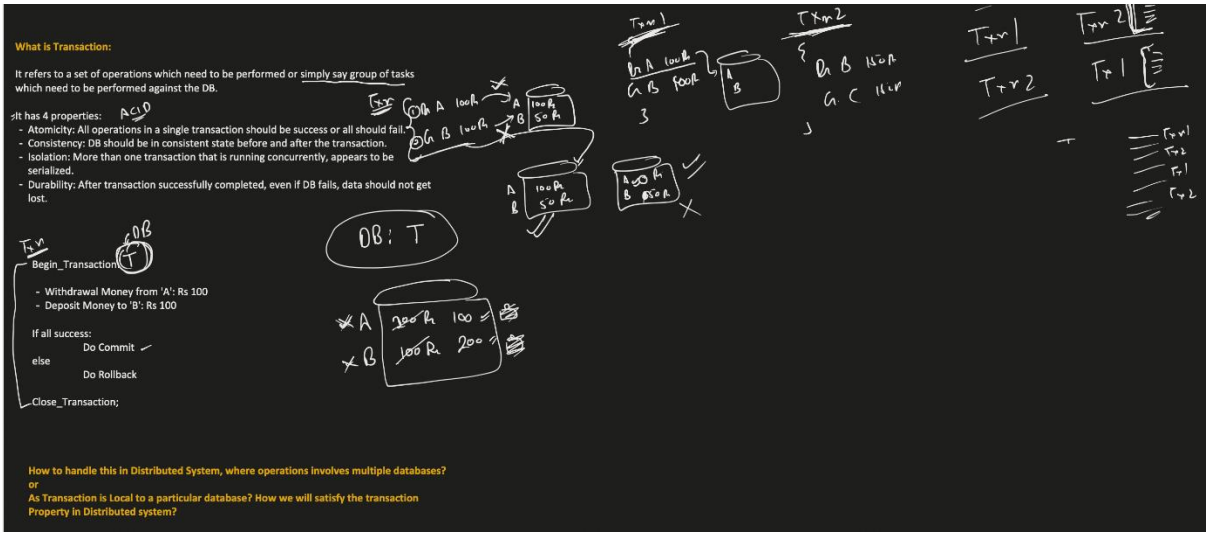
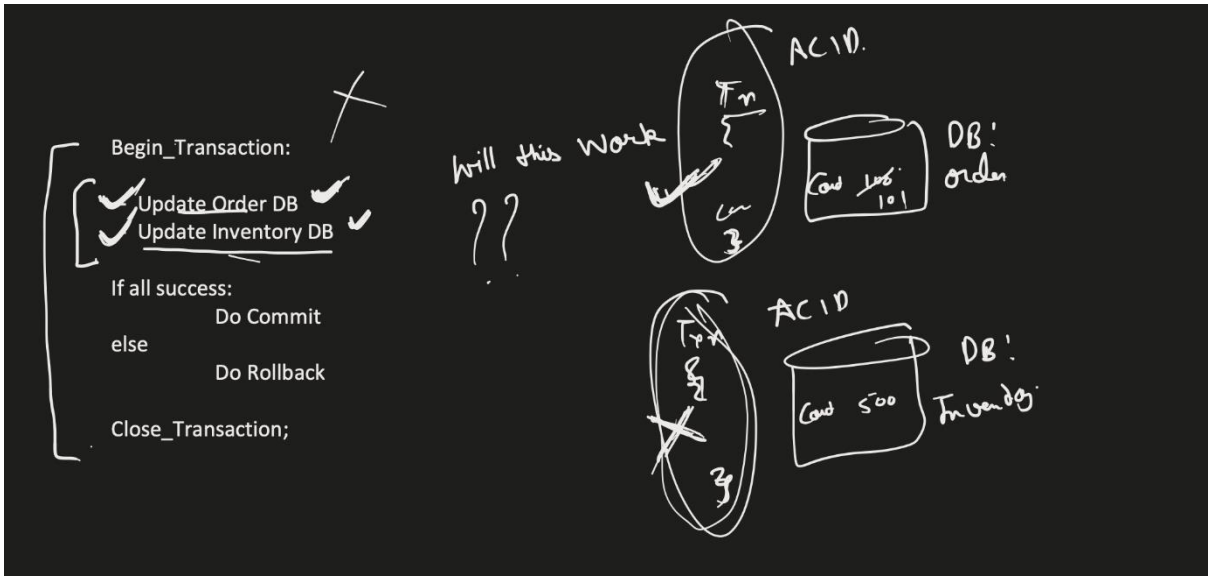
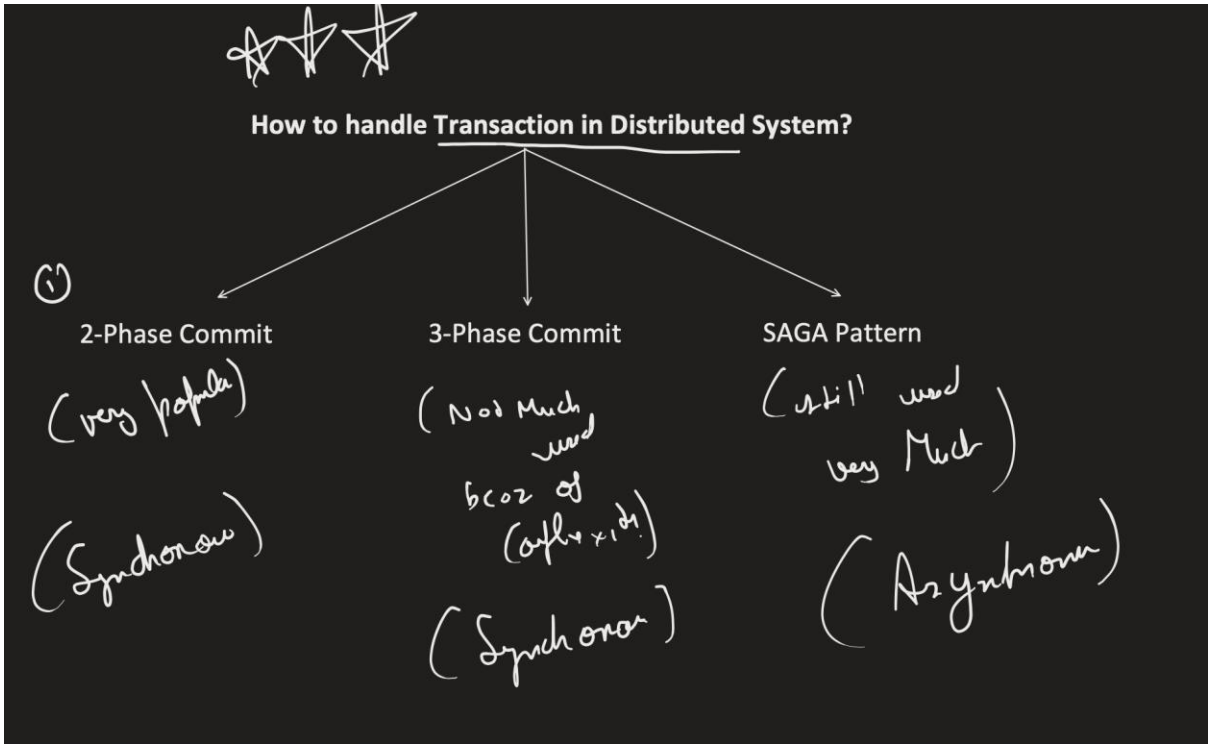
Table of Contents

HLD: Distributed Transaction Handling (2PC, 3PC and SAGA)



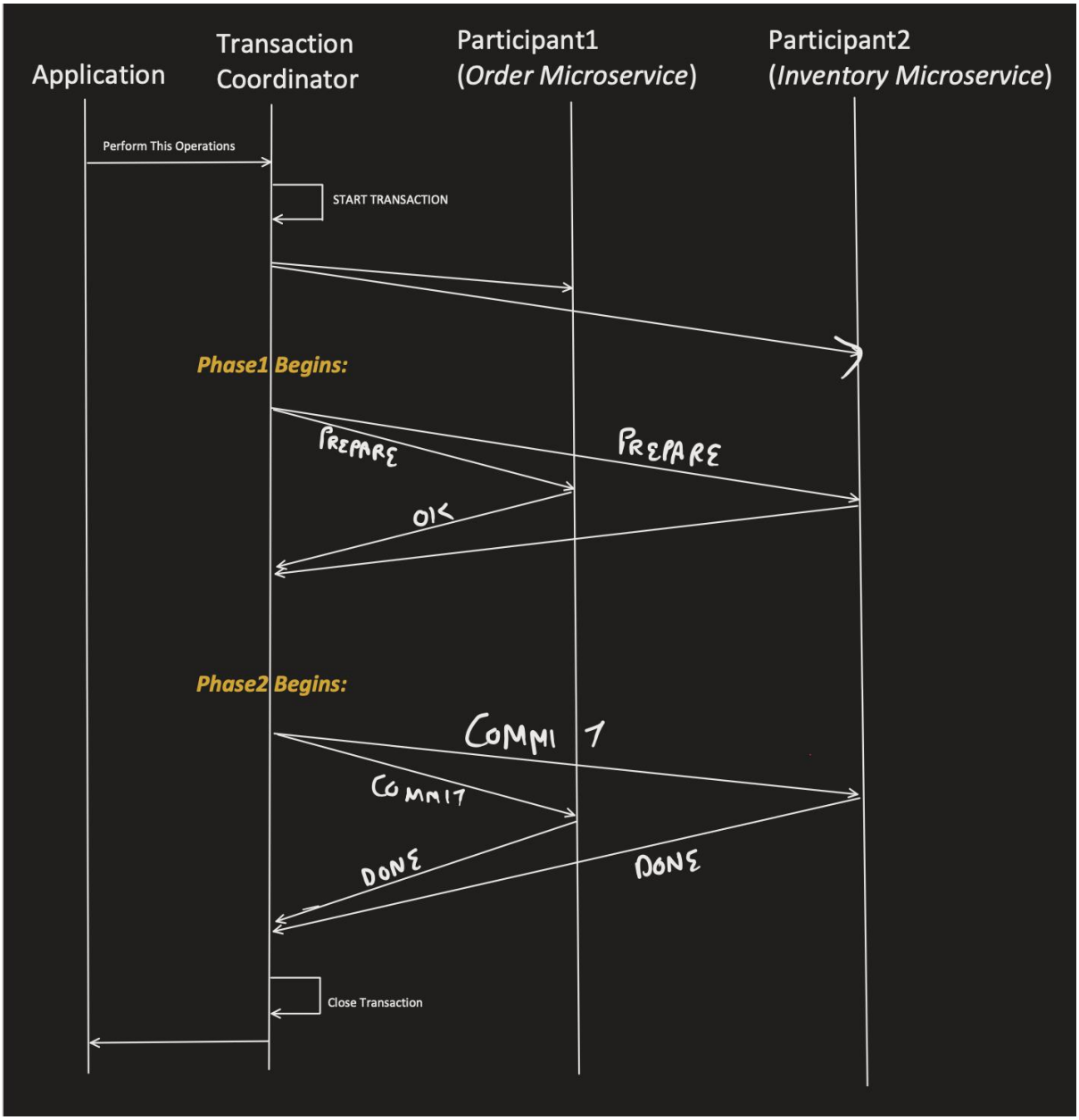


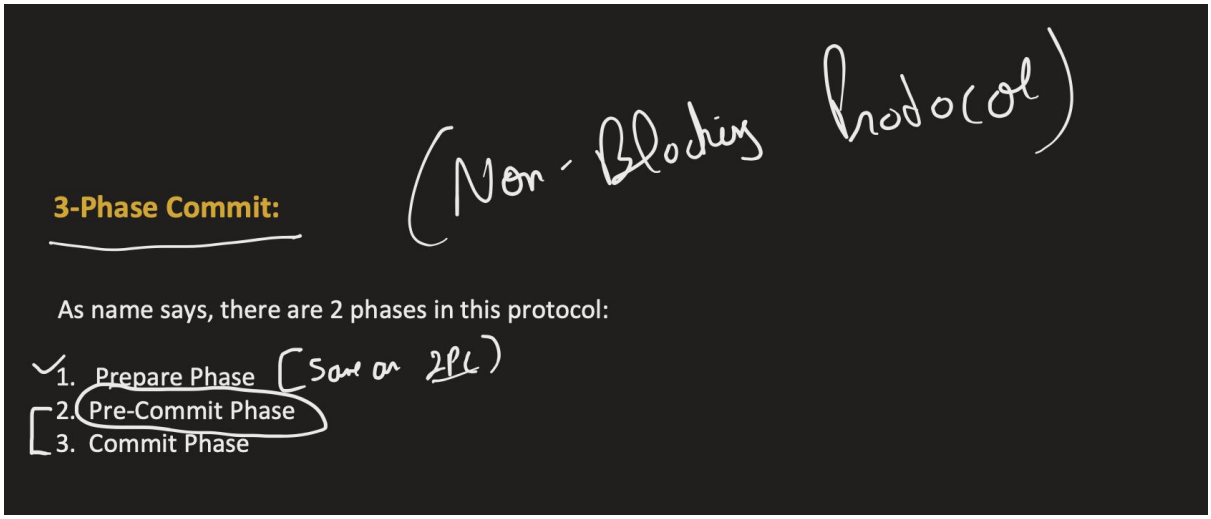


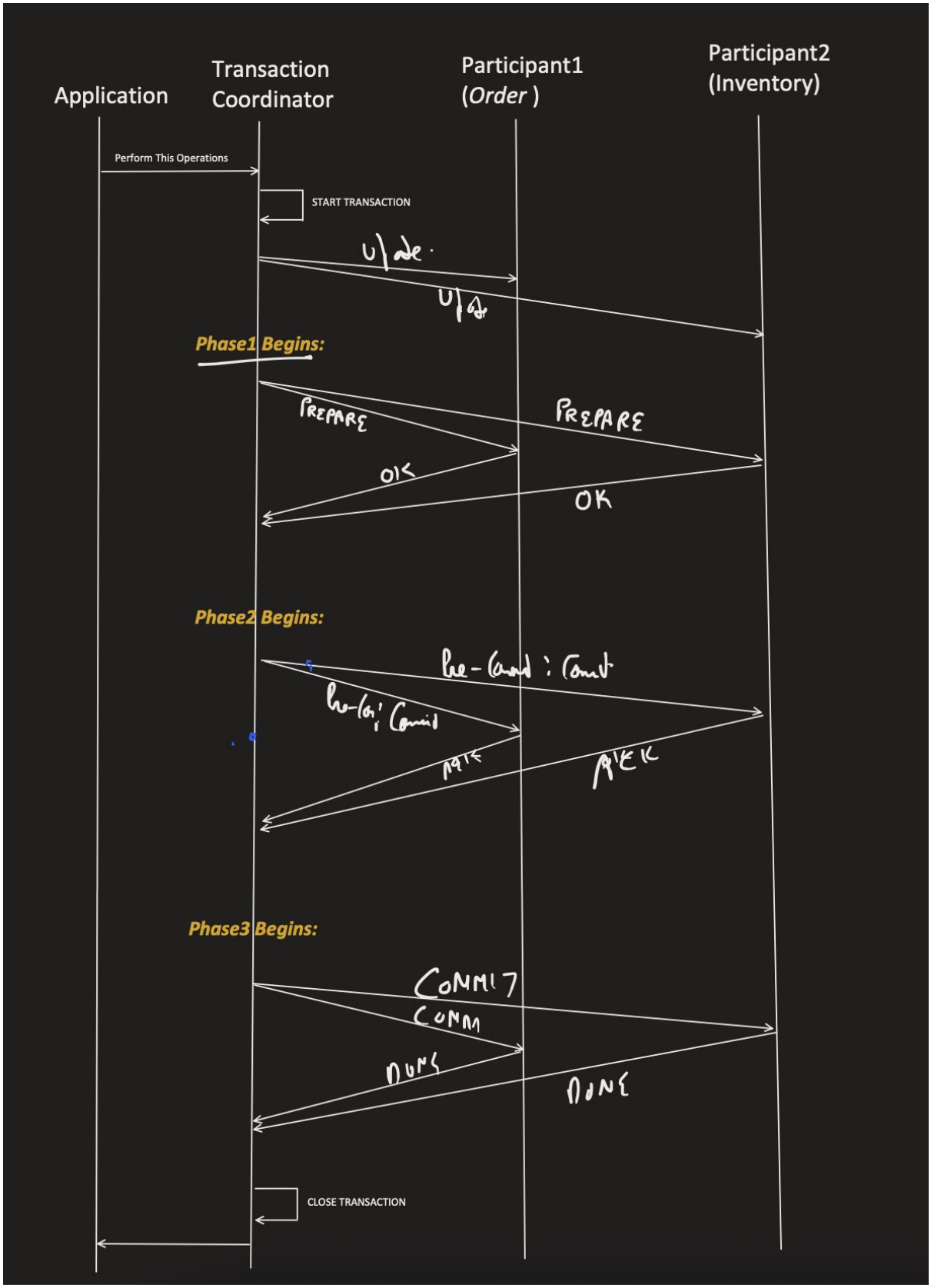
2-Phase Commit:

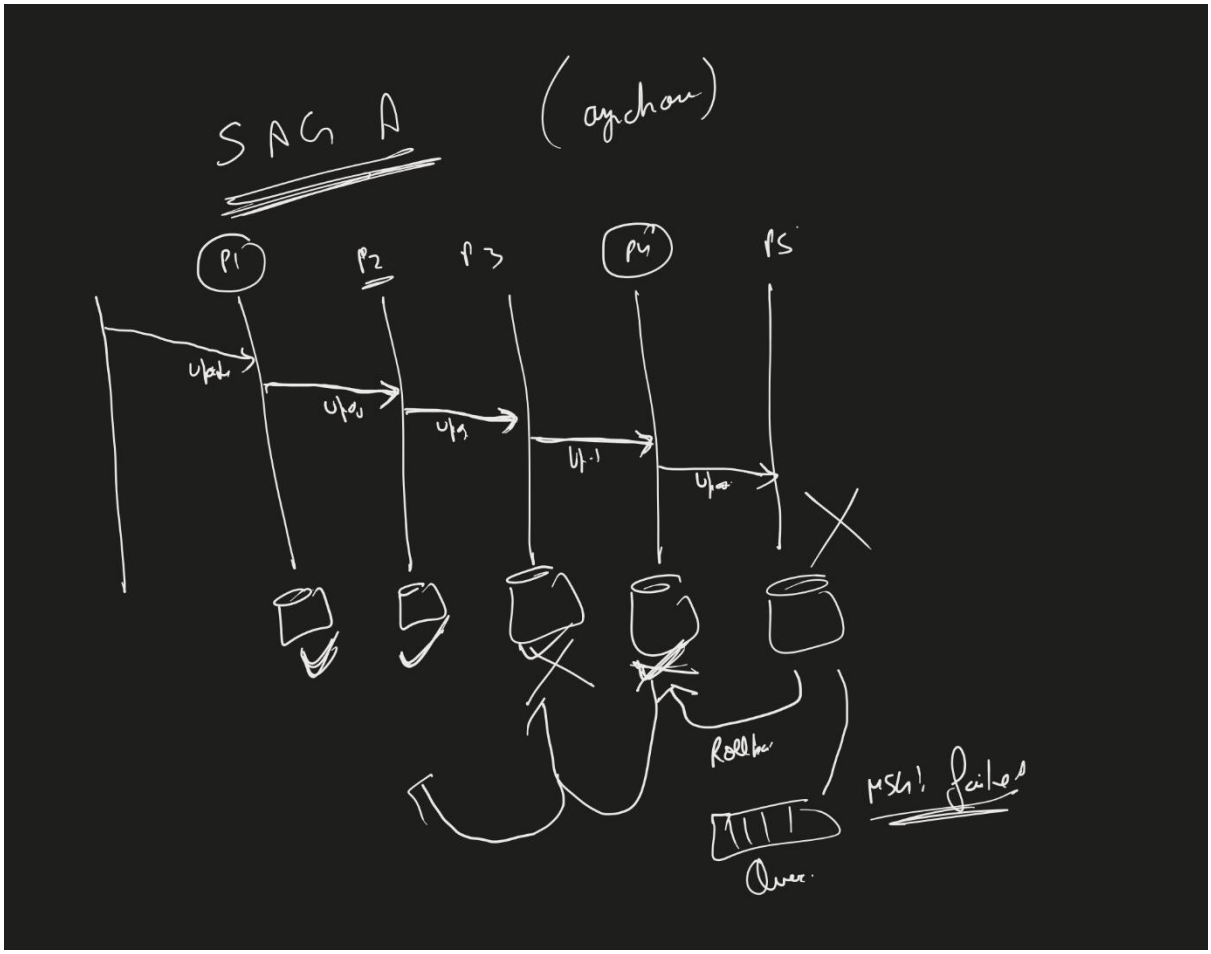
As name says, there are 2 phases in this protocol:

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

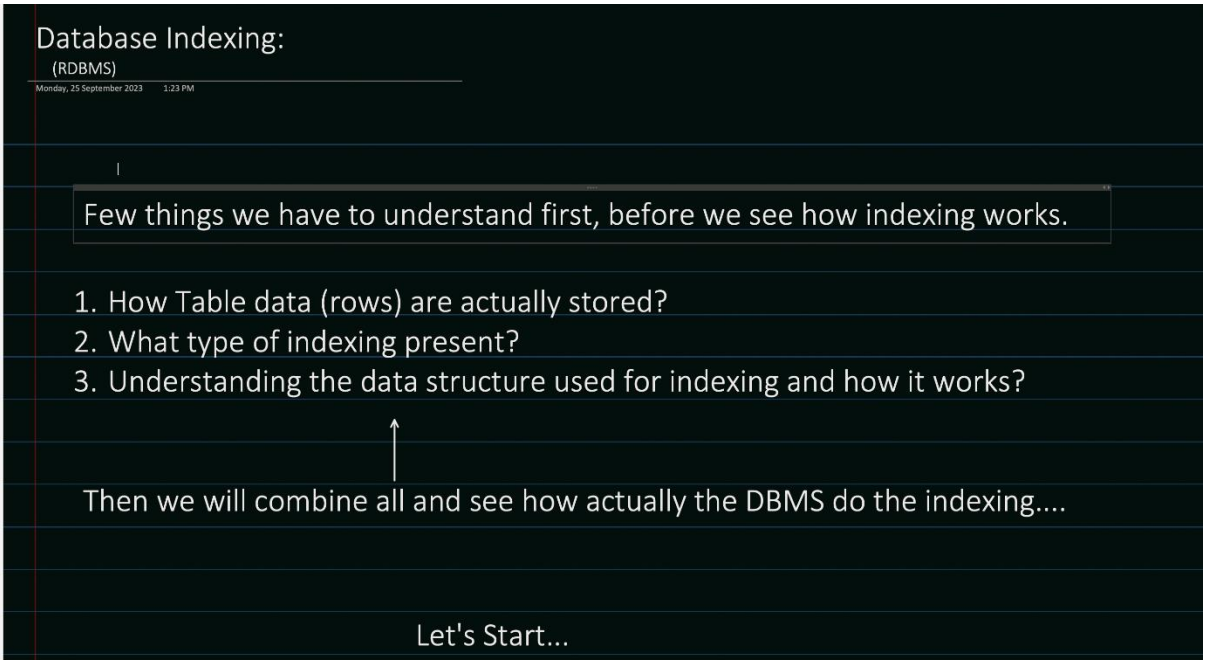

Application
Transaction Coordinator
Participant1 (Order Microservice)
Participant2 (Inventory Microservice)
Perform This Operations
START TRANSACTION
Phase1 Begins:
PREPARE
PREPARE
OK
Phase2 Begins:
COMMIT
COMMIT
DONE
DONE
Close Transaction


3-Phase Commit:
(Non-Blocking Protocol)
As name says, there are 2 phases in this protocol:
1. Prepare Phase
2. Pre-Commit Phase
3. Commit Phase


Application
Transaction Coordinator
Participant1 (Order)
Participant2 (Inventory)
Perform This Operations
START TRANSACTION
Phase1 Begins:
PREPARE
PREPARE
OK
OK
Phase2 Begins:
Phase3 Begins:
COMMIT
COMM
DONE
DONE
CLOSE TRANSACTION



DATABASE INDEXING



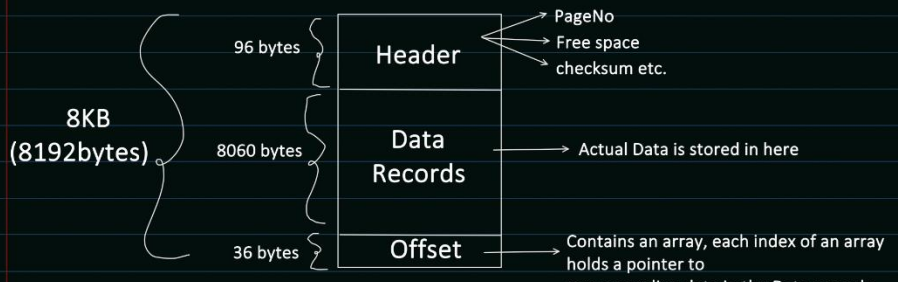
1. How Table data is actually stored?

	Column1	Column2	Column3
	Emp ID	Name	Address
Row1	1	A	City A
Row2	2	B	City B
Row3	3	C	City C
Row4	4	D	City D

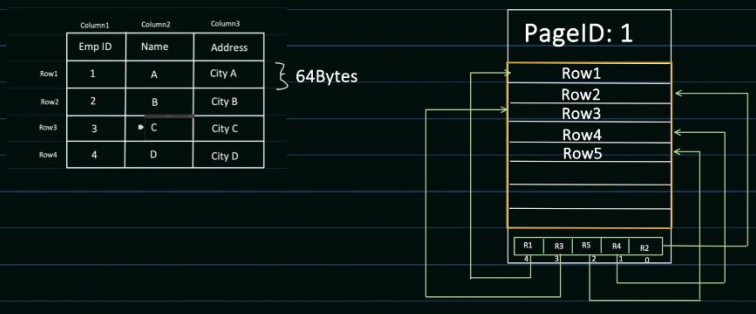
This is just a logical representation.
Actual Data is not ✗ stored in this way.

- DBMS creates Data Pages (generally its 8KB but depends upon DB to DB)
- Each Data Page can store multiple table rows in it.

Just an example of Data page.



So if 1 table row is lets say 64Bytes, so in 1 data page we can have ~125 DB rows



DBMS creates and manage these data pages. As for storing 1 table data, it can create many data pages.

These data pages ultimately gets stored in the Data Block in physical memory like disk.

What is Data block?

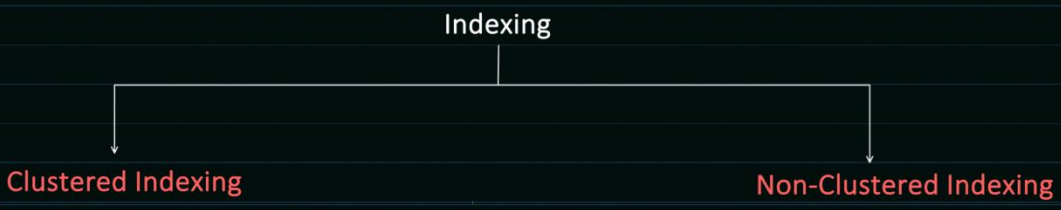
- Data Block is the minimum amount of data which can be read/write by an I/O operation.
- It is managed by underlying storage system like disk. Data Block size can range from 4kb to 32kb (common size is 8KB).
- So based on the data block size, it can hold 1 or many Data page.

DBMS maintains the mapping of DataPage and Data Block.

Data Page1	Data Block 1
Data Page2	Data Block 1
Data Page3	Data Block 2
Data Page4	Data Block 2

Remember, DBMS controls Data pages(like what Row goes in which page or sequence of pages etc.) but has no control on Data Blocks (data blocks can be scattered over the disk)

2. What type of indexing present in RDBMS?



Text Mode

Now before we understand Clustered Indexing and Non-Clustered Indexing, we have to first understand,
What is Indexing?
What Data Structure it uses and how it works?

Indexing:

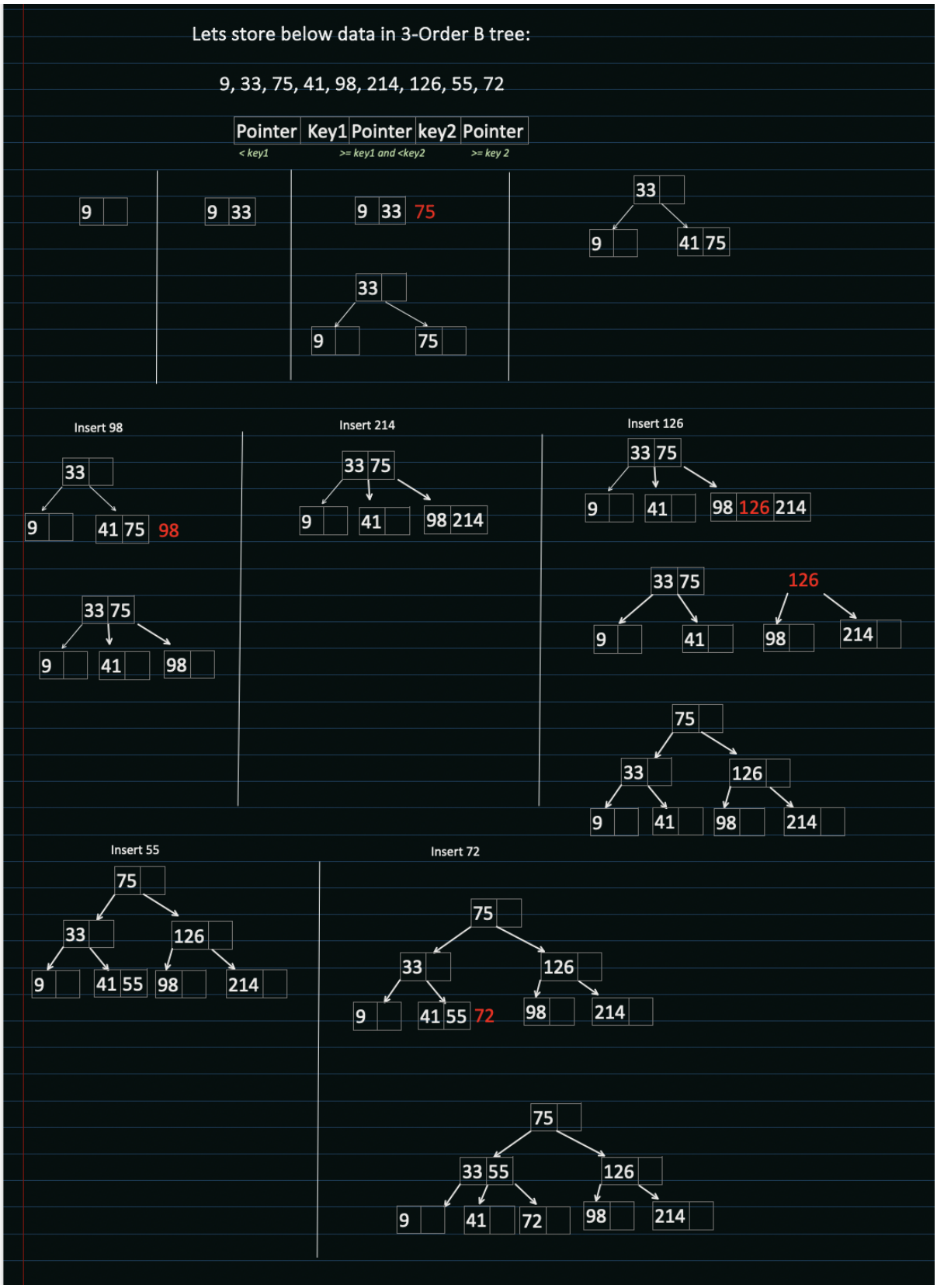
It is used to increased the performance of the database query. So that data can be fetched faster.
Without indexing, DBMS has to iterate each and every table row to find the requested data.
i.e O(N) , if there are millions of rows, query can take some time to fetch the data.

Which Data Structure provides better time complexity then O(N) ?

B+ Tree, it provides O(log N) time complexity for insertion, searching & deletion.

How B (Balanced) Tree works?

- It maintains sorted data.
- All leaf are at the same level
- M order B tree means, each node can have at most M childrens.
- And M-1 Keys per node.


Lets store below data in 3-Order B tree:
9, 33, 75, 41, 98, 214, 126, 55, 72
Pointer Key1 Pointer key2 Pointer
< key1
>= key1 and <key2
>= key 2
9
9 33
9 33 75
33
9
75
41 75
Insert 98
33
9
41 75 98
33 75
9
41
98
Insert 214
33 75
9
41
98 214
Insert 126
33 75
9
41
98 126 214
33 75
126
9
41
98
214
75
33
126
9
41
98
214
Insert 55
75
33
126
9
41 55
98
214
Insert 72
75
33
126
9
41 55 72
98
214
75
33 55
126
9
41
72
98
214

B+ tree is exactly same as B tree, with additional feature like child nodes are also linked to each other.

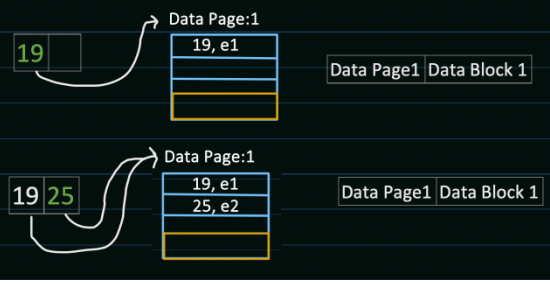
DBMS uses B+ Tree to manage its Data Pages and Rows within the pages.

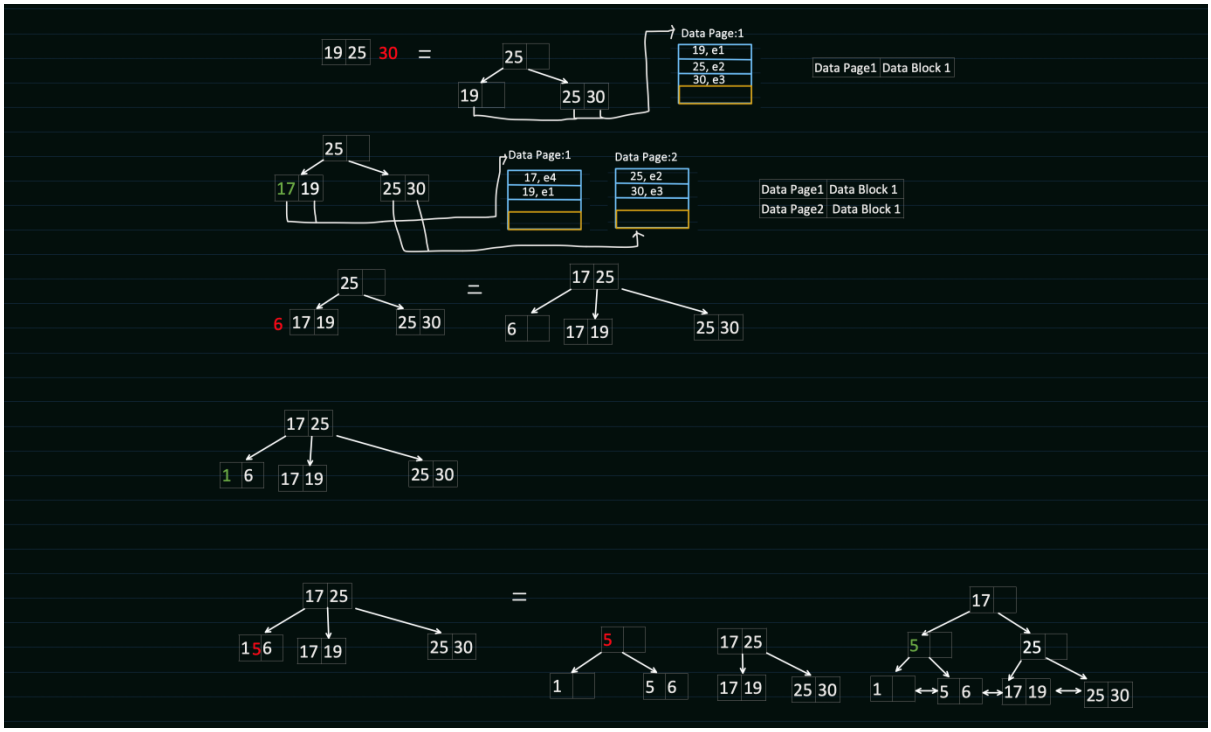
- Root node or Intermediary node hold the Value which is used for faster searching the data. Possible that value might have deleted from DB, but its can be used for sorting the tree.
- Leaf node actually holds the indexed column value.

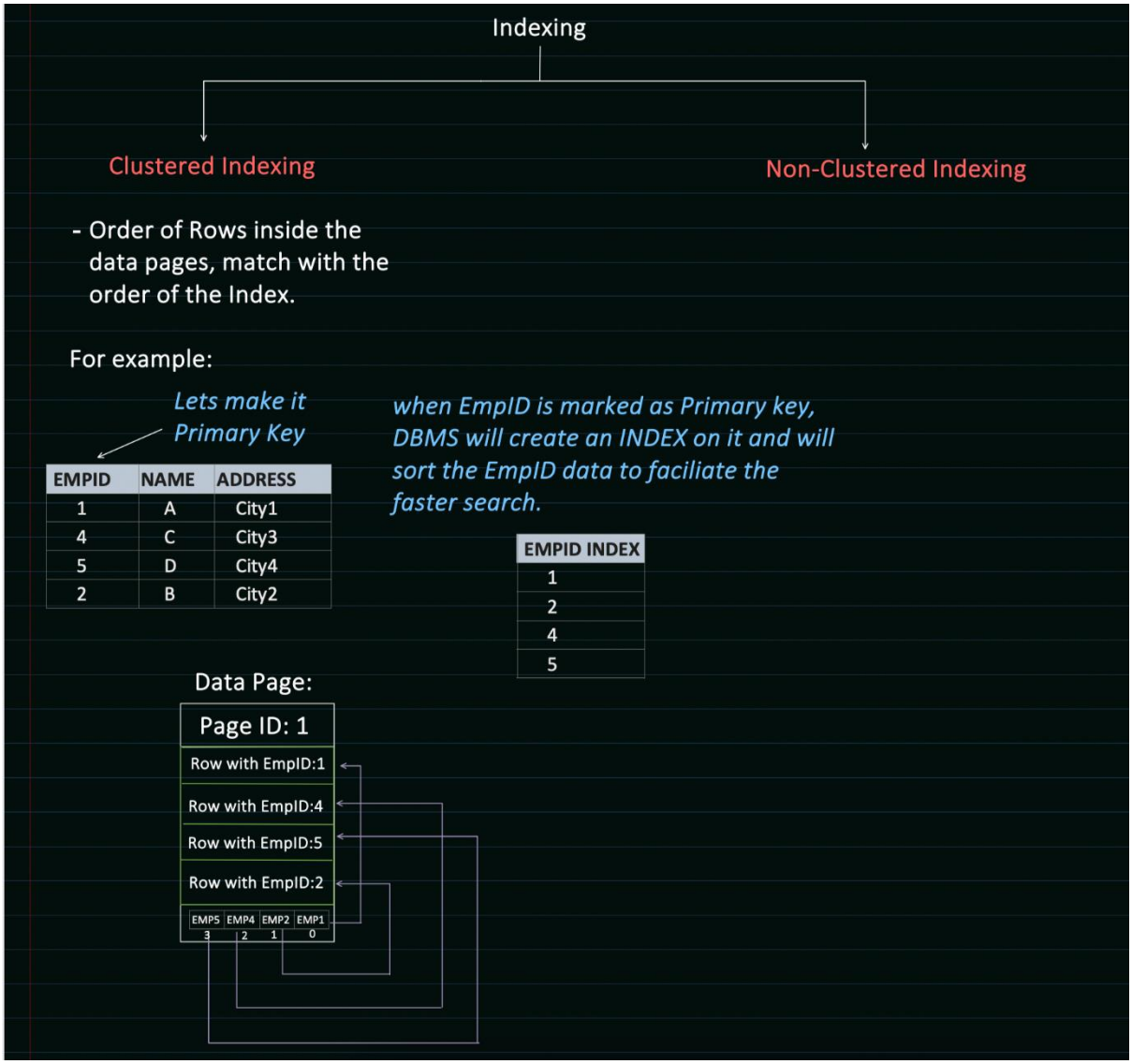
Indexed Column

EmpID	EmpName
19	e1
25	e2
30	e3
17	e4
6	e5
1	e6
5	e7

Same using 3 Order B+ Tree






Indexing
Clustered Indexing
Non-Clustered Indexing
- Order of Rows inside the data pages, match with the order of the Index.
For example:
Lets make it Primary Key
when EmpID is marked as Primary key, DBMS will create an INDEX on it and will sort the EmpID data to faciliate the faster search.
EMPID NAME ADDRESS
1 A City1
4 C City3
5 D City4
2 B City2
EMPID INDEX
1
2
4
5
Data Page:
Page ID: 1
Row with EmpID:1
Row with EmpID:4
Row with EmpID:5
Row with EmpID:2
EMP5 EMP4 EMP2 EMP1
3 2 1 0

So,

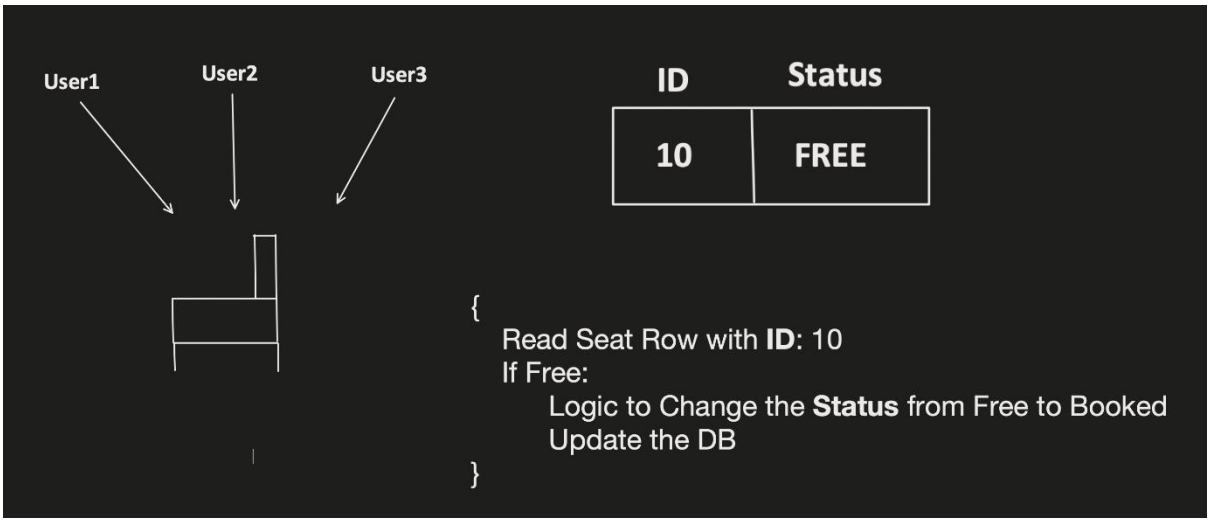
- There can be only 1 clustured index present / table bcoz ordering of pages can be done based on one index only.

- If manually you have not specified any Clustered index, then DBMS looks for PRIMARY KEY which is UNIQUE and NOT NULL and use it as a Clustured key.

- If in table there is no PRIMARY KEY avilable, then internally it creates a hidden Column which is used as Clustured index. (This columns just increse sequentially, so gurantted unique and not null)

LLD: Concurrency Control

Scenario: Many concurrent request tries to book same Movie theatre Seat



1. Using "SYNCHRONIZED" for the Critical Section:

```
synchronized ()
{
    Read Seat Row with ID: 10
    If Free:
        Logic to Change the Status from Free to Booked;
        Update the DB;
}
```

ID	Status
10	FREE

Will this solution works for Distributed System?

2. Using Distributed Concurrency Control

Optimistic Concurrency Control (OCC)

Pessimistic Concurrency Control (PCC)

But before we learning this, we MUST know below IMPORTANT things first.

1. What is the usage of **Transaction**?
2. What is **DB Locking**?
3. What are the **Isolation Level present**?

1. What is the usage of **Transaction**?

Transaction helps to achieve **INTEGRITY.** Means it help us to avoid **INCONSISTENCY** in our database.

For example:

Debit the Money from A and Credit the money to B

BEGIN_TRANSACTION:

- Debit the Money from A
- Credit the Money to B
If all success:
COMMIT;
else:
ROLLBACK;

END_TRANSACTION;

2. What is **DB Locking**?

DB Locking, help us to make sure that no other transaction update the locked rows.

Lock Type	Another Shared Lock	Another Exclusive Lock
Have Shared Lock	Yes	NO
Have Exclusive Lock	NO	NO

3. What are the **Isolation Level present**?

Isolation Level	*Dirty Read Possible*	*Non-Repeatable Read Possible*	*Phantom Read Possible*
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

Consistency High

Consistency Low

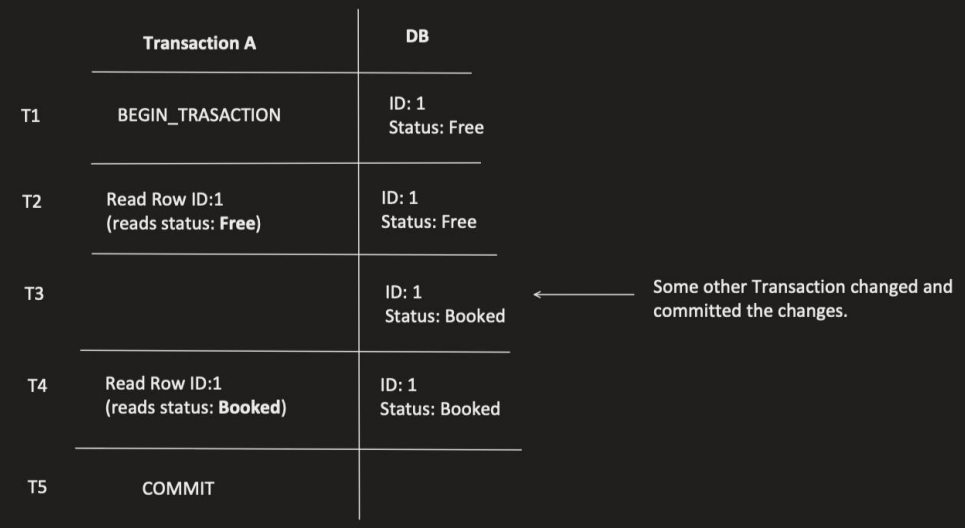
Dirty Read :

If Transaction A is reading the data which is writing by Transaction B and not yet even committed.
If Transaction B does the rollback, then whatever data read by Transaction A is know dirty read.

	Transaction A	Transaction B	DB
T1	BEGIN_TRASACTION	BEGIN_TRASACTION	ID: 1 Status: Free
T2	Update Row ID:1, Status: Booked		ID: 1 Status: Booked (Not Committed by Transaction B Yet)
T3		Read Row ID:1 (Got status as Booked)	ID: 1 Status: Booked (Not Committed by Transaction B Yet)
T4	Some Issue faced here	Some Computation	ID: 1 Status: Booked (Not Committed by Transaction B Yet)
T5	Rollback	Commit	ID: 1 Status: Free

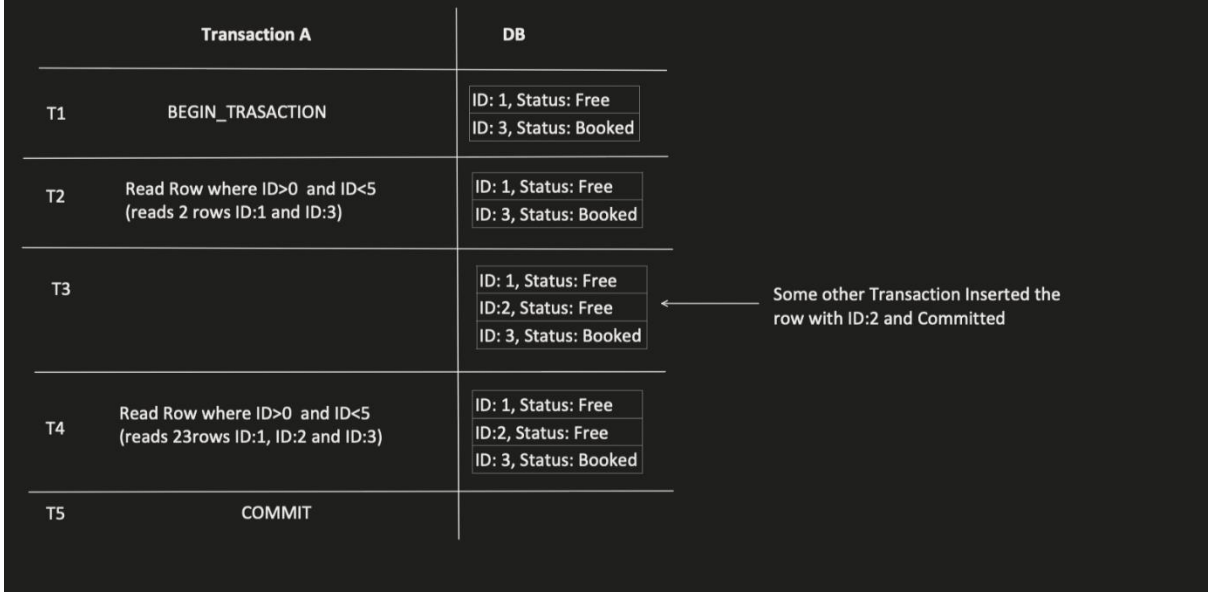
Non-Repeatable Read :

If suppose **Transaction A**, reads the same row several times and there is a chance that it reads different value.



Phantom Read :

If suppose **Transaction A**, executes same query several times and there is a chance that the rows retuned are different.



Lets come back to the Table Again:

Isolation Level	*Dirty Read Possible*	*Non-Repeatable Read Possible*	*Phantom Read Possible*
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

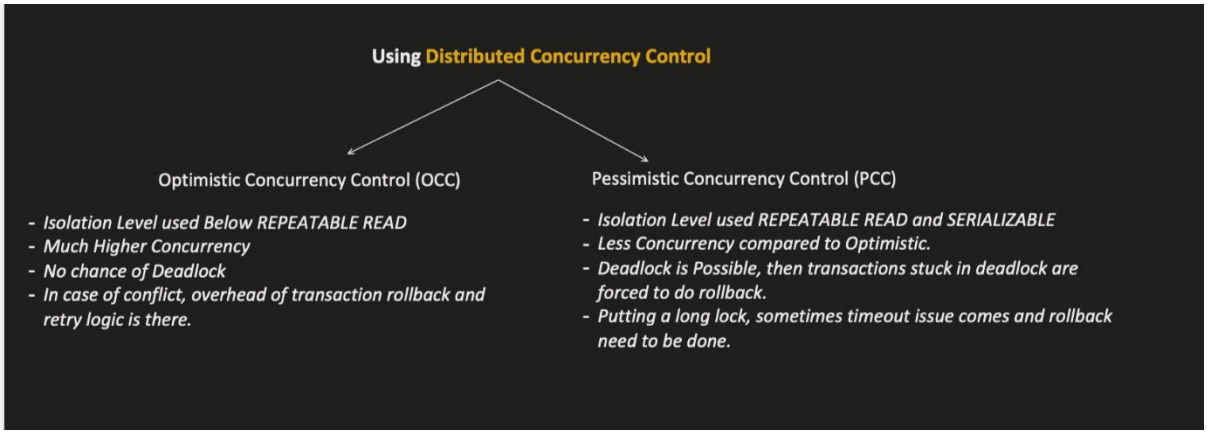
Consistency High

Consistency Low

ISOLATION LEVEL	Locking Strategy
Read Uncommitted	**Read** : No Lock acquired **Write**: No Lock acquired
Read Committed	**Read**: Shared Lock acquired and Released as soon as Read is done **Write**: Exclusive Lock acquired and keep till the end of the transaction
Repeatable Read	**Read**: Shared Lock acquired and Released only at the end of the Transaction **Write**: Exclusive Lock acquired and Released only at the end of the Transaction
Serializable	Same as Repeatable Read Locking Strategy + apply Range Lock and lock is release only at the end of the Transaction.

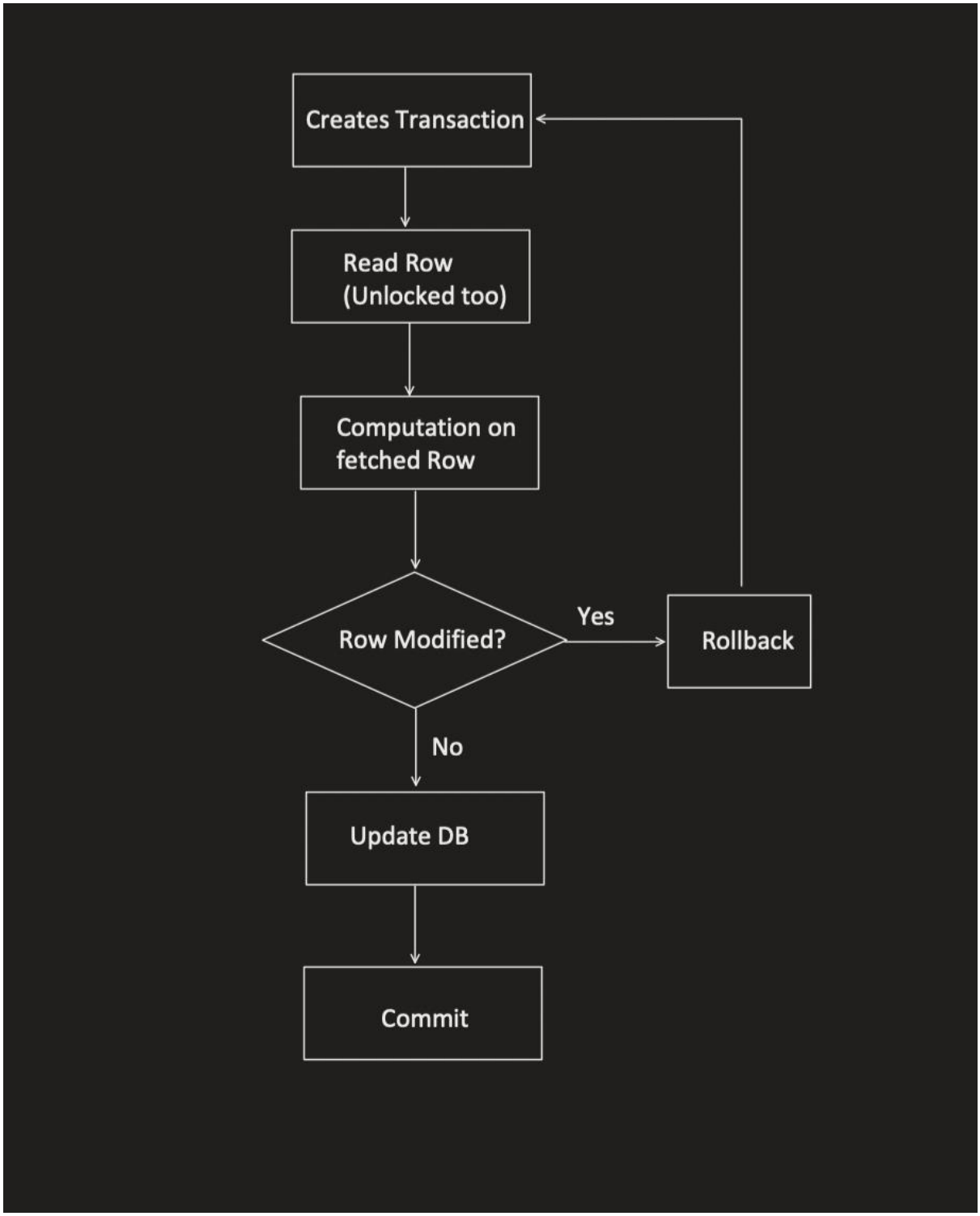
```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN_TRANSACTION;

SELECT Query;
Update Query;
If Success:
    COMMIT TRANSACTION;
else:
    ROLLBACK TRANSACTION;

END_TRANSACTION
```

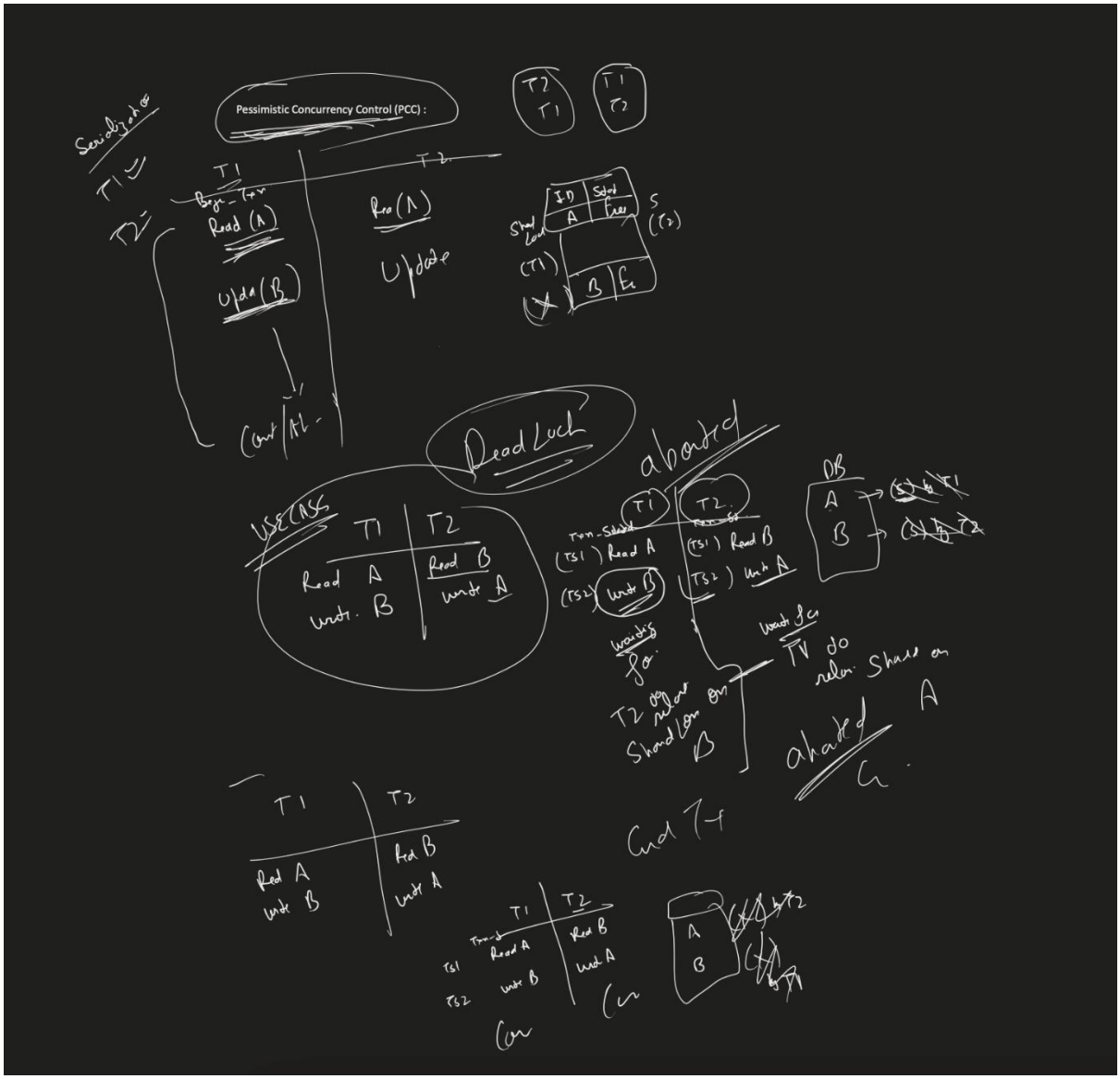
Lets come back to the main Solution of the problem "How to resolve concurrency issue"



Optimistic Concurrency Control (OCC) :

	Transaction A	Transaction B	DB
T1	BEGIN_TRASACTION	BEGIN_TRASACTION	ID: 1 Status: Free Version: 1
T2	Read Row ID:1 (Row Version is 1)	Read Row ID:1 (Row Version is 1)	ID: 1 Status: Free Version: 1
T3	Select for Update (Version Validation happens)		ID: 1 Status: Free (Exclusive Lock by Txn A) Version: 1
T4	Update Row ID:1, Status: Booked, Version:2		ID: 1 Status: Booked (Exclusive Lock by Txn A) Version:2
T5	COMMIT		ID: 1 Status: Booked Version:2
T6		Select for Update (Version Validation happens)	ID: 1 Status: Booked (Exclusive Lock by Txn B) Version:2
T7		ROLLBACK	ID: 1 Status: Booked Version:2


Creates Transaction
Read Row
(Unlocked too)
Computation on
fetched Row
Row Modified?
Yes
Rollback
No
Update DB
Commit


Pessimistic Concurrency Control (PCC) :
Read Lock
aborted
Read A
Write B
Read B
Write A

Two Phase Locking (2PL)

Before this video, do checkout the previous ***"Distributed Concurrency"*** video in this ***HLD*** playlist....

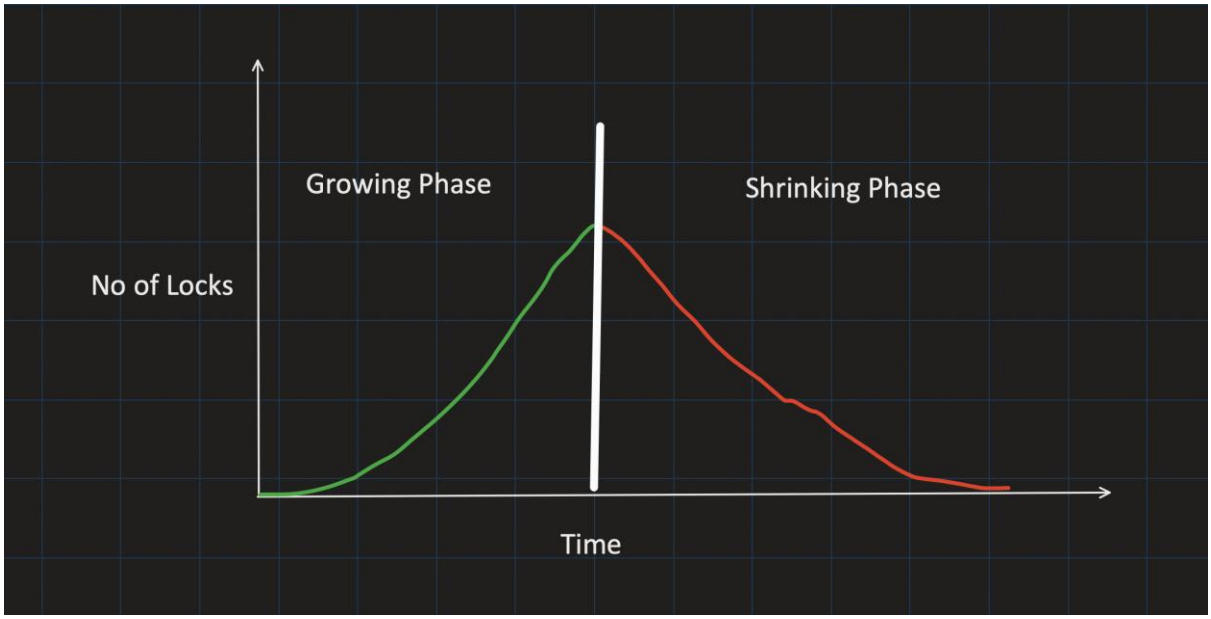
Basic Two Phase Locking:

Phase1: Growing Phase

- Txn request for the lock by Lock manager.
- Lock Manager either grant or denied the lock request *(denied if other txn has placed the locked on it already)*

Phase2: Shrinking Phase

- Txn can not acquire any new locks.
- Txn is only allowed to release the lock which is taken previously.



Time	Transaction 1	Transaction2
t0	BEGIN TXN	
t1	X(A)	
t2	X(B)	
t3	process some work	
t4	Unlock (A)	BEGIN TXN
t5	Unlock (B)	X(B)
t6	COMMIT	X(A)
t7		COMMIT

(commit will unlock A and B internally)

2 big issue of Basic 2PL and its way around:

1. Deadlock:

Time	Transaction 1	Transaction2
t0	BEGIN TXN	BEGIN TXN
t1	X(A)	X(B)
t2	X(B) (waiting for lock to be released on B)	X(A) (waiting for lock to be released on A)
t3		

OR

Time	Transaction 1	Transaction2
t0	BEGIN TXN	BEGIN TXN
t1	S(A)	S(A)
t2	X(A) (lock conversion only happens if Txn2 Read lock is released)	X(A) (lock conversion only happens if Txn1 Read lock is released)
t3		

Deadlock Prevention Strategies:

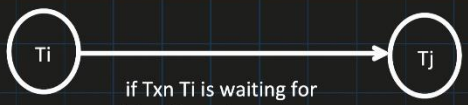
1. Timeout:

In this strategy, scheduler find out that TXN is waiting too long for the lock, it simply assumes that there might be a deadlock involving this transaction and thus abort it.

Schedule can make mistake (what if TXN1 wait for the lock, which is acquired by other TXN2 which is taking just long time to finish).

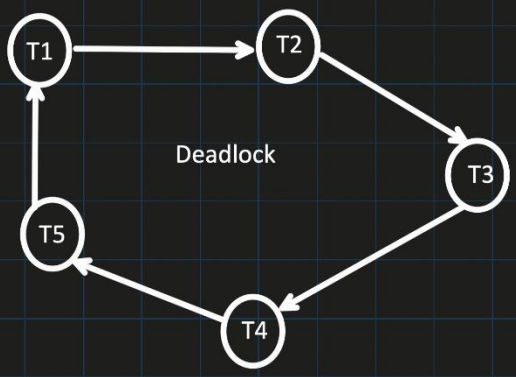
2. Direct graph called Wait-for-Graph(WFG):

There will be an edge from node Ti to Tj .



if Txn Ti is waiting for Txn Tj to release some lock

Scheduler delete the edge, whenever lock is release by particular txn which causing some other txn to wait.



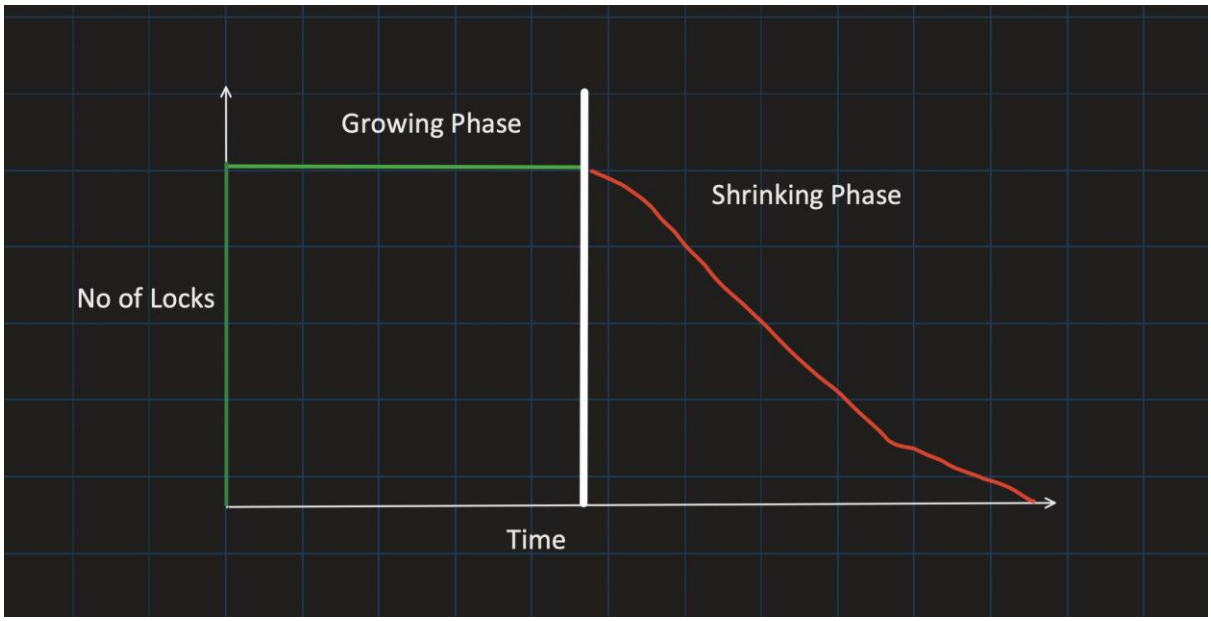
Scheduler looks for cycle in the WFG and try to identify the deadlock.

When deadlock is identified, TXN is choosen from the cycle in WFG that need to be ABORTED (these txns are called victim)

Scheduler check below things to identify the victim:
- The amount of effor txn has put till now
- The amount of effort required to finish the txn
- The cost of Aborting the txn (cost generally means here is how many updates already has been done)
- The number of cycles that contains the transaction.

3. Conservative 2PL (*also know as Static 2PL*):

- Avoid deadlock by requiring each txn to aquire all the locks at start of the txn itself.
- Each txn, predaclared its Read and Write operation to the scheduler.
- Scheduler tries to set all the lock needed by txn.
- If Scheulder fails to acquire any lock, none of the lock will be granted to Txn and it will wait.



Cons of Conservative 2PL:

- Less concurrency
- Extra overhead for Scheduler to know all the Read and Write operation of txn before starting the operation.

4. TimeStamp based Deadlock detection:

Old TimeStamp of Txn = High Priority of the Txn

- Wait-Die :

- Old txn waits for the New txn
- ts(T1) < ts(T2), then T1 will wait for T2 to get completed, otherwiase txn will get aborted.

Wound-Wait:

- Old txn give wound to the new txn and make them aborted

ts(T1) < ts(T2), then T2 abort and releases its lock otherwise txn will wait.

T1	T2
Begin	
	Begin
	X(A)
X(A)	

ts(T1) < ts(T2) = T1 has higher priority than T2

Wait-Die = T1 will wait

Wound-Wait = T2 will be aborted

T1	T2
Begin	
X(A)	
	Begin
	X(A)

ts(T1) < ts(T2) = T1 has higher priority than T2

- Wait-Die = T2 will abort

Wound-Wait = T2 will wait

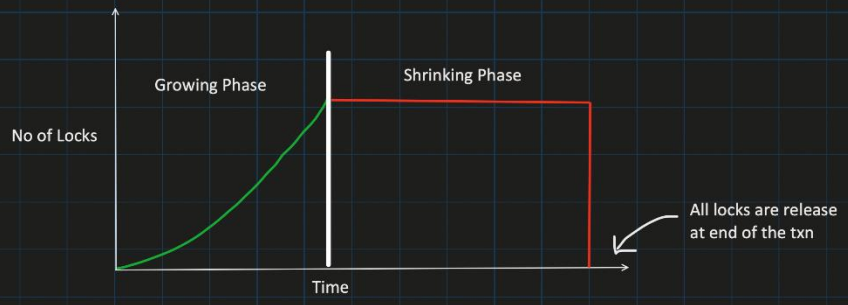
2. Cascading Aborts:

Time	Transaction 1	Transaction2
t0	BEGIN TXN	BEGIN TXN
t1	X(A)	
t2	X(B)	
t3	Update the value of A	
t4	Unlock(A)	
t5		X(A)
t6		Some operation
t7	ABORT	
t8		
t9		This has to be ABORTED Too, as it read the dirty value of A

Cascading Aborts Prevention Strategy:

1. Strong Strict 2PL *(also know as Rigorous 2PL):*

- Txn is not allowed to acquire or upgrade more lock after growing phase.
- Release all the locks at the end of the txn only.



Cons of Stict 2PL:

- Less concurrency
- Deadlock

Example:

T1: Send 10Rs from A to B

T2: Sum of A balance + B balance

Initial DB values =
A = 100, B = 100

Basic 2PL: (deadlock and cascading abort, both issue present)

Time	Transaction 1	Transaction2
t0	BEGIN TXN	BEGIN TXN
t1	X(A)	
t2	A = A-10 i.e (100-10)	
t3	Update A value	
t4	Unlock(A)	
t5		S(A)
t6		Read value of A i.e 90
t7		Unlock(A)
t8		S(B)
t9		Read value of B i.e 100
t10		Unlock(B)
t11	X(B)	COMMIT
t12	B = B + 10	
t13	Update B value	
t14	Unlock(B)	
t15	Comit	

Output of T2:
A + B = 90+100 = 190

Conservative 2PL: (cascading abort issue present)

Initial DB values =
A = 100, B = 100

Time	Transaction 1	Transaction2
t0	BEGIN TXN	BEGIN TXN
t1	X(A)	
t2	X(B)	
t3	A = A-10 i.e (100-10)	
t4	B = B+10 i.e (100+10)	
t5	Update B value	
t6	Unlock (A)	
t7		S(A)
t8		S(B) (waiting.....)
t9		(waiting.....)
t10	Unlock (B)	S(B) (Success)
t11	Commit	Read value of A and B
t12		Unlock (A)
t13		Unlock (B)
t14		Commit
t15		

Output of T2:
A + B = 90+110 = 200

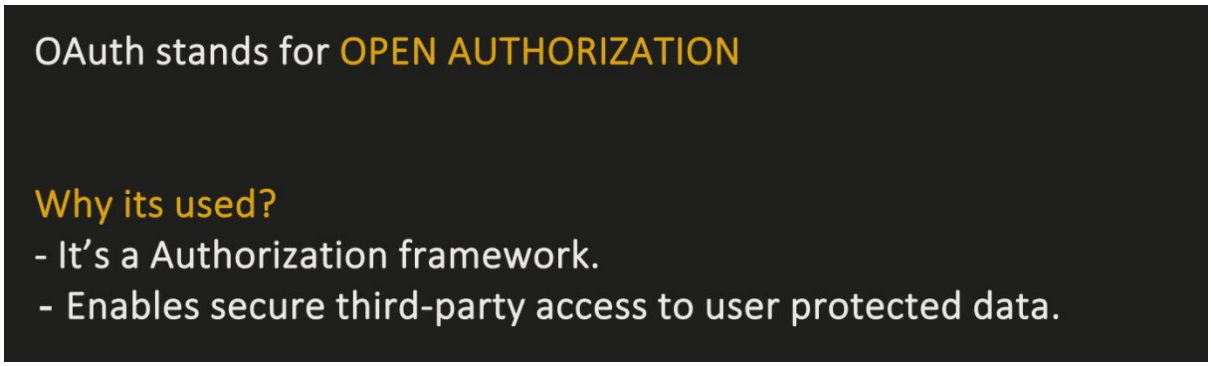
Strong Strict 2PL:

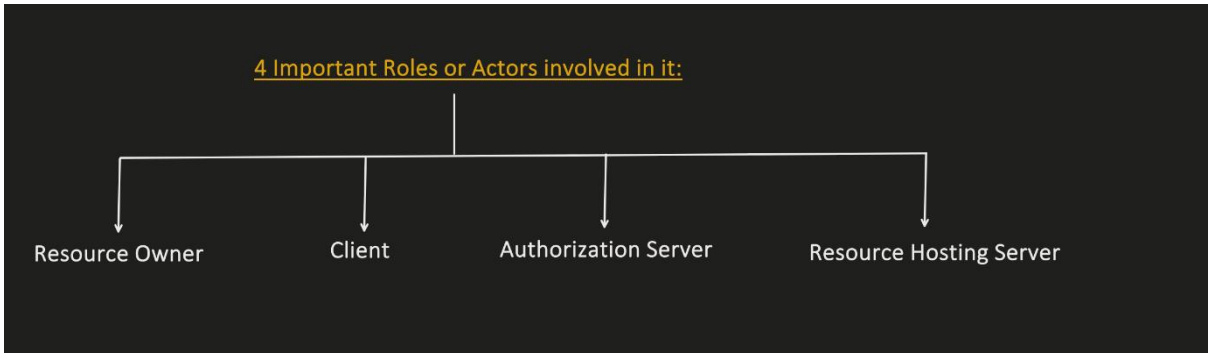
Initial DB values =
A = 100, B = 100

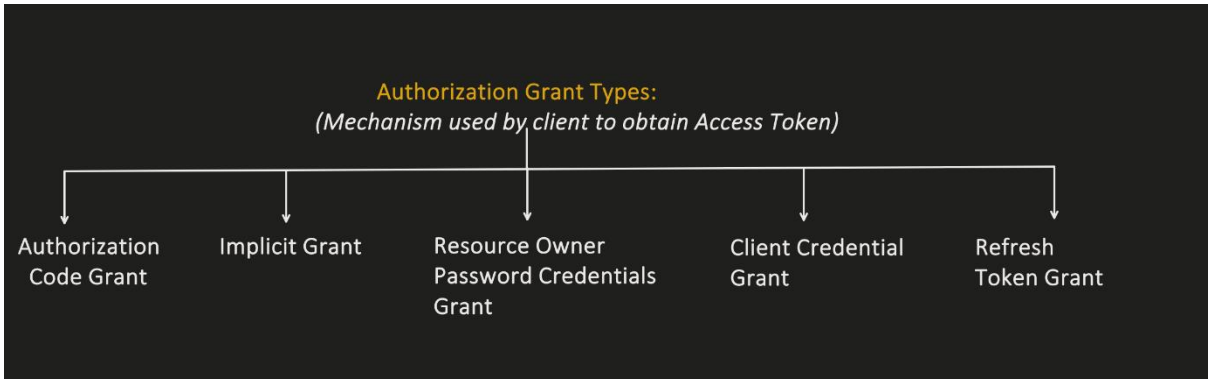
Time	Transaction 1	Transaction2
t0	BEGIN TXN	BEGIN TXN
t1	X(A)	
t2	A = A-10 i.e (100-10)	S(A) (waiting...)
t3	Update A value	
t4	X(B)	
	b = B+10 i.e (100+10)	
t5	Update B value	
t6	Commit	
t7		S(A) (Success)
t8		Read A value
t9		S(B)
t10		Read B Value
t11		Commit
t12		
t13		
t14		
t15		

Output of T2:
A + B = 90+110 = 200

HLD: OAuth 2.0


OAuth stands for OPEN AUTHORIZATION
Why its used?
- It's a Authorization framework.
- Enables secure third-party access to user protected data.


4 Important Roles or Actors involved in it:
Resource Owner
Client
Authorization Server
Resource Hosting Server


Authorization Grant Types:
(Mechanism used by client to obtain Access Token)
Authorization Code Grant
Implicit Grant
Resource Owner Password Credentials Grant
Client Credential Grant
Refresh Token Grant

Lets see sample API request and response:

NOTE: pls follow the official documentation for exact API names and more details about request and response.

1. Registration Process:

Request:

POST /register

body:

```
{
   "client_name" : {my app name},
   "redirect_uris" : ["https://mysamplewebsite.com/", "https://mysamplewebsite.com/"]
}
```

Response:

```
{
   "client_name" : {my app name},
   "client_id" : {unique id assigned to client}
   "client_secret" : {Confidential info, known only to client and authorization server, use for client authentication}
}
```

2. Fetch Authorization Code:

Request:

GET /authorize?

response_type=code

&client_id={id which client gets during registration}

&redirect_uri={callback URI like https://mysamplewebsite.com/callback}

&scope=scope1 scope2 scope3

&state={some random value like sj111}

Query Parameters:

- response_type
  - REQUIRED
  - Value must be set to "code".
  - This field specifies the desired response type from the authorization server.
- client_id
  - REQUIRED
- redirect_uri
  - OPTIONAL
  - If provided, it should match, with one of URI provided during registration
  - If not provide, one will be picked from list provided during registration.
- scope
  - OPTIONAL
  - Space separated list
  - If not provided, each authorization server has their own default scope, that will be used.
  - Some authorization server also provide option to define "default scope" during registration.
- state:
  - RECOMMENDED (but not mandatory)
  - Unique Random no, sent to authorization server and server echo it back to client.
  - PREVENT CSRF ATTACKs

Response (after user authenticate and provide consent to authorization server):

https://mysamplewebsite.com/callback?code={Authorization code here}
&state=sj111

Good Practice:

- Authorization code should be used only Once.
- If code used more than once, Authorization server deny the request and also make all tokens INVALID which is based on this authorization code.
- All Authorization code should be short lived (10 minutes recommended).

3. Fetch Token:

Request:

POST /token?
grant_type=authorization_code
&code={authorization code here}
&redirect_uri={callback URI like https://mysamplewebsite.com/callback}
&client_id={id which client get during registration}
&client_secret={secret which client get during registration}

Response:

```
{
    "access_token": {access token value here},
    "token_type":"Bearer",
    "expires_in": 3600,
    "refresh_token": {refresh token value here},
}
```

Bearer: is a security mechanism, and it means that client should add the TOKEN in the authorization header, whenever it want to access the protected resources.

Token Value: it can either be JWT token or plain string.

Expiry Time : generally its in seconds. After expiry Refresh token should be used to get new Tokens.

4. Generate New Token:

Request:

POST /token?
grant_type=refresh_token
&refresh_token={refresh token value here}
&redirect_uri={callback URI like https://mysamplewebsite.com/callback}
&client_id={id which client get during registration}
&client_secret={secret which client get during registration}

Response:

```
{
    "access_token": {new access token value here},
    "token_type":"Bearer",
    "expires_in": 3600,
    "refresh_token": {new refresh token value here},
}
```

Implicit Grant:

Request:

GET /authorize?
response_type=token
&client_id={id which client gets during registration}
&redirect_uri={callback URI like https://mysamplewebsite.com/callback}
&scope=scope1 scope2 scope3
&state={some random value like sj111}

Response:

https://mysamplewebsite.com/callback?
access_token={access token value here}
&token_type=bearer
&expires_in=3600

There is no REFRESH TOKEN in Implicit Grant.

Resource Owner Password Credentials Grant:

Request:

GET /token?
grant_type=password
&client_id={id which client gets during registration}
&client_secret={secret which client get during registration}
&username=owner Username
&password=owner Password
&scope=scope1 scope2 scope3

Response:

```
{
  access_token={access token value here}
  &token_type=Bearer
  &expires_in=3600
  &refresh_token={refresh token value here}
}
```

- There is no **separate Authorization call** in ROPC Grant and
- Username and Password is not required during refresh token usage

Client Credentials Grant:

Request:

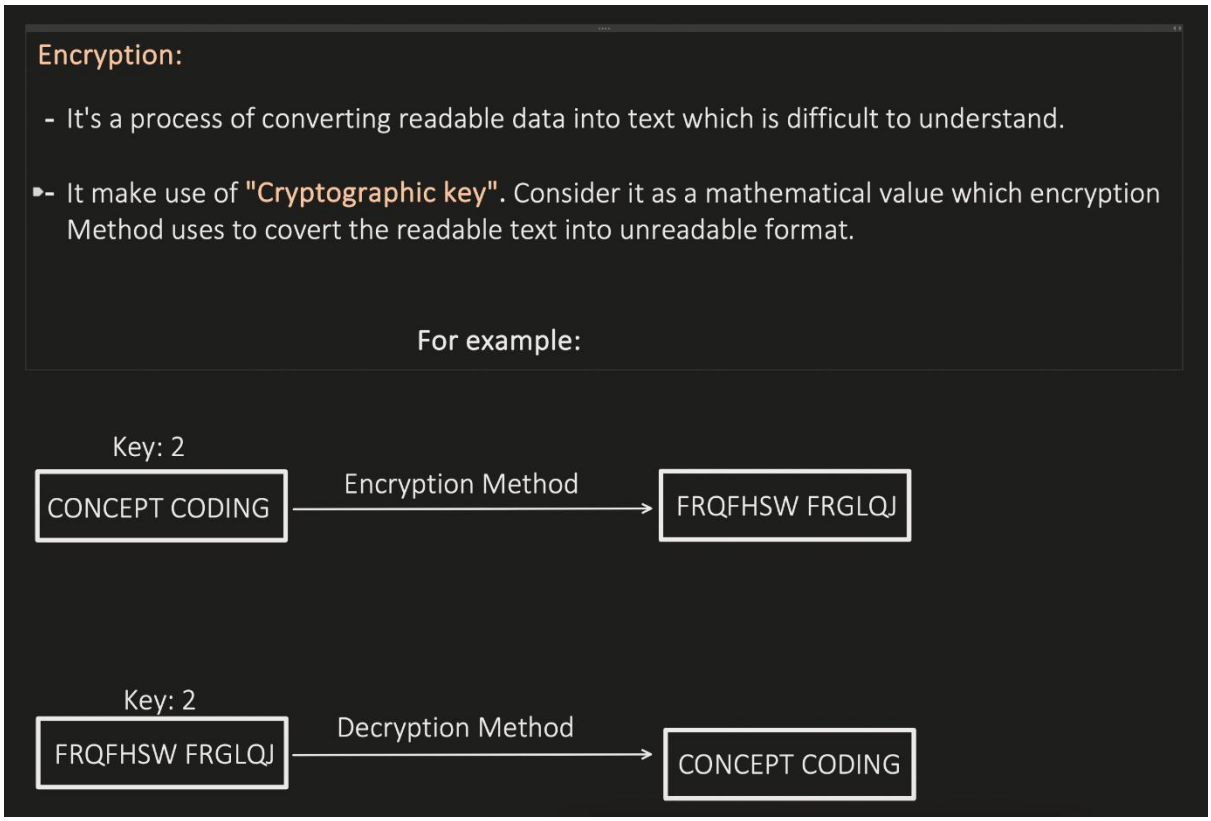
GET /token?
grant_type=client_credentials
&client_id={id which client gets during registration}
&client_secret={secret which client get during registration}
&scope=scope1 scope2 scope3

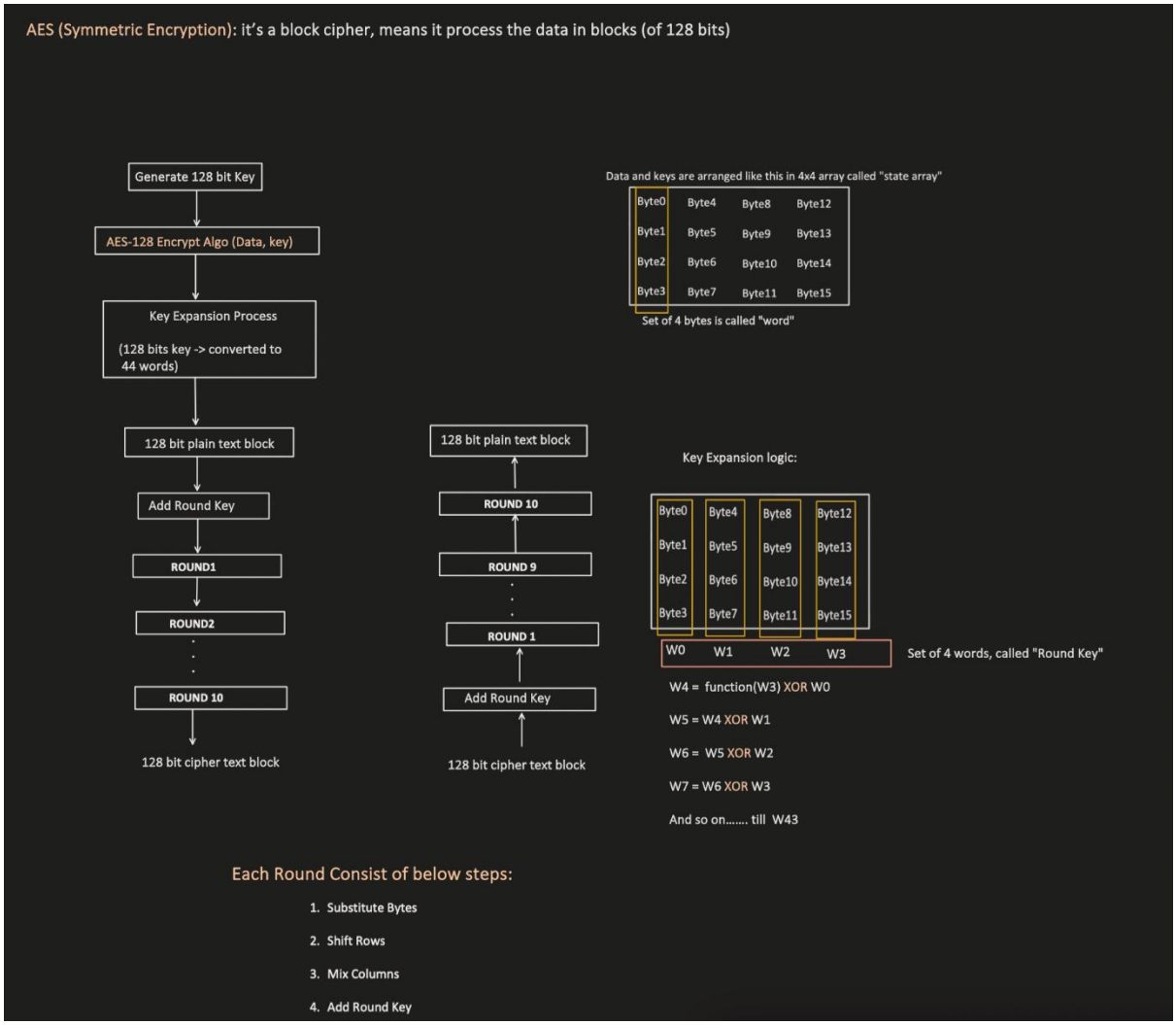
Response:

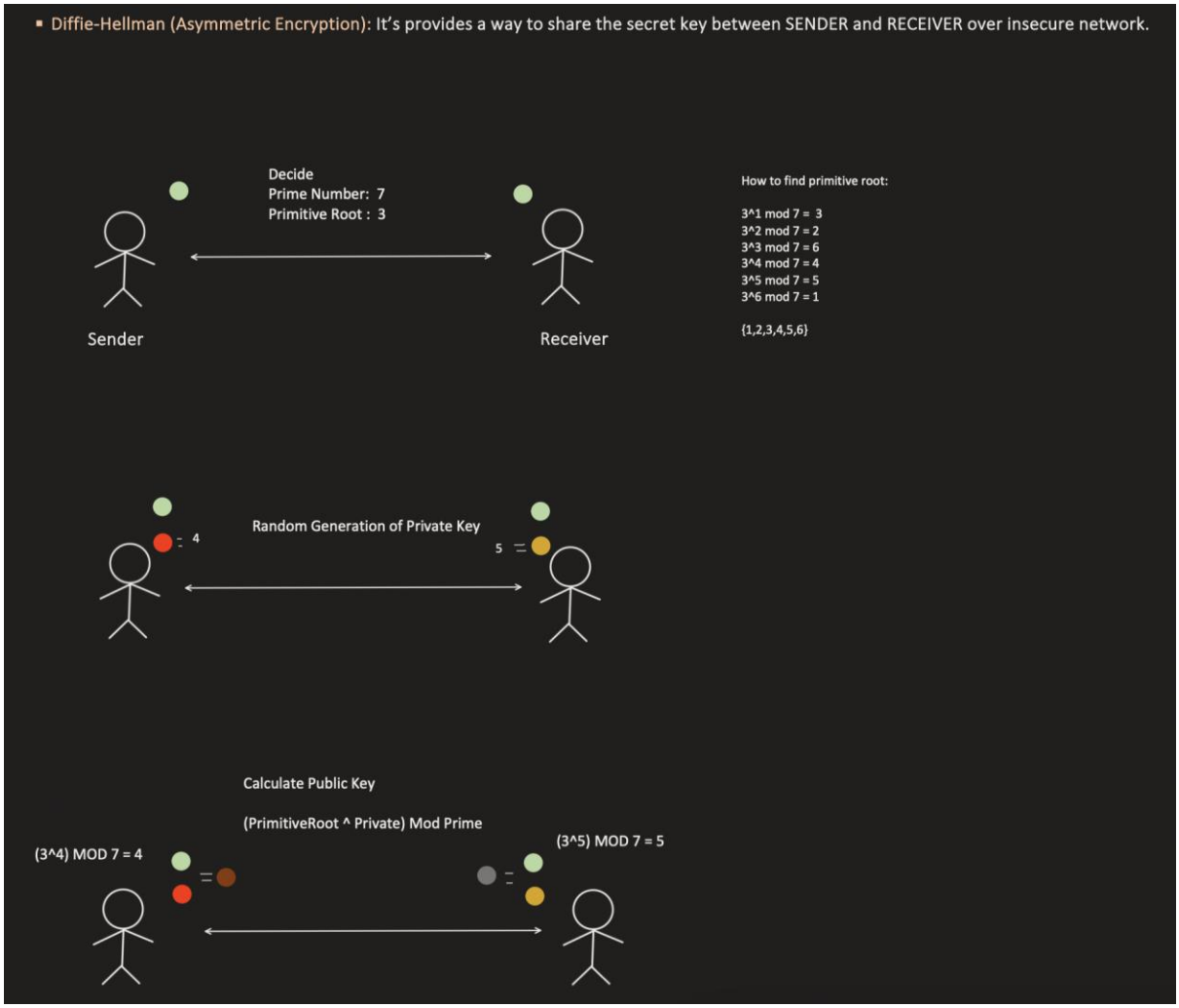
```
{
   access_token={access token value here}
   &token_type=Bearer
   &expires_in=3600
}
```

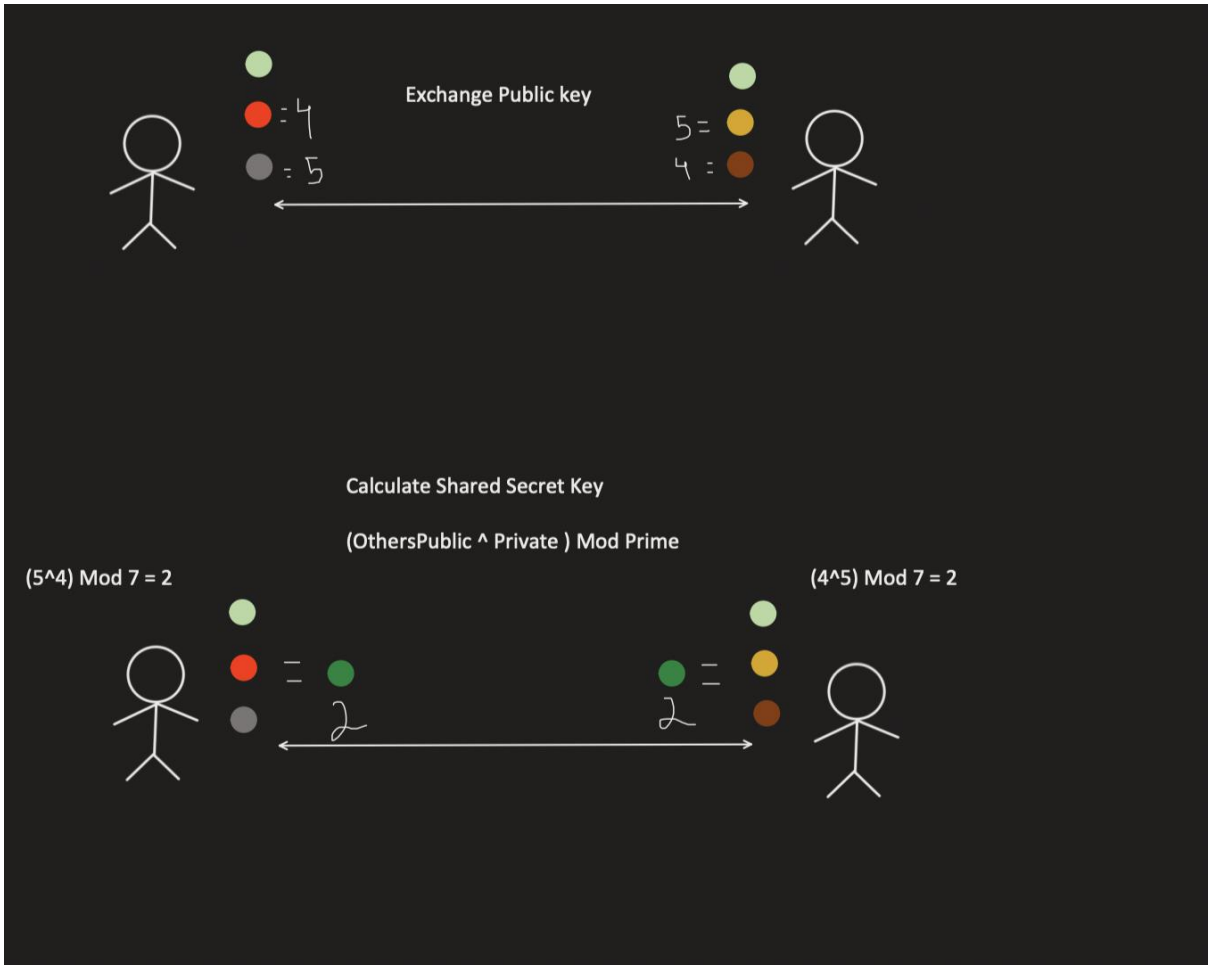
- There is no **Refresh token required in this.**
- There is no Authorization call too.

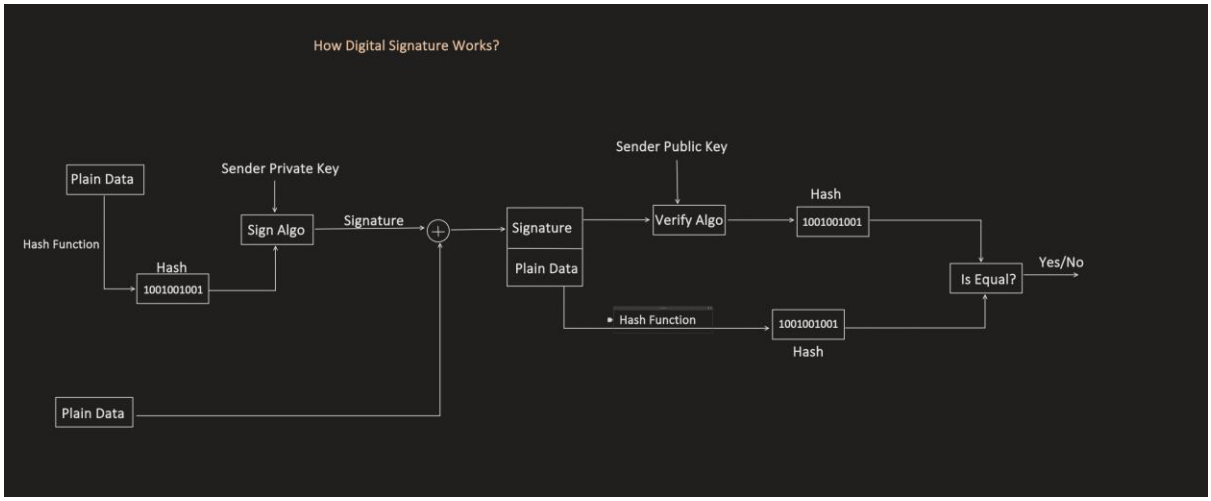
Cryptography




AES (Symmetric Encryption): it's a block cipher, means it process the data in blocks (of 128 bits)
Generate 128 bit Key
AES-128 Encrypt Algo (Data, key)
Key Expansion Process
(128 bits key -> converted to 44 words)
128 bit plain text block
Add Round Key
ROUND1
ROUND2
ROUND 10
128 bit cipher text block
Data and keys are arranged like this in 4x4 array called "state array"
Byte0 Byte4 Byte8 Byte12
Byte1 Byte5 Byte9 Byte13
Byte2 Byte6 Byte10 Byte14
Byte3 Byte7 Byte11 Byte15
Set of 4 bytes is called "word"
128 bit plain text block
ROUND 10
ROUND 9
ROUND 1
Add Round Key
128 bit cipher text block
Key Expansion logic:
Byte0 Byte4 Byte8 Byte12
Byte1 Byte5 Byte9 Byte13
Byte2 Byte6 Byte10 Byte14
Byte3 Byte7 Byte11 Byte15
W0 W1 W2 W3
Set of 4 words, called "Round Key"
W4 = function(W3) XOR W0
W5 = W4 XOR W1
W6 = W5 XOR W2
W7 = W6 XOR W3
And so on..... till W43
Each Round Consist of below steps:
1. Substitute Bytes
2. Shift Rows
3. Mix Columns
4. Add Round Key

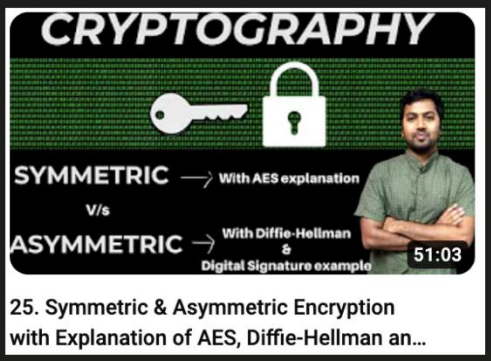

Diffie-Hellman (Asymmetric Encryption): It's provides a way to share the secret key between SENDER and RECEIVER over insecure network.
Decide
Prime Number: 7
Primitive Root : 3
Sender
Receiver
How to find primitive root:
3^1 mod 7 = 3
3^2 mod 7 = 2
3^3 mod 7 = 6
3^4 mod 7 = 4
3^5 mod 7 = 5
3^6 mod 7 = 1
{1,2,3,4,5,6}
Random Generation of Private Key
4
5
Calculate Public Key
(PrimitiveRoot ^ Private) Mod Prime
(3^4) MOD 7 = 4
(3^5) MOD 7 = 5


Exchange Public key
: 4
: 5
5 =
4 :
Calculate Shared Secret Key
(OthersPublic ^ Private) Mod Prime
(5^4) Mod 7 = 2
(4^5) Mod 7 = 2
=
2
=
2


How Digital Signature Works?
Plain Data
Sender Private Key
Sender Public Key
Hash Function
Sign Algo
Signature
Signature
Plain Data
Verify Algo
Hash
1001001001
Hash
1001001001
Is Equal?
Yes/No
Hash Function
1001001001
Hash
Plain Data

JWT(JSON Web Token)

24. OAuth 2.0: Explained with API Request and Response Sample | High Level System...



25. Symmetric & Asymmetric Encryption with Explanation of AES, Diffie-Hellman an...

What is JWT (JSON Web Token)

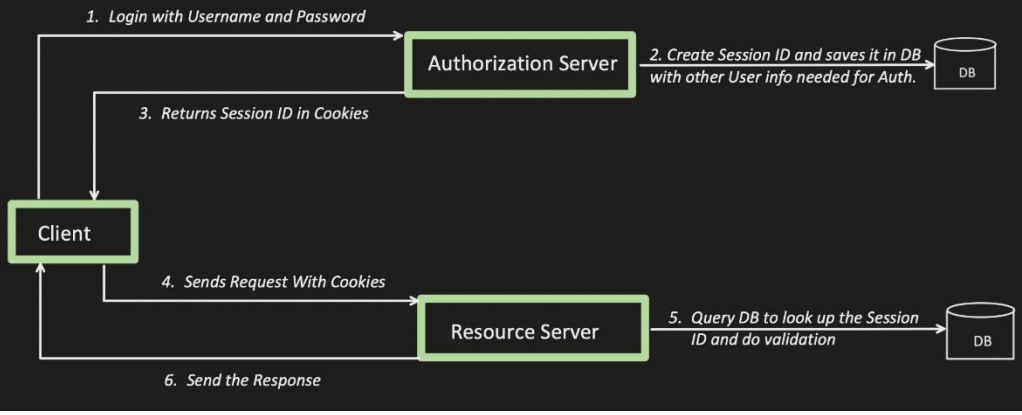
- It's provides a secure way of transmitting information between parties as a JSON object.
- This information can be verified because its digitally signed using RSA (public/private key pair) etc.

Advantages:

- **Compact**: Because of its size, it can be sent inside an HTTP header itself. And, due to its size its transmission is fast.
- **Self Contained / Stateless**: The payload contains all the required information about the user, thus it avoid querying the database.
- Can be signed using Symmetric (HMAC) or Asymmetric (RSA).
- Built in expiry mechanism.
- Custom claim (additional data) can be added in the JWT.

Before we understand more about JWT, lets first understand, what was popular before JWT and what are the problems with it?

Session ID (or JSessionID) :



Disadvantage:

- Stateful: it rely on server side state management, it cause problem in distributed systems.
- Its just a unique random string, when server get this id, it has to perform DB query to fetch the details.

JWT Structure

(aaaaaaaaaa.bbbbbbbbbb.ccccccccccc)

Header

```
{
  "typ": "JWT",
  "alg": "RSA",
  "kid": "sfdsf234234"
}
```

Payload

```
{
  "iss": "https://example.com/auth",
  "sub": "1234567890",
  "exp": 1711945909,
  "email": "conceptcoding@xyz.com,
  "jti": "unique_id_12312312"
}
```

Signature

```
RSASHA256(
Base64UrlEncode(Header) + "." + Base64UrlEncode(Payload) ,
Key
)
```

Challenges with JWT:

1. Token Invalidation : lets say, I have blacklisted one user, how to invalidate its Token before its expiration?
    a) Server need to keep the list of blacklisted tokens and then DB/cache lookup is required while validating.
    b) Or, Change the secret key, but this will make the JWT invalidate for all users.
    c) Or, Token should be very short lived.
    d) Or, Token should be used only once.

2. JWT token is encoded, not encrypted. So its less secure.
    a) Use JWE, Means encrypt the payload part.

3. Unsecured JWT with "alg" : none, such JWT should be rejected.

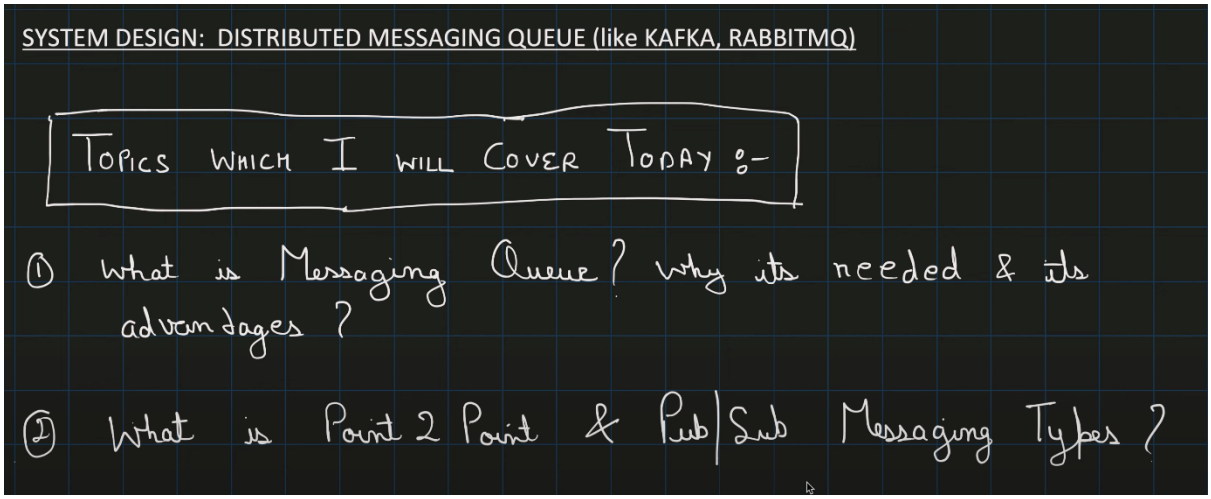
4. Jwk exploit: public key shared in this, should not be used to verify the signature.

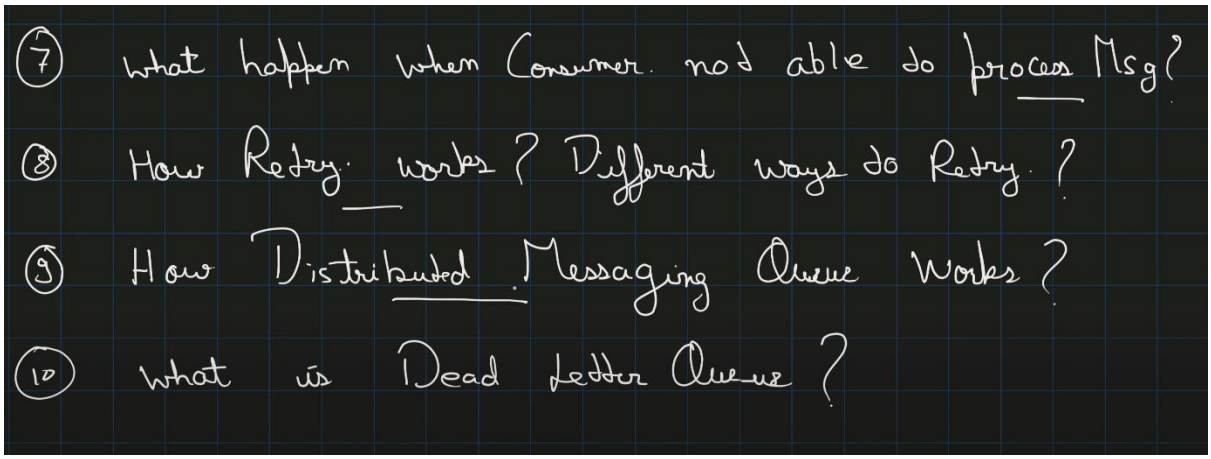
e stands for "exponent" and n stands for "modulus" and combined together they form Public key.

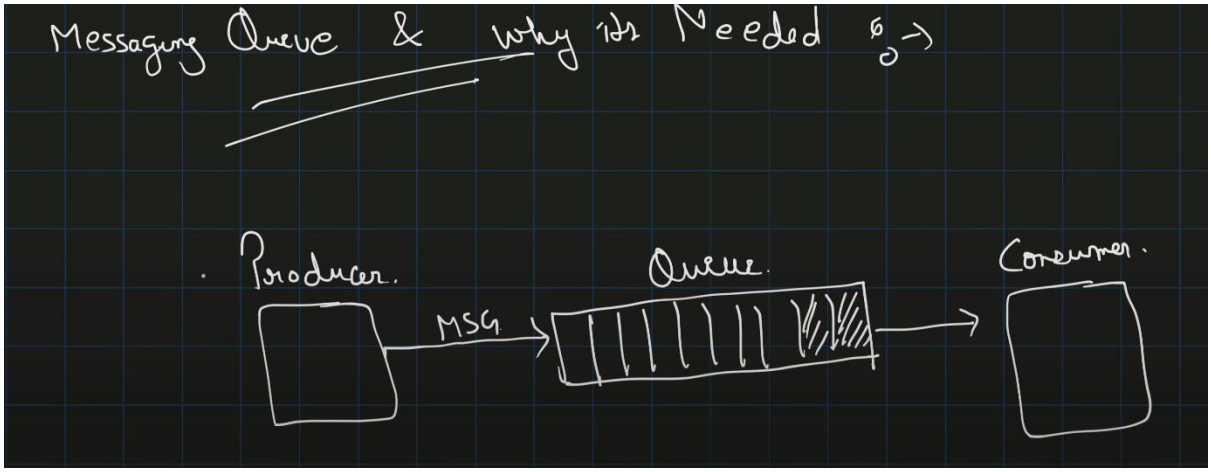
```
{
  "typ": "JWT",
  "alg": "RSA",
  "jwk": {
            "n" : "sfsdf234324324fsd4sfdsfsdf23",
            "e" : "ABC4ED",
            "kid" : "sdfds3432432432fwfdwfsfwf"
          }
}
```

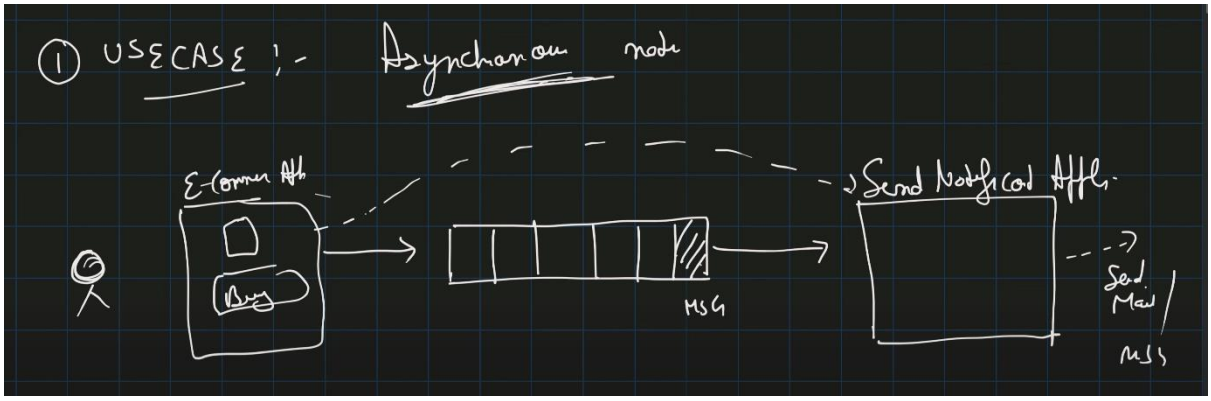
5. Use "Kid" in the Header to look up the https://{Auth server domain}/.well-known/jwks.json to find the Public key.

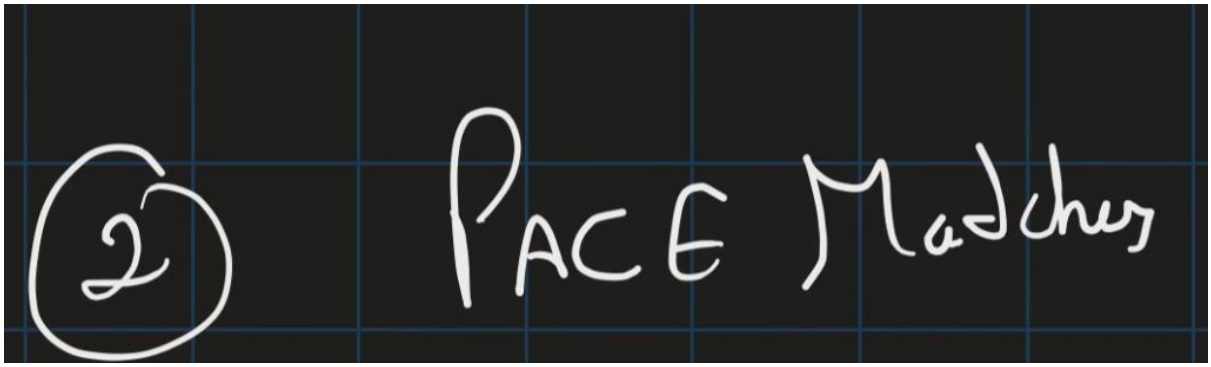
HLD: Kafka, RabbitMQ(Distributed messaging queue)

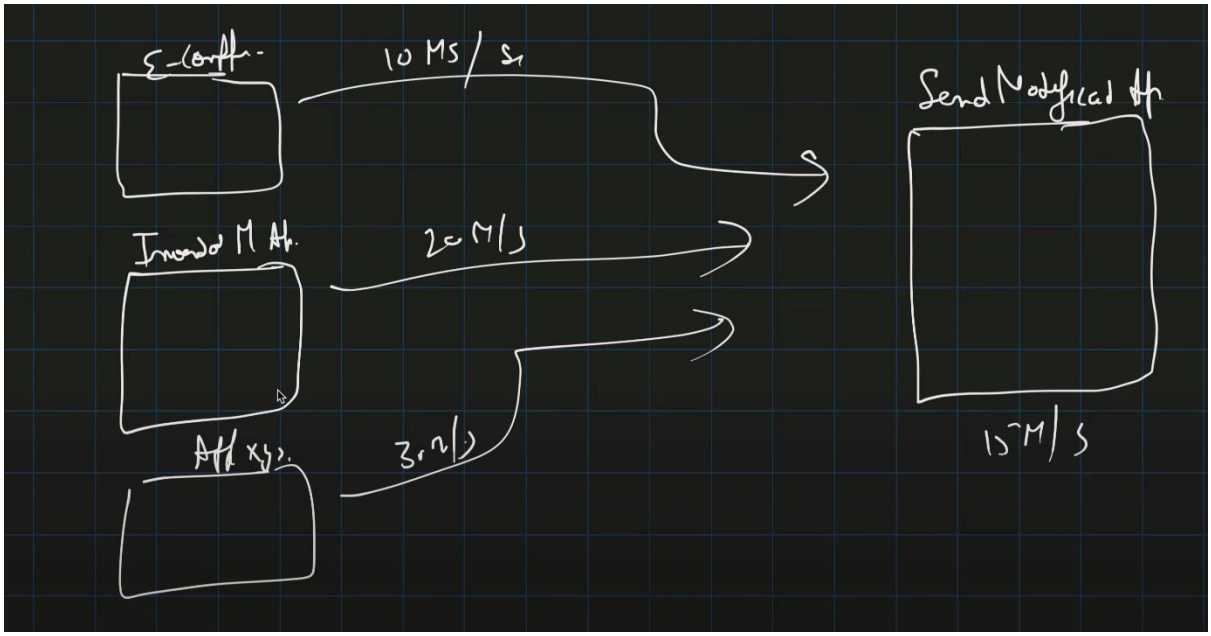


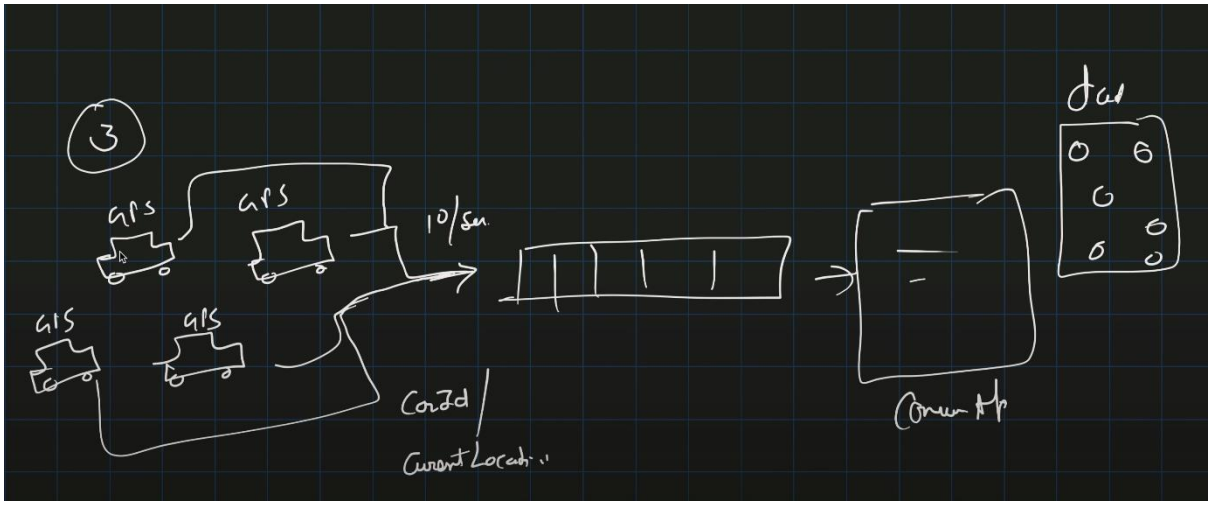



Messaging Queue & Why its Needed ?
Producer.
MSG
Queue.
Consumer.

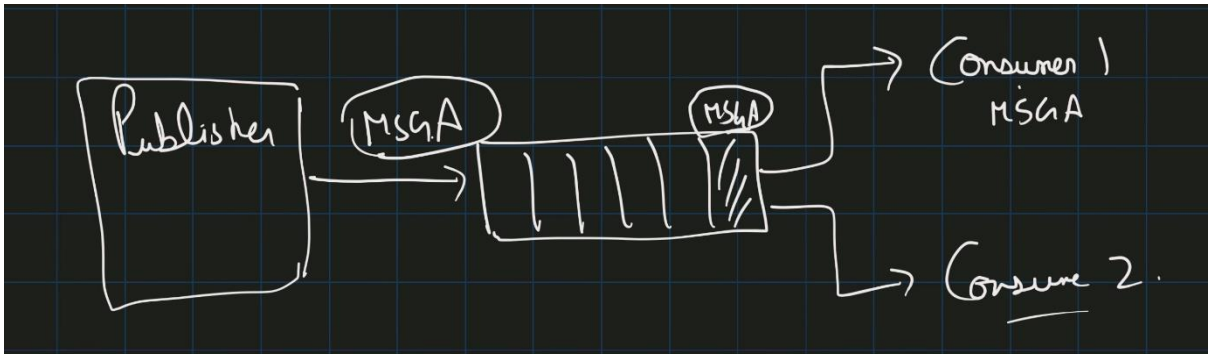

① USECASE :- Asynchronous mode
E-Comm Apk
Buy
MSG
Send Notification Apk.
Send Mail / MSG

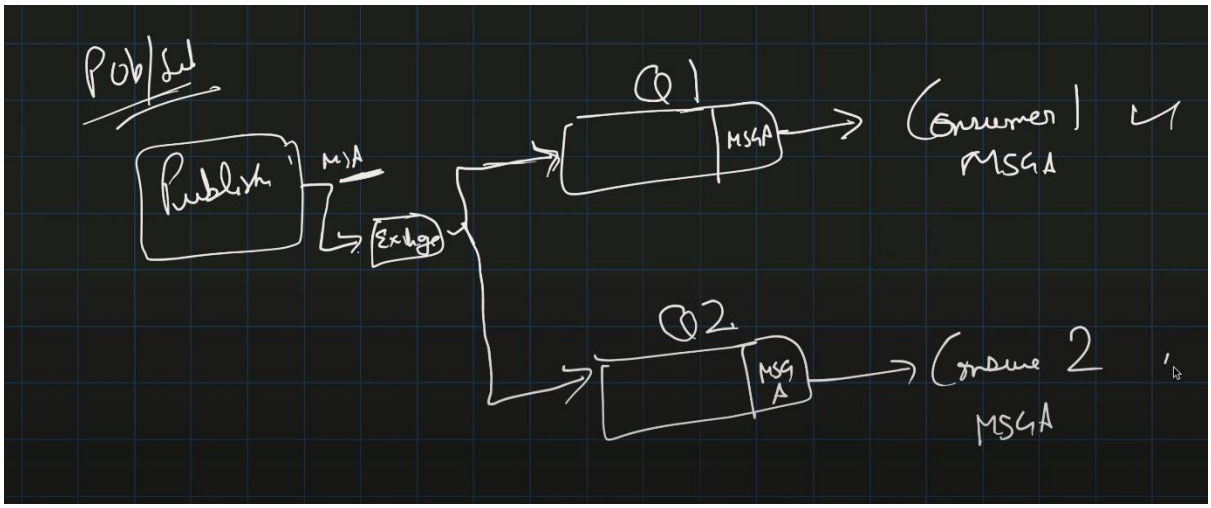

2
PACE Matches

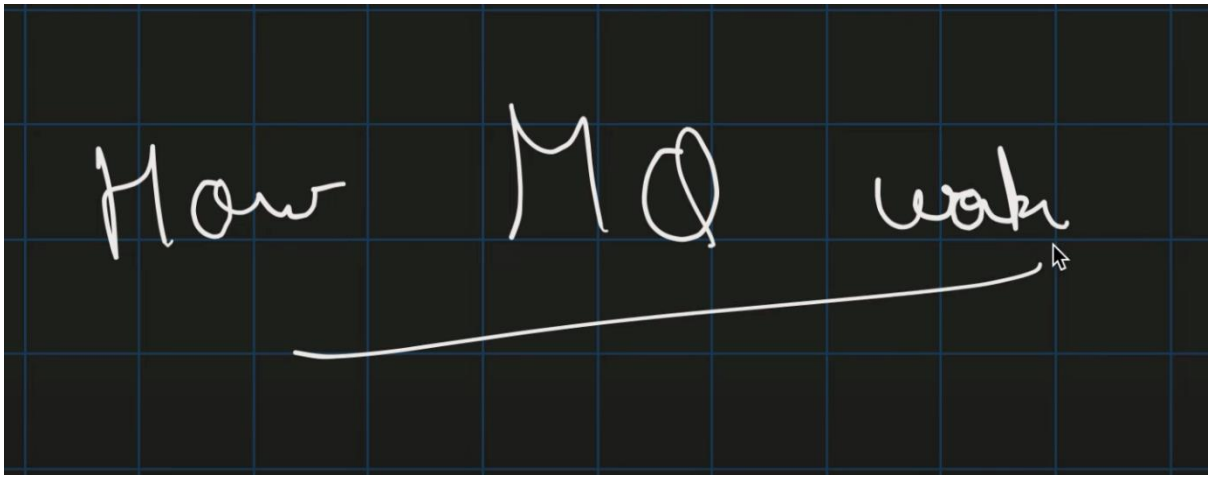




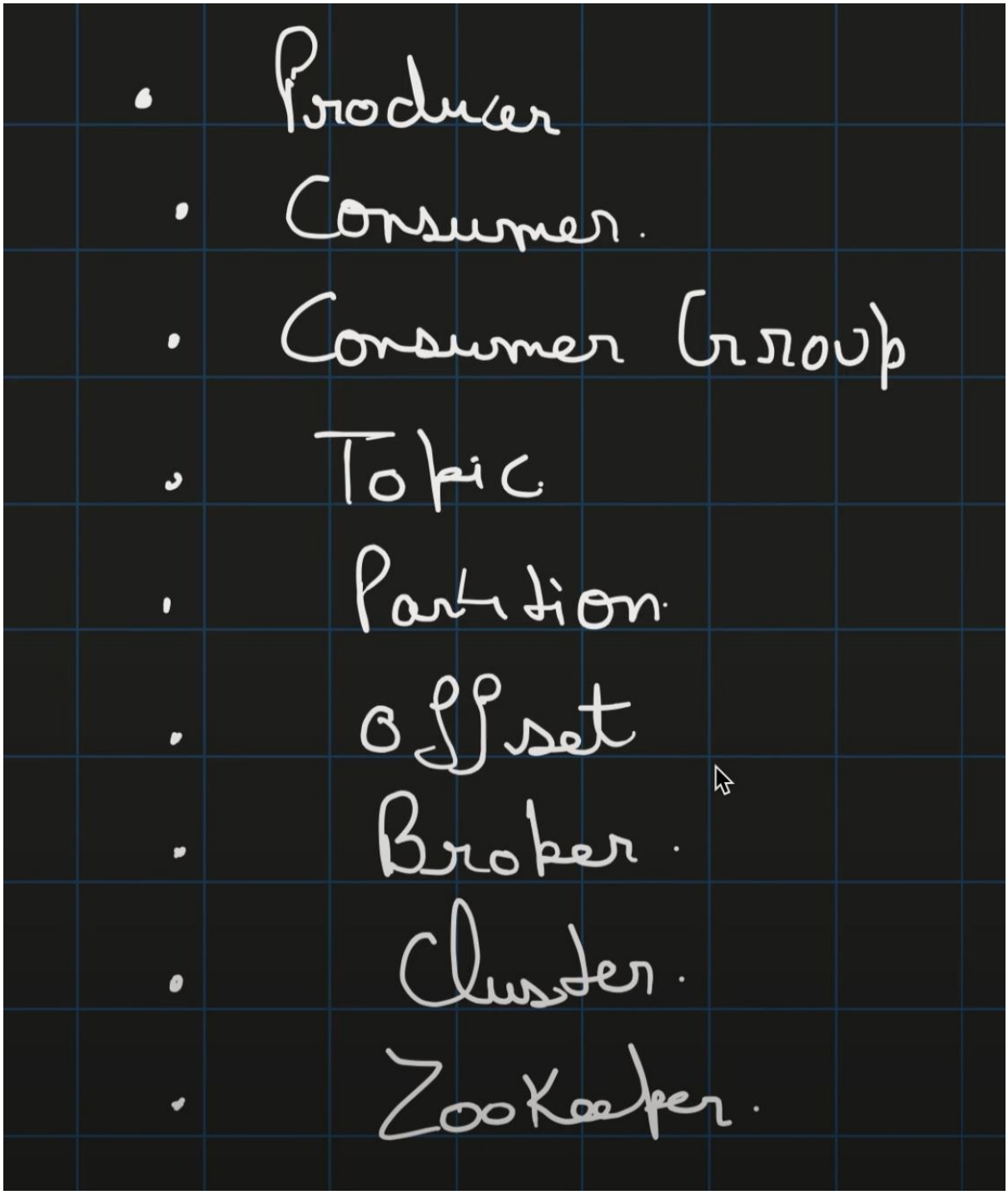
P2P & Pub / Sub

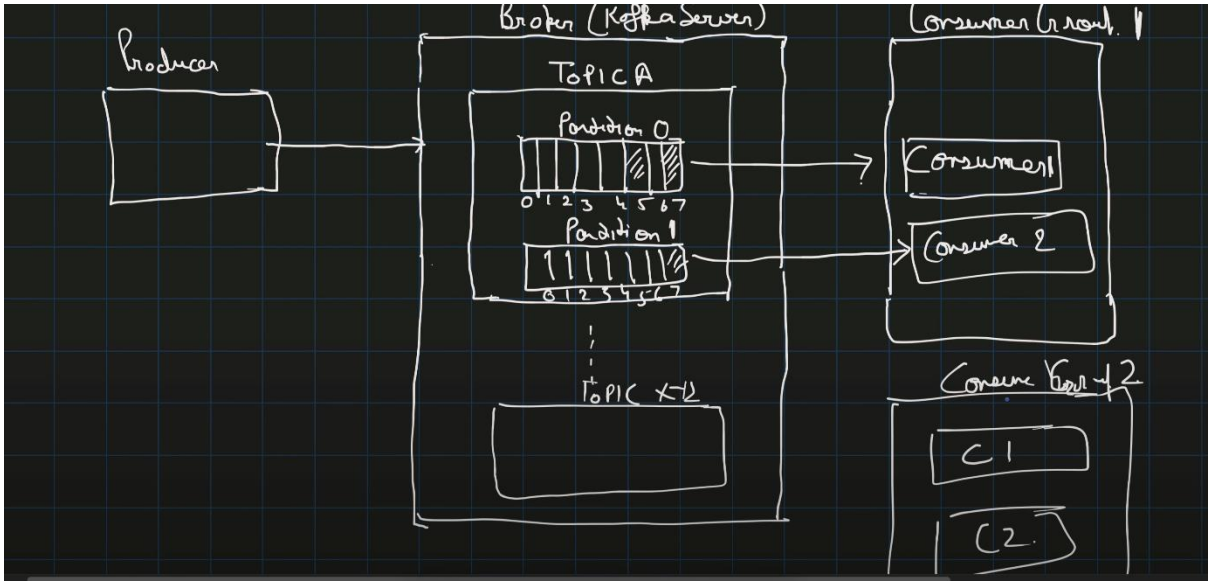

Publisher
MSGA
MSGA
Consumer 1
MSGA
Consume 2.

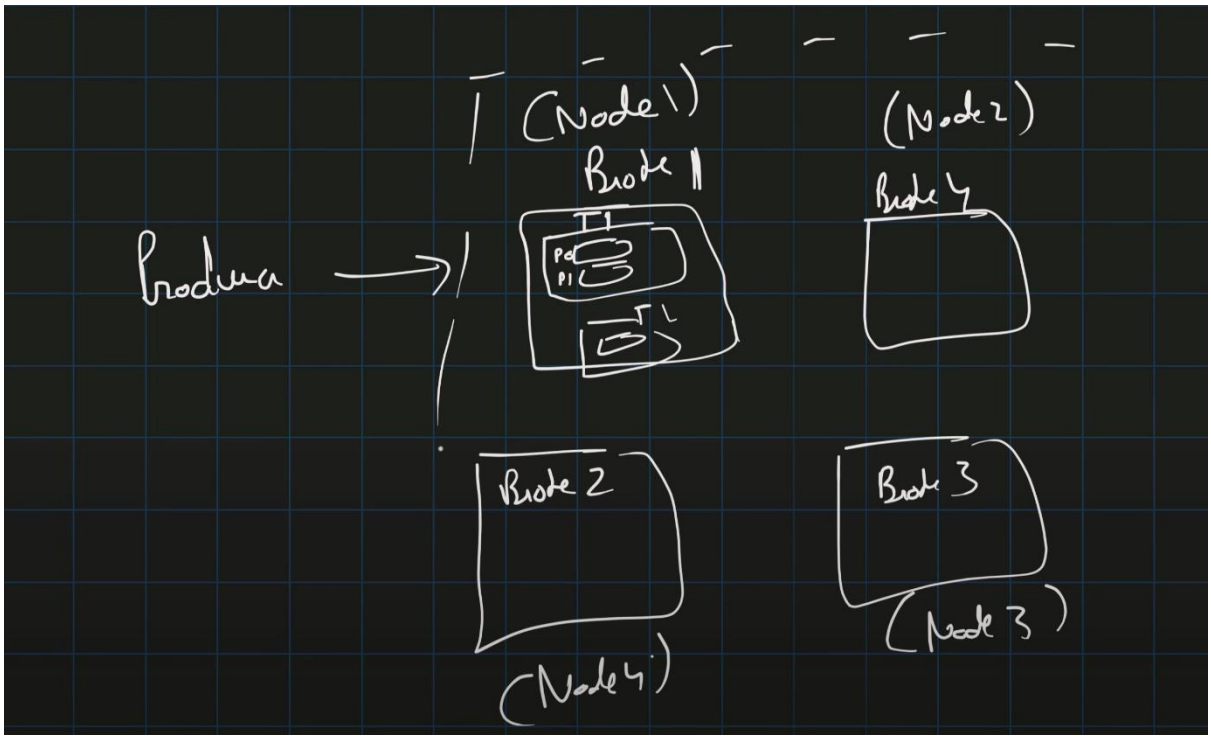

Pub/Sub
Publisher
MSA
Exchange
Q1
MSGA
Consumer 1
MSGA
Q2.
MSG A
Consume 2
MSGA

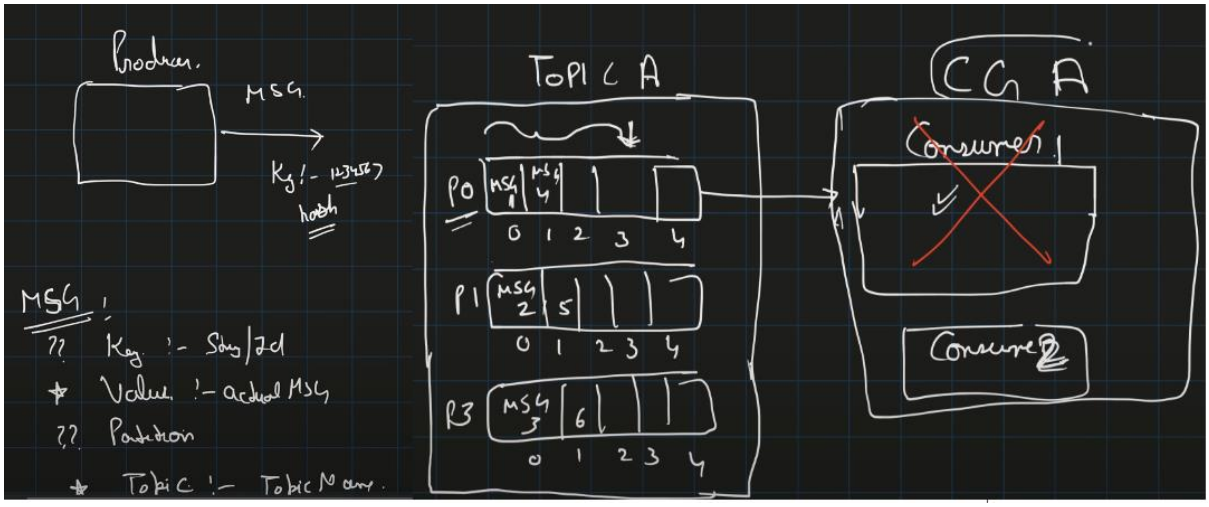

How MQ work

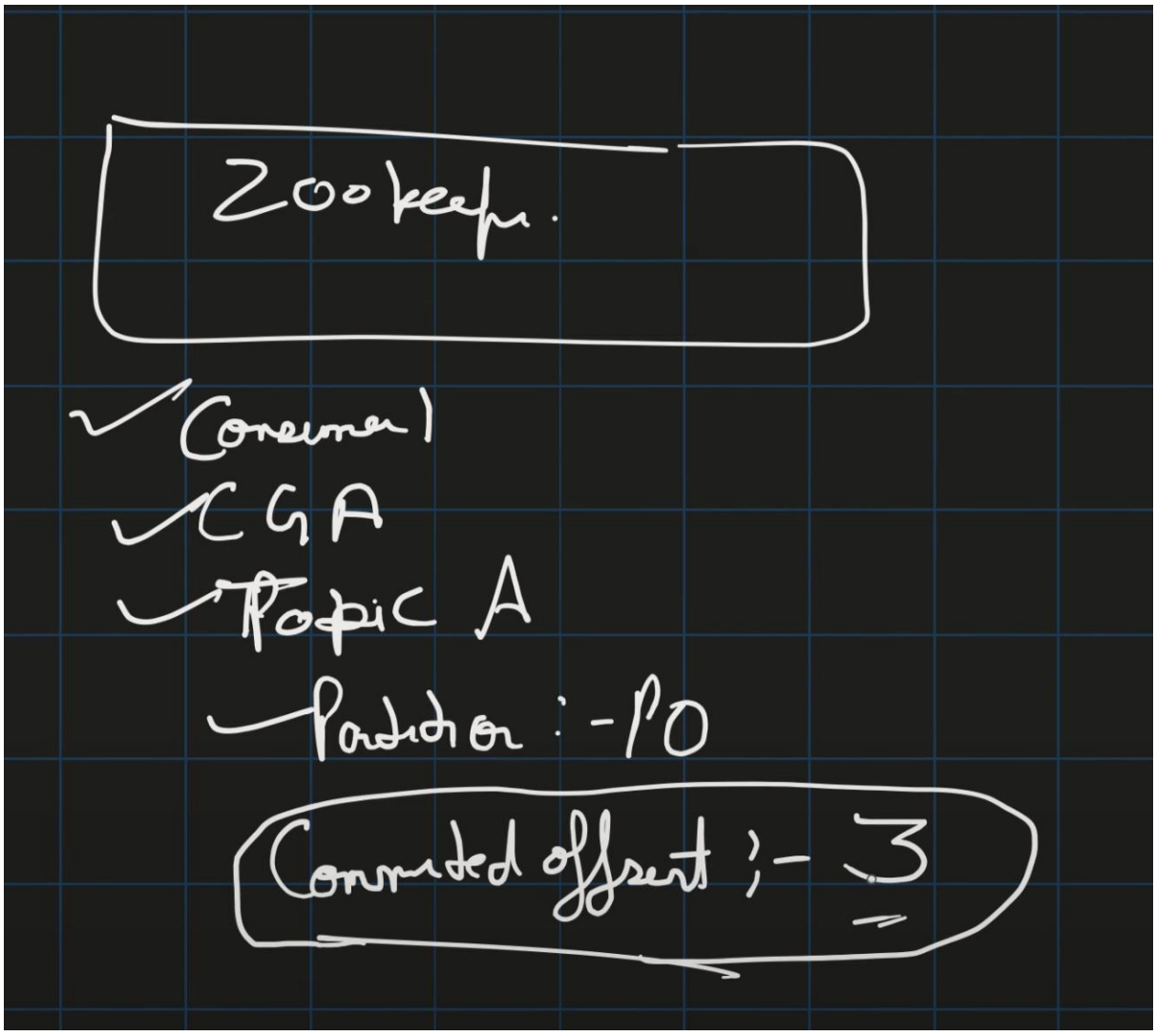

1
KAFKA

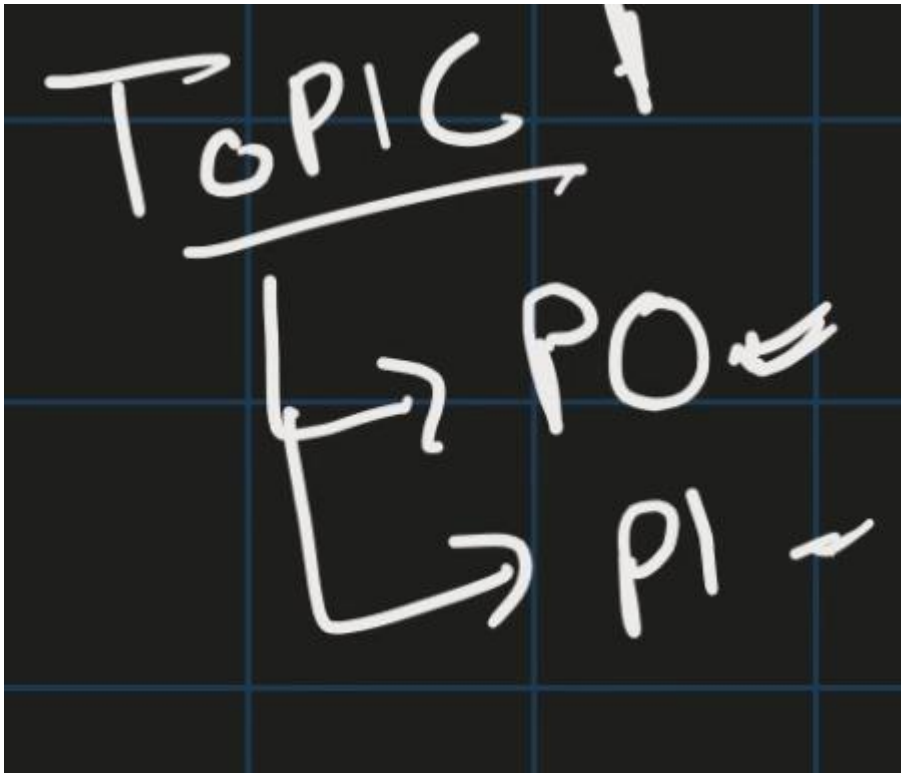

Producer
Consumer.
Consumer Group
Topic
Partition.
offset
Broker.
Cluster.
ZooKeeper.

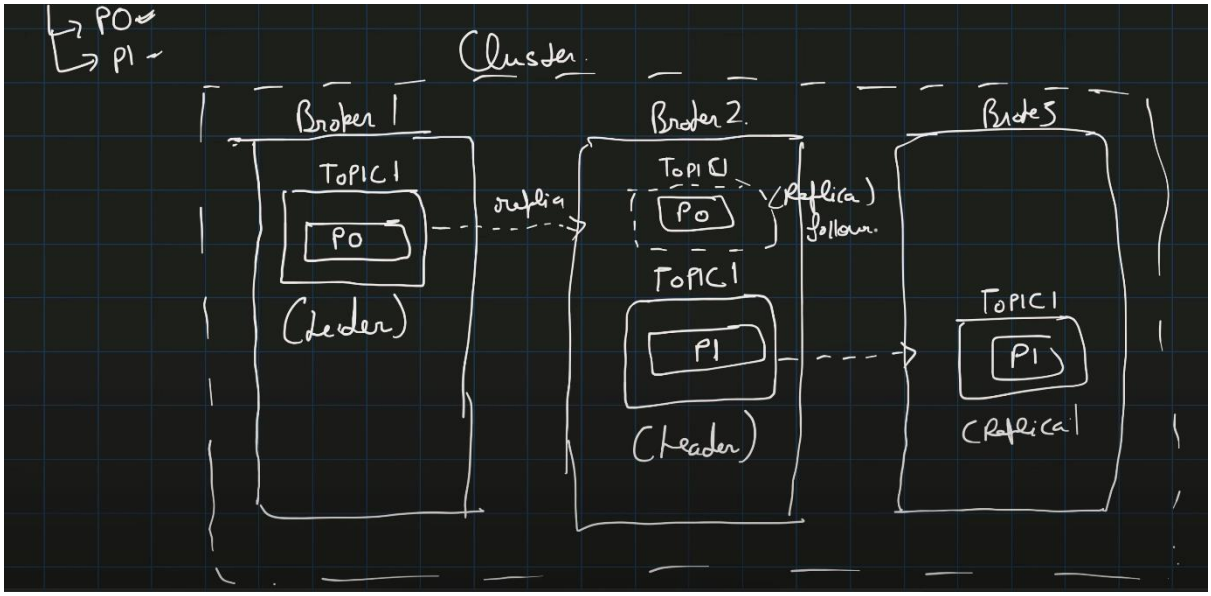

Producer
Broker (Kafka Server)
TOPIC A
Partition 0
0 1 2 3 4 5 6 7
Partition 1
0 1 2 3 4 5 6 7
TOPIC X-Z
Consumer (group 1)
Consumer 1
Consumer 2
Consumer Group 2
C1
C2

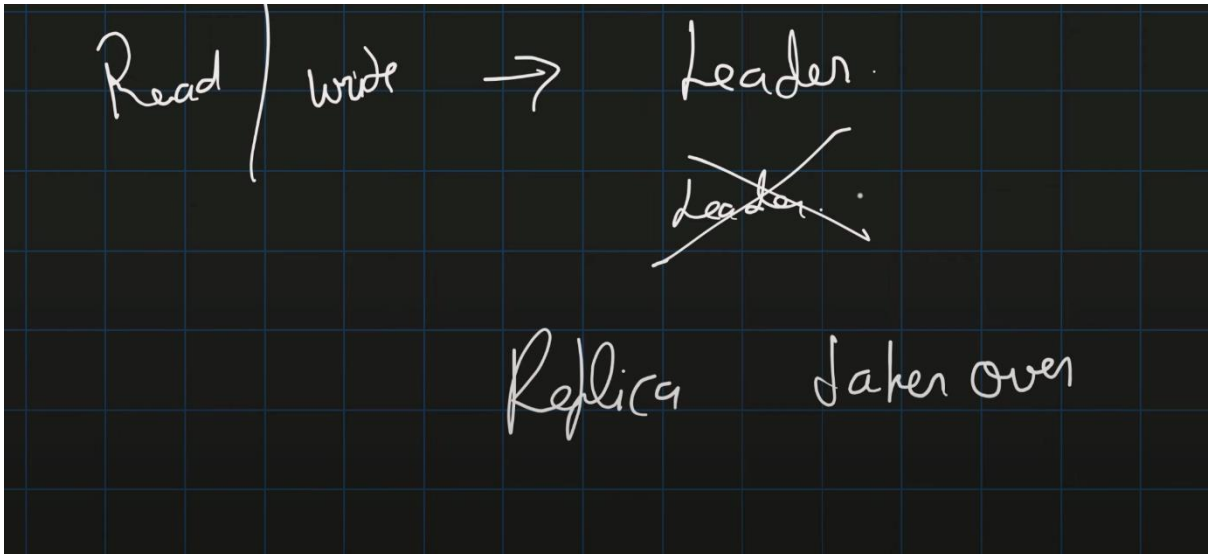

Produce
(Node 1)
Broke 1
T1
P0
P1
T2
(Node 2)
Broke 4
Broke 2
(Node 4)
Broke 3
(Node 3)

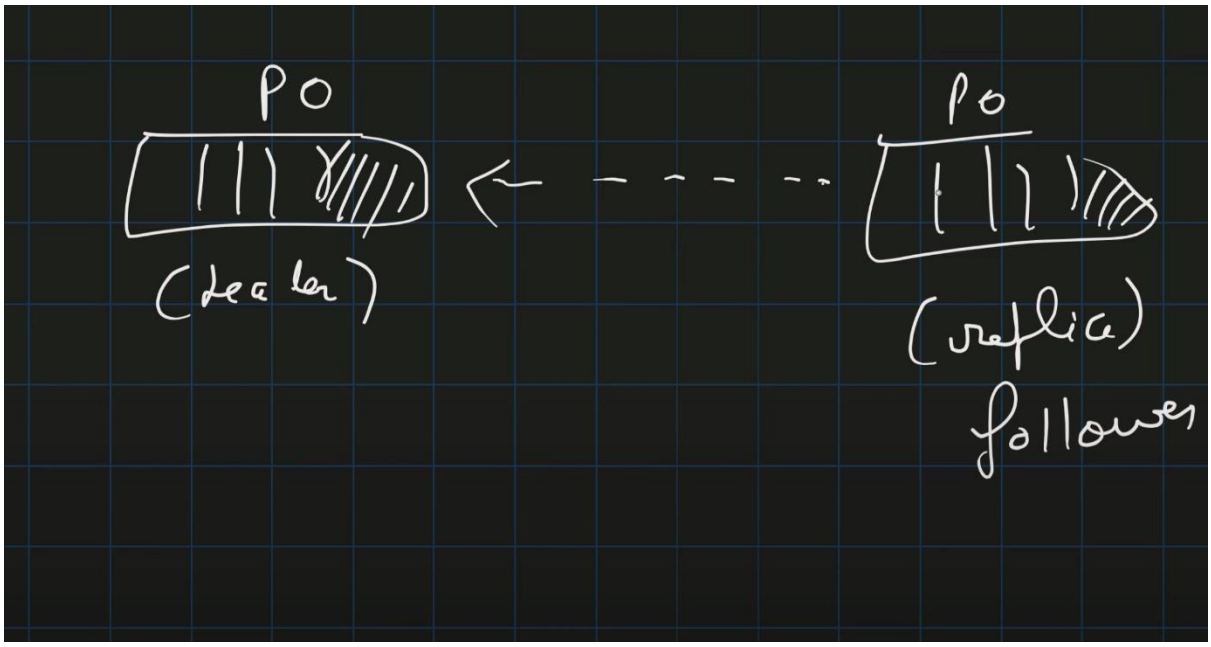

Producer.
MSG.
Key :- 123456
hash
MSG :
?? Key :- String/Int
* Value :- actual MSG
?? Partition
* Topic :- Topic Name.
TOPIC A
P0
P1
P3
0 1 2 3 4
CG A
Consumer 1
Consumer 2

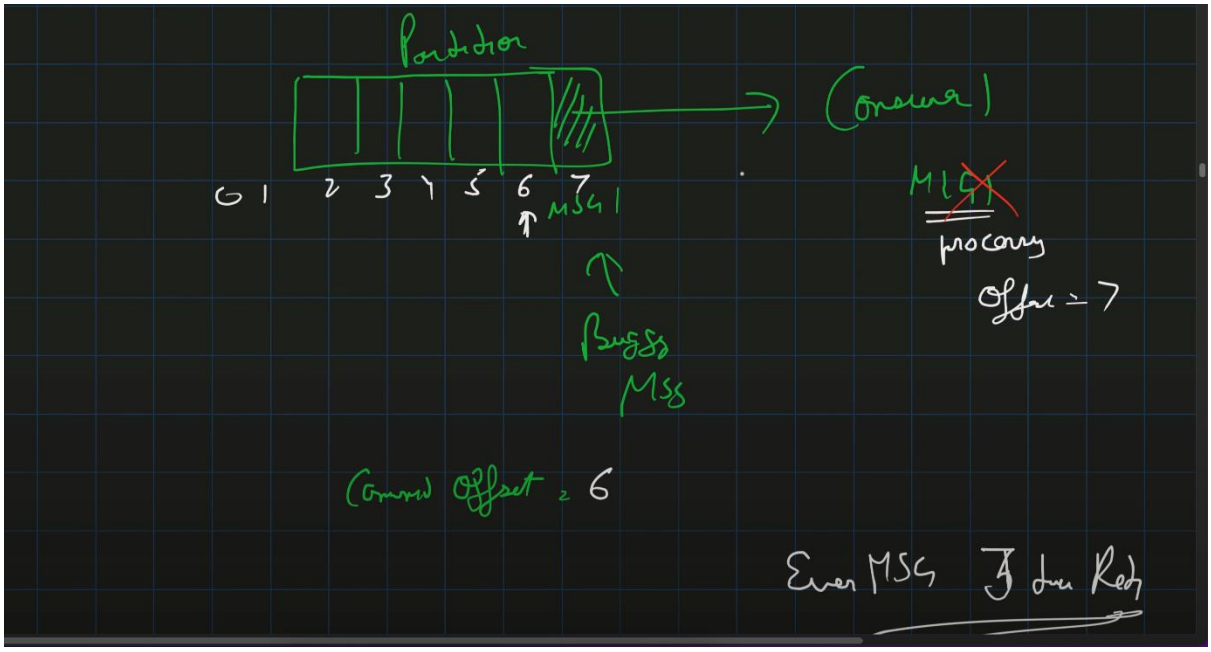

Zookeeper.
Consumer 1
CG A
Topic A
Partition :- P0
Committed offset :- 3

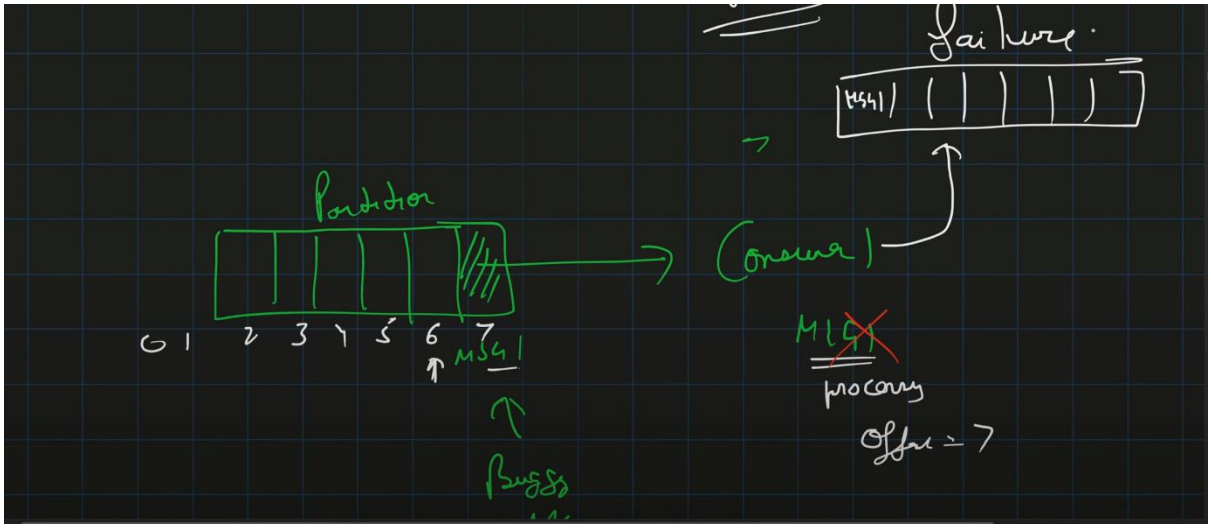

TOPIC 1
P0
P1

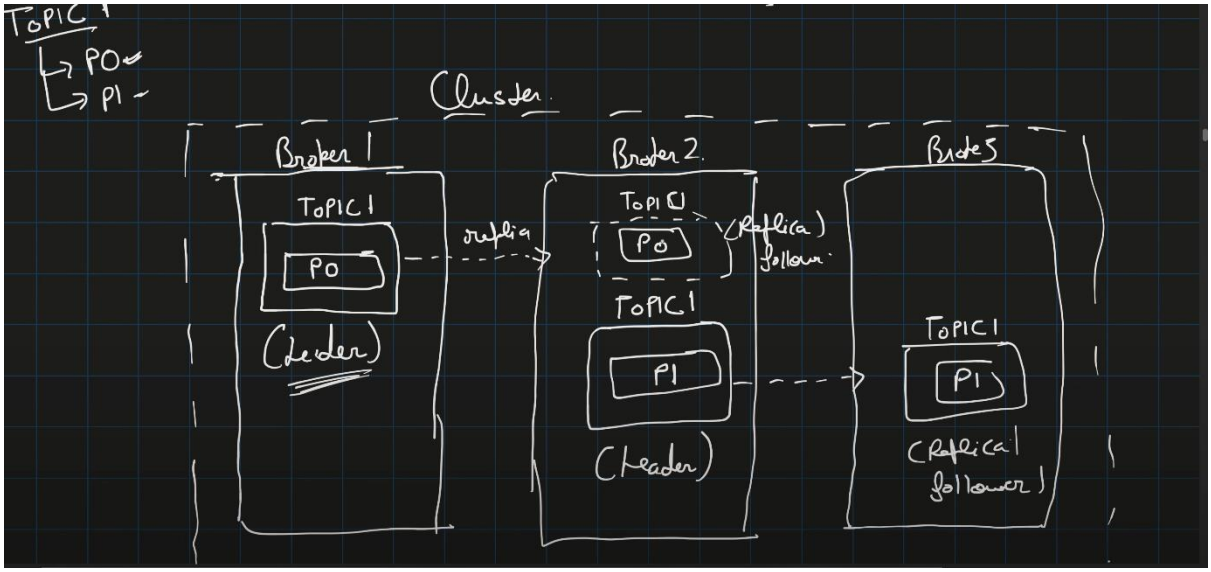

P0
P1
Cluster
Broker 1
Broker 2
Broker 3
TOPIC 1
P0
(Leader)
replica
TOPIC 1
P0
(Replica)
follow.
TOPIC 1
P1
(Leader)
TOPIC 1
P1
(Replica)

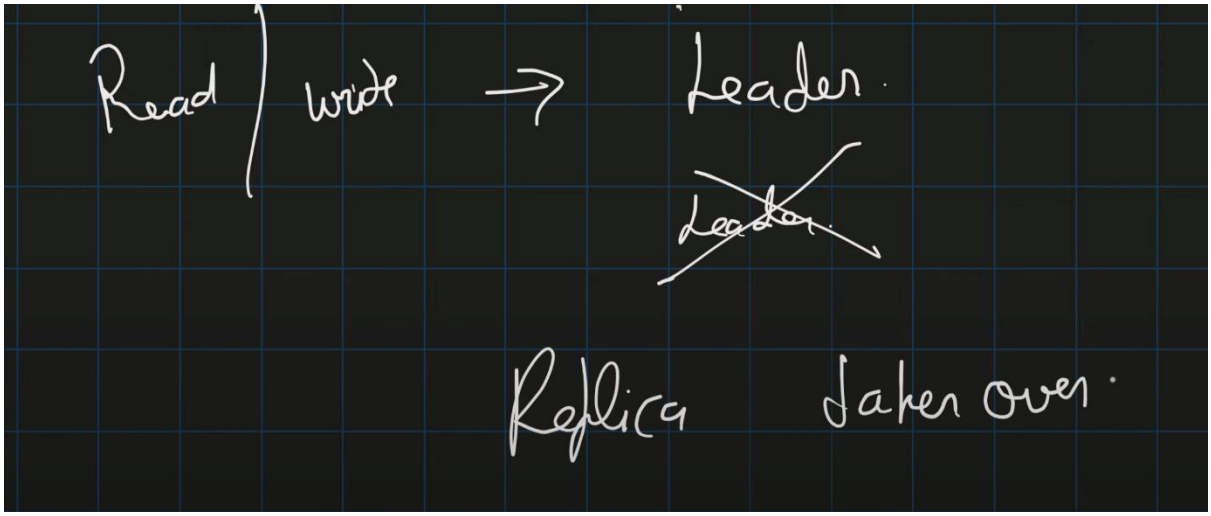

Read / Write → Leader
Leader
Replica taken over

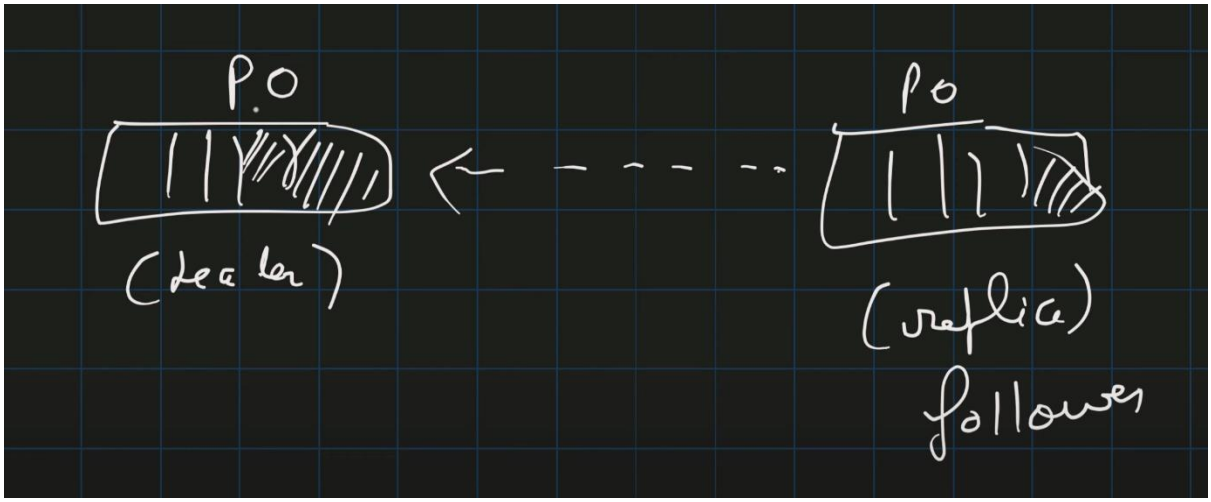

P0
P0
(leader)
(replica)
follower

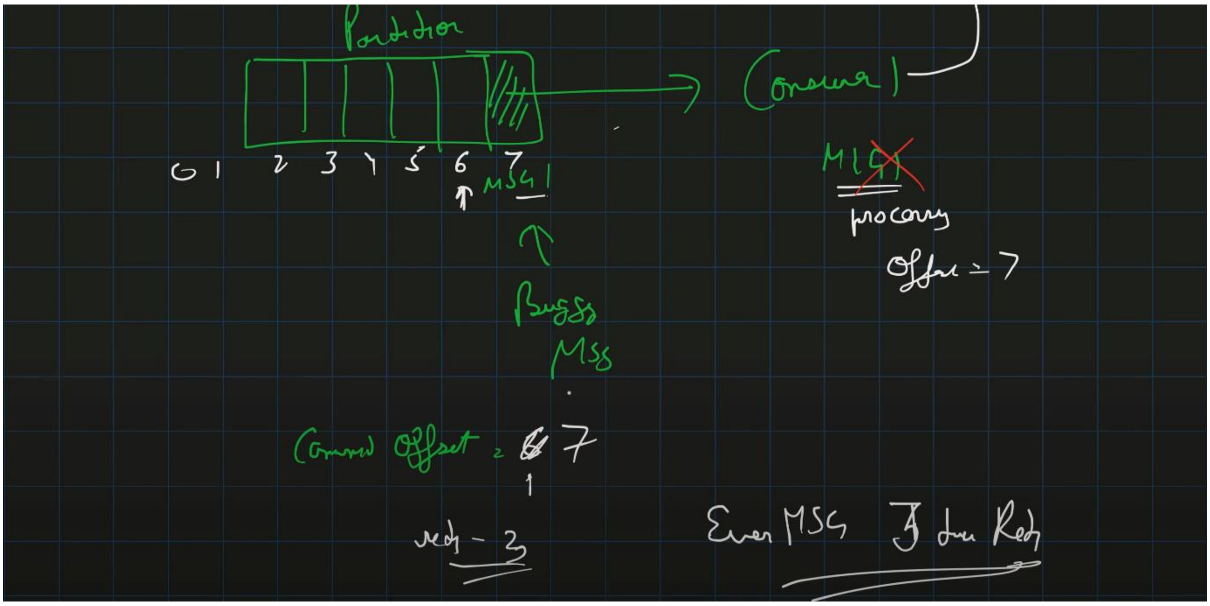

Partition
Consumer
0 1 2 3 4 5 6 7
Msg1
Msg1
processing
Offset = 7
Buggy Msg
Consumer Offset = 6
Even Msg Jaha Rahe

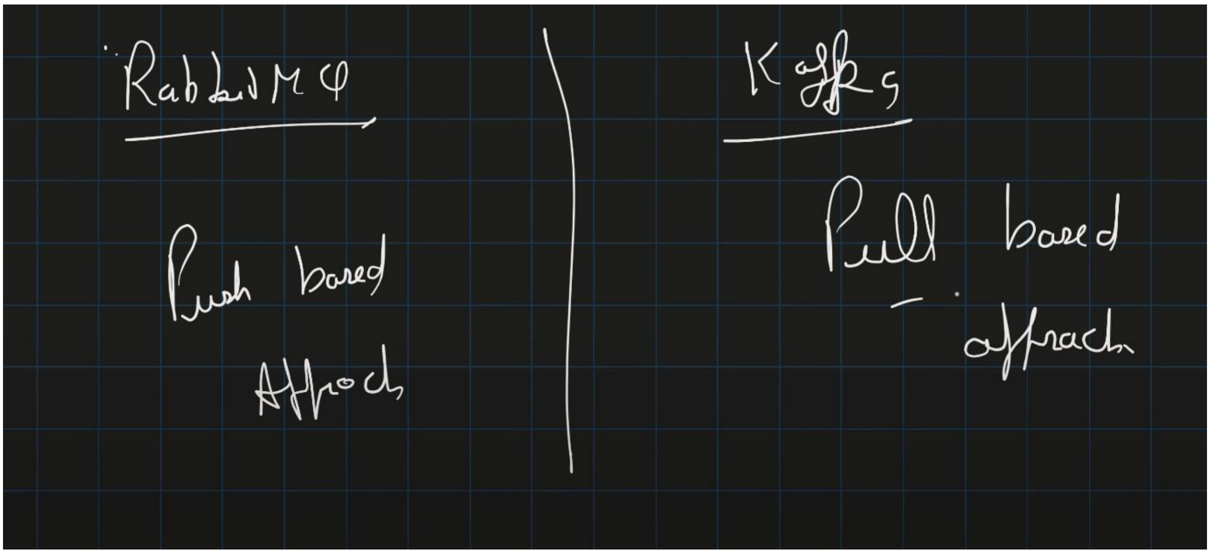

Failure
Msg1
Partition
Consumer
0 1 2 3 4 5 6 7
Msg1
Msg1
processing
Offset = 7
Buggy


Topic 1
P0
P1
Cluster
Broker 1
Broker 2
Broker 3
Topic 1
P0
(Leader)
replica
Topic 1
P0
(Replica)
follower
Topic 1
P1
(Leader)
Topic 1
P1
(Replica)
follower)

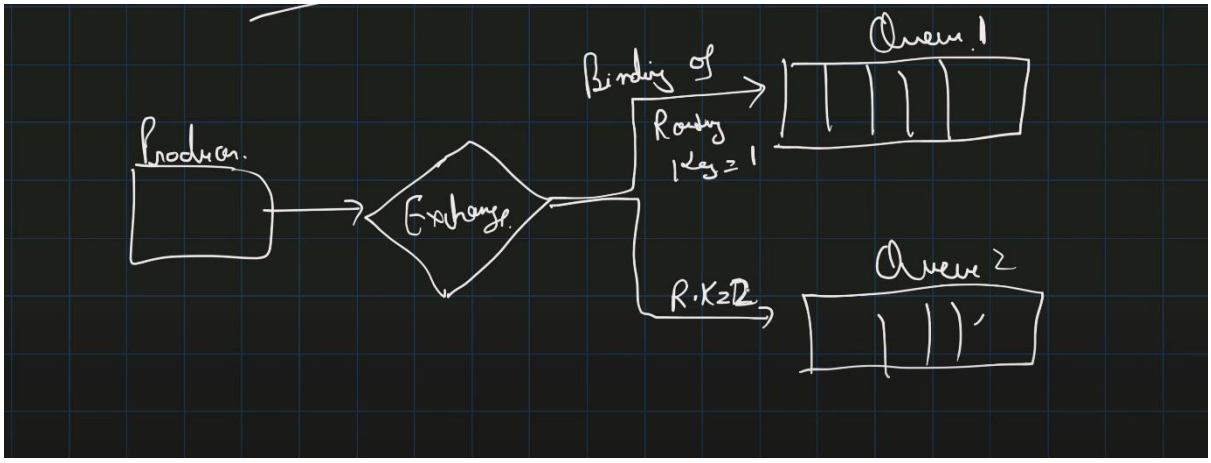

Read / write → Leader
Leader
Replica taken over.

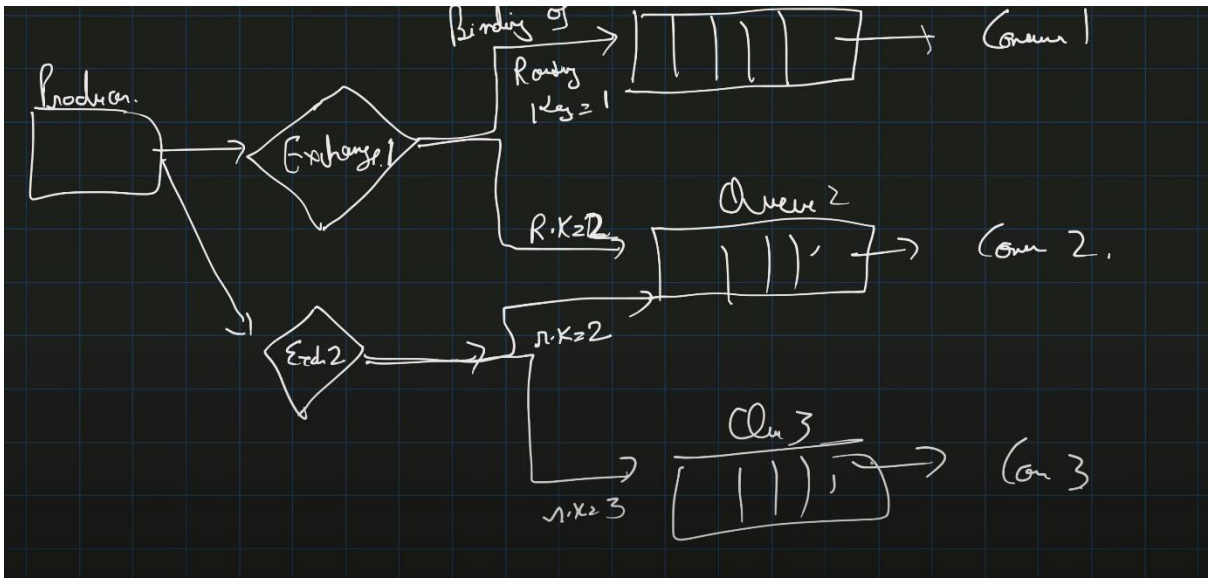

P.0
(Leader)
P0
(replica)
follower

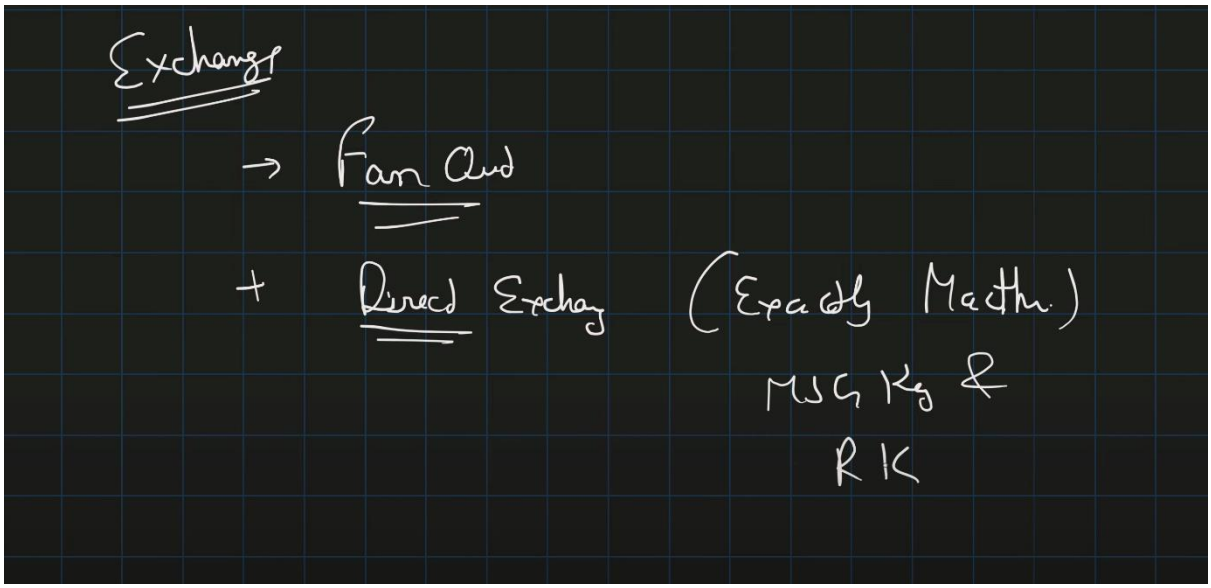

Partition
Consumer
0 1 2 3 4 5 6 7
processing
Offset = 7
Commit Offset

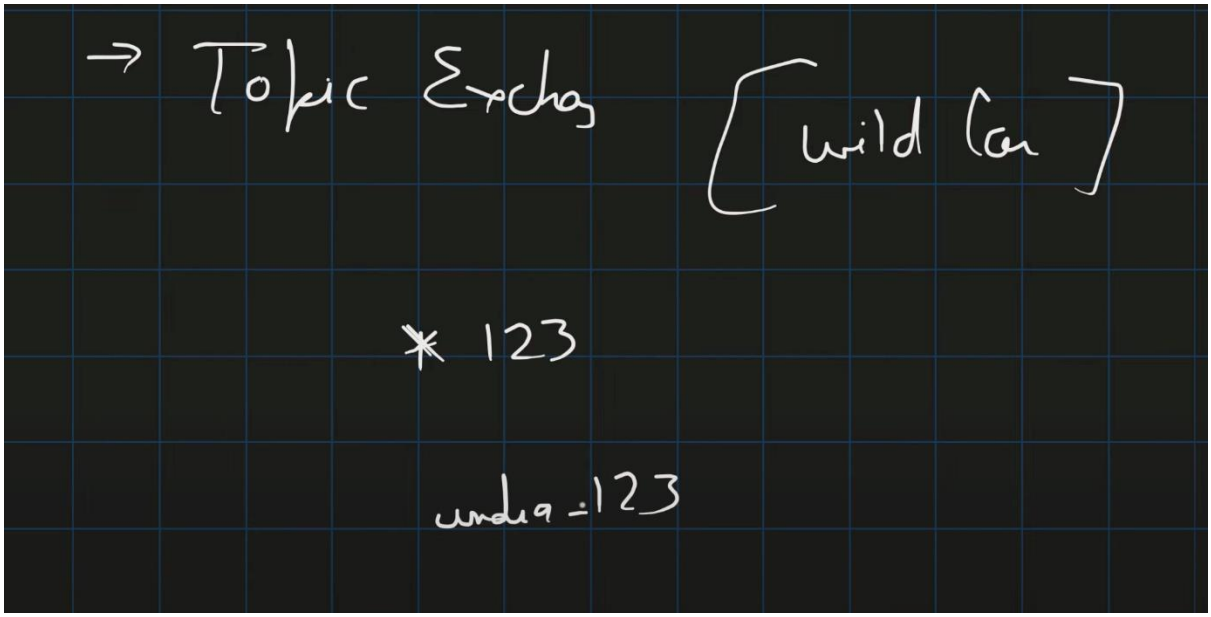

RabbitMQ
Kafka
Push based
Pull based

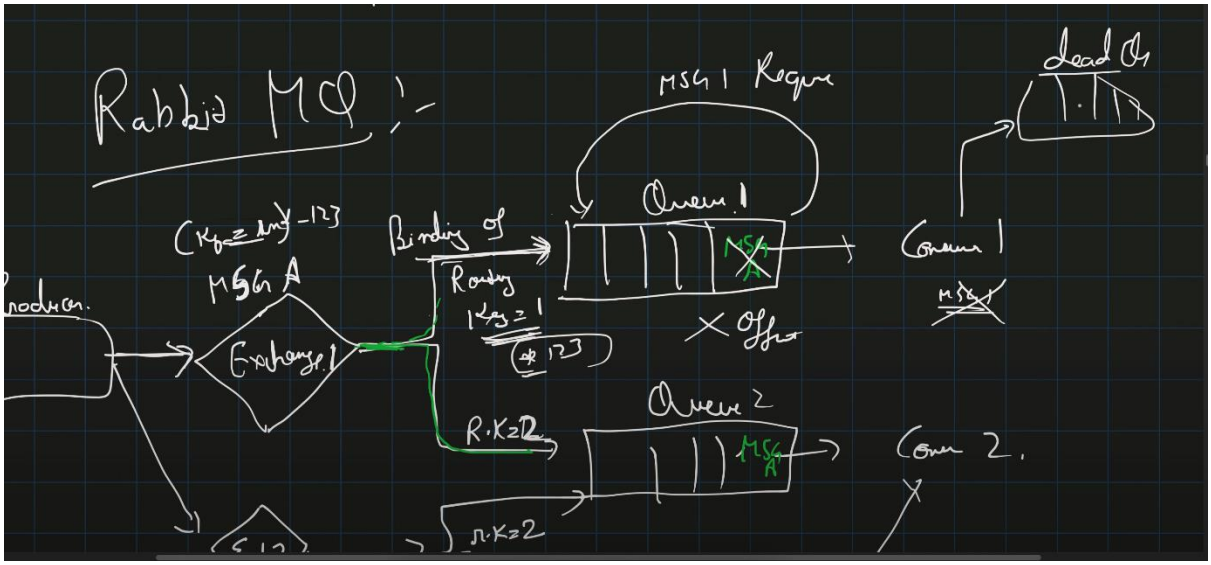
Rabbit MQ :-


Queue 1
Binding of
Routing
Key = 1
Producer
Exchange
R.K=2
Queue 2


Binding of
Routing
Key = 1
Queue 1
Producer
Exchange 1
R.K=2
Queue 2
Cons 2.
Exch 2
R.K=2
Qu 3
Cons 3
R.K=3


Exchanges
→ Fan Out
+ Direct Exchg (Exactly Matching)
MSG Key &
RK


→ Topic Exchg [wild Card]
* 123
india.123


Rabbit MQ :-
MSG1 Requ
dead Q
Queue 1
Binding of
Routing
Key = 1
Exchange 1
R.K=2
Queue 2
Cons 2.
R.K=2