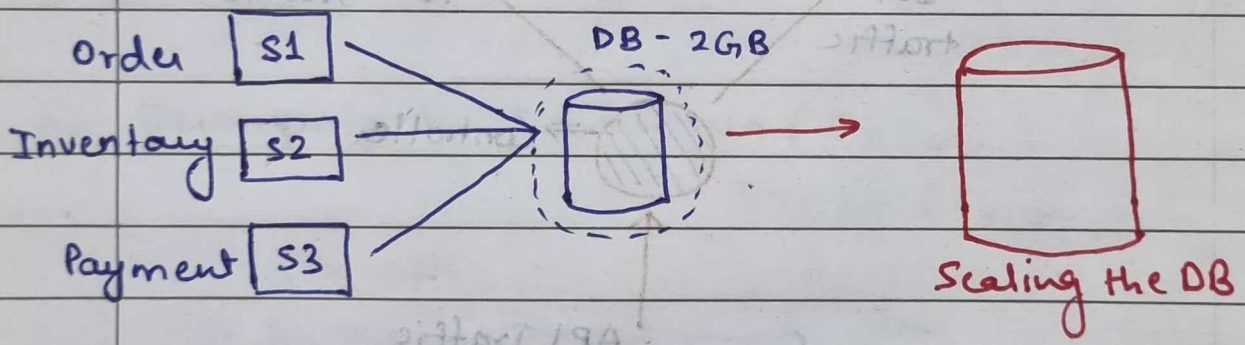
a) Database for each individual service

b) Shared Database



Disadvantage of Shared Database:

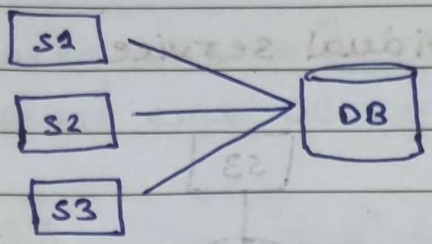
i) Individual Component scaling is difficult.



Suppose if order service has 1 Million request but Inventary and Payment has very less. But the issue is for Order Service we can't scale individual DB, Instead we need to scale the whole DB, which is a bit difficult.

Date: __/__/____

ii)



----→ It has $T_1, T_2, T_3, \ldots T_{10}$ tables

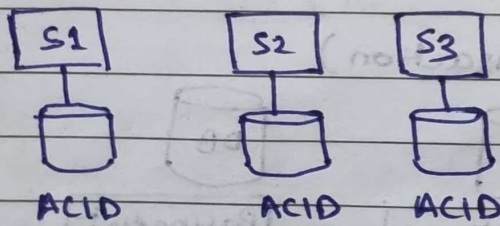
Now suppose S3 wants to modify/delete one column in T3, but since that column is used by other services S1 and S2, so S3 can't modify that column, because there is dependency from other services.

Advantages of Shared Database:

i) Query Join is easy

ii) Transactional Property (ACID) is easy.

→ Now the advantages of shared Database becomes challenges for Database for each individual service.

- 

Transaction is spanning over a distributed DB

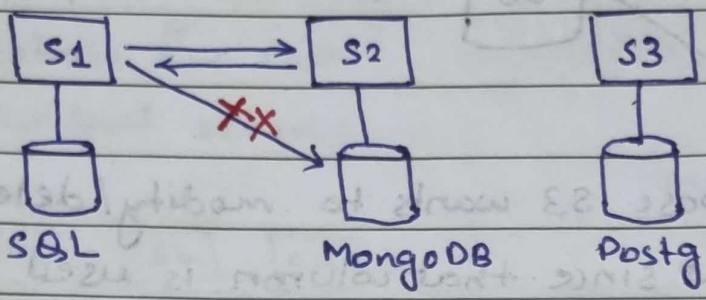
→ Solved by SAGA Pattern

- $(T_1 \ldots T_5)$ $(T_6 \ldots T_8)$ $(T_9 \ldots T_{10})$

Since tables are present in different DB so query join is a challenge: → Solved by CQRS

Date: __/__/____

→ Database for each individual service



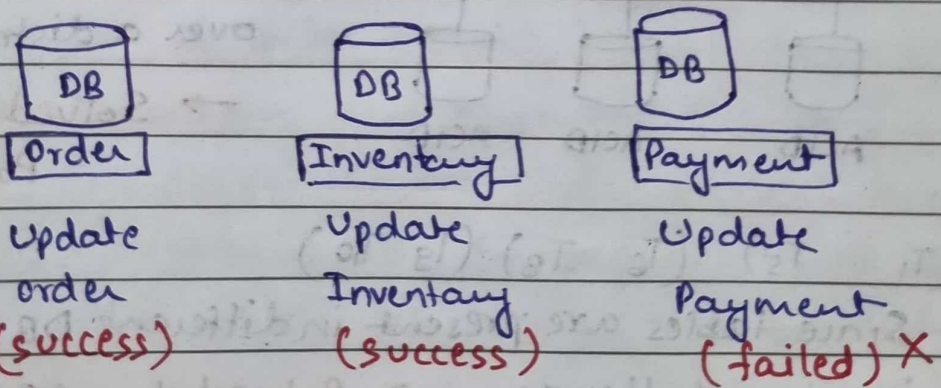
(Based on requirements)

Advantage:

- One Service cannot directly access DB of other service. S1 can call API of S2 and get the data from DB.
- Based on requirements each service can have their own DB (Relational or NoSQL etc.)
- Modifications on DB is easier.
- Scaling individual service DB is easy to handle more transactions.

SAGA Pattern:

e.g. Place an Order (Transaction)

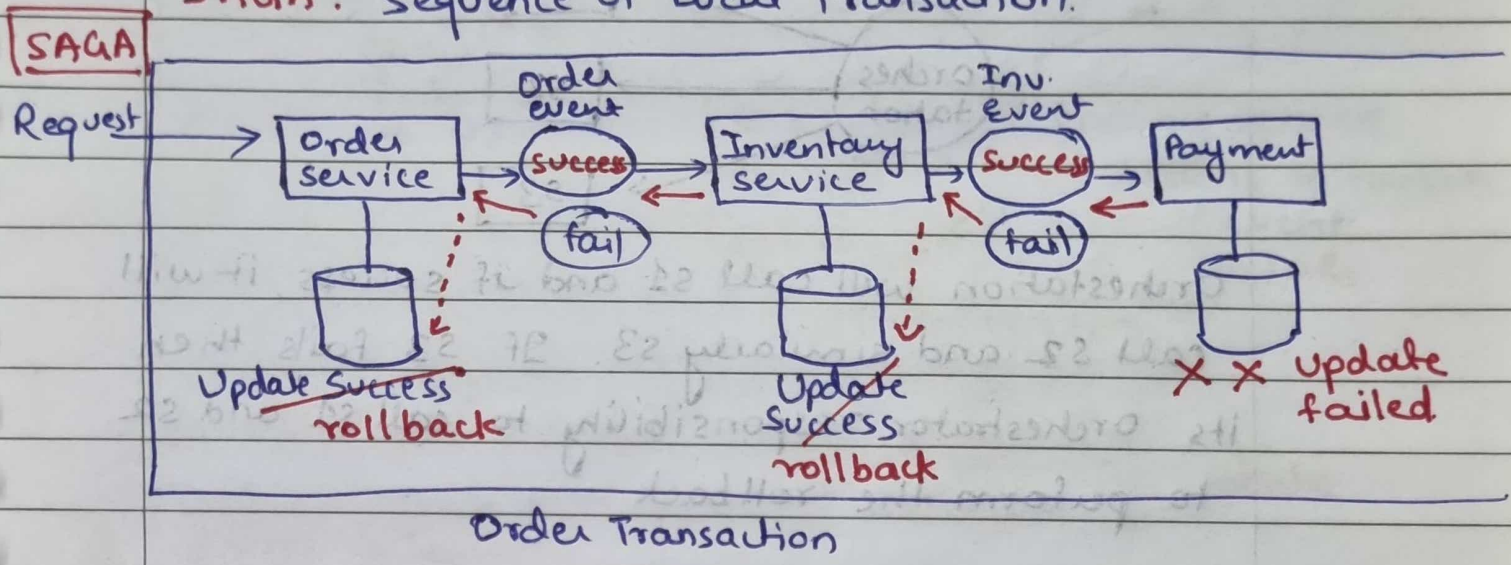


How to rollback?

Date: __/__/____

⇒ How to rollback? → Challenge
because each DB has individual ACID property.

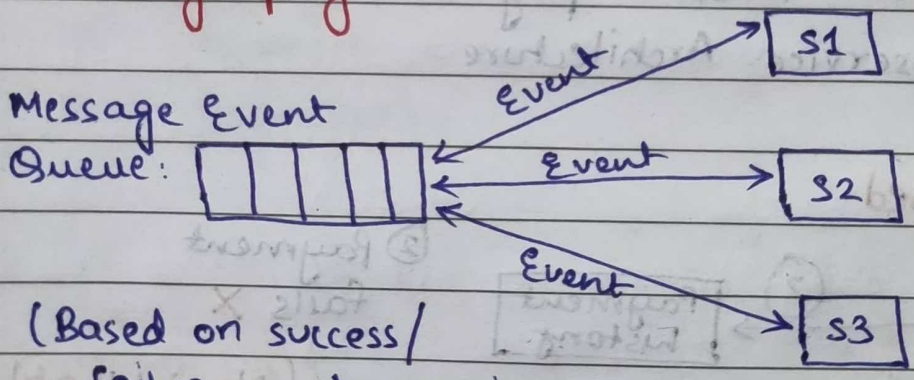
SAGA: Sequence of Local Transaction.



Order Transaction

Types of SAGA:

a) Choreography:

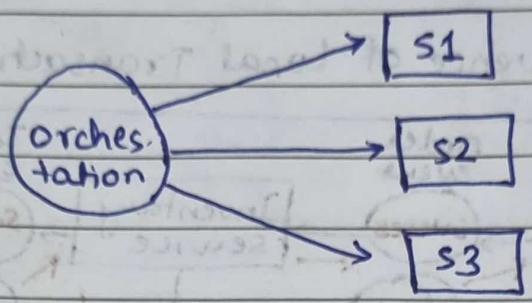


(Based on success/ failure event, services can do the transactions update)

Drawback: Cycle dependency → S1 gives event to S2, S2 gives again to S1, again same loop...

Date: __/__/____

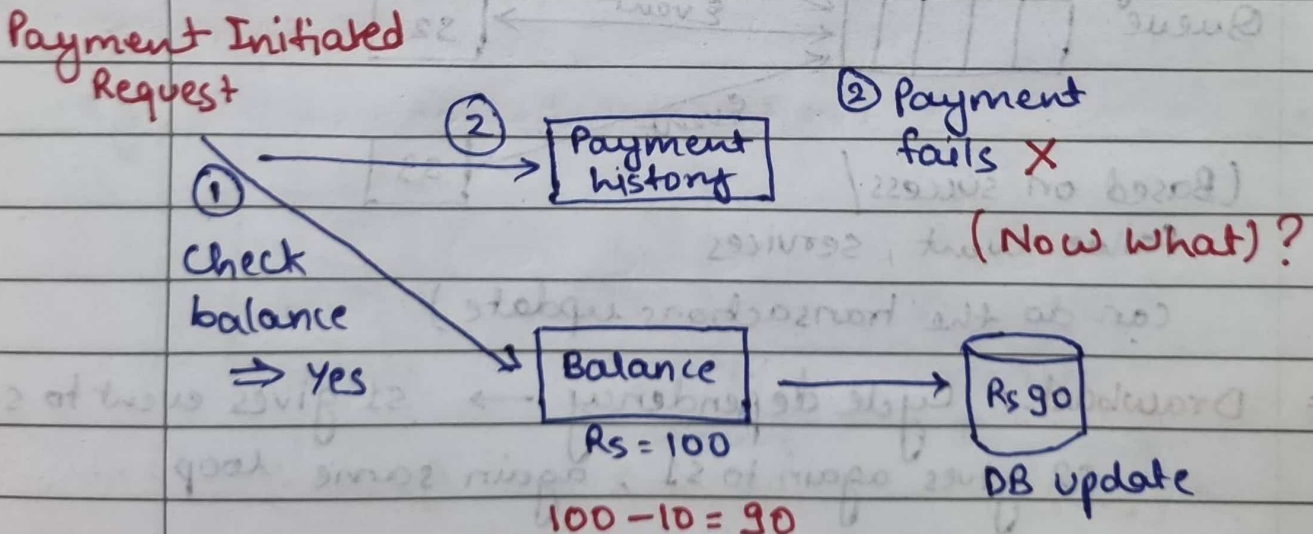
b) **Orchestation:** To overcome cycle dependency we have orchestation.



Orchestation will call S1 and if success, it will call S2 and similarly S3. If S3 fails then its orchestrator responsibility to call S1 and S2 to perform the rollback.

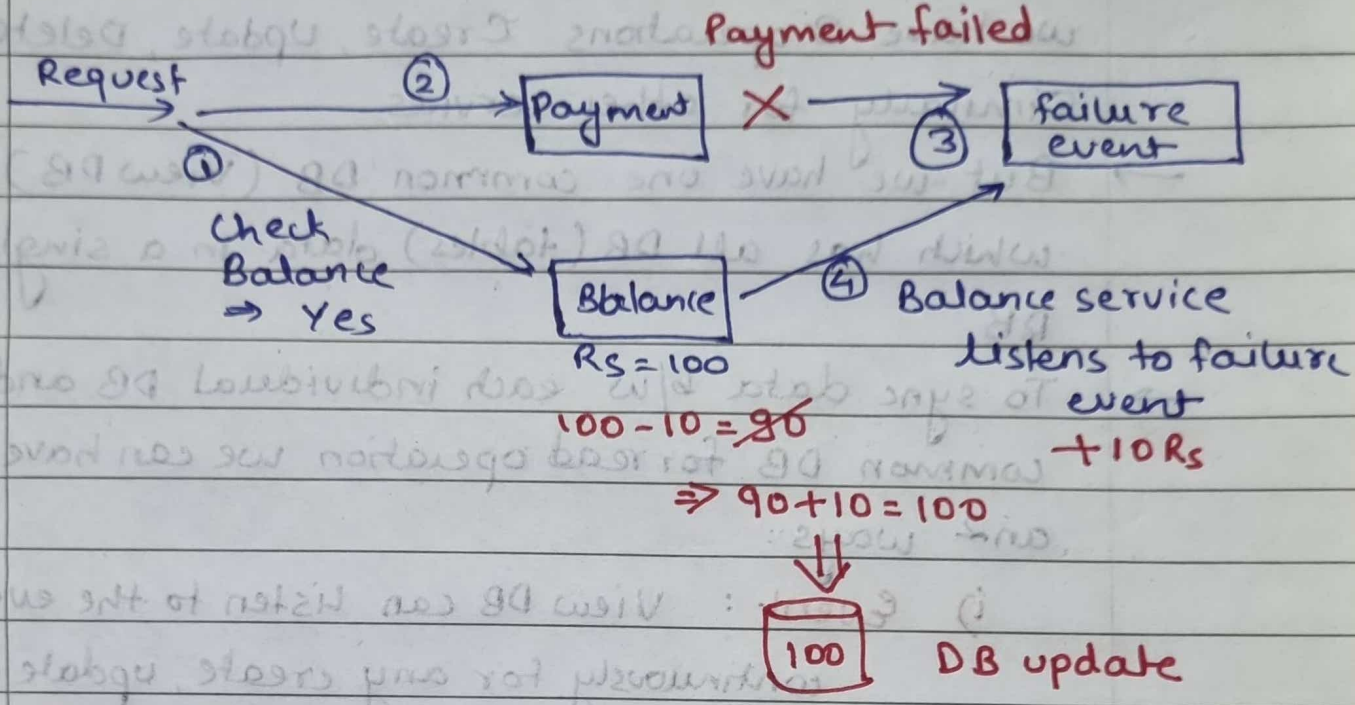
Interview Question:

- Scenario: Person A pays Rs 10 to Person B.
- Microservice Architecture:



Date: __/__/____

To resolve this issue we use SAGA:



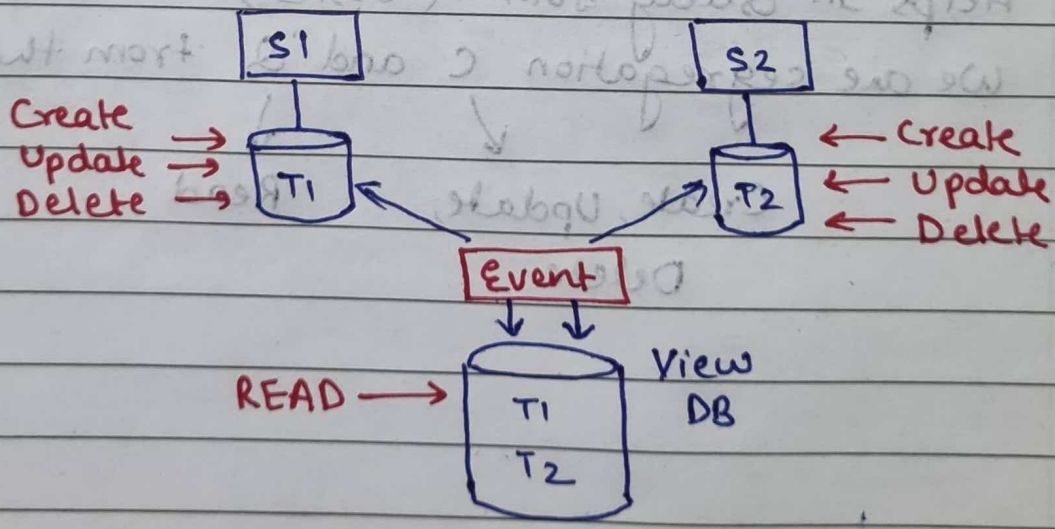
CQRS Pattern:

C : Command (Create, update, delete)

Q : Query (Select)

R : Request

S : Segregation



Date: __/__/____

→ In CQRS, for each individual microservice DB we have 3 operations Create, Update, Delete. Similarly for other services.

→ But we have one common DB (View DB) which has all DB (tables) data in a single DB.

→ To sync data b/w each individual DB and the common DB for read operation we can have an 2 ways:

i) Event: View DB can listen to the event continuously for any create, update and delete operations.

ii) DB Trigger / Procedure: Whenever there will be Create/Update/Delete, DB trigger will update View DB.

→ Helps in Query Join (CQRS)

→ We are segregation C and Q from the request.

↙ ↘

Create, Update, Delete | Read