Lab 4: Concurrency Control in SQL Server

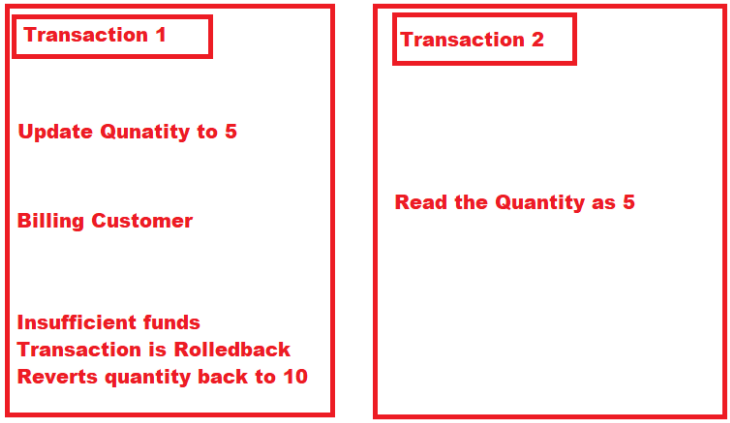
We are going to use the following Products table to understand this concept.

Id	Name	Quantity
1001	Mobile	10
1002	Tablet	20
1003	Laptop	30

```
-- Create Products table
CREATE TABLE Products
(
    Id INT PRIMARY KEY,
    Name VARCHAR(100),
    Quantity INT
)

-- Insert test data into Products table
INSERT INTO Products values (1001, 'Mobile', 10)
INSERT INTO Products values (1002, 'Tablet', 20)
INSERT INTO Products values (1003, 'Laptop', 30)
```

1. Dirty Read Concurrency Problem in SQL Server with Examples



In the above example, we have two transactions (Transaction 1 and Transaction 2) that are going to work with the same data. Currently, the available Quantity of the Product whose productId is 1001 is 10. Transaction 1, updates the value of Quantity to 5 for the Product whose productId is 1001. Then it starts to bill the customer or doing some other tasks - so we intentionally delay the execution to 15 seconds by using **Waitfor Delay** statement.

Copy the following query (query window 1) - Transaction 1

```
BEGIN TRANSACTION
  UPDATE Products SET Quantity = 5 WHERE Id=1001

  -- Billing the customer
  Waitfor Delay '00:00:15'
  -- Insufficient Funds. Rollback transaction

ROLLBACK TRANSACTION
```

Copy the following query (query window 2)- Transaction 2

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SELECT * FROM Products WHERE Id=1001
```

Execute both one by one. (First Transaction 1 then Transaction 2)

While Transaction 1 is in progress, Transaction 2 starts, and it reads the Quantity value of the same Product whose Id is 1001 which is 5 at the moment. At this point in time, Transaction 1 fails because of insufficient funds or for some other reason, and transaction 1 is rolled back. Quantity is reverted back to the original value of 10, but Transaction 2 is working with a different value i.e. 5 which does not exist in the database anymore and that data is said to be dirty data that does not exist anymore.

Note: `TRANSACTION ISOLATION LEVEL READ UNCOMMITTED` - try to read the uncommitted data. This is the only Transaction Isolation Level provided by SQL Server which has the **Dirty Read Concurrency Problem**.

How to Overcome the Dirty Read Concurrency Problem Example in SQL Server?

If you want to restrict the dirty read concurrency problem in SQL Server, then you have to use any Transaction Isolation Level except the Read Uncommitted Transaction Isolation Level. So, modify transaction 2 as shown below and hear we are using Read Committed Transaction Isolation Level.

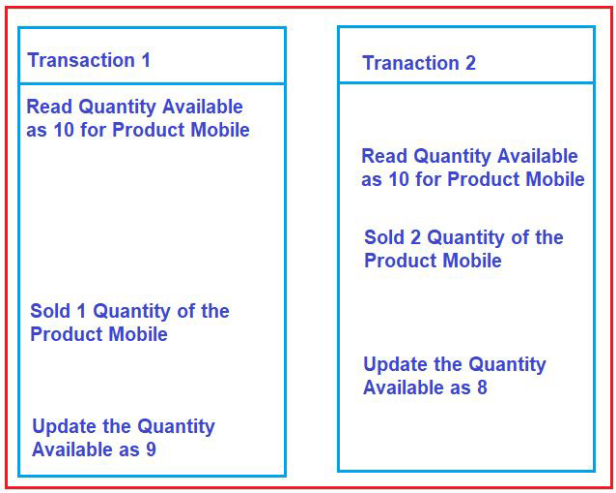
```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT * FROM Products WHERE Id=1001
```

Now run the first transaction and then immediately run the second transaction. You will see that until the first is not completed, you will not get the result in the second transaction. Once the first transaction execution is completed, then you will get the data in the second transaction and this time you will not get the uncommitted data rather you will get the committed data that exist in the database.

Another option provided by SQL Server to read the dirty data is by using the **NOLOCK** table hint option. The below query is equivalent to the query that we wrote in Transaction 2.

```
SELECT * FROM Products (NOLOCK) WHERE Id=1001
```

2. Lost update concurrency problem



Open 2 instances (query windows) of SQL Server Management Studio. From the first instance execute the Transaction 1 code and from the second instance, execute the Transaction 2 code.

Copy the following query (query window 1)

```
-- Transaction 1
BEGIN TRANSACTION
  DECLARE @QunatityAvailable int
  SELECT @QunatityAvailable = Quantity FROM Products WHERE Id=1001

  -- Transaction takes 10 seconds
  WAITFOR DELAY '00:00:10'

  SET @QunatityAvailable = @QunatityAvailable - 1
  UPDATE Products SET Quantity = @QunatityAvailable  WHERE Id=1001
  Print @QunatityAvailable
COMMIT TRANSACTION
```

Copy the following query (query window 2)

```
-- Transaction 2
BEGIN TRANSACTION
  DECLARE @QunatityAvailable int
  SELECT @QunatityAvailable = Quantity FROM Products WHERE Id=1001

  SET @QunatityAvailable = @QunatityAvailable - 2
  UPDATE Products SET Quantity = @QunatityAvailable WHERE Id=1001
  Print @QunatityAvailable
COMMIT TRANSACTION
```

First, run the transaction 1 code and then immediately run the transaction 2 code. You will see that Transaction2 is completed first and then the second transaction is completed. Now, if you verify the Products table, then you will see that the quantity for the product Mobile is 9 by executing the below

query.

```
SELECT * FROM Products WHERE Id = 1001
```

How to overcome the Lost Update Concurrency Problem?

We can overcome using the **Repeatable Read Transaction Isolation Level** in SQL Server. The Repeatable Read Transaction Isolation Level uses **additional locking on rows** that are read by the current transaction which prevents those rows to be updated or deleted by other transactions. This solves the Lost Update Concurrency Problem. Update the Quantity as 10 for the Product Mobile by executing the below update query.

```
Update Products SET Quantity = 10 WHERE Id = 1001
```

Modify both the transactions as shown below to use the **Repeatable Read Isolation Level** to solve to Lost Update Concurrency Problem.

Transaction 1

```
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
  DECLARE @QunatityAvailable int
  SELECT @QunatityAvailable = Quantity FROM Products WHERE Id=1001

  -- Transaction takes 10 seconds
  WAITFOR DELAY '00:00:10'

  SET @QunatityAvailable = @QunatityAvailable - 1
  UPDATE Products SET Quantity = @QunatityAvailable  WHERE Id=1001
  Print @QunatityAvailable
COMMIT TRANSACTION
```

Transaction 2

```
-- Transaction 2
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
  DECLARE @QunatityAvailable int
  SELECT @QunatityAvailable = Quantity FROM Products WHERE Id=1001

  SET @QunatityAvailable = @QunatityAvailable - 2
  UPDATE Products SET Quantity = @QunatityAvailable WHERE Id=1001
  Print @QunatityAvailable
COMMIT TRANSACTION
```

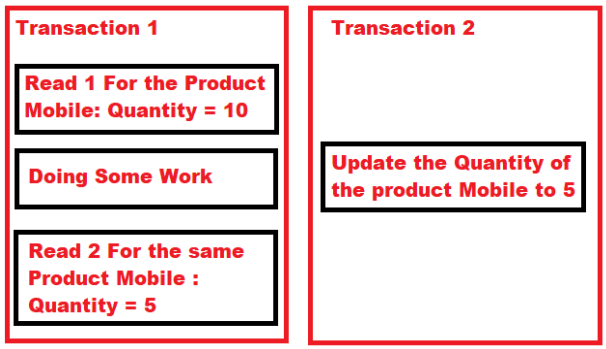
Now run Transaction1 first and then run the second transaction and you will see that Transaction 1 was completed successfully while Transaction 2 competed with **the error. The transaction was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.**

Once you rerun Transaction 2, the Quantity will be updated correctly as expected in the database table.

Reset the table for the next example- update the Quantity as 10 for the Product Mobile by executing the below update query.

```
Update Products SET Quantity = 10 WHERE Id = 1001
```

3. Non-Repeatable Read Concurrency Problem



Non-Repeatable Read Concurrency Problem Example in SQL Server:

Open 2 instances of SQL Server Management Studio. From the first instance execute the Transaction 1 code and from the second instance, execute the Transaction 2 code.

Transaction 1:

```
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRANSACTION
SELECT Quantity FROM Products WHERE Id = 1001
-- Do Some work
WAITFOR DELAY '00:00:15'
SELECT Quantity FROM Products WHERE Id = 1001
COMMIT TRANSACTION
```

Transaction 2:

```
 -- Transaction 2
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
UPDATE Products SET Quantity = 5 WHERE Id = 1001
```

Notice that when Transaction 1 is completed, it gets a different value for reading 1 and reading 2, resulting in a non-repeatable read concurrency problem. As we already discussed READ COMMITTED and READ UNCOMMITTED Transaction Isolation Level produces the Non-Repeatable Read Concurrency Problem. So, here, we need to set the Transaction Isolation Level either READ

COMMITTED and READ UNCOMMITTED. Both transactions use READ COMMITTED to introduce the Non-Repeatable Read Concurrency Problem.

How to Solve the Non-Repeatable Read Concurrency Problem in SQL Server?

In order to solve the **Non-Repeatable, Read Problem in SQL Server**, we need to use either **Repeatable Read Transaction Isolation Level** or any other higher isolation level such as **Snapshot** or **Serializable**. So, let us set the transaction isolation level of both Transactions to repeatable read (you can also use any higher transaction isolation level). This will ensure that the data that Transaction 1 has read will be prevented from being updated or deleted elsewhere. This solves the non-repeatable read concurrency issue. Let us rewrite both the transactions using the Repeatable Read Transaction Isolation Level.

Modify the Transaction 1 code as follows:

```
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
SELECT Quantity FROM Products WHERE Id = 1001
-- Do Some work
WAITFOR DELAY '00:00:15'
SELECT Quantity FROM Products WHERE Id = 1001
COMMIT TRANSACTION
```

Modify the Transaction 2 code as follows:

```
-- Transaction 2
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
UPDATE Products SET Quantity = 5 WHERE Id = 1001
```

With the above changes in place, now run transaction 1 first and then the second transaction and you will see that it gives the same result for both the read in transaction 1. When you execute Transaction 1 and 2 from 2 different instances of SQL Server management studio, Transaction 2 is blocked until Transaction 1 completes, and at the end of Transaction 1, both the reads get the same value for the Quantity of the same product mobile.

4. Phantom Read Concurrency Problem in SQL Server with Examples

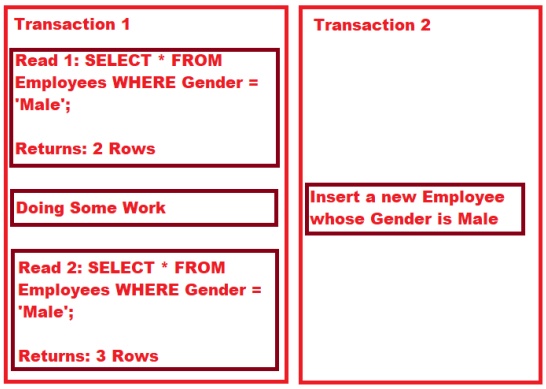
We are going to use the following Employees table to understand this concept.

Id	Name	Gender
1001	Anurag	Male
1002	Priyanka	Female
1003	Pranaya	Male
1004	Hina	Female

Please use the below SQL Script to create and populate the Employees table with the required sample data.

```
-- Create Employee table
 CREATE TABLE Employees
 (
     Id INT PRIMARY KEY,          Name VARCHAR(100),
     Gender VARCHAR(10)
 )
 Go
 -- Insert some dummy data
 INSERT INTO  Employees VALUES(1001,'Anurag', 'Male')
 INSERT INTO  Employees VALUES(1002,'Priyanka', 'Female')
 INSERT INTO  Employees VALUES(1003,'Pranaya', 'Male')
 INSERT INTO  Employees VALUES(1004,'Hina', 'Female')
```

The Phantom Read Concurrency Problem happens in SQL Server when one transaction executes a query twice and it gets a different number of rows in the result set each time. This generally happens when a second transaction inserts some new rows in between the first and second query execution of the first transaction that matches the WHERE clause of the query executed by the first transaction



Let say we have two transactions Transaction 1 and Transaction 2. Open 2 instances of the SQL Server Management Studio. From the first instance execute the Transaction 1 code and from the second instance execute the Transaction 2 code.

Transaction 1

```
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
SELECT * FROM Employees where Gender = 'Male'
-- Do Some work
WAITFOR DELAY '00:00:10'
SELECT * FROM Employees where Gender = 'Male'
COMMIT TRANSACTION
```

Transaction 2

```
-- Transaction 2
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
INSERT into Employees VALUES(1005, 'Sambit', 'Male')
COMMIT TRANSACTION
```

Notice that when Transaction 1 is completed, it gets a different number of rows for reading 1 and reading 2, resulting in a phantom read problem. The Read Committed, Read Uncommitted, and Repeatable Read Transaction Isolation Level causes Phantom Read Concurrency Problem in SQL Server. In both Transactions, we use REPEATABLE READ Transaction Isolation Level, even you can also use Read Committed and Read Uncommitted Transaction Isolation Levels.

How to solve the Phantom Read Concurrency Problem in SQL Server?
You can use the **Serializable or Snapshot Transaction Isolation Level** to solve the Phantom Read Concurrency Problem in SQL Server.

In our example, to fix the Phantom Read Concurrency Problem let set the transaction isolation level of Transaction 1 to serializable. The Serializable Transaction Isolation Level places a range lock on the rows returns by the transaction based on the condition. In our example, it will place a lock **where Gender is Male**, which prevents any other transaction from inserting new rows within that Gender. This solves the phantom read problem in SQL Server.
When you execute Transaction 1 and 2 from 2 different instances of SQL Server Management Studio. Transaction 2 is blocked until Transaction 1 completes and at the end of Transaction 1, both the reads get the same number of rows. Modify the Transaction 1 code as follows:

```
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION
SELECT * FROM Employees where Gender = 'Male'
-- Do Some work
WAITFOR DELAY '00:00:10'
SELECT * FROM Employees where Gender = 'Male'
COMMIT TRANSACTION
```

Your Turn!

Implement a database application that simulates the concurrent access to a shared database table by multiple clients. The database table should store information about a bank account, including account number, account holder name, and balance. The application should allow clients to perform the following operations:

1. Deposit money into the account
2. Withdraw money from the account
3. Check the balance of the account

The application should handle concurrency problems by using lock-based synchronization or other concurrency control techniques. The program should also report any errors or conflicts that occur during the transactions.

Deliverables:

Query of the implementation
A brief report explaining the design and implementation of the application, including the concurrency control techniques used and the results of testing the program.
Screen shots or outputs of the program demonstrating its functionality.