Transactions

Contents

[1. Agenda 2](#_Toc187097105)

[2. What are Transactions? 3](#_Toc187097106)

[3. ACID Properties of Transactions 4](#_Toc187097107)

[4. Atomicity in Transactions 5](#_Toc187097108)

[5. Consistency in Transactions 5](#_Toc187097109)

[6. Isolation in Transactions 8](#_Toc187097110)

[7. Durability in Transactions 10](#_Toc187097111)

[8. Transactions 13](#_Toc187097112)

[SQL Transactions - Practice 14](#_Toc187097113)

[9. Transaction Isolation Levels 21](#_Toc187097114)

[a. Read Uncommitted - Isolation Level 21](#_Toc187097115)

[Read Uncommitted – Practice 22](#_Toc187097116)

[b. Read Committed Isolation Level 25](#_Toc187097117)

[Read Committed – Practice 26](#_Toc187097118)

[c. Repeatable Read Isolation Level 30](#_Toc187097119)

[Repeatable Read – Practice 33](#_Toc187097120)

[Transaction Isolation Levels Explained 35](#_Toc187097121)

[d. Serializable Isolation Level 37](#_Toc187097122)

[Serializable – Practice 41](#_Toc187097123)

[Transaction Isolation Levels - Serializable 44](#_Toc187097124)

[Deadlocks 45](#_Toc187097125)

[Deadlock Practical 46](#_Toc187097126)

# Agenda

### Session 1: Introduction to Transactions and ACID Properties

1. **Introduction to Transactions**
   * Definition and significance of transactions.
   * Real-world examples of transactions.
2. **ACID Properties**
   * Atomicity
   * Consistency
   * Isolation
   * Durability

### Session 2: Isolation Levels and Practical Demonstrations

1. **Isolation Levels**
   * Understanding transaction isolation.
   * Overview of isolation levels: Read Uncommitted, Read Committed, Repeatable Read, and Serializable.
2. **Practical Demonstrations**
   * Simulating transactions in action.
   * Identifying and resolving deadlocks.

# **What are Transactions?**

* A **transaction** is a **set of logically related SQL queries**.
* These queries are grouped together to implement a specific operation.
* Example: A **money transfer** in a banking application involves:
  + Deducting money from one account (e.g., Account A).
  + Adding money to another account (e.g., Account B).

### **Example of a Transaction: Money Transfer**

1. **Read the balance** of Account A.
2. Check if Account A has sufficient funds.
3. **Deduct the amount** from Account A.
4. **Update the database** with the new balance of Account A.
5. **Read the balance** of Account B.
6. **Add the amount** to Account B.
7. **Update the database** with the new balance of Account B.

* If executed independently without proper transaction handling, failures (e.g., database crashes) may lead to:
  + Deduction from Account A without addition to Account B.
  + Inconsistent and unreliable outcomes.

### The Problem with Independent Queries

* Executing these queries individually can lead to inconsistency. For instance, if the process is interrupted after deducting money from the sender but before adding it to the recipient, the database will be left in an invalid state.

# ACID Properties of Transactions

* To ensure reliability and consistency, transactions must exhibit the following **ACID** properties:

1. **Atomicity**:
   * The transaction must be treated as a single, indivisible unit.
   * Either all operations within the transaction are completed successfully, or none of them are executed.
   * For example, if the transfer fails midway, any changes made (e.g., money deducted from the sender's account) must be rolled back.
2. **Consistency**:
   * The database must transition from one valid state to another.
   * The integrity of the database (e.g., constraints, rules) must not be violated.
   * In the money transfer case, the sum of both accounts’ balances before and after the transaction must remain the same.
3. **Isolation**:
   * Transactions must not interfere with each other.
   * Intermediate states of a transaction should not be visible to other transactions.
   * For example, while money is being transferred, no other transaction should access or modify the accounts involved.
4. **Durability**:
   * Once a transaction is committed, its effects must persist, even in the event of a system failure.
   * For example, if the money transfer completes, the changes must be permanently saved in the database.

# Atomicity **in Transactions**

* Atomicity ensures that for the outside world, a transaction appears as a single operation. Internally, a transaction might involve multiple steps, but it either succeeds as a whole or fails without leaving any trace of partial execution.
* This means:
  + If the transaction is interrupted (e.g., power failure, system crash), any completed operations must be undone, leaving the database unchanged.
* For example:
  + If money is deducted from Account A but not credited to Account B due to a failure, the system must roll back the deduction.

### Ensuring Atomicity

* Databases use mechanisms like **transaction logs** to track changes. If a failure occurs, the log helps undo incomplete changes or redo committed ones.
* This makes transactions a crucial concept for maintaining data integrity in real-world applications.

# **Consistency in Transactions**

* Consistency is one of the fundamental **ACID** properties of database transactions. It ensures that a database remains in a **valid state** before and after a transaction, even in the presence of failures.

### **What is Consistency?**

* **Consistency guarantees** that any transaction will take the database from one valid state to another valid state.
* A **valid state** means the database adheres to all the defined **rules, constraints, and relationships** (e.g., primary keys, foreign keys, and other integrity constraints).

### **Example to Understand Consistency**

* Let’s revisit the **bank transfer example**:

1. **Initial State (Valid)**:
   * Account A: $2000
   * Account B: $700
2. **Transaction:**
   * Deduct $100 from Account A.
   * Add $100 to Account B.
3. **Outcome (Valid State):**
   * Account A: $1900
   * Account B: $800

Here:

* The sum of balances ($2000 + $700 = $2700) before the transaction equals the sum of balances after the transaction ($1900 + $800 = $2700).
* This ensures **consistency**.

### **How Consistency Works**

#### **Constraints Validation**

* All **integrity constraints** (e.g., foreign keys, unique constraints) must be satisfied.
* Example: If Account B doesn’t exist, the database shouldn’t allow transferring money to it.

#### **Business Logic Rules**

* Consistency also enforces **domain-specific rules**.
* Example: The system should ensure that the balance of any account cannot go below zero. If Account A has less than $100, the transaction should fail.

### **Why Consistency Matters**

Without consistency:

* The database could end up in an **invalid state**, leading to:
  + Incorrect results.
  + Violated constraints.
  + Logical errors in the application.

### **Ensuring Consistency in Practice**

#### **Atomic Operations:**

* Break transactions into small, valid steps, and ensure they complete successfully or rollback entirely.

#### **Database Constraints:**

* Use constraints like **foreign keys, check constraints, and unique indexes** to enforce rules.

#### **Transaction Management:**

* Use **transaction blocks** (BEGIN TRANSACTION, COMMIT, ROLLBACK) to ensure intermediate states are not saved if the transaction fails.

### **Key Difference Between Atomicity and Consistency**

* **Atomicity** ensures that a transaction either completes entirely or doesn’t occur at all.
* **Consistency** ensures that, whether a transaction succeeds or fails, the database **always remains valid**.

For example:

* If the transfer from A to B fails midway, atomicity will roll back the transaction, ensuring consistency by leaving the database in its original valid state.

# **Isolation in Transactions**

### **Definition**

* Isolation ensures that a transaction is **executed independently** of other concurrent transactions. It guarantees that the results of a transaction are **not affected by other transactions** running at the same time.

### In simpler terms:

* A transaction should feel as though it is the **only one** operating on the database at that moment.
* Even though multiple transactions might execute concurrently, **isolation ensures correctness** by preventing conflicts or inconsistencies.

### **Why Is Isolation Important?**

* Without isolation, concurrent transactions can interfere with each other, leading to:
  + **Incorrect results** (e.g., double deductions, missed updates).
  + **Data corruption**.
  + **Unpredictable behavior**.

### **Real-World Example**

* Imagine two transactions involving the same bank account, happening at the same time:

1. **Transaction A**: B receives ₹1 crore from KBC.
2. **Transaction B**: B sends ₹50 to a shopkeeper.

### Initial State:

* Balance of **B**: ₹100.

### Without Isolation:

1. **Transaction A** reads B's balance as ₹100, adds ₹1 crore, and writes ₹1,00,00,100 to the database.
2. **Transaction B** reads B's balance as ₹100 (before A’s update), subtracts ₹50, and writes ₹50 to the database.
3. **Final Balance of B**: ₹50 (incorrect).
   * B should have ₹1,00,00,050, but due to lack of isolation, B lost the ₹1 crore deposit.

### With Isolation:

1. Transaction A and Transaction B are **executed in isolation**.
   * A completes first: Updates B's balance to ₹1,00,00,100.
   * B then executes on the updated balance: Deducts ₹50, resulting in ₹1,00,00,050.
2. **Final Balance of B**: ₹1,00,00,050 (correct).

### **In** Summary

* Isolation ensures that **each transaction operates independently** without being negatively affected by other concurrent transactions. It protects data integrity and correctness in multi-transaction systems, especially in scenarios involving simultaneous updates, reads, or inserts.

# **Durability in Transactions**

### **Definition**

* Durability ensures that once a transaction is successfully completed, its effects are **permanently saved** in the database. The changes made by the transaction will persist even in the event of:
  + A system crashes.
  + A power failure.
  + Server restarts.
* In simple terms:
  + **"Once done, always done."**

### **Why is Durability Important?**

* Without durability:
  + Transaction results could be lost due to unexpected failures.
  + Critical operations like financial transactions might "disappear," causing discrepancies and user dissatisfaction.

### **Real-World Example**

Imagine you transfer money from **Account A** to **Account B**:

* You transfer ₹10,000 from A to B.
* The system deducts ₹10,000 from A and adds ₹10,000 to B.

If durability is not ensured:

* After completing the transfer, a **system crash** occurs before saving the transaction results to disk.
* When the system restarts, the changes are **lost**.
* Account A still shows ₹10,000, but Account B never received the money.

With durability:

* The transaction results are saved to persistent storage (e.g., disk) **before the transaction is marked as complete**.
* After the system restarts, both accounts reflect the correct balances.

### **How Durability is Achieved**

* Durability is typically implemented using **persistent storage** and **logging mechanisms**:

1. **Write-Ahead Logging (WAL)**:

* Changes made during a transaction are written to a **log file** before being applied to the database.
* If a crash occurs, the database can use the log to recover and reapply the changes.

1. **Checkpointing**:

* Periodically, the database saves a snapshot of its current state to disk.
* This ensures that recent transactions are safely stored.

1. **Transaction Commit**:

* A transaction is considered committed only after its changes are successfully written to disk.
* Until then, it is not "durable."

1. **RAID and Replication**:

* Systems use RAID (Redundant Array of Independent Disks) or replication across multiple servers to protect data against hardware failures.

### **Durability and System Crashes**

If a system crash occurs:

* **Uncommitted Transactions**: Changes are discarded.
* **Committed Transactions**: The database uses the log to reapply these transactions, ensuring their results persist.

### **Real-Life Scenario**

When you make a **bank transaction**:

* After transferring money, you receive a confirmation message.
* This message indicates that the transaction is **committed** and saved in the bank’s system.
* Even if the bank’s server restarts or crashes later, your transaction is **durable** and will not be lost.

### **Key Takeaways**

1. **Durability ensures persistence** of committed transaction results.
2. It is achieved through mechanisms like **logging, checkpoints**, and **replication**.
3. It guarantees data reliability even in the case of hardware failures or system crashes.
4. **Durable transactions enhance user trust** in systems like banking, e-commerce, and databases.

**Durability ensures: "Once it's written, it's there forever."**

# **Transactions**

1. **Transaction:**

* A transaction consists of multiple SQL statements executed as a single unit.
* Defined using the SQL statement: START TRANSACTION.

1. **Commit:**

* Marks the successful completion of a transaction.
* Ensures that all changes made during the transaction are persisted in the database.
* Implements the **Durability** property of ACID:
  + Data remains stored even in the event of a system failure after the transaction commits.

1. **Rollback**:

* Reverts all changes made during a transaction if an error occurs.
* Restores the database to the state it was in before the transaction started.
* Implements the **Consistency** property of ACID:
  + Ensures that the database remains valid and consistent after a rollback.

### **ACID Properties in Transactions**

1. **Atomicity**:

* A transaction is an all-or-nothing operation.
* Either all SQL statements within the transaction execute successfully, or none of them have any effect.
* Supported by both COMMIT (everything executed) and ROLLBACK (everything reverted).

1. **Consistency**:

* Ensures the database remains in a valid state before and after a transaction.
* Maintained through rollback if an error occurs.

1. **Isolation**:

* Controls the visibility of transaction changes to other concurrent transactions.
* Maintained using isolation levels (e.g., read uncommitted, read committed).

1. **Durability**:

* Ensures that once a transaction commits, its changes are permanently saved in the database.

### **Behavior of Commit and Rollback**

* **Commit**:
  + Saves all changes made during the transaction to the database.
  + Changes become persistent and are protected from data loss due to system failures.
* **Rollback**:
  + Undoes all changes made during the transaction.
  + Restores the database to its state before the transaction began.

## **SQL Transactions - Practice**

### **Understanding Transactions:**

* A **transaction** in SQL is a sequence of operations performed as a single logical unit of work.
* A transaction begins with a START TRANSACTION and ends with either a COMMIT or ROLLBACK.

### **Start of Transaction:**

* Any SQL operation executed after a transaction starts and before it ends (COMMIT or ROLLBACK) is part of the transaction.

### **Example Queries in a Transaction:**

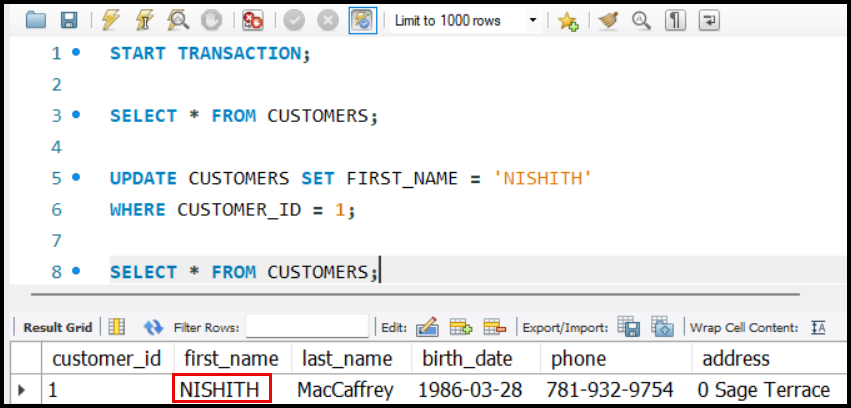
* SELECT \* FROM CUSTOMERS; – Reads data.
* UPDATE CUSTOMERS SET FIRSTNAME = 'NAMAN' WHERE CUSTOMERID = 1; – Modifies data.

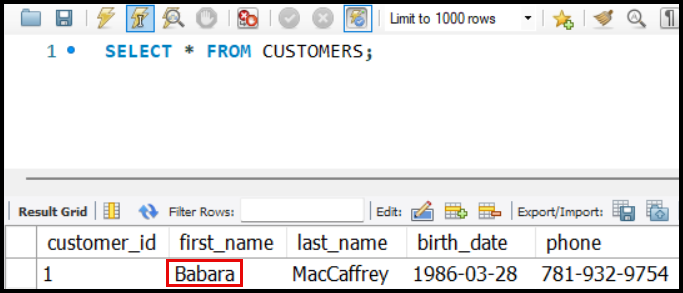
### **Transaction Termination:**

* A transaction is completed by either:
  + **COMMIT**: Saves all changes made during the transaction permanently.
  + **ROLLBACK**: Undoes all changes made during the transaction.

### **Scenario 1: Session Closure Without Commit or Rollback**

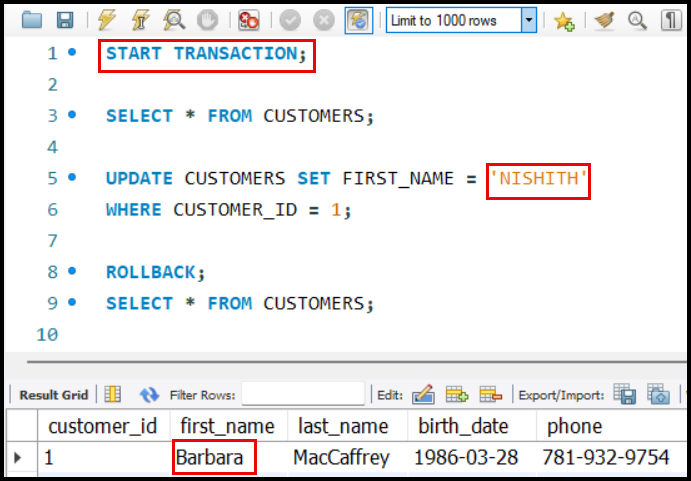
* If a session is closed unexpectedly (due to machine shutdown, network error, etc.):
* **Transaction is automatically rolled back**.
* All changes made during the transaction are lost.
* Example:
  + After updating CUSTOMER\_ID = 1 with FIRST\_NAME = 'NISHITH' but closing the session without a COMMIT, reopening the database and checking shows FIRST\_NAME remains as 'BARBARA' (original value).





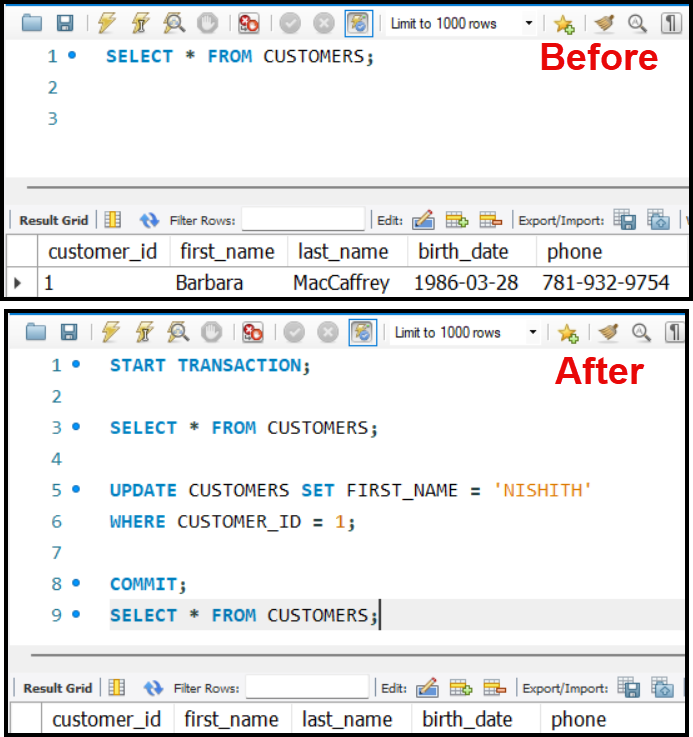
### **Scenario 2: Explicit Rollback**

* If ROLLBACK is explicitly called:
  + All changes made during the transaction are undone.
  + Example:
    - Update CUSTOMERID = 1 to FIRST\_NAME = 'NISHITH'.
    - Call ROLLBACK.
    - After rollback, checking the value shows 'BARBARA' (original value).



### **Scenario 3: Commit**

* If COMMIT is called:
  + Changes are saved permanently, even if the session is closed or the machine shuts down.
  + Example:
    - Update CUSTOMERID = 1 to FIRST\_NAME = 'NISHITH'.
    - Call COMMIT.
    - After reopening the database, checking shows FirstName remains as 'NISHITH' (updated value).



### **Key Takeaways:**

* **Automatic Rollback**: If a session closes unexpectedly, the transaction is rolled back by default.
* **Explicit Rollback**: Use ROLLBACK to undo changes.
* **Commit**: Use COMMIT to make changes permanent.
* Transactions provide a mechanism to ensure **atomicity** (all-or-nothing execution).

### **Failures and Transaction Rollback:**

1. **Session Closure Due to Failure:**

* If a failure occurs (e.g., network issue, server crash, or power outage):
  + Both client and server recognize the session has stopped.
  + Any ongoing transaction **automatically rolls back** to maintain data integrity.

1. **How to Identify Failure:**

* The session closing is an indicator of a failure.
* Any uncommitted transaction at the time of failure will be reverted.

### **Behavior of** COMMIT **and** ROLLBACK**:**

1. COMMIT **Without an Active Transaction:**

* If no transaction has been started, using COMMIT has no effect.
* COMMIT only works in the context of a transaction, saving changes made during that transaction.

1. **Automatic** COMMIT **Without Transaction:**

* If a query (e.g., SELECT or UPDATE) is executed outside of a transaction:
  + The database **automatically commits** the changes immediately after execution.
  + Example: (without a transaction).

UPDATE CUSTOMERS SET

    FIRST\_NAME = 'NISHITH'

WHERE

    CUSTOMER\_ID = 1;

* + The change is stored permanently as soon as the query is executed.

### **Scenarios:**

* **No Transaction, Query Execution:**
  + If no transaction is started and queries are executed:
    - The database handles each query independently.
    - Each query **auto-commits** immediately after execution.
    - Failure after the query execution has no effect since the changes are already stored.
* **Transaction Started, Failure Occurs:**
  + If a transaction is started, and failure occurs before a COMMIT:
  + The transaction **automatically rolls back**, reverting all changes.
* **Transaction Not Started,** COMMIT **Called:**
  + If no transaction is started but COMMIT is called:
  + No operation occurs as there are no changes to commit.

### **Key Takeaways:**

1. **Transaction Dependency:**

* COMMIT and ROLLBACK are meaningful only when a transaction is active.
* Without a transaction, queries auto-commit upon execution.

1. **Automatic Commit Behavior:**

* Any SQL operation executed outside a transaction is automatically persisted.

1. **Failure Handling:**

* When a session ends unexpectedly (failure), uncommitted transactions are automatically rolled back.
* Queries outside a transaction remain unaffected, as they commit immediately upon execution.

1. **Practical Usage:**

* Use transactions for operations requiring **atomicity** (all-or-nothing).
* Be aware of auto-commit behavior for standalone queries outside transactions.

### **Atomicity in Action**

* **Atomicity Implementation**: Achieved through **commit** and **rollback** mechanisms.
  + **Commit**: Ensures all changes made by a transaction are saved permanently.
  + **Rollback**: Ensures no partial changes are left in case of failure.
* **Atomicity in Action**:
  + If a transaction commits, all changes are saved.
  + If a transaction rolls back, no changes are applied.

### **Consistency**

* **Consistency Mechanisms**:
  + Transactions help ensure consistency by maintaining database rules.
  + Commit and rollback contribute to consistency.
  + **Isolation** also plays a crucial role in maintaining consistency.

### **Durability**

* **Durability in Action**:
  + Changes made by a committed transaction are permanent and durable.

### **Challenges with Concurrent Transactions**

* Issues arise when **multiple transactions access the same data concurrently**.
* Problems like inconsistency can occur due to uncoordinated actions by simultaneous transactions.

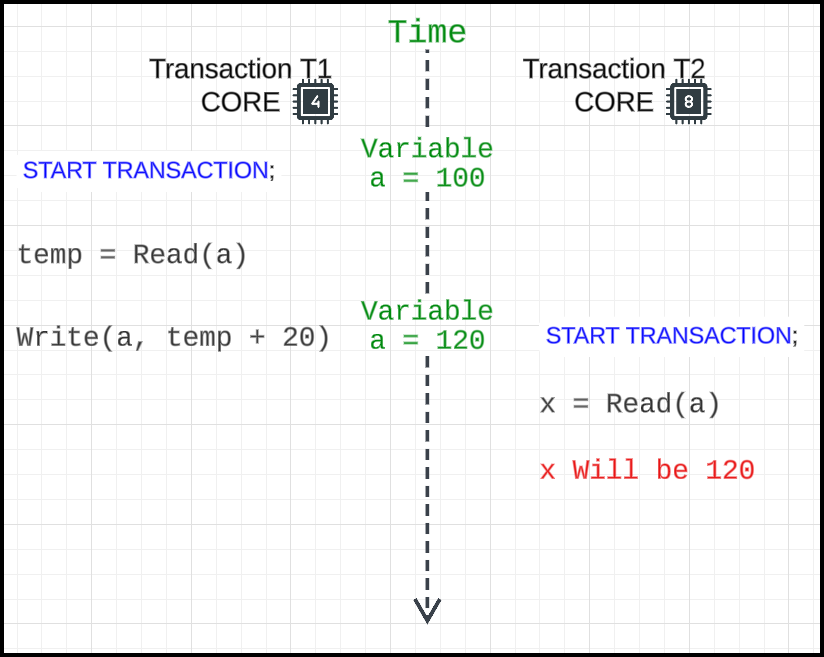
Isolation levels determine how transactions interact when multiple transactions occur concurrently.

# **Transaction Isolation Levels**

* Isolation levels determine how transactions interact when multiple transactions occur concurrently.

## **Read Uncommitted - Isolation Level**

* **Definition**: The most lenient isolation level, with no safeguarding between transactions.
* **Behavior**:
  + Allows reading **uncommitted data** from other transactions.
  + Example:
    - T1 modifies ‘a’ (from 100 to 120) but hasn’t committed.
    - T2 can read the modified value (120) despite no commit.



* **Advantages**:
  + Fast due to no locking or overhead.
* **Use Cases**:
* Suitable for systems where **exact accuracy** is not critical (e.g., social media
* **Problems**:
  + Can lead to **consistency issues** in critical systems (e.g., banking).

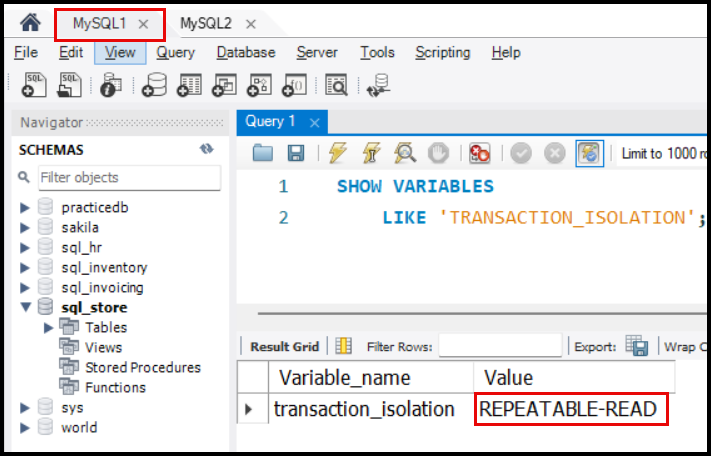
## Read Uncommitted – Practice

**Database Session 1 – Name MySQL1**

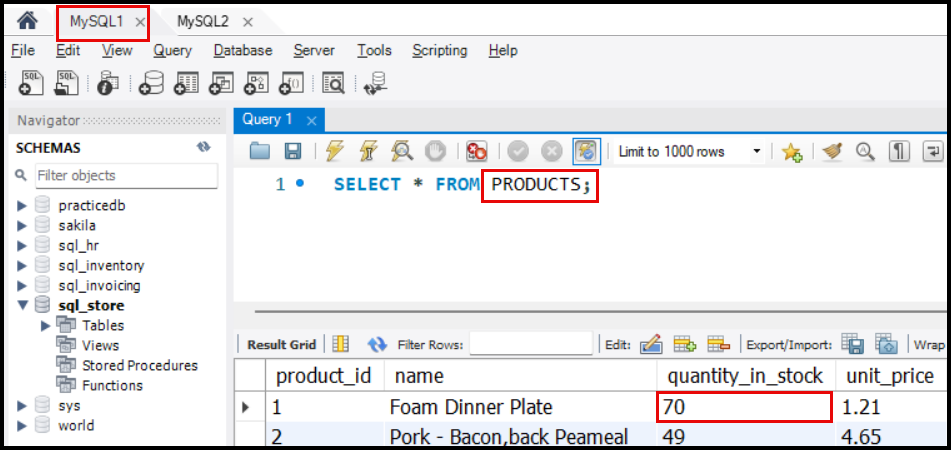
* See the current Transaction Isolation Level that is set…

SHOW VARIABLES

    LIKE 'TRANSACTION\_ISOLATION';



* Let’s use PRODUCTS table from SQL\_STORE database.



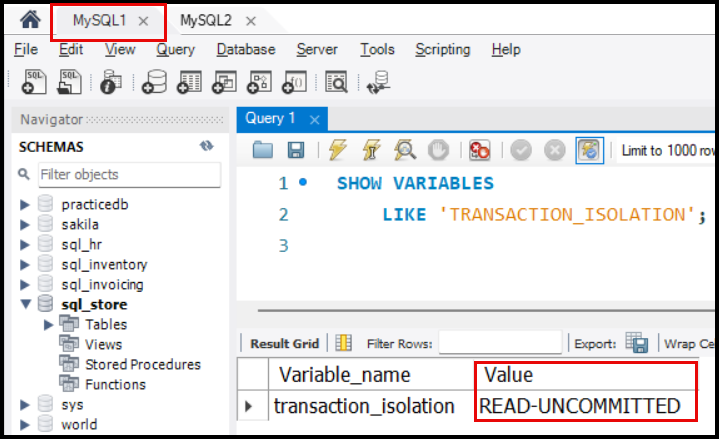
* Set the Transaction Isolation Level to ‘READ UNCOMMITTED’.

SET SESSION

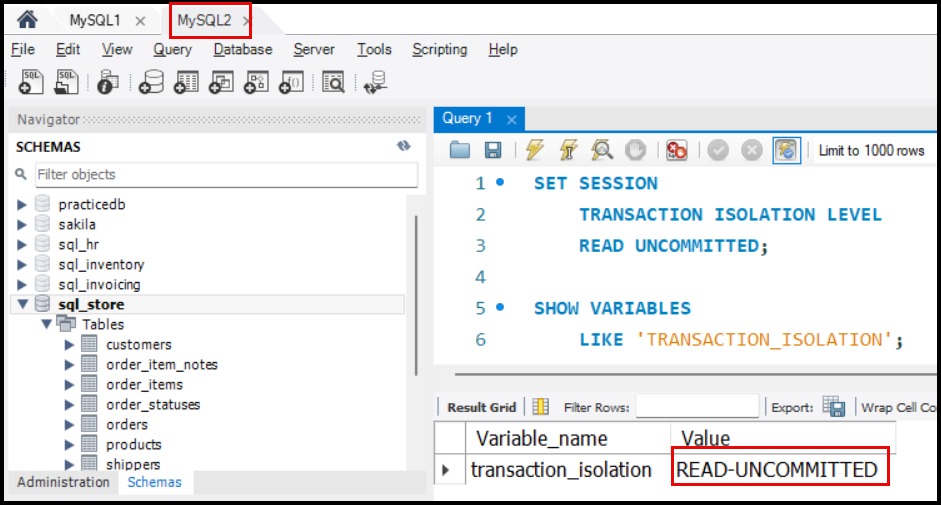
    TRANSACTION ISOLATION LEVEL

    READ UNCOMMITTED;

* Check whether ‘Transaction Isolation Level’ is set to ‘READ UNCOMMITTED’.



* Do the same thing in another database session MySQL2.



* Let’s start a transaction and update the ‘QUANTITY\_IN\_STOCK’ column value to 60 for the ‘PRODUCT\_ID = 1’ in PRODUCTS table (Session 1: MySQL1).

UPDATE

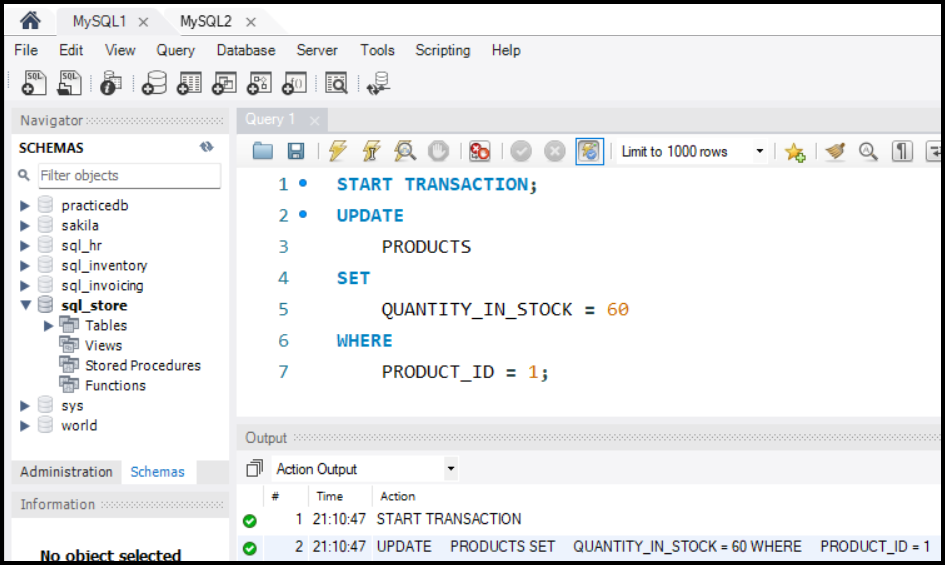
    PRODUCTS

SET

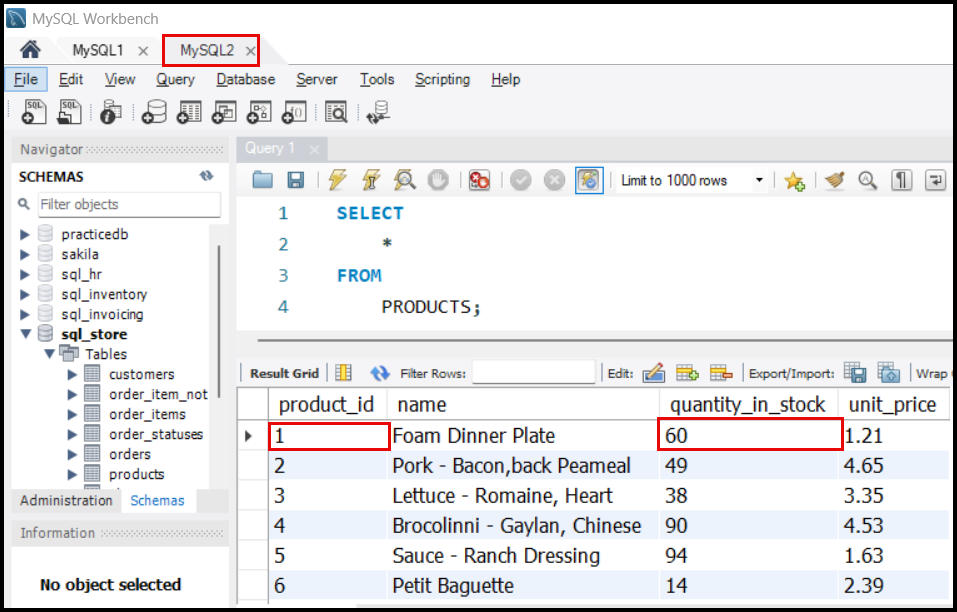
    QUANTITY\_IN\_STOCK = 60

WHERE

    PRODUCT\_ID = 1;



* Now, let’s check the PRODUCTS table in database session MySQL2.



* In database session MySQL2 shows that for the ‘PRODUCT\_ID = 1’, the ‘QUANTITY\_IN\_STOCK’ value is 60, which was set in db session MySQL1 and was not committed.
* Hence ‘READ UNCOMMITTED’ allows reading **uncommitted data** from other transactions.

## Read Committed Isolation **Level**

* **Read Committed Isolation Level**: Ensures that a transaction sees only committed data of other transactions or the latest uncommitted data of its own transaction, whichever is the most recent.

### **Key Behaviors**

1. **Committed Data Visibility**:

* A transaction operating under this isolation level can only read data that has been committed by other transactions.
* Uncommitted data from other transactions is not visible.

1. **Uncommitted Data from Own Transaction**:

* If a transaction updates a value, it sees its own uncommitted updates, even if they are not yet committed.

1. **Priority of Data**:

* For a transaction:
  + It sees its own latest uncommitted updates if present.
  + Otherwise, it reads the latest committed data from other transactions.

### **Examples**

1. **Transaction with Read Committed Isolation Level**:

* Transaction begins with the isolation level set to READ COMMITTED.
* Initial value of a product is 60.
* Transaction updates the value to 50 but **does not** COMMIT.
* It will see the updated value 50 because it is the latest uncommitted data of the same transaction.

1. **Effect of Committing Changes:**

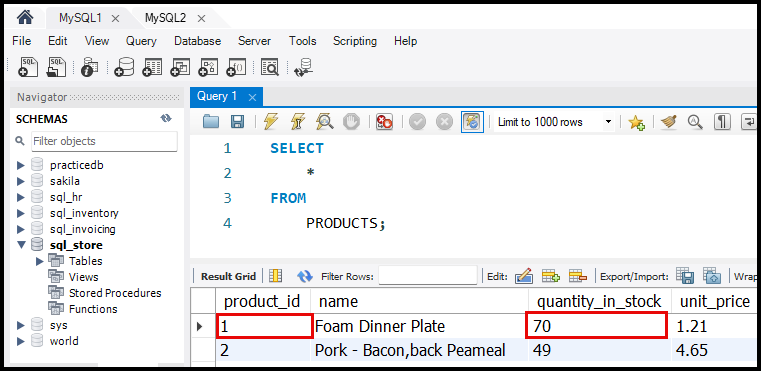
* If another transaction changes the value (e.g., to 71) but does not commit, the READ COMMITTED transaction will still see its own uncommitted value 50.
* If the other transaction commits, the READ COMMITTED transaction will then see the committed value 71 after its own changes are committed or rolled back.

### **Summary of Behaviours**

| **Isolation Level** | **What It Reads** |
| --- | --- |
| **Read Uncommitted** | Latest value (committed or uncommitted). |
| **Read Committed** | - Latest committed value of other transactions. |
|  | - Latest uncommitted value of the same transaction. |

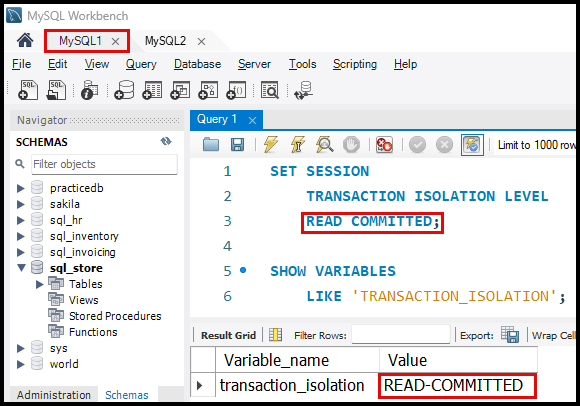
## Read Committed – Practice

* Initial state of the PRODUCTS table in the database…



### Session 1: MySQL1

* Let’s change the transaction isolation level to ‘READ COMMITTED’ for this MySQL1 session.



### Session 2: MySQL2

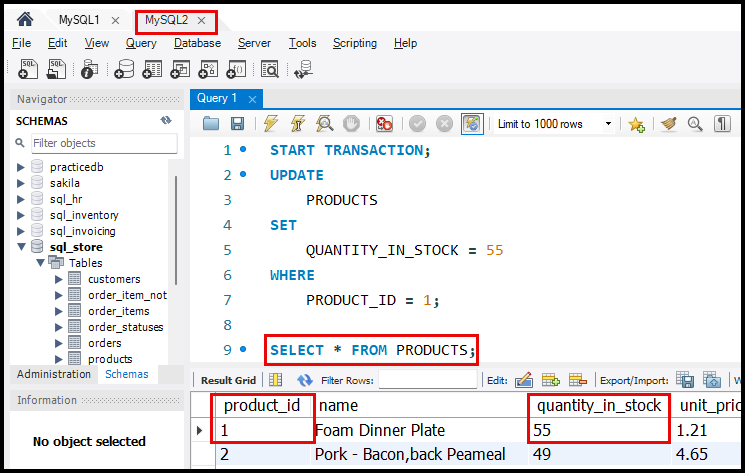
* Let’s change the transaction isolation level to ‘READ UNCOMMITTED’ for the MySQL2 session.



* Configuration is done. Now let’s modify the data for the PRODUCTS table is MySQL2 session and observer it in MySQL1 session.

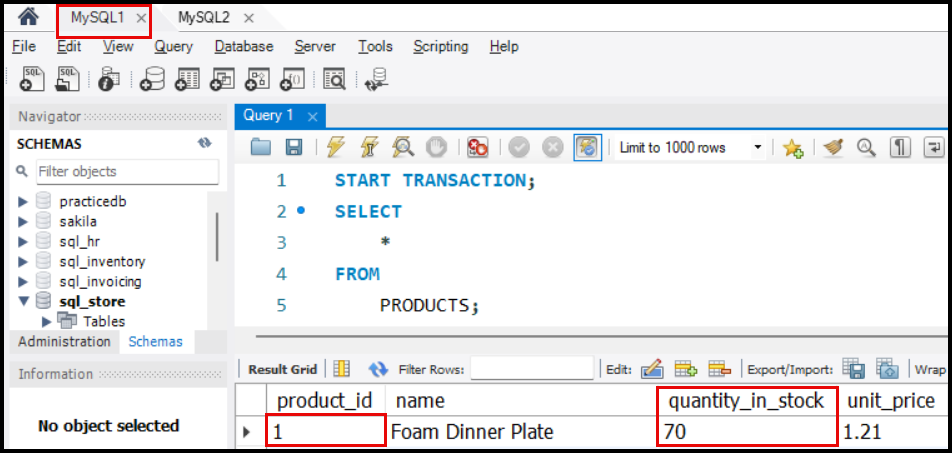
### Session 2: MySQL2

* We have updated the ‘QUANTITY\_IN\_STOCK = 55’ in the PRODUCTS table in MySQL2 session and if we see the value, it shows 55 here in this session.



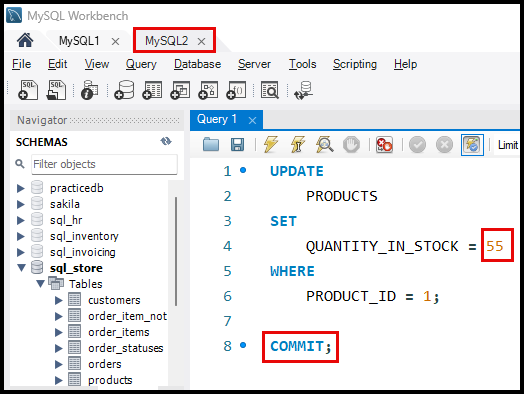
### Session 1: MySQL1

* If we check the value of ‘QUANTITY\_IN\_STOCK’ column in session MySQL1, it will still show the old value of 70 as the session MySQL2 has not committed the data yet.



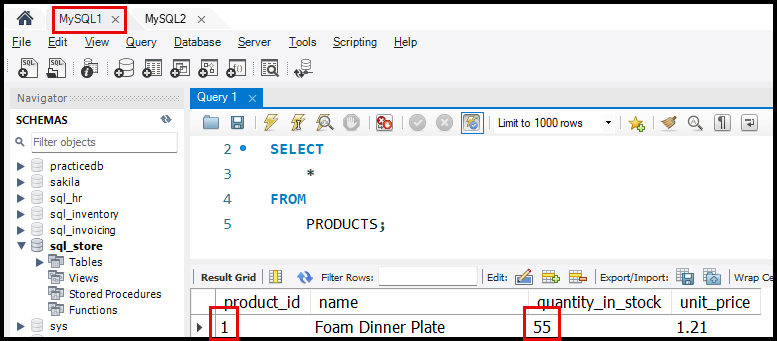
### Session 2: MySQL2

* Now let’s COMMIT the changes in the MySQL2 session.



### Session 1: MySQL1

* Let’s verify the data in MySQL1 session. Since the value was COMMITted in MySQL2 session, in MySQL1 session it shows the updated value of 55 as the ‘TRANSACTION ISOLATION LEVEL’ is ‘READ UNCOMMITTED’.



## Repeatable Read Isolation **Level**

* Ensures that the value of a row remains consistent across all reads within a transaction.
* **Behavior**:
  + Reads the latest committed value on the first read.
  + Subsequent reads within the transaction return the same value as the first read, even if the data is updated by another transaction.
* Prevents dirty reads and non-repeatable reads but **does not handle phantom reads**.

### **Key Problems and Examples**

#### **Dirty Read (Problem in Read Uncommitted)**

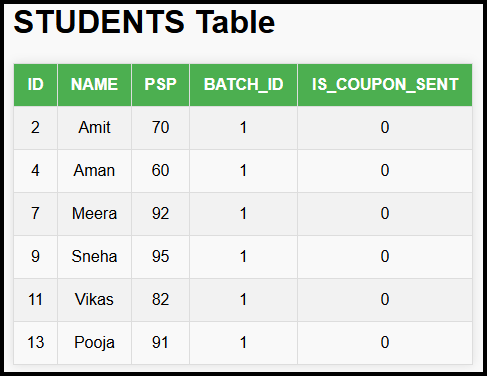
* **Definition**: Reading a value that is not yet committed.
* Example:
  + Transaction A writes a value but does not commit.
  + Transaction B reads this uncommitted value, which may later change or be rolled back.

#### **Non-Repeatable Read (Problem in Read Committed)**

* **Definition**: A scenario where the value read by a transaction changes because another transaction modifies the data in between.
* Example Use Case:
  + Selecting STUDENTS from a batch and sending them coupons.
  + **Steps**:
    1. Query all STUDENTS from batch ID 2 (SELECT \* FROM STUDENTS WHERE BATCH\_ID = 2;).
    2. Send coupons to the queried students.
    3. Update IS\_SENT to TRUE for these students (UPDATE STUDENTS SET IS\_COUPON\_SENT = TRUE WHERE BATCH\_ID = 2;).
  + **Problem**:
    - If a new student is added to the batch after the initial query but before the update, the new student will not receive the coupon, but their IS\_COUPON\_SENT value will still be updated.

### Problem Simulation

* To simulate the **Non-Repeatable Read** problem in the READCOMMITTED isolation level, setup the environment with STUDENTS table.



CREATE TABLE STUDENTS (

    ID INT AUTO\_INCREMENT PRIMARY KEY,

    NAME VARCHAR(50),

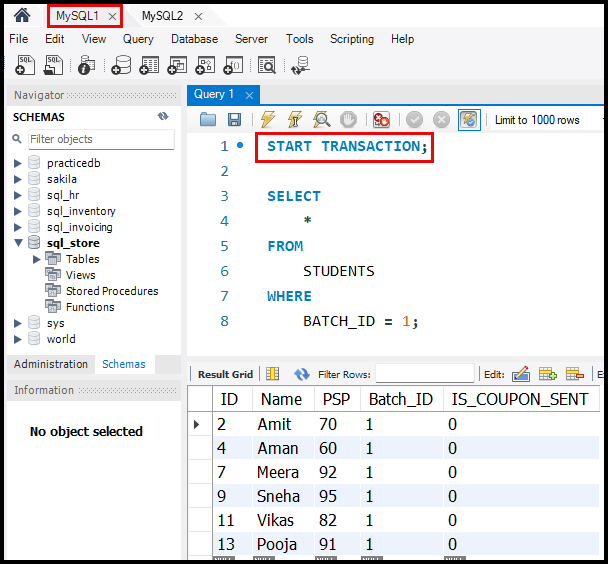
    PSP INT,

    BATCH\_ID INT,

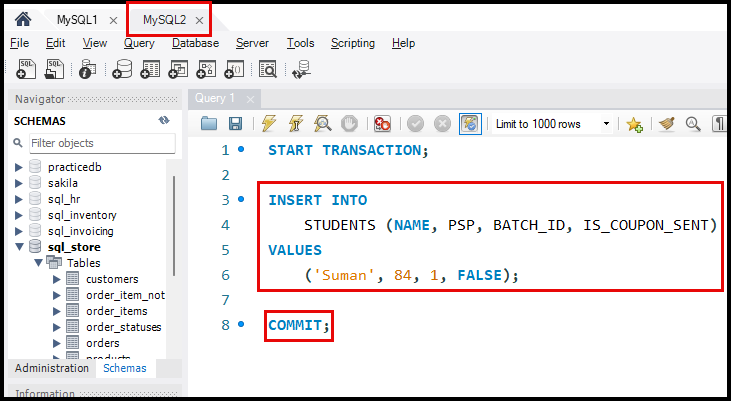
    IS\_COUPON\_SENT BOOLEAN DEFAULT FALSE

);

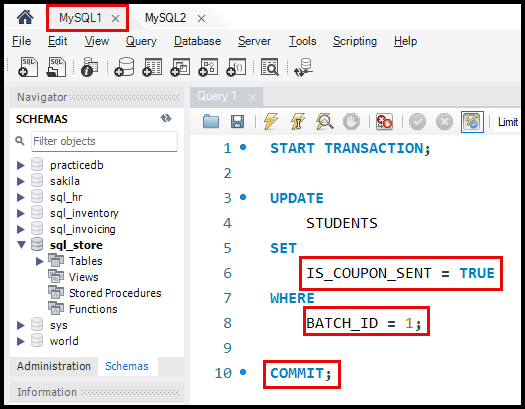
#### Start Transaction T1 (MySQL1)



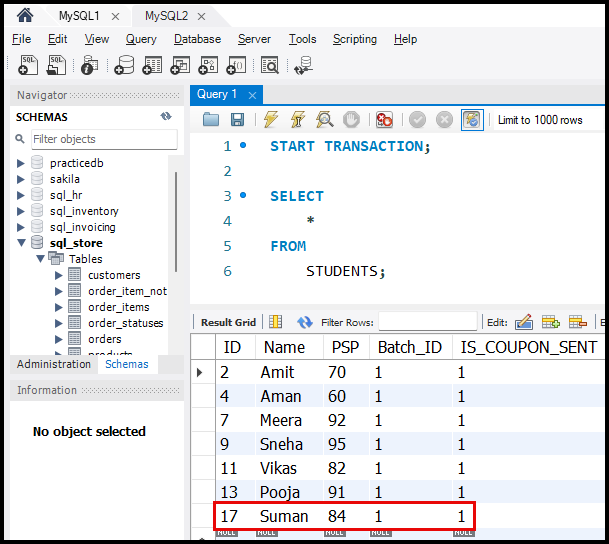
#### Start Transaction T2 (MySQL2)



#### Update Data in Transaction T1 (MySQL1)



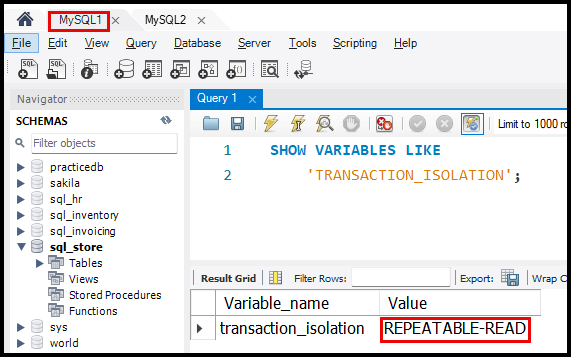
#### Check the data in Transaction T2 (MySQL2)



#### **Result of Non-Repeatable Read**

* **Problem**:
  + - * The initial SELECT in T1 (MySQL1) only retrieved all the students.
      * New Student ‘SUMAN’ was added in T2 (MySQL2) transaction.
      * However, the UPDATE in T1 (MySQL1) affects all students in BATCH\_ID = 1, including ‘SUMAN’.
* **Impact**:
  + ‘SUMAN’ receives the IS\_COUPON\_SENT = TRUE update even though he did not receive the coupon because he was not included in the initial query.

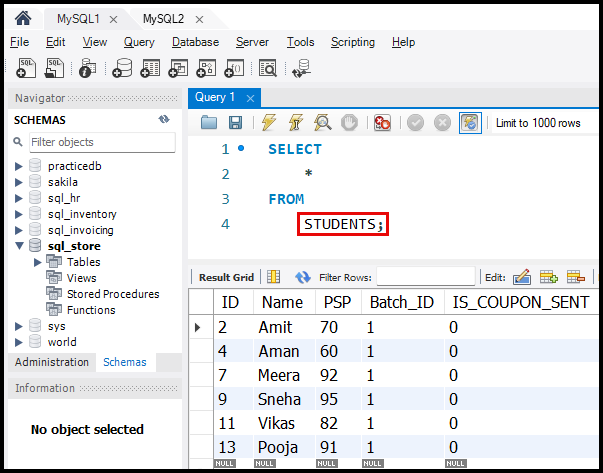
## Repeatable Read – Practice

* In MySQL, when a new session starts, transactions default to the REPEATABLE READ isolation level unless explicitly changed by using…

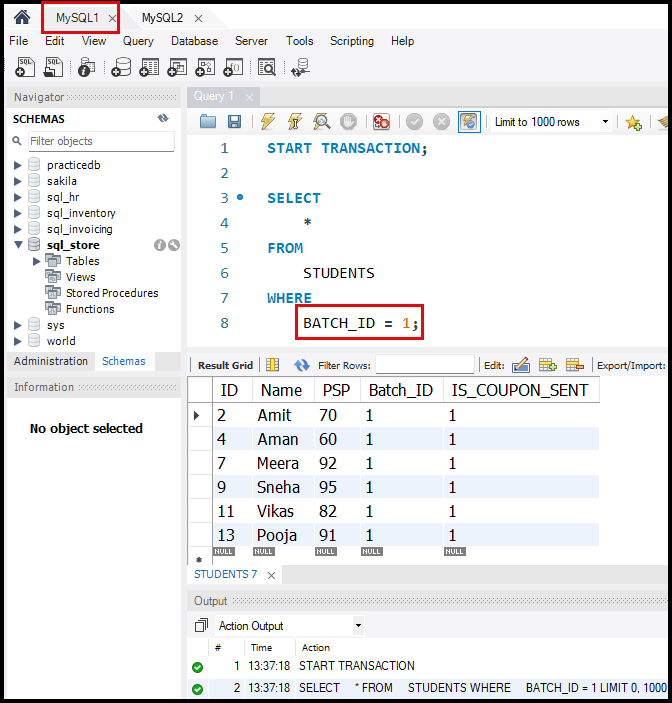
SET TRANSACTION ISOLATION LEVEL

    REPEATABLE READ;

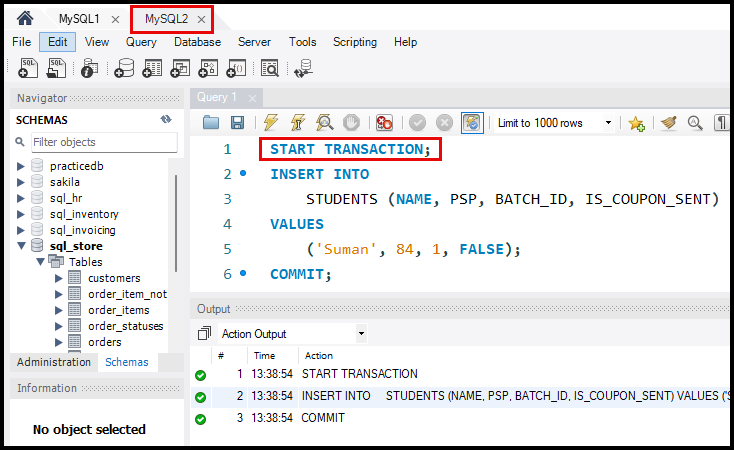
* Initial state of the STUDENTS table…



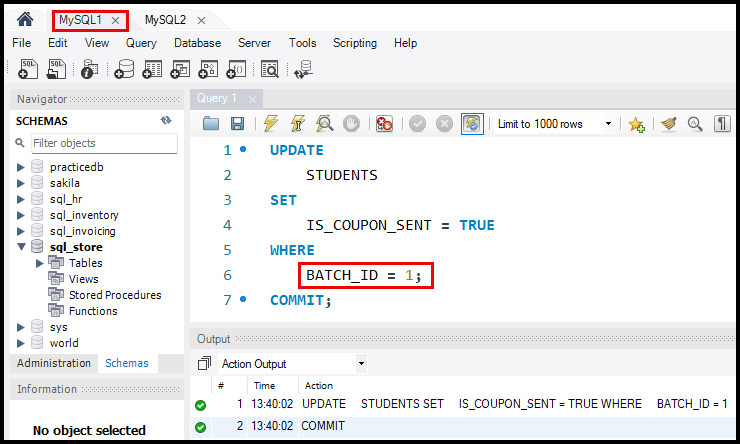
#### Start Transaction T1 (MySQL1)



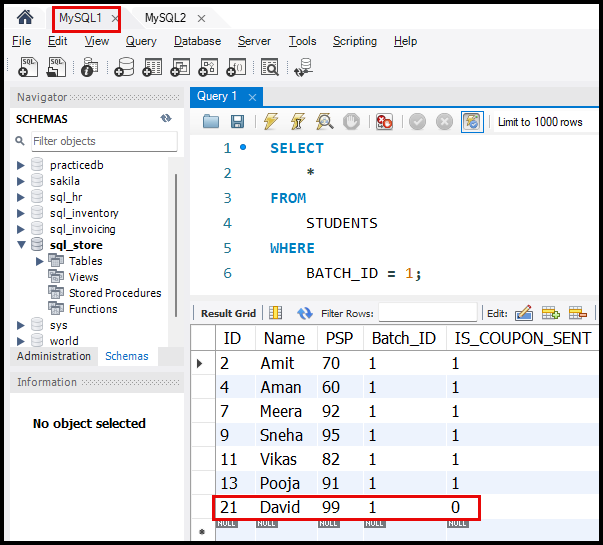
#### Start Transaction T2 (MySQL2)



#### Update Data in Transaction T1 (MySQL1)



#### Check the data



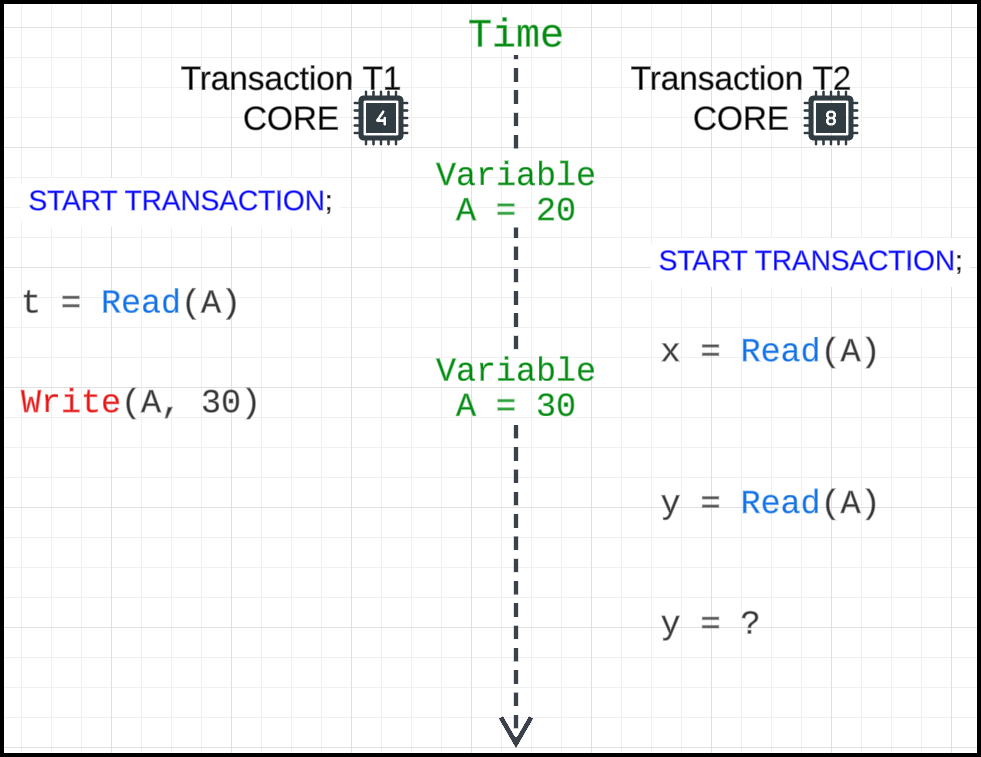
## Transaction Isolation Levels Explained

#### **Scenario Setup**

* **Initial Conditions**:
  + Value of A = 20.
  + Two transactions: T1 and T2.

#### **Steps in the Transactions:**

1. **Transaction T1**:

* Starts and reads the value of A (current value = 20) and stores it in a variable t.

1. **Transaction T2**:

* Starts and reads the value of A (current value = 20) and stores it in a variable x.

1. **Write in T1**:

* T1 updates the value of A to 30.

1. **Read in T2**:

* T2 attempts to read the value of A into variable y.

#### **Behaviour Across Isolation Levels**

1. READ UNCOMMITTED:

* Reads the latest uncommitted value.
* **y in T2**: 30.

1. READ COMMITTED**:**

* Reads the latest committed value.
* **y in T2**: 20 (value of A remains the last committed value).

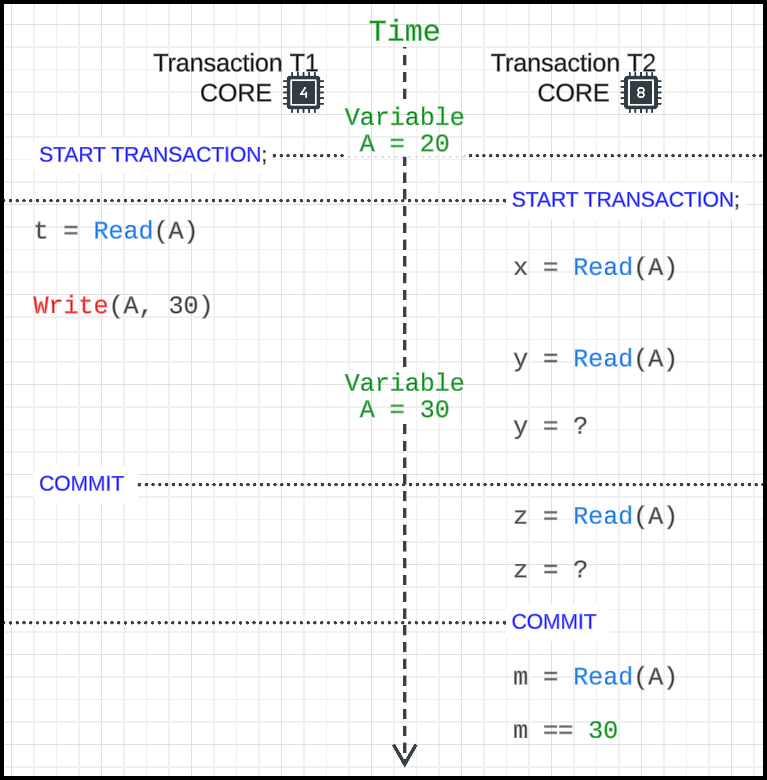
1. REPEATABLE READ:

* Ensures that the value of a row remains consistent across all reads within a transaction.
* For the **first read**, it behaves like Read Committed (reads the latest committed value).
* **y in T2**: 20 (value of A remains the same as the first read within T2).

#### **Committing Transactions**

1. **After T1 Commits**:

* T2 performs z = Read(A).
* Behavior:
  + READUNCOMMITTED: Reads the latest value (30).
  + READ COMMITTED: Reads the latest committed value (30).
  + REPEATABLE READ:
    - If A was previously read in T2, it keeps the old value (20).
    - The value of a row does not change within the same transaction.



#### **Post-Transaction Reads**

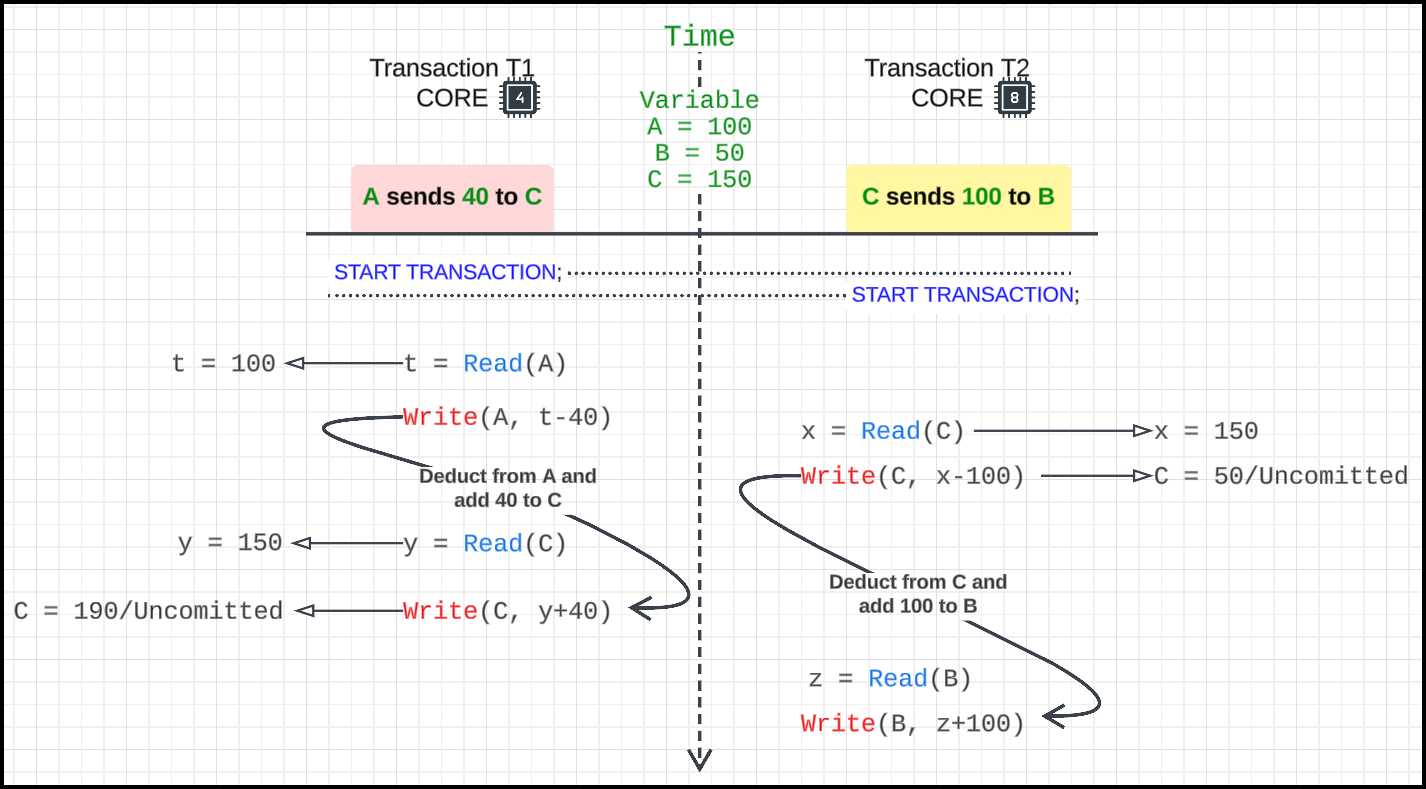
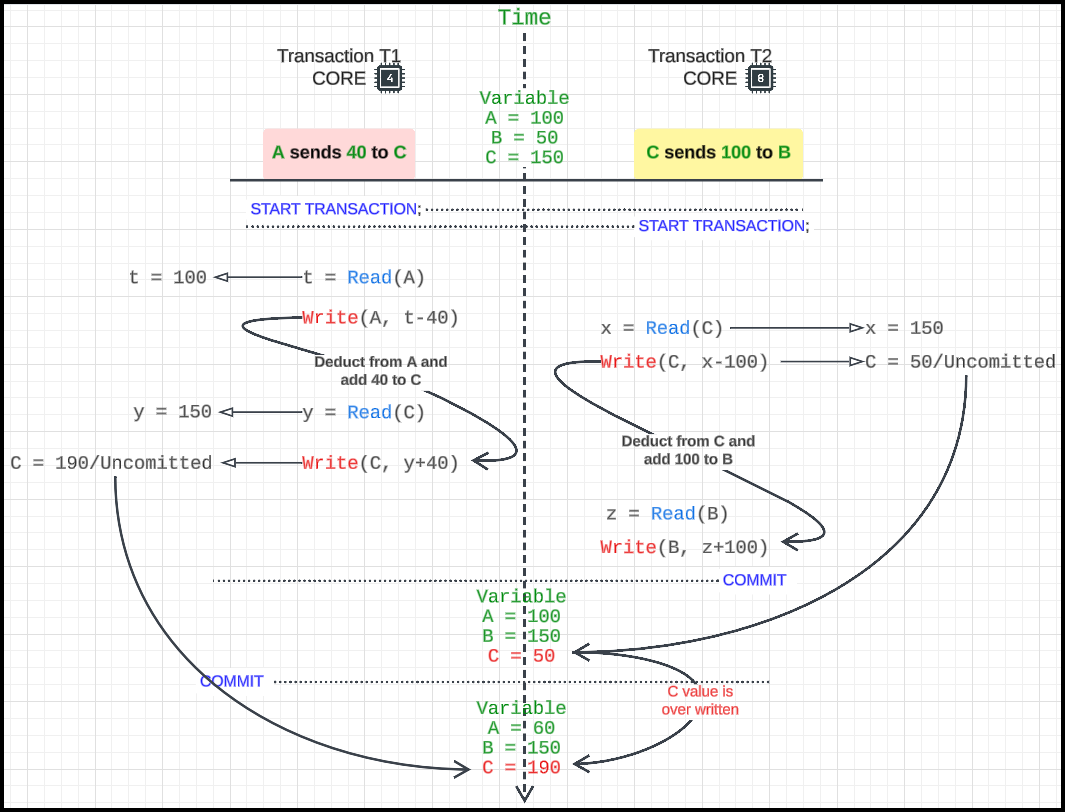
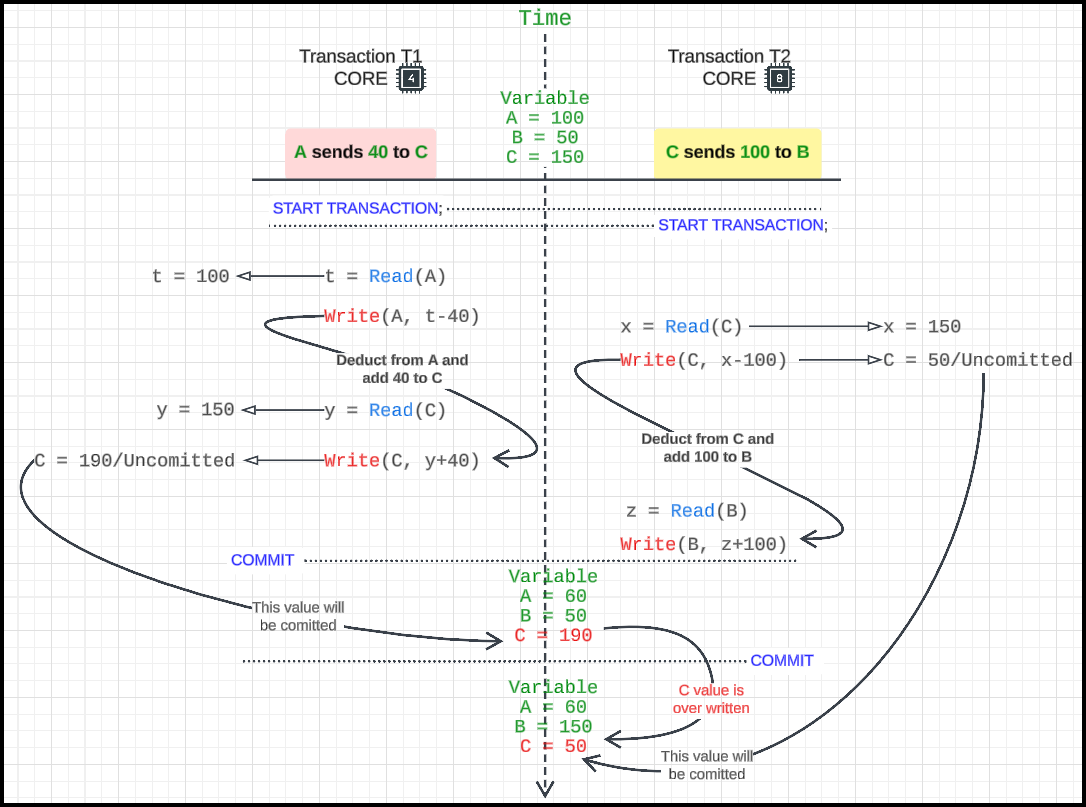
1. **After Both T1 and T2 Commit:**

* Any subsequent read of m = Read(A) will return the latest committed value (30).

## Serializable Isolation Level

### Examples and Scenarios:

Scenario 1 - Bank Example (Issues with Repeatable Read):

* Initial Setup:
  + Account balances: A = 100, B = 50, C = 150.
* Transactions:
  + T1: A sends 40 to C.
  + T2: C sends 100 to B.
* Actions:
  + T1 modifies A to 60 (uncommitted).
  + T2 reads C = 150 and modifies it to 50 (uncommitted).
  + T1 writes C = 190 (uncommitted).
  + T2 writes B = 150 (uncommitted).
* Commit Order Impact:
  + If T2 commits first, C = 50 is overwritten by T1 to 190.
  + If T1 commits first, C = 190 is overwritten by T2 to 50.

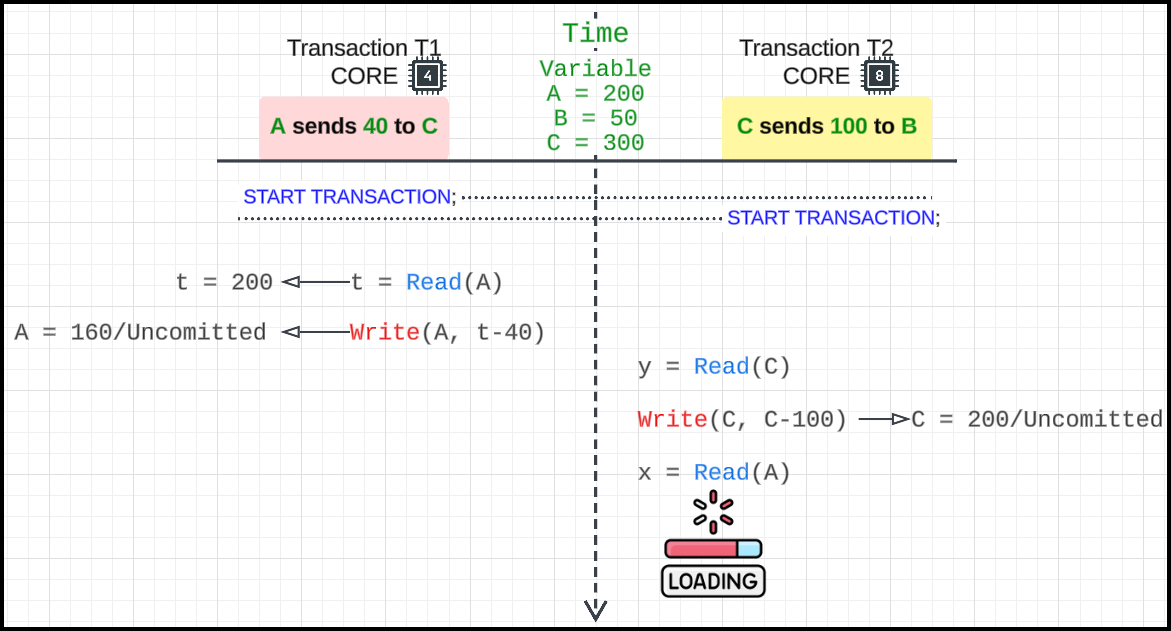
### Definition - Serializable Isolation Level

* Ensures transactions are executed in a **sequential order**, eliminating concurrency-related inconsistencies.
* Transactions are **serialized** (executed one after the other, not parallel).

### Mechanism:

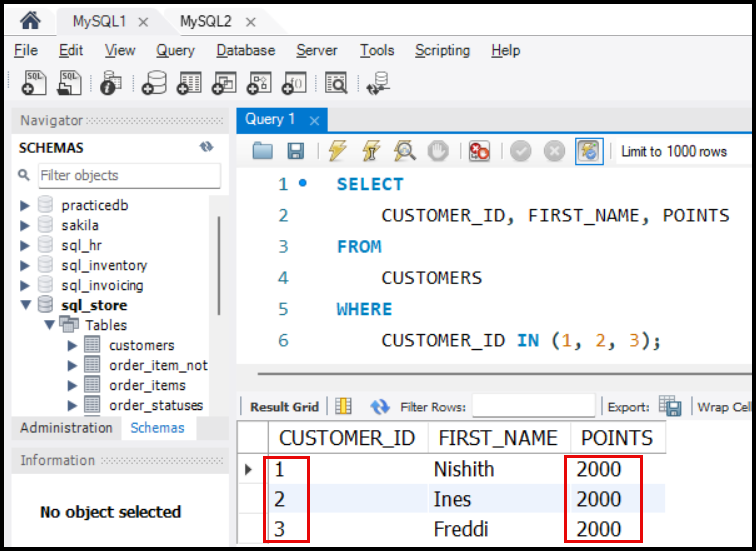
* Locks:
  + A row being used by one transaction cannot be read or written by another until the first transaction is committed or rolled back.
* Prevents interference between transactions.

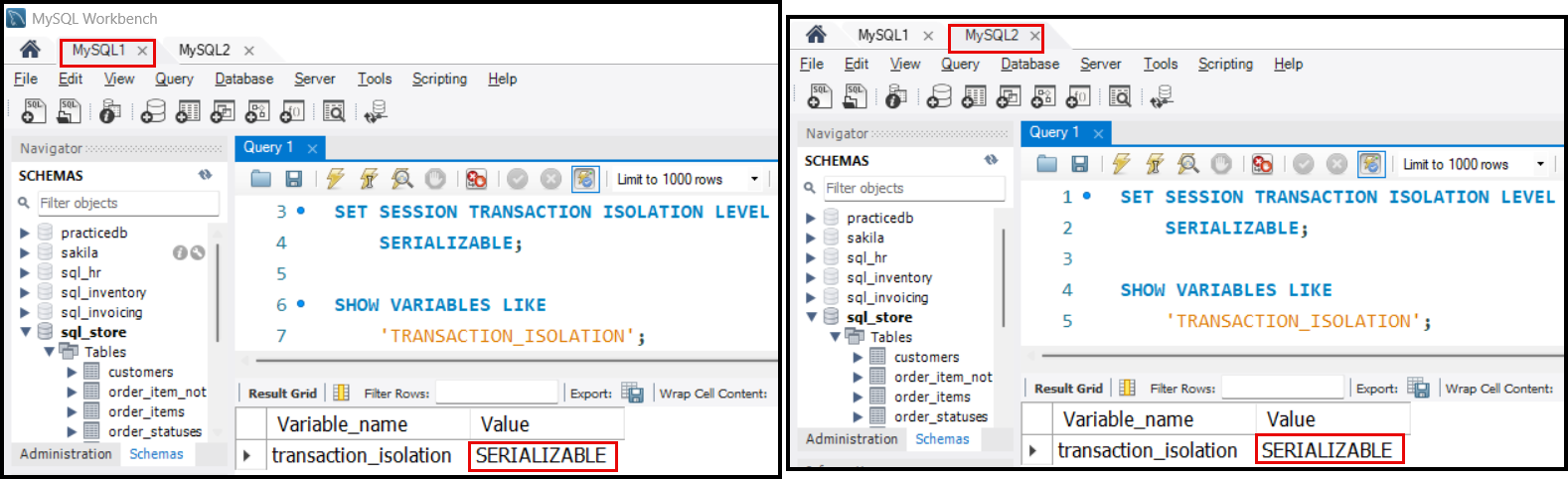
### Example:

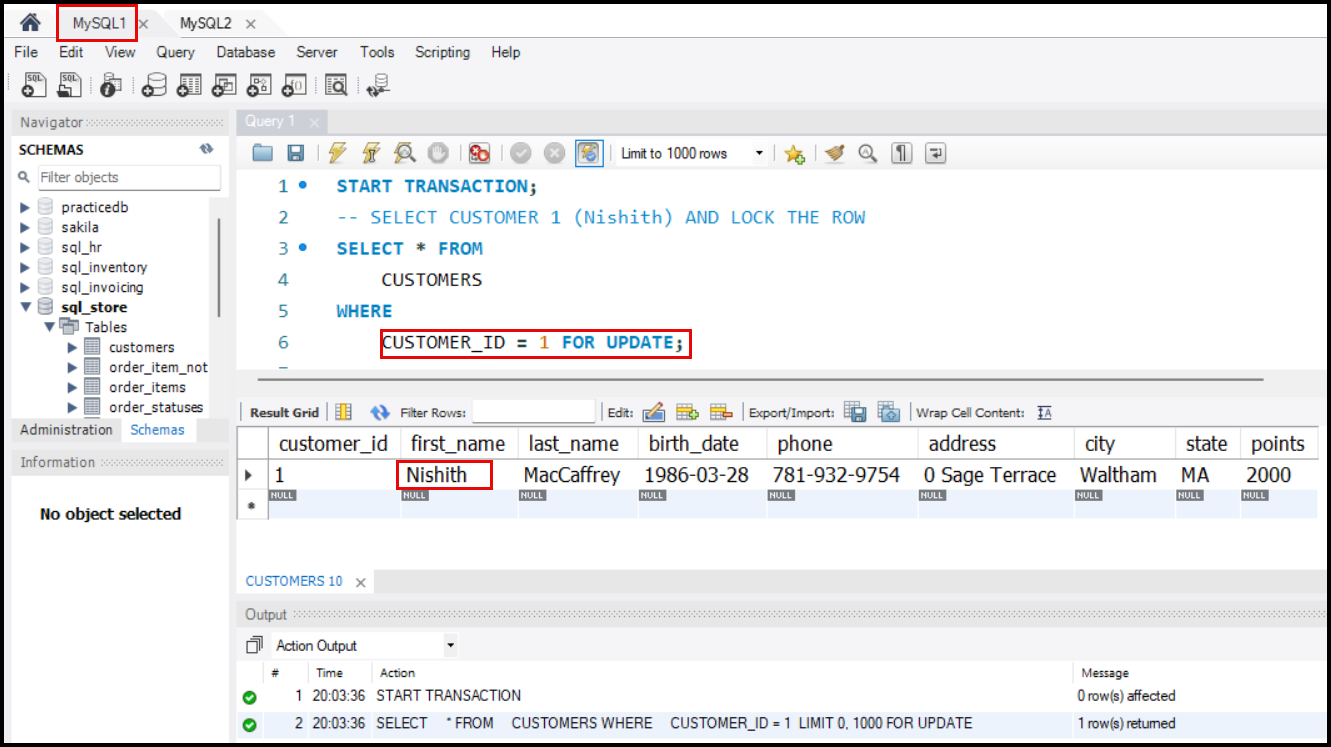
* Scenario:
  + T1 reads A = 200, deducts 40 and updates A with 160.
  + T2 reads C in variable y, which is 300.
  + T2, deducts 100 from C and updates it to 200 which will be allowed.
  + T2 attempts to read A while T1 is active.
* What Happens:
  + T2 waits until T1 COMMITs or ROLLBACKs before it can read A.

## Serializable – Practice

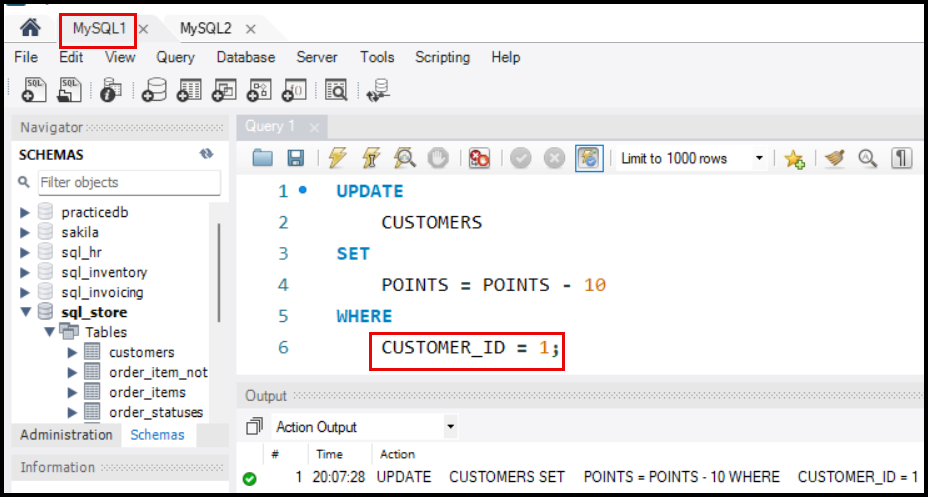
* Initial state of CUSTOMERS table in database…



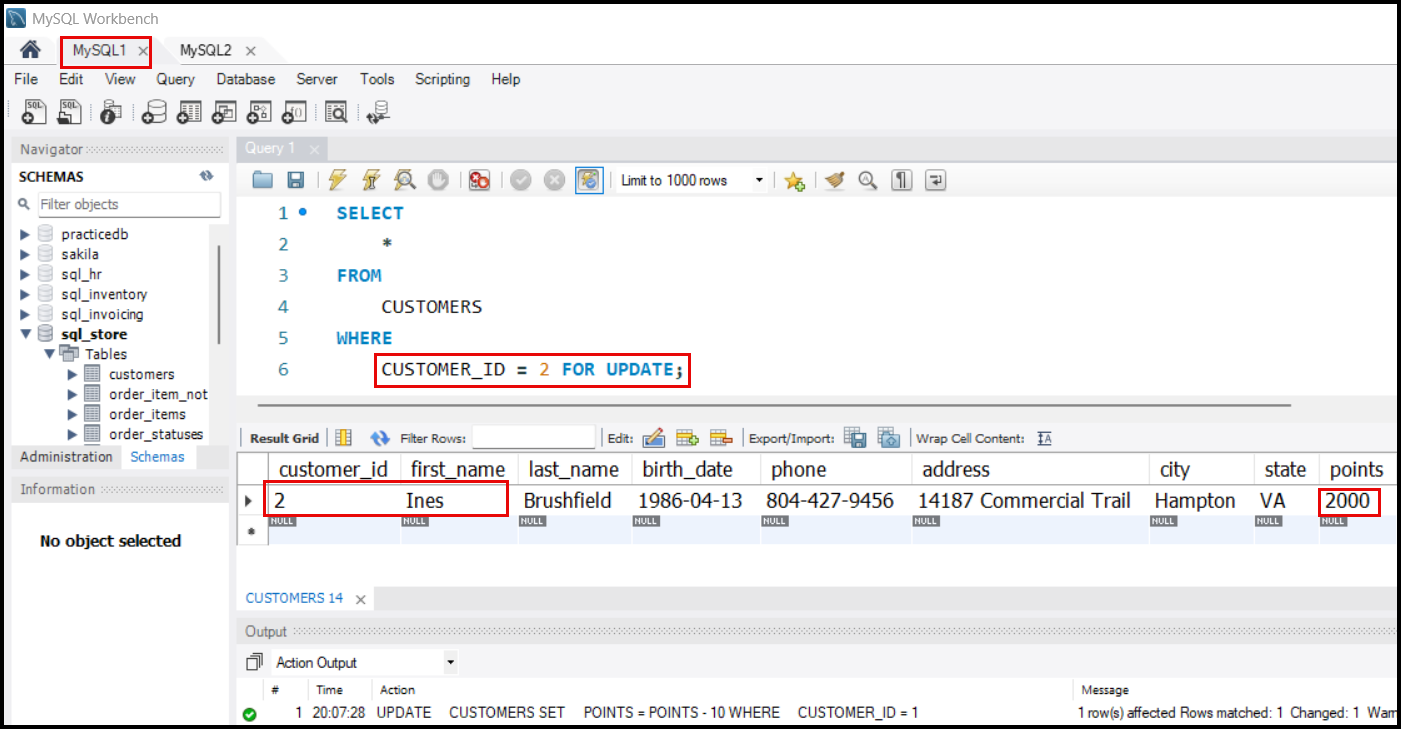
* Set Transaction isolation level SERIALIZABLE in both T1 (MySQL1) and T2 (MySQL2) session.
* Select customer 1 (Nishith) and lock the row using FOR UPDATE in T1 (MySQL1) session.



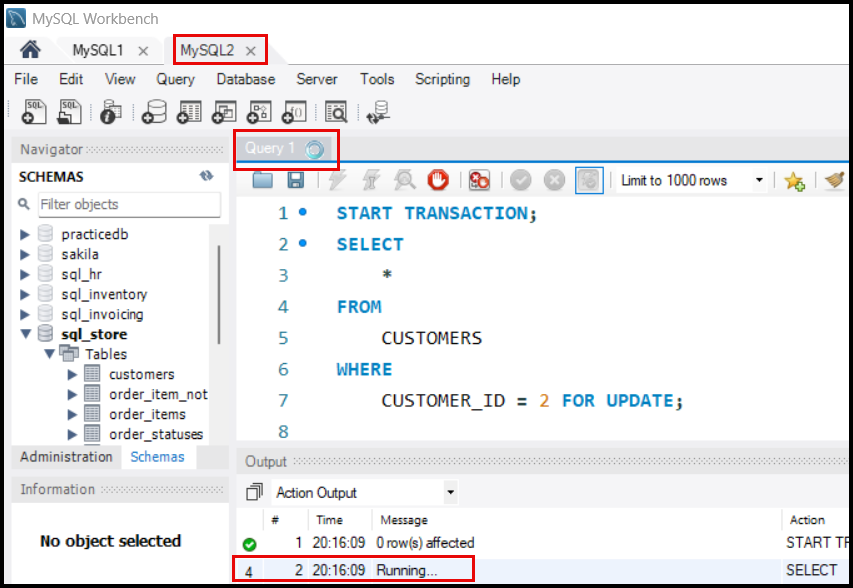
* Deduct points from customer 1 (Nishith)

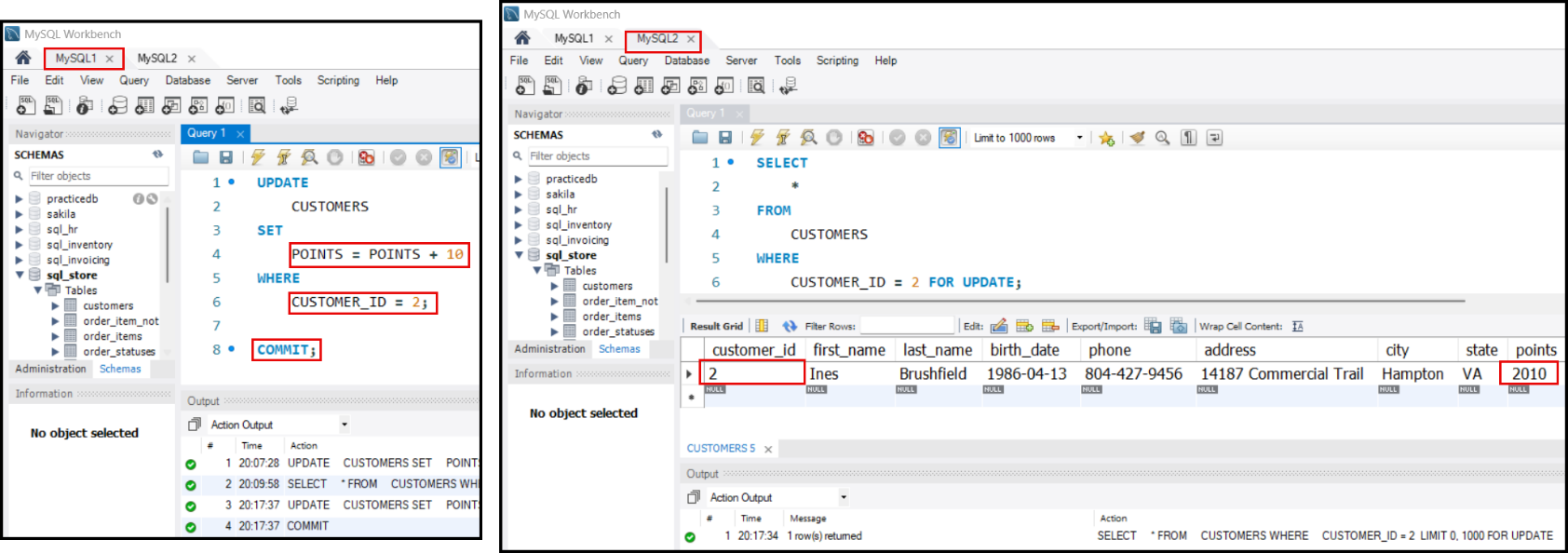


* Select customer 2 (Ines) and lock the row.



* In session T2 (MySQL2), start transaction and select customer 2 (Ines) and lock the row. When you try to lock the row using FOR UPDATE, the T2 (MySQL2) sessions goes into waiting state as shown below…



* Only after doing COMMIT or ROLLBACK in T1 (MySQL1) session, the T2 (MySQL2) sessions goes out of wait state and starts executing…

## **Transaction Isolation Levels - Serializable**

* **Serializable** is the strictest isolation level in databases.
* Transactions are executed sequentially, ensuring complete isolation.
* Ensures no other transaction can read/write a row that a current transaction is working on.
* Used in critical systems like banks to prevent data inconsistencies.

### **Row Locking with Serializable**

* During a transaction, a **lock** is placed on the rows being read or updated.
* This lock ensures:
  + No other transaction can read or write to the locked row until the current transaction completes (commits or rolls back).

### **Using** FORUPDATE **Clause**

* The FOR UPDATE clause explicitly locks rows that will be updated later in the transaction.
* This prevents other transactions from even reading these rows, ensuring complete isolation.
* Lock remains until the transaction ends.

### **Behaviour in Serializable Transactions**

* Other transactions attempting to access locked rows will go into a waiting state.
* Example: Transaction 2 must wait until Transaction 1 commits or rolls back.

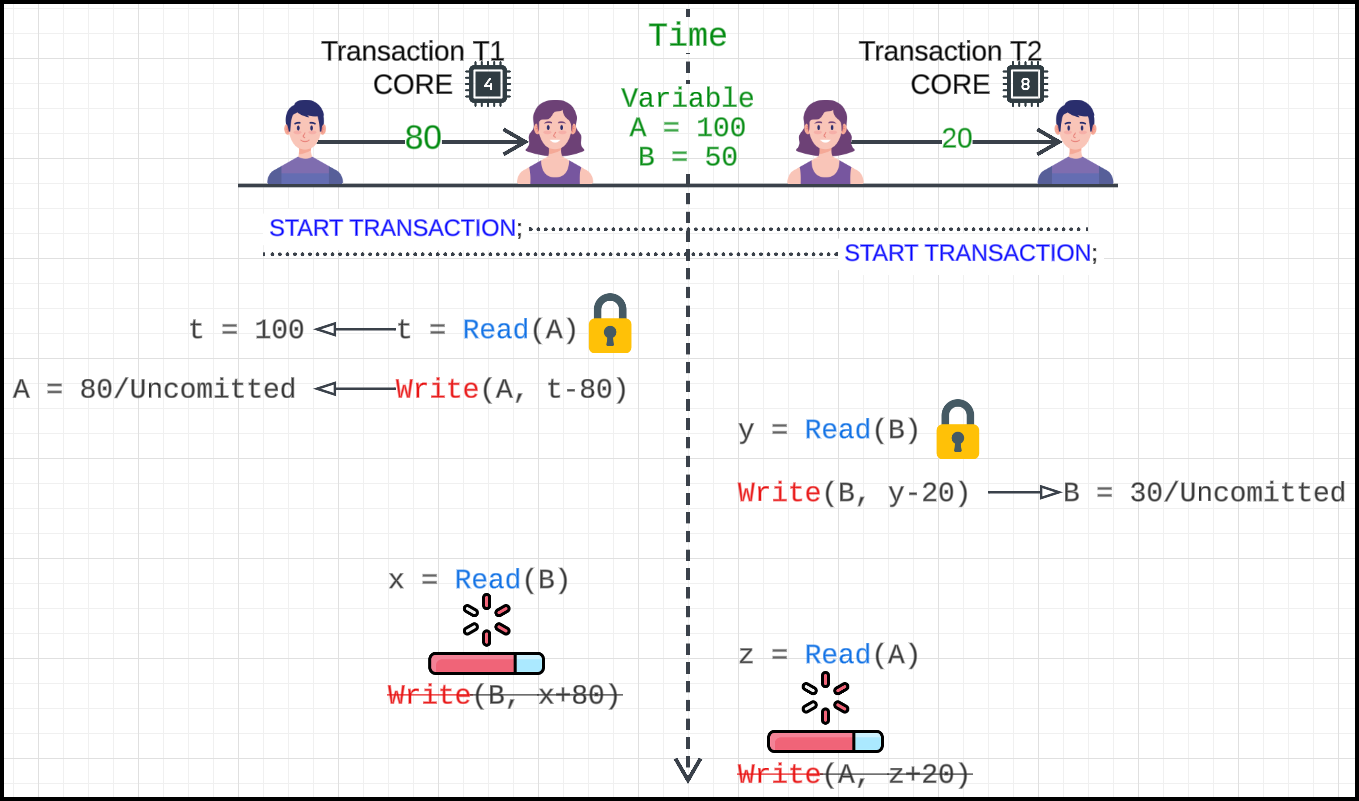
### **Timeout for Lock Waiting**

* Default timeout in MySQL is **30 seconds**.
* If a transaction cannot acquire a lock within this time, it will roll back automatically.

## Deadlocks

* A **deadlock** occurs when two or more transactions are waiting for each other to release locks, leading to a state where no progress is made.

### Example:

* **Transaction T1**:
  + Reads and locks Account A, deducts money 80 from A.
  + Tries to lock Account B but waits because it's locked by Transaction T2.
* **Transaction T2**:
  + Reads and locks Account B, deducts money 20 from A.
  + Tries to lock Account A but waits because it's locked by Transaction T1.
* Both transactions wait indefinitely, causing a **deadlock**.

### Deadlock Resolution in MySQL:

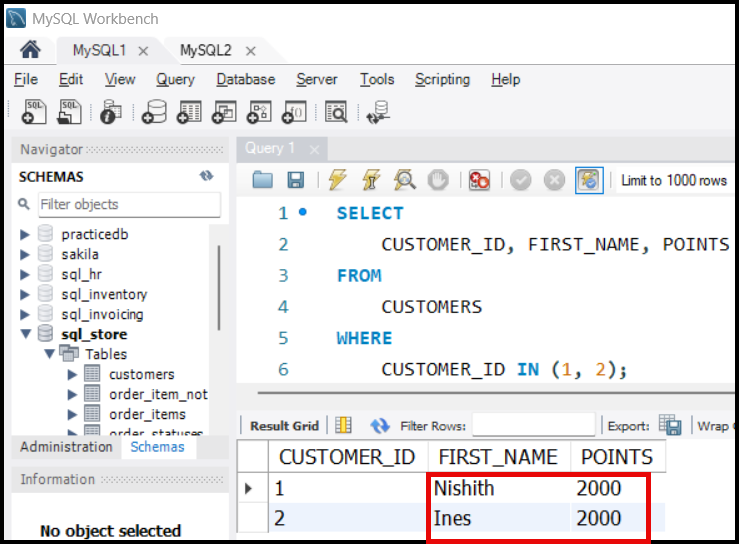
* MySQL detects deadlocks automatically.
* It rolls back one of the transactions to break the deadlock.
* After rollback, the other transaction can proceed.

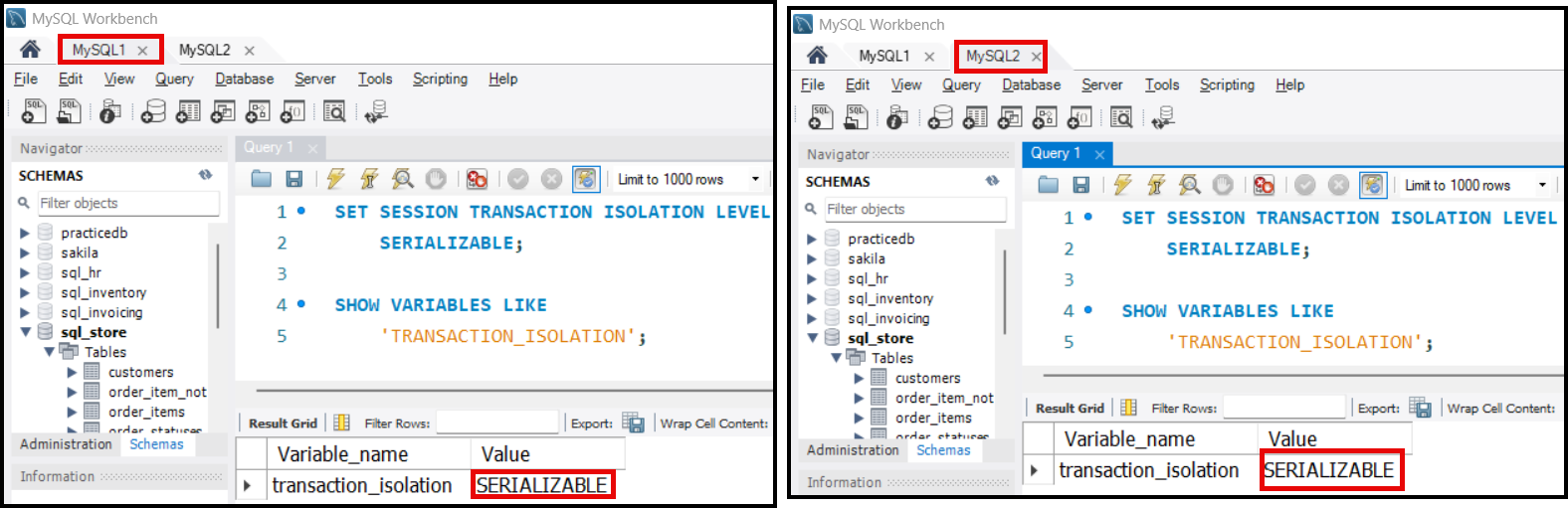
### Deadlock Detection:

* Deadlocks can be detected using **graph-based algorithms**, where transactions and locks are represented as nodes and edges in a graph.
* If a **cycle** is detected in the graph, a deadlock is present.

## Deadlock Practical

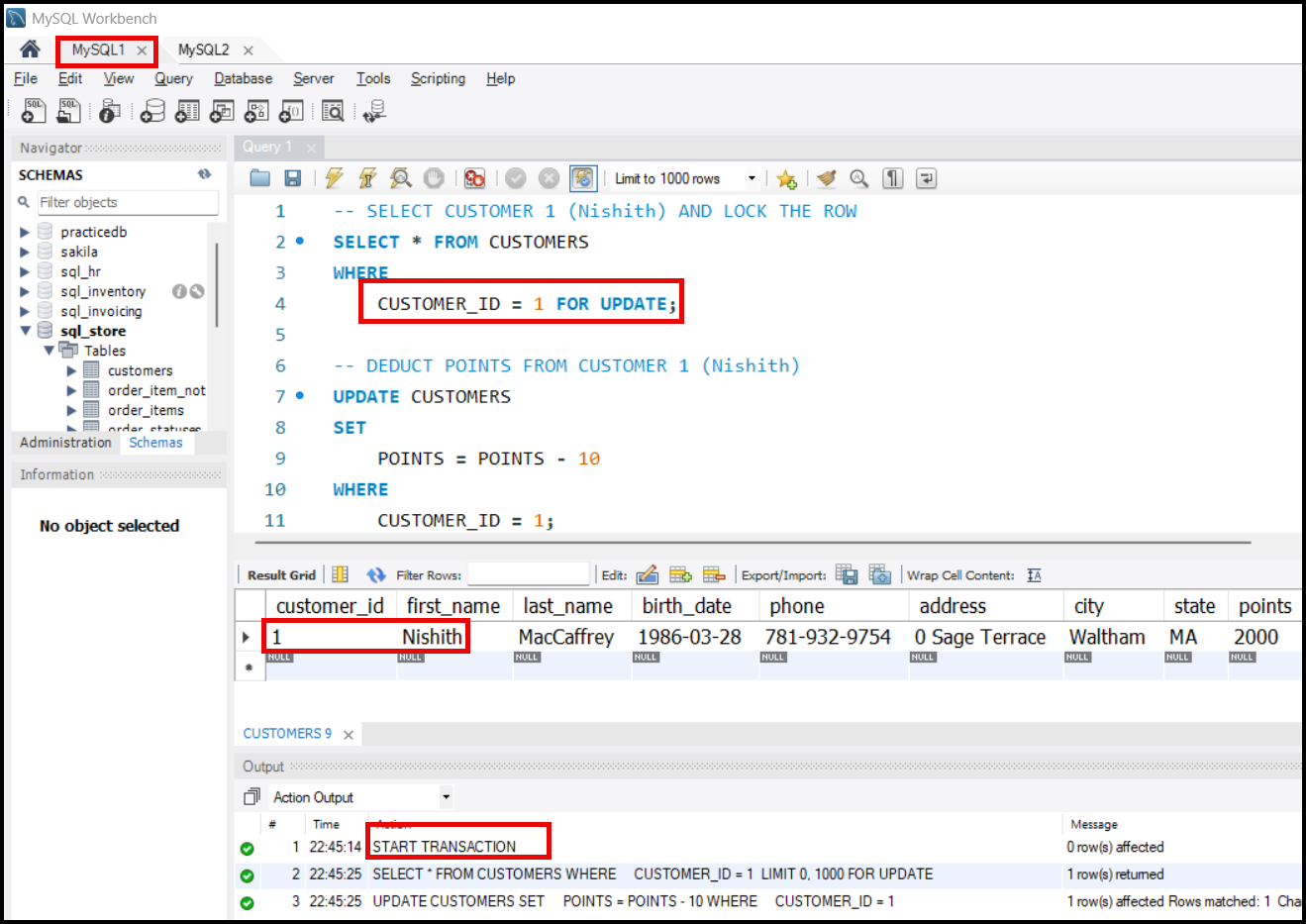
* Initial state of CUSTOMERS table for CUSTOMER\_ID 1 (‘Nishith’) and 2 (‘Ines’).



* Set the session transaction isolation level to SERIALIZABLE in both T1 (MySQL1) session and in T2 (MySQL2) session.

### Session 1: MySQL1

* Start a transaction first.
* Select Customer 1 (Nishith) and lock the corresponding row. Deduct points from Customer 1 (Nishith).



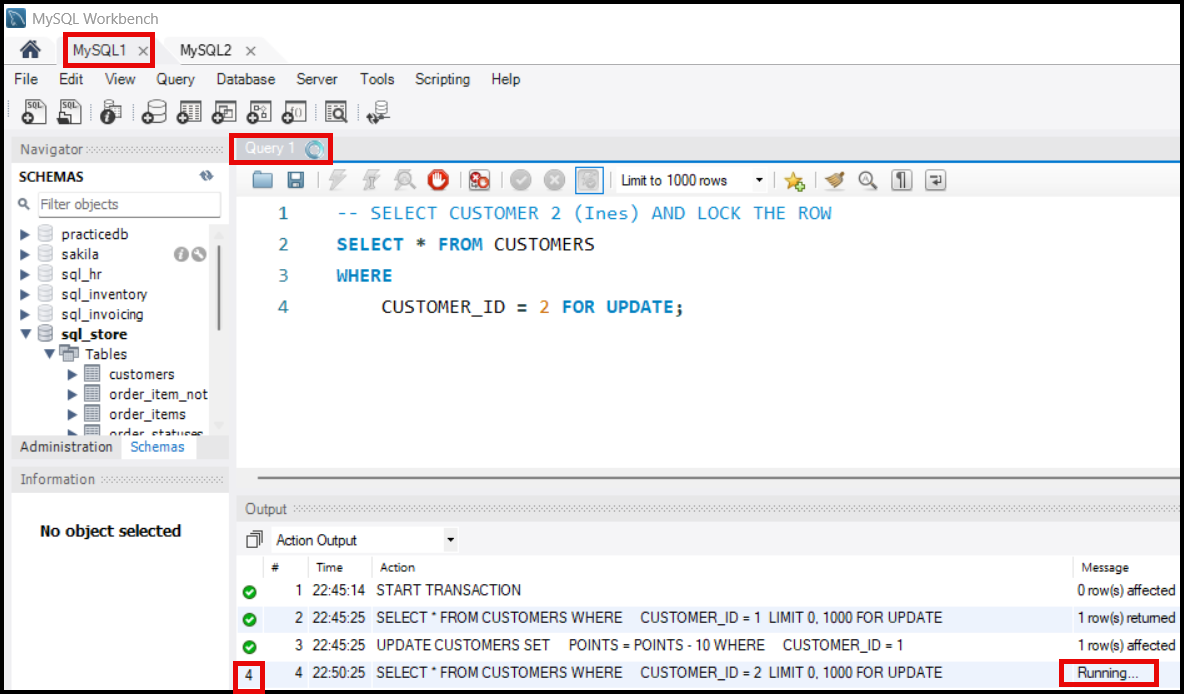
### Session 2: MySQL2

* Start a transaction first. Select Customer 2 (Ines) and lock the corresponding row. Deduct points from Customer 2 (Ines).



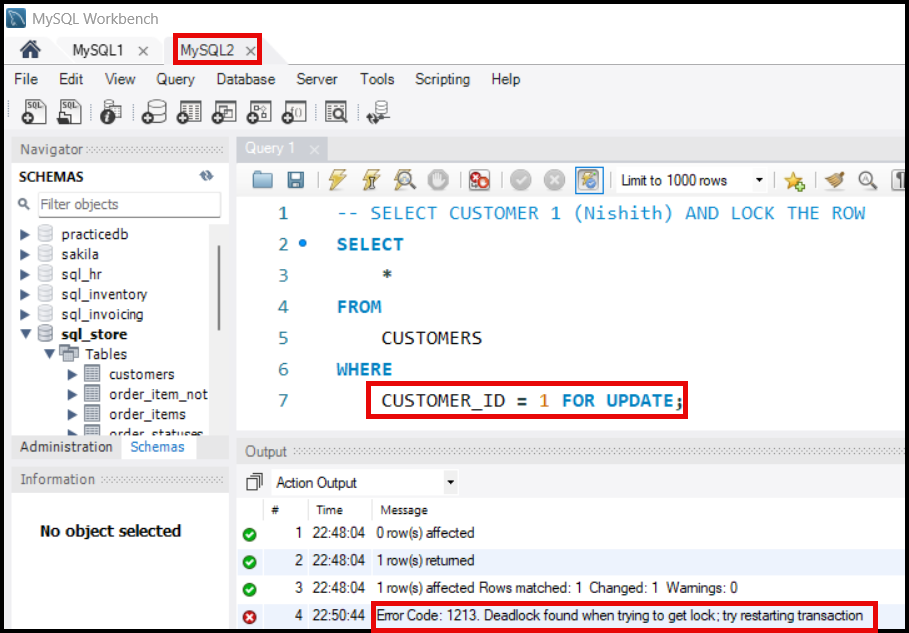
### Session 1: MySQL1

* Now we have to update the point which we have deducted from Customer 1 (Nishith) in Customer 2 (Ines). So read the points from Customer 2 first.



* Since session 2(MySQL2) has locked on the customer 2, session 1 (MySQL1) is waiting…

### Session 2: MySQL2

* Now try to read the Customer 1 in session 2 (MySQL2).
* Since Customer 1 is already locked in session 1 (MySQL1), now both sessions 1 and 2 enter deadlock and is detected by MySQL.
* MySQL automatically detected the deadlock, rolled back the transactions in Session 2 (MySQL2), and as a result, the other transaction in Session 1 (MySQL1) was able to acquire the lock and proceed to the next operation as shown below.

