distributed concurrency control :-

* Suppose you are trying to book a seat in theater and other n users are also doing same.

↳ Here we want to avoid booking of same seat by multiple users.

Solution (1) : using Syncronized for critical section

```
Syncronized()
{
    Read SeatId
    If free:
        change state to Booked
    end
}
```

works well with single process system where we have multiple threads.

But won't work in distributed system where multiple instances of Service (Process) is deployed!

* what is DB Locking ?
  - Shared lock (R only)
  - Exclusive lock (R/W only)

	Another S	Another X
Have Shared Lock	✓	X
Have exclusive lock	X	X

dirty read: If a row is updated by a Tx. and and read by another Tx. before commiting. (As it can rollback as well)

non-Repetable read: If some Tx. is reading values multiple times and in bet^ it got commited by other Tx. then it leads diff outcomes. (Read only commited)

Phantom Reads: same "query returns diff number of rows at diff point in Tx. because some other Tx. added a row

Read uncommited: No Locks acquired in Read/write

Read commited: Read: Shared Lock acquired, and released as soon as Read is done.
write: Ex. lock acquired and keep till end of Tx

Repeatable Read: Read: shared Lock acquired and released at the end of Tx.
write: Ex. lock acquired and released only at the end of Tx.

serializable: Same as Repetable read but applied on the range of values

only for RO applications

	Dirty Read	non-rep. read	Phantom read
Read Uncommited	✓	✓	✓
Read commited	×	✓	✓
Repetable Read	×	×	✓
Serializable	×	×	×

concurrency ↑ as we move up.

⇒ lock entries are eventually available after Tx. but if other Tx. Times out on acquiring Lock/reading then that Tx. would be failed.

Using distributed Concurrency Control

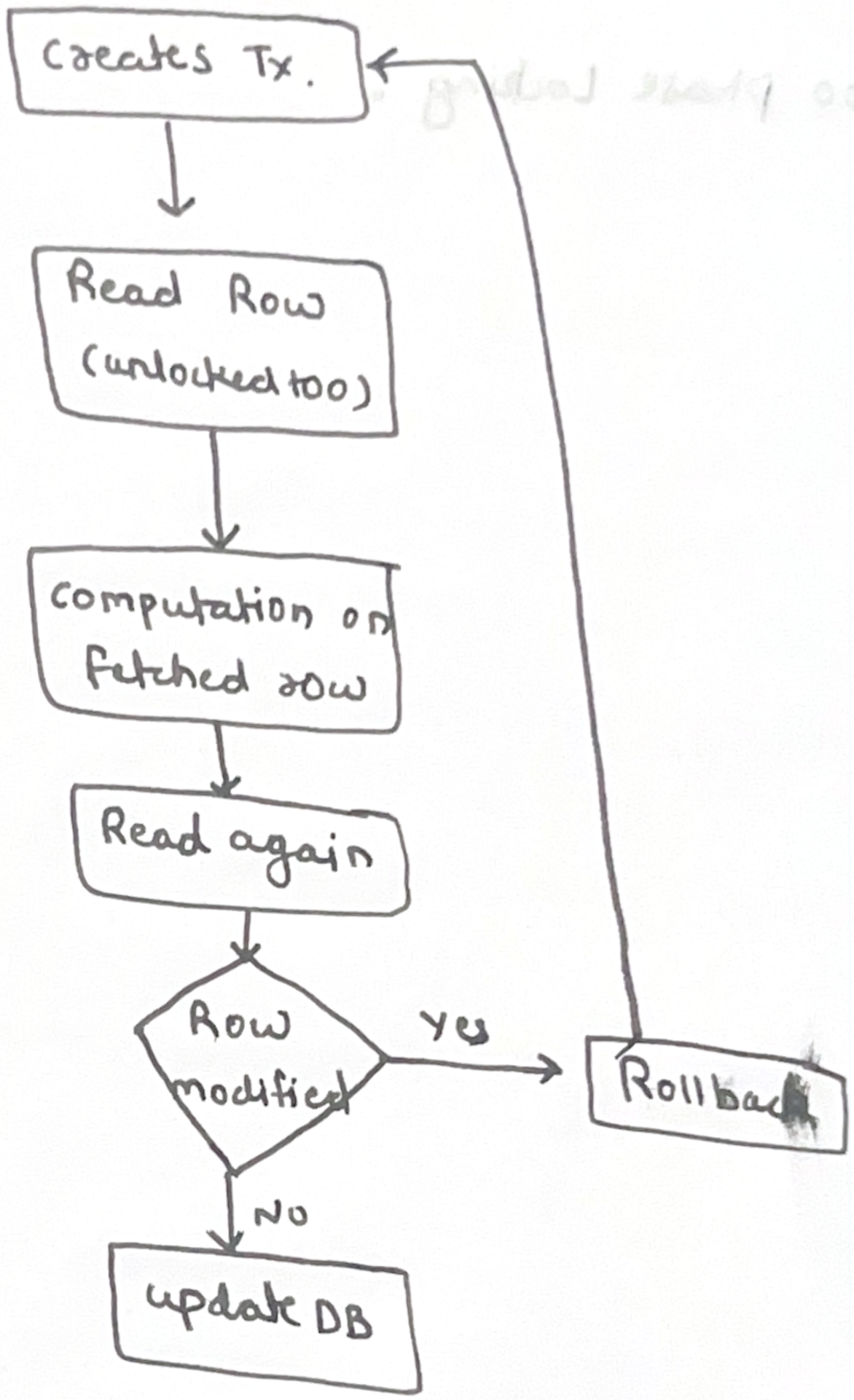
Optimistic concurrency control

- Iso. Level used below Repetable read
- much higher concurrency
- No chance of deadlock
- In case of conflict, overhead of Tx. rollback and retry logic is there.

Pessimistic concurrency control

- Isolation level used are Repetable read or seri.
- Less concurrency as comp. to opti.
- Deadlock is possible, Tx. in deadlock are forced to be rollback
- Putting long lock can cause timeout.

⊛ Optimistic cc :



⊛ pessimistic cc :

T_1	T_2
Read (A)	Updat (A)
Update (B)	
⋮	
COMMIT	

↖ this would have to wait till Tx. 1 is commit

deadlock case

T_1	T_2
Read A	Read B
write B	write A

- waiting for each other's lock to get removed