
Agenda

- What are transactions?
- Properties of transactions?
 - Atomicity
 - Consistency
 - Isolation
 - Durability
- How do transactions work?
 - Read
 - Write
- Commits and Rollbacks

We will try to cover most of these topics here and the remaining in the next part of Transactions.

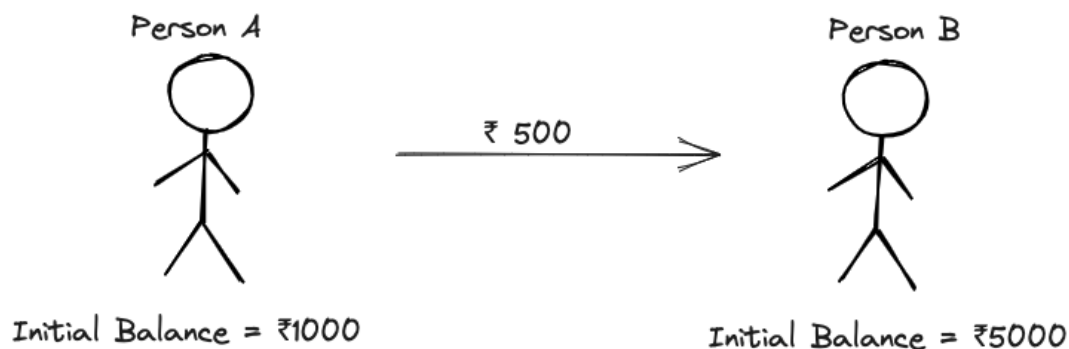
It is going to be a bit challenging, advanced, but very interesting topic that is asked very frequently in interviews, such as ACID, lost updates, dirty and phantom reads, etc.

So let's start.

What are transactions?

Till now, we have written a query. We executed it, and it worked. Sometimes, instead of writing one simple query, we might need multiple queries to execute together.

For example - **Transferring Money in a Bank**



Now, to transfer the money from A to B, what all changes are needed in the database of the bank?

[Activity:] Where do you think the information regarding the balance etc., of a customer is stored in the bank? in which table?

--> Probably a table called `account` with a schema such as:

`accounts`

--	--	--	--	--	--

id	name	balance	created_at	branch	...
1	A	1000	-	-	-
2	B	5000	-	-	-

Where - represents some value (not relevant to our example).

Now, let's see what all things need to happen in this `accounts` table for the transaction to complete.

[Time to think:] Can the first step be "reduce this much money" from A? **No** because what if A doesn't even have that much money to transfer?

The steps will be as follows:

1. Get the balance of A. (DB call)
2. Check if the balance \geq 500 (no DB call)
3. Reduce the balance of A by 500. (DB call)
4. Increase the balance of B by 500. (DB call)

Now, to do this, we will probably have a function as follows in our application code:

```
transfer_money(from, to, amount) {
    // multiple SQL queries with conditionals
}
```

Will that be enough? Let's think about what could go wrong using such a function call.

[Let us think:] Do you think, at one time, only one person would be calling the `transfer_money()` function?

--> **No**

There might be a situation where `transfer_money()` may be getting executed by multiple people at the same time :

- A --> B (read --> as "transferring money to")
- C --> D
- E --> B
- D --> A

Let's understand what can go wrong in such a situation with an example.

Let's say two people (A and C) are transferring money to the same person (say, B) at the same time.

- A --> B
- C --> B

Such a situation can happen in the case of **Amazon** where multiple people are sending payment to the same merchant at the same time.

Now before continuing with our example, let's first ask ourselves - is updating the balance of B a single operation?

```
B += 500 --> B = B + 500
```

Which behind the scenes converts to:

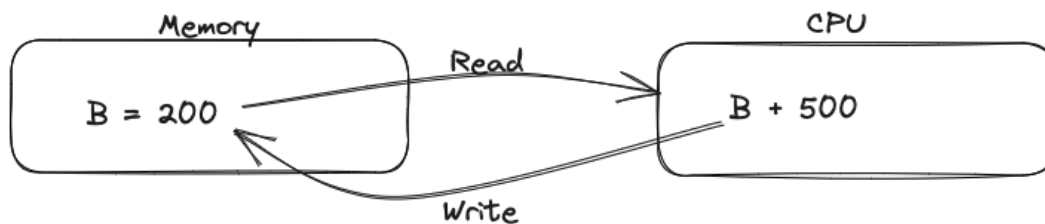
- read the current value of B in a temporary variable temp .
- Increase the value of temp by 500.
- Write the value in temp back to B .

Even though it looked to us as 1 update statement, behind the scene, it was multiple (read and write).

Let's write it as a pseudocode (Note to instructor - write non DB statements differently than DB statements) -

```
transfer_money(a, b, amount) {
  read A -> x
  if (x >= amount) {
    write A <- x - amount
    read B -> x
    x = x + amount
    write B <- x
  }
}
```

To be clear here, we can't update the value of B directly because any calculation happens in the CPU and before updating, the CPU must first get the value from memory/disk.



What could go wrong - 1

Now back to the topic. What could go wrong in our example?

- A --> B
- C --> B

Initially:

- A = 1000
- B = 5000
- C = 15000

[Time to think:] Two people are performing the same operations at the same time, but at the end, can two things be written on the memory at the same time?

No, one will happen just before the other.

Let's run through our function line by line assuming that A --> B (represented in orange) transfer is running one step ahead of C --> B (represented in blue).

A --> B

a = 1000, b = 5000, amount = 500

C --> B

a = 15000, b = 5000, amount = 10000

```
transfer_money(a, b, amount) {
  read A -> x
  if (x >= amount) {
    write A <- x - amount
    read B -> x
    x = x + amount
    write B <- x
  }
}
```

x = 1000

x = 15000

A = 500

C = 5000

x = 5000

x = 5000

x = 5500

x = 15000

B = 5500

B = 15000

Note: Here, we are running these queries concurrently/Parallelly step by step.

[Time to think:]

- Before transactions, what was the total money in the bank?
 - ₹ 21000
- After both transactions are done, what's the total money in the bank?
 - ₹ 20500

What happened here?

Money went missing.

The A --> B transaction made the balance in A 500 from 1000, and 5500 in B from 5000, but before this transaction ended, the C --> B read the value of B as 5000 instead of 5500 and made it 15000.

So, do you see, when one particular operation requires us to do multiple things, it can actually lead to wrong answers unless we take care.

What could go wrong - 2

What else could go wrong?

Let's say the DB machine goes down after the `Write A <- X` statement. What will happen?

Money is gone from one person, but the other person never received it.

What is a transaction?

We saw the following two problems that can happen when executing multiple queries as part of a single operation.

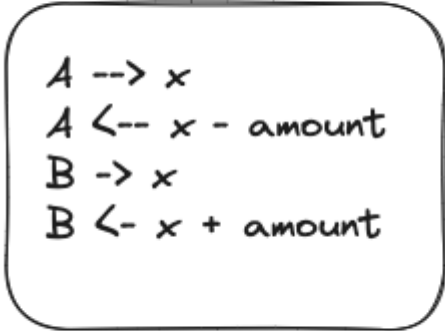
1. Inconsistent/ Illogical State
2. Complete operation may not execute.

These are the problems a **transaction** tries to solve.

Transaction: A set of database operations logically grouped together to perform a task.

Example - Transfer Money

Transaction: transfer money

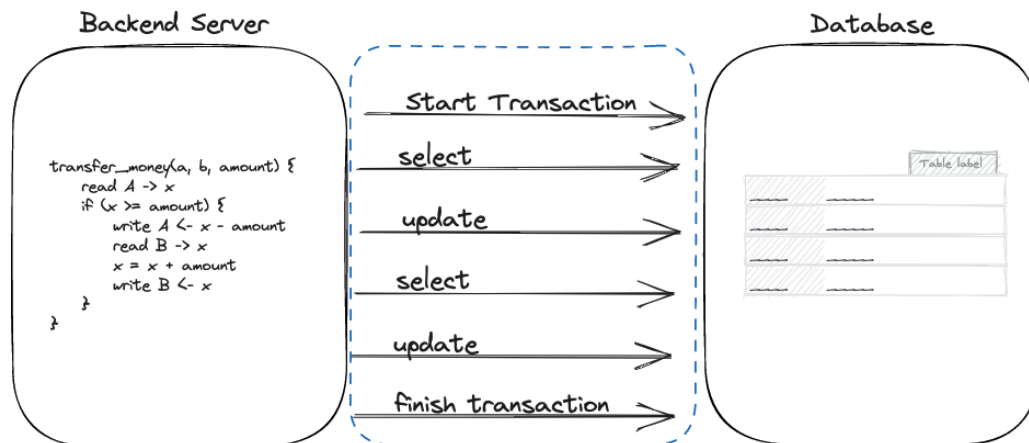


A \rightarrow x
A \leftarrow x - amount
B \rightarrow x
B \leftarrow x + amount

What do I mean by "Logically Grouped Together"?

It means they are grouped together to achieve a certain outcome, like transferring money.

When you have to perform such a task, you say, "I am starting a transaction", after that you perform the queries and then you "finish the transaction".



Basically, a transaction is a way to tell the DB that all these following operations are part of doing the same thing.

Now that we know what a transaction is. We will be discussing what are some properties a transaction guarantees us so that things that can go wrong don't actually go wrong.

ACID Properties

Now, let's discuss what are the expectations from a transaction.

Most of the time, when you are reading a book, they will talk about the "properties" of a transaction but we have used the word "expectations" for a reason.

[Time to think:] How many of you have heard about ACID properties?

[Time to think:] How many of you think ACID properties **must** always be there in a transaction?

So here's the thing - ACID properties may or may not be there in a transaction. It depends on our usecase.

ACID properties are 4 expectations that an engineer might have when they create a transaction or execute a group of SQL queries together.

They are:

- Atomicity
- Consistency
- Isolation
- Durability

Now let's discuss these expectations one by one.

ACID Properties - Atomicity

Atomicity

[Time to think:]

- The word atomicity is formed out of which word? **Atom** which means the smallest **indivisible** unit (at least for a long time before the discovery of sub-atomic particles)

Now how does it relate to Databases?

For us, atomicity does not mean the smallest unit, it means the indivisible unit.

The property of atomicity means the transaction should appear atomic or indivisible to an end user.

Analogy

Take yourself back to your childhood. When you were a kid, did "transferring money" mean checking balances, reducing from one account and adding to another?

No.

It was a simple give-and-take for us.

This is how a transaction should appear to an end user.

- **To an outsider, it should feel that either nothing has happened or everything has happened.**
- This means that a transaction should never end in an intermediary state.

Activity: Go back to the `transfer_money()` function example and observe the opening and closing curly braces. They ensures Atomicity and keeps overall functionality of a Transaction under a single hood.

Usecase

The property prevents cases where money gets deducted from one account but didn't credit to another account.

[Time to think:]

- Can you think of a usecase where atomicity may not be required?
 - **Google Profile Picture Update:** It may happen that after updating the profile picture, it is updated in Gmail but not in youtube and may take up to 48 hours to update accross different services.

ACID Properties - Consistency

Consistency means:

- Correctness
- Exactness
- Accuracy
- Logical correctness.

What do we mean by this?

[Time to think:]

- In our original example - was atomicity handled for both accounts A and C?
 - Yes. it was.
 - Money transfer was completed for both of them. The transfer was not left in between.

Now, consistency comes into play when you start a work, and you complete it **but** you don't do it in the correct or accurate way.

[Time to think:]

- In our original example - `transfer_money`, both A and C completed, but was the outcome correct?
 - No.
 - The money was somehow lost.
 - It leads the DB to an inaccurate state.

In the bank example, consistency is important, but is it always the case?

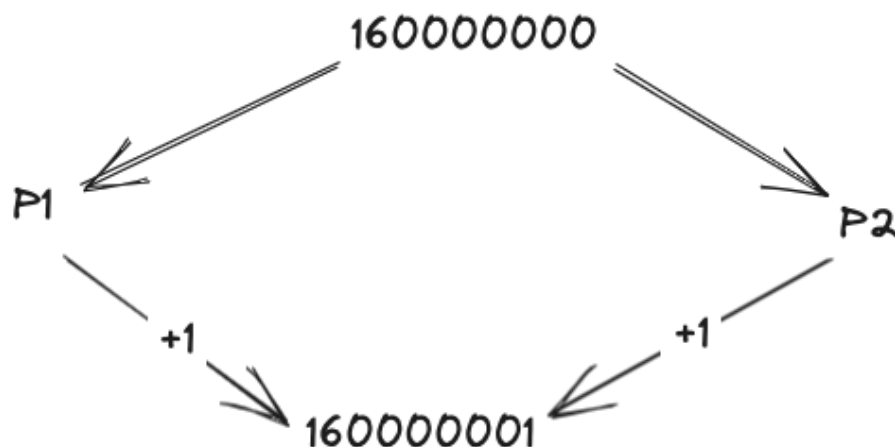
Let's take the example of **Hotstar**. Let's say we a `live_streaming` table.

stream_id	count
1	160000000
...	...

Let's say a particular stream with 1.6 crore people watching. Now, 2 people at the exact same time start watching the stream too.

So for both of them, how would the SQL query look like?

1. Get the current count of viewers.
2. Update count.



Both update the count to 160000001 but is that a critical situation?

- Will the world go gaga over this?

It doesn't matter whether it's 160000001 or 160000002 if it is improving the performance.

So, consistency is something that might be expected but not always. It depends on the use-case.

ACID Properties - Isolation

Now, let's discuss the third one. That is isolation.

Isolation means one transaction shouldn't affect another transaction running at the same time on the same DB in the wrong way.

[Time to think:]

- In the `transfer_money` example if there was just one transaction happening would there have been any problem?
 - No.
 - The things went wrong because there was another transaction that started happening on the same person.
 - The two operations were not completely separate.

So what does the word Isolated mean? It means **separate**. That is not aware or completely apart from each other.

In our example, the transactions were interfering with each other.

The inconsistency that happened was actually because of the isolation not being present.

We need to understand that two transactions can happen at the same time but it is important to keep them isolated and not let them interfere with each other.

In the hotstart example, if there was no isolation, would that have been okay? - Yes.

There are multiple levels of isolation that we will be discussing later.

ACID Properties - Durability

Now, let's discuss the last one. Durability.

Durability is once a transaction is completed. We would want its work to stay persistent.

It should not happen that you make a money transfer today and it is completed, but tomorrow that transfer is no longer present in the history logs.

This means that whenever a transaction is complete, whatever the result was should be stored on a disk.

- Saving on a disk takes time.
- By default, every DB may not update the disk.

It depends on our use-case and how we have configured the DB.

There are DBs that give us fast writes. How? By just updating in-memory instead of the disk.

- Would they have durability? No.
- Will they be fast for writes? Yes.

[Time to think:]

- Can a bank use such a DB that provides fast write by avoiding writing on disk?
 - No.
 - It is important for a bank to ensure the changes are persistent on a disk.

"Everything in engineering comes with a tradeoff and we should not assume that a DB provides something without properly checking and understanding how it works."

Now, durability has different levels:

- What if disk gets corrupted after saving?
- Then we might need to store on multiple disks.
- What if there is a flood in the data center?
- Then we might need to store in multiple data centers
- and so on...

Theoretically, we can't have 100% durability. How much durability we should have depends on the use-case.

ACID Properties - Summarized

Let's summarize the ACID properties:

- Atomicity - Everything happens or nothing happens.
- Consistency - Whatever happens should happen in the correct way.
- Isolation - One transaction should not interfere with another transaction, consequently giving a wrong outcome.
- Durability - Once the outcome has been achieved, the outcome should persist.

Commits and Rollbacks

Now, we will discuss the practical part. How to actually use the transactions in the DB.

Let's say you have a students table:

id	name	psp	batch
1	Naman	70	2
...

If I write a SQL query:

```
UPDATE students
SET psp = 80
WHERE id = 1;

SELECT * FROM students;
```

[Time to think:]

- What are you expecting the psp of id 1 to be?

- 80.
- We are saying 80 assuming that the query got written to the DB.

When I write a SQL query, it automatically gets updated to the DB. Why is that?

That happens because of something called **auto commit**.

Whenever you write a simple SQL query, it starts a transaction, after that it executes itself and then save the changes to DB automatically.

We only have to start a transaction when it happens for multiple queries.

Here "saves the changes" means "commit". Committing is similar to a promise or a guarantee.

Commit is a keyword/clause/statement to persist the results of a SQL query.

[Time to think:]

- Which property of ACID is taken care of by `commit` ?
 - Durability.

By default, in MySQL and most of the SQL DBs, `autocommit` is set to true. But we can keep it off based on the use-case.

Let's see an actual example in the MySQL visualizer.

- Create a file `transact.sql` .
- Add the first command and execute it:

```
set autocommit = 0;
```

- Now, let's open a table.

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

- Which film do you see?
 - "Aladdin Calendar"
- Now let's update it:

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

[Time to think:]

- Now, if we open a new session and try to execute the following statement:

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

What will we see?

- We will still see "Aladdin Calendar" and not "Rahul" because we haven't committed it.

Other people or different sessions won't be able to see the changes we made until we commit the changes manually. So how to do that?

- Write and execute the following query:

```
commit;
```

Now if we try to check again in a different session, it will be the updated value of "Rahul" in the title attribute.

Activity: Run the above example again with `autocommit` set to 1.

[Time to think:]

- Why is commit needed separately, and why we can't always have an autocommit?
 - Because it depends on use-case.
 - There may be a situation where we do a few changes but then realize that we don't want to commit them.

For example - You were doing the money transfer from A to B, and after deducting the money from A you realized that the transfer may be spam or suspicious. In this case, we will have to undo or revert the changes done till that point.

- Now how you can do that?
 - By using another keyword/clause/statement called **Rollback**.

As commit allows us to save the changes, rollback allows us to revert the changes.

Rollback allows us to revert the changes since last commit.

Let's see an example:

- Run the following statements:

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

- Currently, the title is "Rahul". Let's change it to "Rahul2".

```
UPDATE film
SET title = "Rahul2"
WHERE film_id = 10;
```

- Now if we read in the same session the title will be "Rahul2". But if we rollback instead of commit. It will go back to "Rahul".

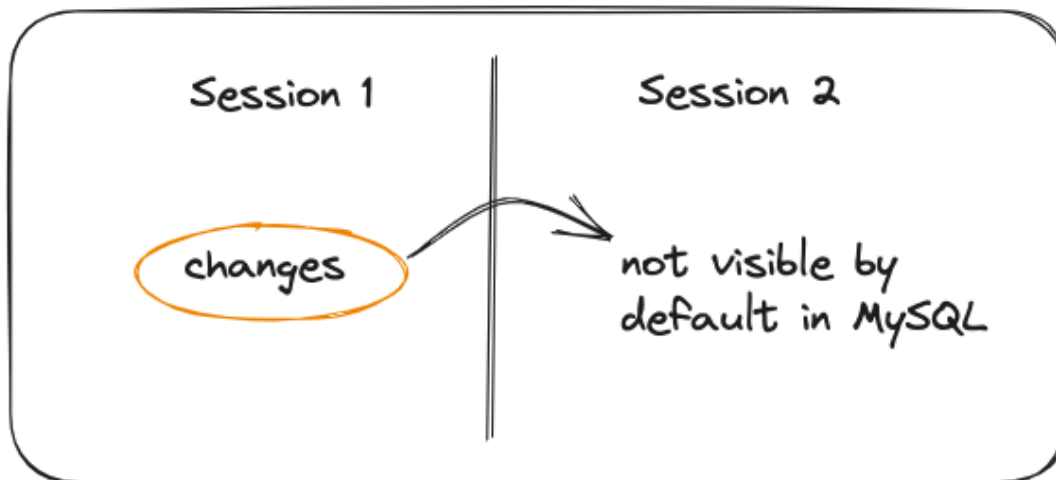
```
rollback;
```

Summarize:

- Commit: like a marriage | persist the changes to DB.
- Rollback: like a break up | undo the changes done since the last commit.

Transaction Isolation Level

Did you notice when we were doing changes in session 1, those changes were not yet visible in session 2.



[Time to think:]

- Was it only because the changes were not yet committed?

That was part of the reason but there are other reasons as well.

This is Isolation

By default, MySQL keeps both of the sessions isolated.

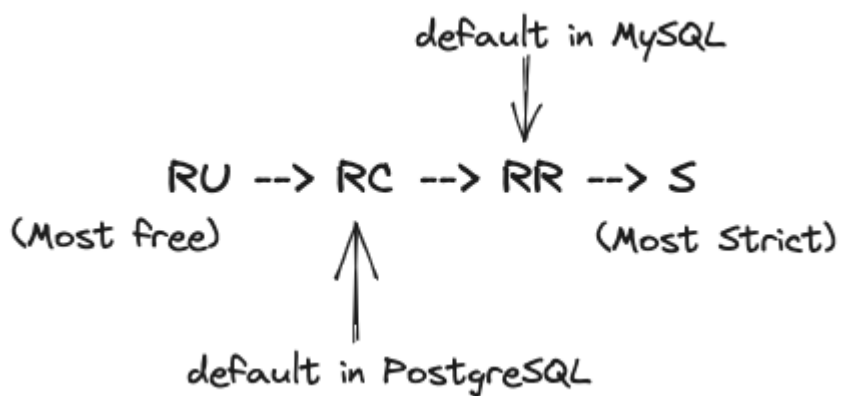
- One session see a snapshot of the table that the other doesn't.

MySQL supports 4 levels of isolation:

1. Read Uncommitted (RU).
2. Read Committed (RC).
3. Read Repeatable (RR).
4. Serializable (S).

Here, we will discuss the first one and the remaining three in the next notes.

The isolations levels are written in order of increasing severity.

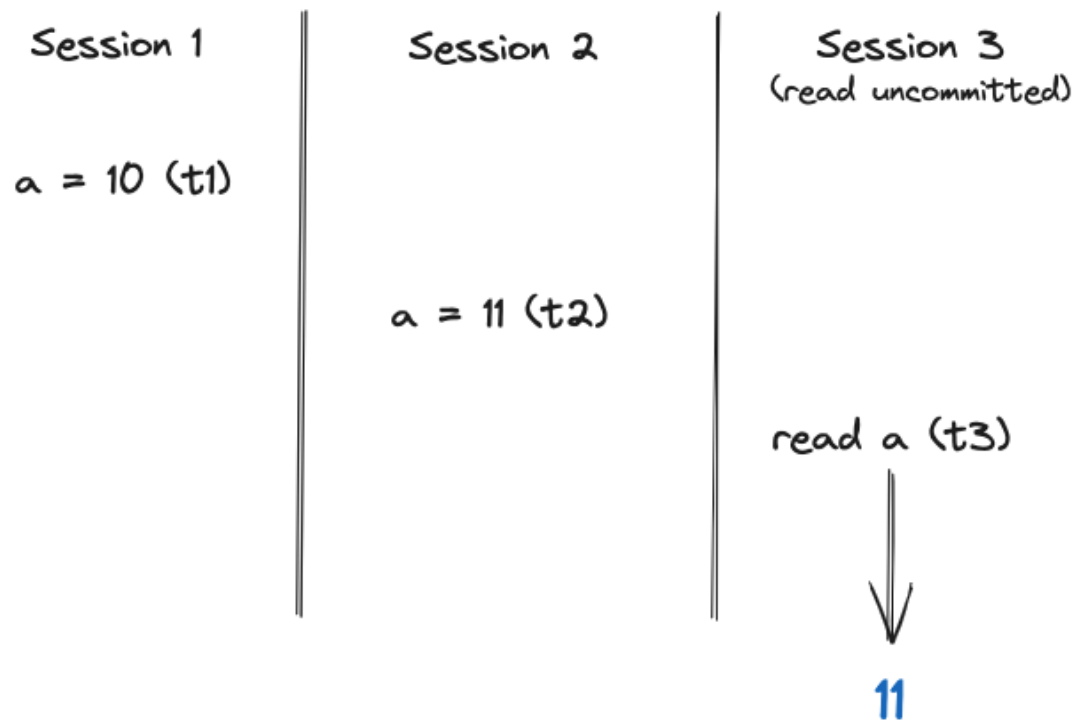


Note that when we talk about isolation, we are talking about the restrictions on reading the data and not updating it.

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 UNCOMMITTED;
```

- Now, when we try to read it, it will be "READ-UNCOMMITTED". (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 "Naman2"
- Let's change it to "Naman3"

```
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;
```

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

Pros

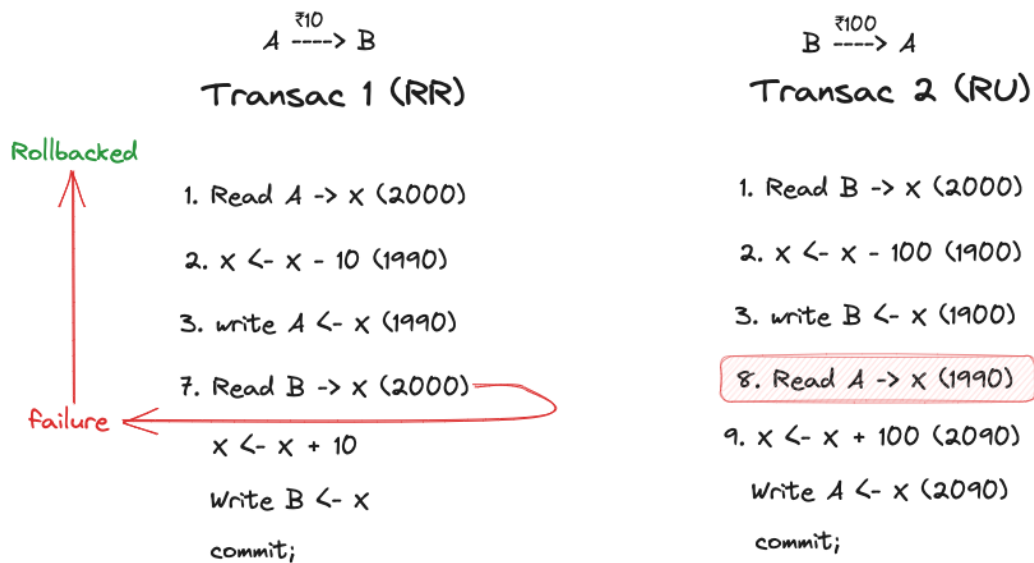
- Fast

Cons

- Read uncommitted may lead to inconsistency.

Let's understand the con with an example.

A = ₹2000
B = ₹2000



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.