

Video 14: Spring Boot @Transactional Annotation

~Part 3 | Isolation

(Isolation Level.)

It tells, how the changes made by one transaction are visible to other transactions running in parallel

How to use?

```
@Transactional(propagation = Propagation.REQUIRED, isolation = ISOLATION.  
                READ_COMMITTED)  
public void updateUser() {  
    // do some operations  
}
```

If Isolation is not provided, the default Isolation is picked.
Default Isolation Levels depends on the database we are using.

(Most Relational Databases use **READ-COMMITTED**)

ISOLATION LEVELS	Dirty Read Possible	Non-Repeatable Read Possible	Phantom Read Possible
READ-UNCOMMITTED	YES	YES	YES
READ-COMMITTED	NO	YES	YES
REPEATABLE-COMMITTED	NO	NO	YES
SERIALIZABLE	NO	NO	NO.

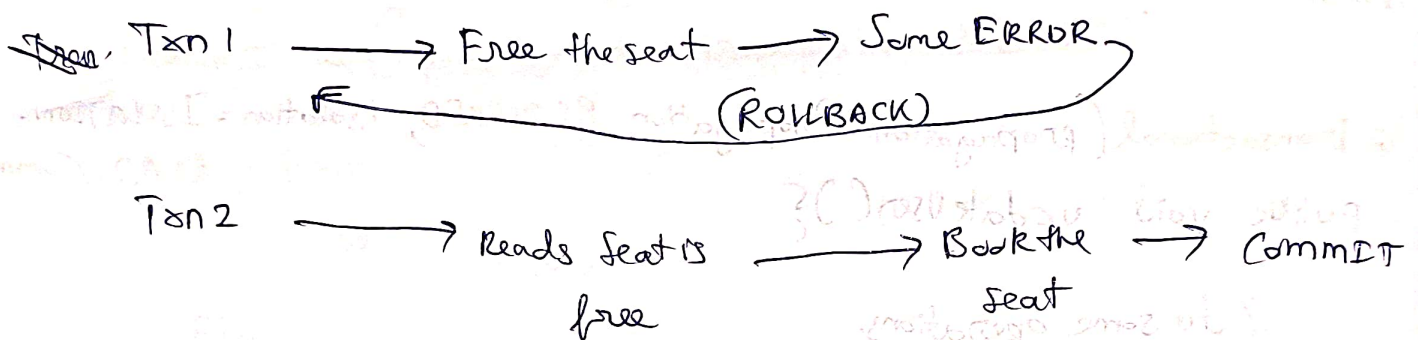
↑ Concurrency
↓ Consistency

[Dirty Read Problem]

Transaction A reads the un-committed data of other transaction.

if other transaction gets ROLLED BACK, the uncommitted data which is read by Transaction A is known as Dirty Read.

Suppose



Now Txn 1, Txn 2 both have seats.

But Txn 2 performed a dirty Read

[Non-Repeatable Read Problem]

If Suppose Transaction A, reads the same row several times and it gets inconsistent result. It got True but some time it gets false, then problem is Non-Repeatable Read Problem.

```
SELECT SALARY FROM EMPLOYEES WHERE
```

```
Emp_ID = '12345';
```

[Phantom Read]

[Example]

Employees EmpID	ROLE	SALARY
1	SDE1	10K
2	SDE2	20K
3	SDE2	20K
4	SDM	50K

Txn A, Txn B are running in parallel

Txn A → SELECT * FROM Employees where ROLE = SDE2

↳ {EmpID 2 and EmpID 3} (will be displayed)

Txn B → Meanwhile, due to promotion of EmpID=1,

(UPDATE ROLE = SDE2 WHERE EmpID=1)

Txn A → Again reading SELECT * FROM Employees where
ROLE = SDE2

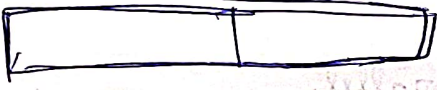
will get [EmpID = 1, 2, 3]

[During same ongoing transaction, for the same read,
the transaction got different results.]

(DB Locking Types)

	Shared Lock	Exclusive Lock
Shared Lock → Read Lock	✓	✗
Exclusive Lock → Write Lock	✗	✗

(Shared Lock) → It's mainly for reading purpose


Row →  If Txn1 is doing some operations on this row

if Txn2 wants to read the content of this row. It can read it.

But if Txn2 wants exclusive lock on it, it cannot do it.

(Exclusive Lock) → If Txn takes exclusive Lock on this

row. Then neither shared Lock, nor exclusive Lock can access that row.

 Mainly used for writing purpose

Read-Uncommitted: (Only to be used when application is ^{Read intensive})
No Read Lock acquired, No write Lock acquired,

Read Committed:

Write Lock \Rightarrow Exclusive Lock acquired and kept till the end of transaction

Read Lock: Shared Lock \rightarrow It is released as soon as read is done. But even though the transaction is complete, but still lock will be released after read and acquired again for another read.

Problem of Repeatable reads occur in this one.

Txn 1

take Lock

Read row where row_id = 1

release Lock

Do some operation.

Txn 2

take Lock

\rightarrow Update row_id = 1

release Lock

take Lock

Read row where row_id = 1

release Lock

Different Result

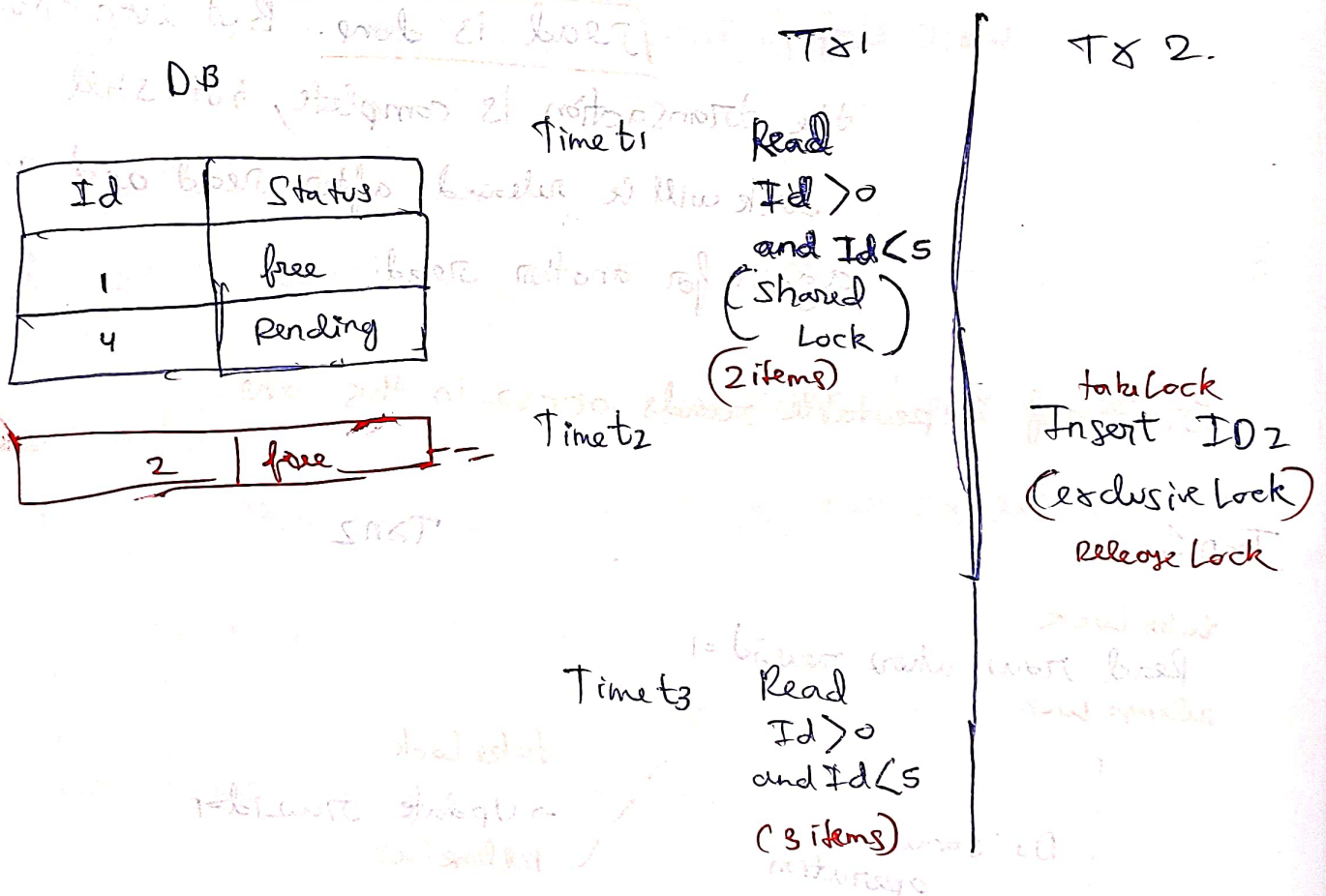
(Non-Repeatable Read Occurred)

[Repeatable Read] ÷

Read :- Shared Lock acquired and released at the end of transaction

Write ÷ Exclusive Lock acquired and released at the end of transaction

(Problem of Phantom Reads happens in this)



[Serializable]

Uses the Locking strategy of [Repeatable Read]

+
apply Range Lock and release at the end of the transaction.

Range Lock is applied.

Suppose

Read $Id > 0$ and

$Id < 5$

(Shared Lock)

$\rightarrow Ids [1-4]$

will be

locked.

T_{N2} at time = T_2

trying to insert $Id=2$. (Exclusive Lock)

\rightarrow [will not be allowed to insert it into the table]

this leads to highest Consistency but lowest concurrency.