
Agenda

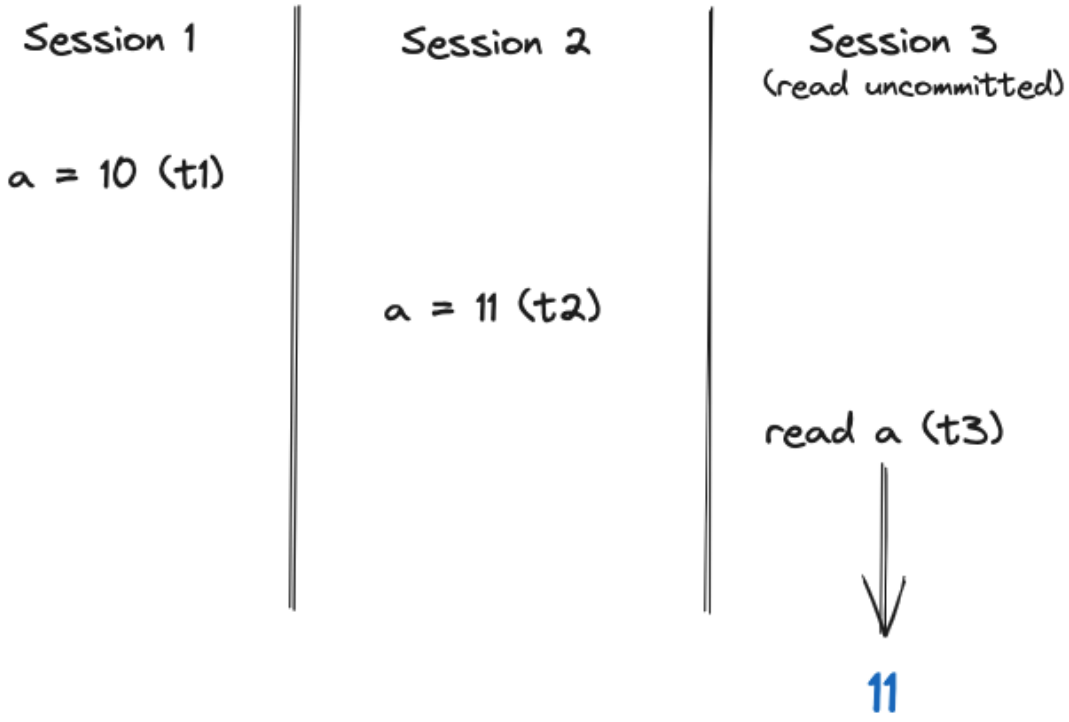
- Read Uncommitted
- Read Committed
- Repeatable Reads
- Serializable
- Deadlocks

> *Quick Recap:*

Transaction Isolation Level - Read Uncommitted

- It allows a transaction to read even uncommitted data from another transaction.
- Reads the latest data (committed or uncommitted).

Assume there are 3 different sessions:



- Isolation level only takes about how your session will work.
- The isolation level of other transactions doesn't matter to you.

Now to understand this, let us see the example.

- To read the current level, execute the following statement:

```
SHOW variables LIKE "transaction_isolation";
```

- It will output the following:

variable_name	value

transaction_isolation	REPEATABLE_READ

- Let's change it in session 2:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITED;
```

- Now, when we try to read it, it will be "READ-UNCOMMITED". (For Session 1 it will remain RR).

Now, let's see another example.

- Run the following statements in session 1:

```
set autocommit = 0;
SELECT * FROM film
WHERE film_id = 10;
```

  - The title is "Rahul2"
- Let's change it to "Rahul3"

```
UPDATE film
SET title = "Rahul3"
WHERE film_id = 10;
```

Now, earlier when we were trying to read this in session 2, it was not showing the updated title. **But** now if we execute the following in session 2:

```
SELECT * FROM film
WHERE film_id = 10;
```

We will see "Rahul3". **Because it is the latest uncommitted value.**

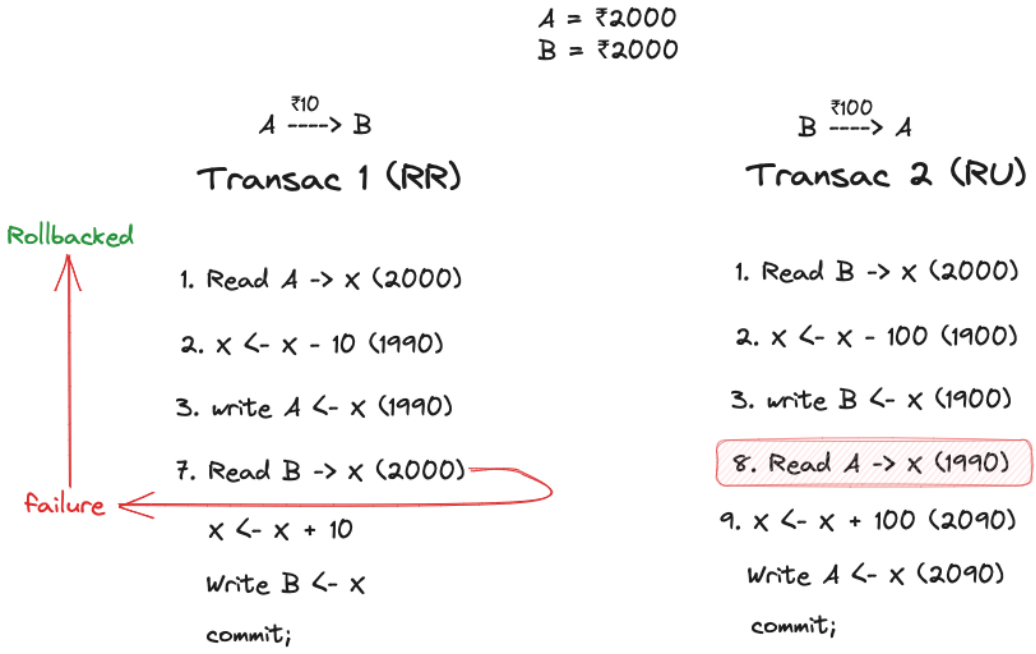
Pros

- Fast

Cons

- Read uncommitted may lead to inconsistency.

Let's understand the con with an example.



In the above example, we ran into a consistency problem because at line 8 in transaction 2, it read the data which was not committed and later rolled back.

Dirty Read

This kind of problem is called a dirty read.

Dirty Read: When a transaction ends up reading data which may not be committed.

Read Committed

We discussed about dirty read.

- Dirty Read: Reading some data that is not confirmed yet.
- A data is confirmed once it is committed.

Now, how do you ensure only committed data is read? By using **Read Committed** isolation level.

Read Committed: Your transaction will only read the latest committed data.

Let's understand with an example:

Start session 1 and execute the following queries:

```
SET autocommit = 0;
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT FROM * film
WHERE film_id = 10; -- This will return "Rahul3"
UPDATE film
SET title = "Rahul4"
WHERE film_id = 10;
```

```
-- Changes not committed yet
```

Now **start session 2** and execute the following queries:

```
SET autocommit = 0;
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM film
WHERE film_id = 10;
```

Now, what do you think will be the output?

It will be "Rahul3". Why not "Rahul4"?

Because the transaction isolation level of the session matters and not the transaction isolation level of the session that updated the value.

Now what if we execute the following query in session 2:

```
UPDATE film
SET title = "Rahul20"
WHERE film_id = 10;
```

> *Note: the above query will keep loading.*

Now do you see some problem is happening? Why is it waiting?

This is because of something called "locking".

As soon as we commit the first transaction, the query in session 2 will execute.

There are two types of locks:

- Shared lock (Caused by a read query)
- Exclusive lock (Caused by a write query)

Someone has	Someone else can
shared lock	read and write the same row (explain with example)
exclusive lock	read the same row but can't write (explain with example)

If one transaction has written on a row, no other transaction will be allowed to write on that row.

> *Note: We will discuss locks again in the later part of the notes again.*

[Time to think:] Now do you understand why performance starts getting affected when you do RC instead of RU?

Because when something is being read or written in RC, a lot of locks and concurrency controls starts happening.

Summarizing the read committed isolation level:

- A transaction will read always the latest committed value.
- It will never read an uncommitted value.
- That means Dirty read will not happen.

[Time to think:] Is this the best solution? Will there be no problems in this?

Let's discuss some problems with read committed.

Problems with read committed.

Let's understand the problem of read committed through an example:

Suppose you have a *users* table:

id	name	psp	email-sent
1	N	60	false
2	A	80	false
3	M	70	false
4	AB	45	false

Suppose for all students having psp less than 80, I want to send an email to them.

[Time to think:] How the flow of above task will look like ??

A structured flow of above task will look like this:

1. Get list of all users having psp<80 `list<user> = SELECT * FROM users WHERE psp<80`
2. Send email to the obtained list of users (Let's assume this operation took 5 minutes)
3. Update the *"email-sent"* column for users to whom the mail has been sent.

```
UPDATE users
SET email_sent = true
WHERE psp < 80;
```

[Time to think:] What can probably go wrong here ? What if someone's psp value get changed in between this process?

Let's clear this by discussing specific user **(User 2)** of above example:

Step 1: Getting list of users having psp<80 *(User with id = 2 will be definietly **not** a part of this list)* **Step 2:** Sending email to the users of obtained list *(Since User 2 is not in the list, they will not receive the email.)* **Before Step 3:** *Let's assume due to some reason the psp value of user 2 got changed to 79.Now, the psp value of User 2 is less than 80.* **Step 3:** While updating the "email_sent" column through the below query:

```
UPDATE users
SET email_sent = true
WHERE psp < 80;
```

it will incorrectly set that User-2 has also received the mail since the user's current PSP is less than 80. However, here we encounter an issue because in reality, User-2 didn't receive the mail. **This is called as non-repeatable read which occurs when a transaction reads the same row twice and gets a different value each time**

- **What should be ideal case here ?** Within a transaction If I read the same row again it must have the same value.

The main problem here is: In the read-committed isolation level, the transaction will always read the latest committed value, even if the latest commit is done after the transaction has started.

[Time to think:] How can we solve this problem of non-repeatable read ?

By using **Repeatable Reads** isolation level.

Repeatable Reads

How does Repeatable Reads work?

In the repeatable reads isolation level, whenever it has to read a row:

- For the very first time, it reads the latest committed value of the current row.
- After that, until the transaction completes, it keeps on reading the same value it read the first time.

Let's understand this with an example:

Start session 1 and execute the following queries:

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN;
SELECT * film where film_id = 13  --> This will return a row of film having name ALI FOREVER
```

Now, start session 2 and execute the following queries:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
START TRANSACTION;
UPDATE film SET title = "Deepak" WHERE film_id = 13
COMMIT; -->The below  mentioned detail will work same with or without this
```

[Time to think:] Now, if you try to run the query from the first session again, what will be the output this time?

The output will still be the same: `ALI FOREVER`, as in this session-1, `ISOLATION LEVEL REPEATABLE READ` is set, and for further reads, it keeps on reading the same value it read the first time until this session's transaction completes.

NOTE: After committing in transaction 1, if you try to read the value again from session-1, it will show film_name as "Deepak".

Summary:

- Dirty Read: This problem was solved by Read Committed
- Non-Repeatable Read: This problem was solved by Repeatable Read.

[Time to think:] Is repeatable read best ? **Ques:** Repeatable read keeps snapshot of which rows ?? Repeatable read doesn't only keep snapshot of the row that you read but also take snaps of other set of nearby rows.

Let's get this point with an example:

Session-1

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
START TRANSACTION;
SELECT * film WHERE film_id = 12; --> Initial film_name: Naman
SELECT * film WHERE film_id = 13; --> Initial film_name: Deepak
```

Now in **Session-2**

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN;
SELECT * film where film_id = 12 -->  Initial film_name = Naman
(Till this moment name of film at film_id = 13 is still Deepak and snap with Deepak is
taken)
```

Let's change film Deepak into Aniket in **session-1**

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
START TRANSACTION;
SELECT * film WHERE film_id = 12; --> Initial film_name: Naman
SELECT * film WHERE film_id = 13; --> Initial film_name: Deepak

UPDATE film SET title = "Aniket" WHERE film_id = 13;
COMMIT;
```

Again read film_name at film_id = 13 in **session-2**

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN;
SELECT * film where film_id = 12;
SELECT * film where film_id = 13; --> film_name is still "Deepak"
```

Even if the value for `film_id = 13` is read the first time, after the update, the latest committed value of this row is not taken into consideration because when initially `SELECT * FROM film WHERE film_id = 12;` was executed, snapshots of a few more rows were already taken.

> ***NOTE:*** *Instead of doing these changes for film_id = 13 if we had done this for some other film_id we might have got updated value as output.*

The new problem: During this process, if someone adds a new row, its snapshot was obviously not taken before. This can create a problem known as a **phantom read.**

Phantom Read: A row comes magically that you have never read earlier.

A phantom read occurs when a transaction reads a set of rows that match a certain condition, but another transaction adds or removes rows that meet that condition before the first transaction is completed. As a result, when the first transaction reads the data again, it sees a different set of rows, as if "phantoms" appeared or disappeared.

How to solve Phantom Read problem ? By Using concept of Serializable

---

Serializable

This technique ensures that multiple transactions or processes are executed in a specific order, one after the other, even if they are requested simultaneously.

> *Activity: Let's understand the difference between the types of isolation with the **"Show Tickets" and "Book Tickets"** functionalities on the BookMyShow website by showing a similar booking in two different tabs. Show Ticket: In the "Show Tickets" functionality, both users can select the same seat. Book Ticket: In the "Book Tickets" functionality, only one person can proceed to book one ticket.*

[Time to think:] What will you read, or do you even read from any row? Will it depend on your isolation level or others' isolation level?

It depends on your isolation level.

In serialization, you are only allowed to read rows that are not locked. In other isolation levels, you can read any row without checking for allowed rows. In another isolation level, which version of the row you'll read was the problem.

Example of Isolation Level in Bank

1. Set isolation level serialisable in two different sessions (In a bank transfer between 2 users if both are not having same isolation level, it doesn't make proper sense)
2. Perform following query in **session-1**

```
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE
START TRANSACTION
SELECT * FROM film WHERE film_id in(12,13) FOR UPDATE;
```

3. Perform following query in **session-2**

```
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE
START TRANSACTION
SELECT * FROM film WHERE film_id in(13,14) FOR UPDATE;
```

> *Note: The above query will keep loading, as lock is already taken on row 13 so even read is not allowed now If FOR UPDATE wasn't used in session-1, then it should be allowed to read in session-2*

Thus, serialization makes the concept of one person at a time, avoiding concurrency and maintaining consistency with complete isolation.

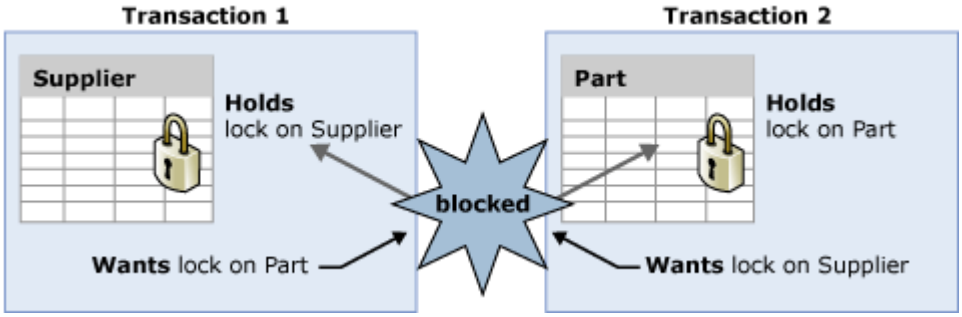
Deadlock

In transaction when we write or we read for updates, do we take a lock on networks ?? Yes.

But this lock can lead to deadlock condition. This problem happens when two people are waiting on resources hold by each other and not able to make a progress.

Example:

1. Suppose there are two users, T1 and T2, each requiring access to resources A and B.



2. User T1 has acquired a lock on resource A and is trying to obtain a lock on resource B to accomplish their action.
3. At the same time, User T2 has acquired a lock on resource B and is trying to obtain a lock on resource A to accomplish their task.
4. Both are waiting for each other to release their occupied lock.
5. T1 is waiting for T2 to complete and T2 is waiting for T1 to complete and no one is making progress. This condition is know as deadlock.

How we handle Deadlock in SQL: It automatically rollsback one of the transaction.

Way to avoid Deadlock: take locks in defined ordered way

> ***More on deadlocks:*** *We'll discuss more about deadlock in further classes of next module.*