DBMS Lecture 7

Transactions:

==Set of logically related operations.==

Example of Bank:

send money(fromaccount, toaccount, amount) {1. t=get current balance of fromaccount 2. t=t-amount. 3. Store t in the database. 4. t= get current balance of toaccount. 5. t=t+amount 6. store t in database. }

So a single operation which is to send money from A to B, it can actually be broken down into multiple operations that happen behind the scenes.

Now, Let's take a scenerio.

In above query, we started sending money, got the current balance of fromaccount you seperated that and store it in the DB, got the current balance of toaccount but after this some crash happened. Now basically the money has been lost, which is not a desirable situation.

Another Scenerio,

Both of this is happening at the exact same time. This is not right, first B has 700000 then 200500. This is an issue of concurrency. When multiple transaction happens at the same data at the exact same time, how to ensure that they don't conflict with each other, so we have to handle concurrency as well.

==So, Transactions are nothing but a set of logically related operations which should work together to achieve the outcome.==

ACID Properties:

A: Atomicity C: Consistency I: Isolation D: Durability

1. Atomicity:

Atom is smallest indivisible particle, smallest unit - cannot break further. - It appears to be a single unit. - Either everything should happen or nothing should happen.

Transaction internally has multiple operations but as an observant it should appear to be like a single unit. *For Example:* **TRANSFER_MONEY**, it should appear like a single unit.

In the sense, there will be not be a case that the transfer_money transaction runs and the money gets deducted from A but is not added in B. In this case observant will feel that internally multiple things are happening that's why something happened but remaining thing didn't happen. - To an observer/ outsider a transaction should appear like a single operation.

2. Consistency:

- A transaction should never leave the DB in an inconsistent state.
- Example: In bank, money got deducted from A but not added in B.
- The system should be consistent before the transaction starts as well as after the transaction ends.
- In b/w the transaction, it is ok if temporarily it is inconsistent. *Example:* In above case, when the money was deducted from A but didn't get added to B yet it was an inconsistent state but it is a temporary internal inconsistent state and after the transaction will end it will be consistent again.

3. Isolation:

Multiple transactions happen on the same data at the same time. - Other transactions running in the system should not affect a given transaction. - Every transaction should be independent and produce desire results. In above example: There was a transaction T1, and the transaction T2 both are working with the shared data B. Both try to change the value of B at the exact same time and because of presence of each other, they ended up producing the wrong result.

4. Durability:

- After a transaction has ended, its result should persist.
- In bank, A sent money to B. After 2 days they check their balance and it's gone. It should not be the case.
- They should avoid hardware failures, etc.

==A-> Everything should run or nothing should run C-> Consistent before and after I-> Not affected by others D-> Once succeeded, no data loss.==

In practice, a transaction can be considered to involve 2 types of operations: 1. Read 2. Write

We have discussed 4 operations: C-> Write R-> Read U-> Write D-> Write

Transaction for bank money transfer.

A-> B (x amount of money).

1. t=Read(A)
2. t-=x
3. Write(A,t)
4. t=Read(B)
5. t+=x
6. Write(B,t)

A transaction can fail at any time, at 2, 4 5.

Failures and Durability are handles via 2 operations: 1. Commit: Transaction has executed successfully and the results are going to persist. 2. RollBack: Undo all the operations that happened in the DB till that time.

After all the transaction operations are executed, once the DB is sure that everything is completed, it will execute 1 last final command: **COMMIT**. Once committed, we cannot reverse the changes. Whatever has happened in transaction till now, store it.

When we are in a non-desirable state(when we cannot proceed ahead with the transaction) we execute the **ROLLBACK** statement.

In MYSQL Workbench: Products table, In 1 transaction I have to increase the value of id=1 by 10 and decrease product with id=8 by 10

```
start transaction;
```

```
 update sql_inventory.products
set co=quantity_in_stock=80
where product_id=1;
```

```
 update sql_inventory.products
set co=quantity_in_stock=16
where product_id=8;
```

```
commit;
```

Now, 1^st^ we run start transaction, then update command, It changes the quantity in stock to 80 for id =1. Now, Something wrong has happened to the system and it shut down.

Again when we read the table, `select * from products;`

It shows 70 in id=1. Transaction has not committed till now.

Now, again start the transaction. update the first statement and second statement again if we close it will get reverted. Still has not committed.

Now, let's commit also. Even if the session closes, Commit means durability.

```
start transaction;
```

```
 update sql_inventory.products
set quantity_in_stock=80
where product_id=1;
```

`rollback;`

```
 update sql_inventory.products
set co=quantity_in_stock=16
where product_id=8;
```

`commit;`

Now, We start the transaction, update the first statement then rollback, it brings the DB to the last committed state.

How transactions handle concurrency:

Transaction Isolation levels:

There are 4 levels.

1. Read Uncommitted.
2. Read committed.
3. Repeatable read
4. Serializable.

These are based on increasing severity.

---

1. Read Uncommitted:

They are running parallely on the same data. T2 hasn't committed. In the DB, the desired number should be 2500. But it is 1500, so it is not the good situation to be in.

- The issue happened because we read the data that was not yet committed=> In future it can rollback.
- This is called **Dirty read**: when you read uncommitted data, and do operations on that.
- Some Dirty read are ok, Example: In facebook news read, Facebook is creating a news feed for the user but it has not committed the news feed yet. It will not affect user much if the post is committed or not. It depends upon use case.
- The isolation level of the DB might have been read uncommitted=> It allow other transaction to read uncommitted data also.
- Why someone add this as an isloation level because Read Uncommitted is most efficient as no locking etc.

---

2. Read Committed:

- The transaction will only read committed data.
- Read Committed isolation levels safegaurds us against Dirty Reads.

Now, Let's look into this scenerio. Now, the final value of x is 1000. That's kind of a weird situation, user wrote something(2500) but that is lost. - This is known as **LOST UPDATE**.

Within the transaction, user just read the value of x twice. This is also not the good situation to be in.

Example: Scaler wants to send everyone a coupon code. Because another student get added in between, scaler will have 1 less coupon code now.

So this is a problem, scaler have not given coupon code to the new student. ISSUE: **Non Repeating Reads.**

---

3. Repeatable Read:

- Within a transaction if a user try to read a value again, it will be same as earlier.
- This is default isolation level in MYSQL.

Bank Scenerio: If there are 2 people who want to work on this example,

Now, whosoever commit first or second, final value is going to be wrong. So in the case of bank, the only way to solve is to have no concurrency at all. Both of these transactions should happen in sequential manner.

---

4. Serializable:

- Uses locks behind the scene.
- If one transaction has taken a lock over a DB row, no other transaction will be allowed to even read that row.
- Each of the other transactions will have to wait till the transaction that has lock completes.
- As locks are involved, system will be slower.
- Most of the bank transactions take time, bookmyshow again is slower.

```
start transaction;
```

```
 update sql_inventory.products
set quantity_in_stock=100
where product_id=1;
```

```
 update sql_inventory.products
set quantity_in_stock=15
where product_id=8;
```

```
commit;
```

To see the current value of isolation level,

```
show variables like 'transaction_isolation';
```

Repeatable Read is the default isolation level of MYSQL.

Change the transaction isolation level:

```
set session transaction isolation level read uncommitted;
```

Create another session, Run this:

```
show variables like 'transaction_isolation';
```

Here it is repeatable read, because both are different sessions.

In session1:

```
start transaction;
```

```
 update sql_inventory.products
set quantity_in_stock=15
where product_id=8;
```

```
commit;
```

In session2:

```
start transaction;
```

```
 select *
from sql_inventory.products
where product_id=8;
```

`commit;`

Start both the transactions.

Here, a user have set the value 15 but in session2 still he can see 16. Because the session2 has isolation level as repeatable read. So what a user will see depend on your isolation level not other's isolation level.

Now, Let's put read uncommitted in session2 So now the user will read 15.

Again, Set read committed now on session2. The user will read 16 only, If user commit in session1, then he will read value 15 in session2.

Serializable:

In session2, make isolation level serializable. Then start the transaction in session1.

When the user start the transaction and try to read the value in session2, he will see a loader because someone else is working on product_id=8. There is a default time out of approx 30 sec after that that the user will see an error.

As soon as session1 commits, loader goes out.

How Serializable works:

1. If another transaction has updated the row, no other transaction is allowed to even read the row.

If the other person has only read the value, then it will work.

Now, Read doen't take a lock. So even though it's working it's not going to give the correct answer.

Here we read because we wanted to update this in future. FOR UPDATE.

```
 select *
from accounts
where name='B'
for update;
```

Whatever the user is reading, he will be going to use this value to update on this row later.

Lock is taken by: 1. Write 2. Read for update.

Transaction for Money transfer in Bank:

Send money(A,B amount) 1. Read(A) for update 2. Read(B) for update So first take the lock over all the entities that we are going to work with and after that perform the actions.

x=Read(A) for update y=Read(B) for update x-=amount y+=amount update(A,x) update(B,y)

It can lead to a problem called Deadlock:

DEADLOCK:

When the system can't proceed ahead because transactions have lock needed by each other.

Solution in MYSQL:

- Detects a deadlock
- ROLLBACK one of the transaction.

Now, In session2, `start transaction;`

```
 select *
from sql_inventory.products
where product_id=8
for update;
```

```
 select *
from sql_inventory.products
where product_id=7
for update;
```

`commit;`

In session1,

`start transaction;`

```
 select *
from sql_inventory.products
where product_id=7;
```

```
 select *
from sql_inventory.products
where product_id=8;
```

`commit;`

So start both the transactions, Read session2 product_id =8, then session1 product_id=7 Now, In session2 try to read product_id=7 it shows loader, because not a deadlock yet. As soon as we read in session2 product_id=8, It shows error, Deadlock found. So it remove session2 transaction and started with session1.
