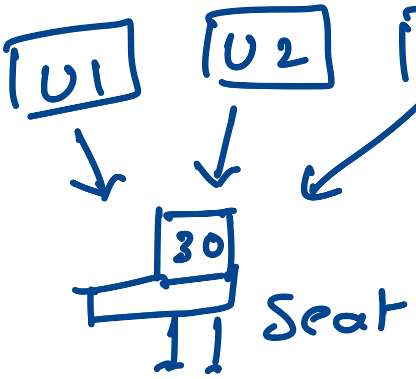
Concurrency Control in Distributed Systems

Concurrency in Non-Distributed Systems:

Scenario:



In Non-distributed systems, if we introduce synchronized block concept -- we may control concurrency issues.

Process
- thread1
- thread2
- thread3
- ⋮

```
synchronized() {
    if Free:
        Book seat
}
```

U1 → free – update to Booked

U2 → Not free – Nothing happen

U3 → Not free – "

Concurrency in Distributed Systems:

In distributed Systems, multiple processes run on multiple servers.

Now imagine multiple users accessing multiple processes to book a same seat. Here, synchronized block of code don't support. Therefore, we need to come up with a better strategy.

Before deep diving, we have to understand a set of concepts.

1) Transaction
2) DB Locking
3) ISolation Levels.

Transaction: Set of instructions combined called a transaction.

If any instruction fails, the whole transaction has to revert back or roll back

```
T1 {
          BankA -20 = BANKA
Failure X Bank B= B+20
Roll Back  Commit
}
```

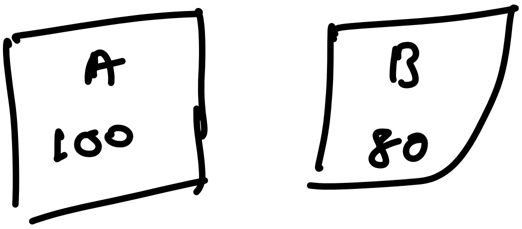
Bank A	Bank B
100	80

Send 20 to B from A

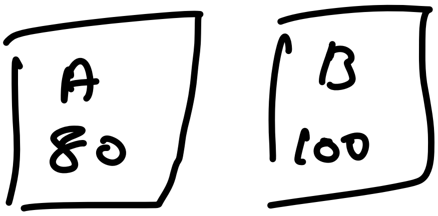
If failure encountered before commit then we need to roll back.

Here we have 2 consistent states

(1)



(2)



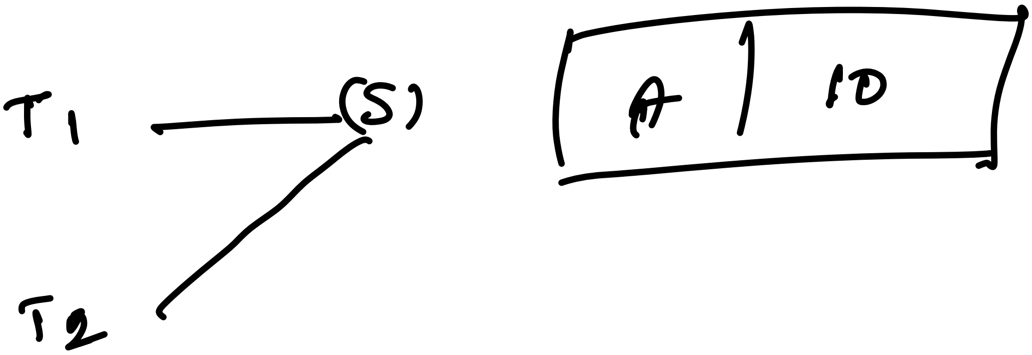
Remaining all are inconsistent states

DB Locking:

1) Shared Locking

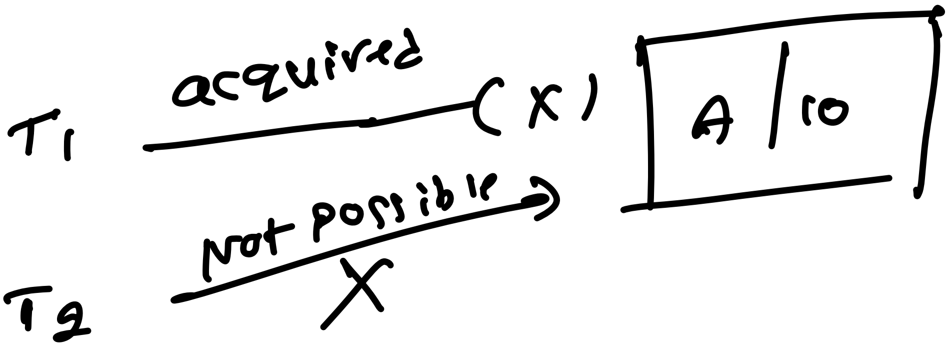
2) Exclusive Locking.

Shared Locking: (Read Lock) (S)



T1 & T2 can acquire shared locked and read data simultaneously.

Exclusive Lock: (Write lock) (X)



Write lock can be acquired one at a time

	has shared lock	has exclusive lock
Shared Lock	✓	✗
Exclusive Lock	✗	✗

1) if T_1 is accessing Shared Lock then $T_2 \ldots T_n$ can access shared lock to read data

2) if T_1 is accessing shared lock then $T_2 \ldots T_n$ cannot acquire exclusive lock.

3) If T_1 is having exclusive lock then $T_2 \ldots T_n$ cannot acquire shared lock

4) If T_1 is having exclusive lock then $T_2 \ldots T_n$ cannot acquire exclusive lock

Isolation Levels in Distributed Systems:

Before deep diving we need to understand

1) Dirty Read
2) Non-repeatable Read
3) Phantom read

Dirty Read:

A	~~10~~	11

```
T1 {
    Read A // 10
    ...
    ...
    Read A // 11
    ...
    ...
    commit
}
```

```
T2 {
    Write A // 11
    ...
    ...   X failure.
    commit ;
}
```

Before commiting write on DB (A) --- T1 read the data updated by T2.

After Roll over --

A	10

A becomes 10

but T1 did some calculations with A = 11

This is Dirty Read.

Non-Repeatable Read:

Read different values which are
Non-Repeatable committed

A	10

T1	T2
read A // 10	write A = 11
	commit
read A // 11	

We read different values of A within a transaction

Phantom Read:

Read different no.of rows when a query is executed (committed)

ID1	A	10
ID2	B	20
⋮		
ID7	E	70

Select ID where ID>2 & ID≤5

First time read

We get

ID3, ID4, ID5

Now, some updates happened in DB.

We added a new row

Same query

select ID where ID>2 & ID≤5

we get

ID3, ID4, $\underline{\text{IDx, IDy}}$, ID5

Types of isolation Levels:

1) Read uncommited
2) Read committed
3) Repeatable Read
4) Serializable

Read Uncommitted:

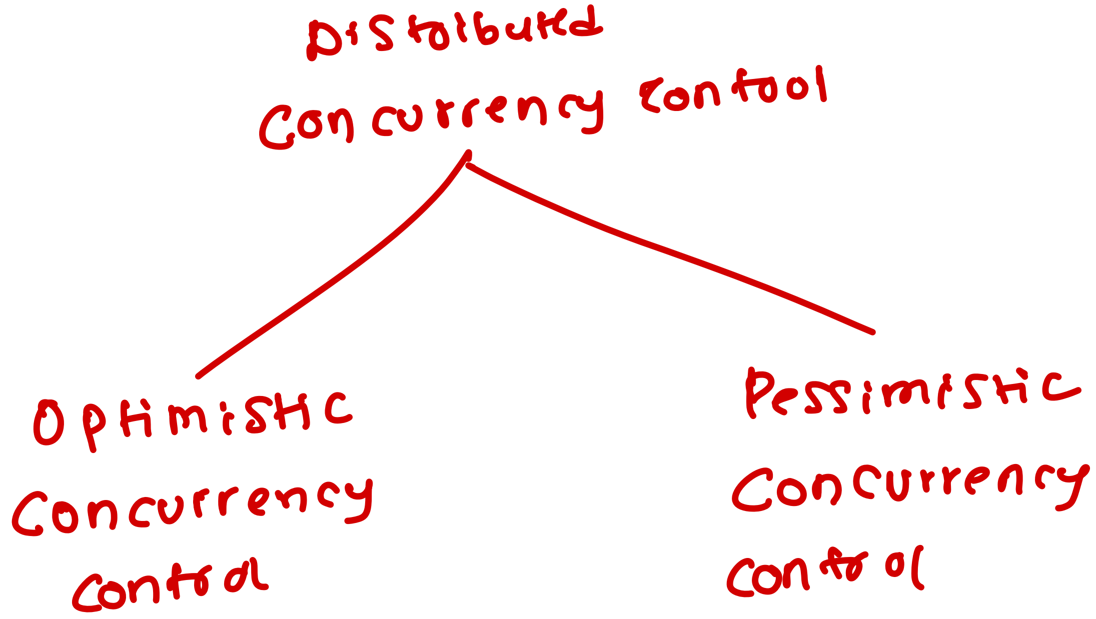
As name suggests we need to support for reading uncommitted data.

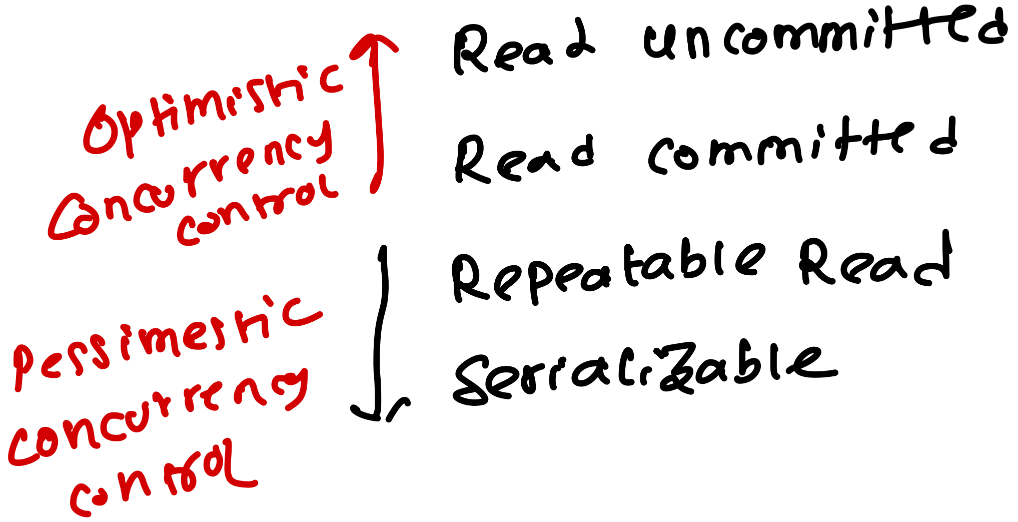
So, We can support

1) Dirty Read
2) Non-repeatable Read
3) phantom Read

Isolation level	Dirty Read	Non-Repeatable read	Phantom Read
Read uncommitted	✓	✓	✓
Read Committed	✗	✓	✓
Repeatable Read	✗	✗	✓
Serializable	✗	✗	✗

Isolation level	Shared lock	Exclusive lock
Read uncommitted	✗	✗
Read Committed	shared lock but release after read	Exclusive lock until write is done in a transaction
Repeatable Read	Shared lock till Transaction ends	Exclusive lock until Transaction ends
Serializable	Shared lock + range lock till Transaction ends	Range lock till Transaction ends


Distributed Concurrency control
Optimistic Concurrency Control
Pessimistic Concurrency control


Optimistic Concurrency control
Pessimestic concurrency control
Read uncommitted
Read committed
Repeatable Read
Serializable

Choose distributed concurrency control as per the requirement in system design.