**Complete Guide MySQL Transactions (With Tables & Examples**

**Concept of Transactions**

🔹 **a transaction is a sequence of SQL statements that must be executed as a single unit**.  
🔹 If any statement fails, the entire transaction should be **rolled back**.  
🔹 If all statements succeed, they should be **committed**.

**Introduce ACID Properties**

| **Property** | **Meaning** | **Real-World Example** |
| --- | --- | --- |
| **Atomicity** | Either all or none of the operations should be executed. | A bank transfer should either complete fully or not at all. |
| **Consistency** | Data should remain valid before and after the transaction. | A student's grades should always be within valid limits. |
| **Isolation** | Transactions should not interfere with each other. | Two people booking the last flight seat should not get the same seat. |
| **Durability** | Once committed, data should not be lost. | Even after a system crash, confirmed orders should remain in an e-commerce system. |

To explain transactions effectively, we’ll create a **banking system** with an accounts table.

**Create the bank database**

CREATE DATABASE bank;

USE bank;

**Create the accounts table**

CREATE TABLE accounts (

account\_id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(50) NOT NULL,

balance DECIMAL(10,2) NOT NULL

) ENGINE=InnoDB; -- Ensure InnoDB for transactions

**Insert Sample Data**

INSERT INTO accounts (name, balance) VALUES ('Alice', 5000);

INSERT INTO accounts (name, balance) VALUES ('Bob', 3000);

SELECT \* FROM accounts;

**📌 Expected Output**

| **account\_id** | **name** | **balance** |
| --- | --- | --- |
| 1 | Alice | 5000.00 |
| 2 | Bob | 3000.00 |

Now, students can **see** and **understand** the data before transactions.

**💰 Bank Transfer without Transaction**

UPDATE accounts SET balance = balance - 1000 WHERE account\_id = 1; -- Deduct 1000 from Alice

UPDATE accounts SET balance = balance + 1000 WHERE account\_id = 2; -- Add 1000 to Bob

📌 **Problem:** If the second query fails, Alice loses money but Bob doesn’t get it.

**💰 Using Transactions (Correct Way)**

START TRANSACTION;

UPDATE accounts SET balance = balance - 1000 WHERE account\_id = 1;

UPDATE accounts SET balance = balance + 1000 WHERE account\_id = 2;

COMMIT; -- Saves the changes permanently

**Explain:** Now, if the transaction completes successfully, both updates are saved.

**Using ROLLBACK**

**Scenario:** What if something goes wrong?

**💰 Failed Transaction Example**

START TRANSACTION;

UPDATE accounts SET balance = balance - 2000 WHERE account\_id = 1;

UPDATE accounts SET balance = balance + 2000 WHERE account\_id = 2;

ROLLBACK; -- Undo all changes

🔹 **ROLLBACK cancels all updates** so the data remains unchanged.  
🔹 Ask students to run SELECT \* FROM accounts; before and after rollback to see no changes.

**Using SAVEPOINT (Partial Rollback)**

🔹 **SAVEPOINT** allows rolling back to a specific point instead of canceling everything.

**Example with Savepoint**

START TRANSACTION;

UPDATE accounts SET balance = balance - 500 WHERE account\_id = 1;

SAVEPOINT first\_update;

UPDATE accounts SET balance = balance + 500 WHERE account\_id = 2;

SAVEPOINT second\_update;

ROLLBACK TO first\_update; -- Undo second update only

COMMIT;

✅ **Only Alice’s deduction remains, Bob’s addition is rolled back.**

**Understanding SET AUTOCOMMIT**

MySQL **auto-commits** transactions by default.

**Disable Auto-Commit (Manual Commit Needed)**

SET AUTOCOMMIT = 0; -- Turns off auto-commit

START TRANSACTION;

UPDATE accounts SET balance = balance - 300 WHERE account\_id = 1;

UPDATE accounts SET balance = balance + 300 WHERE account\_id = 2;

COMMIT;

**📌 Key Takeaway**: Without COMMIT;, changes are **not saved permanently**.

**Explain Transaction Isolation Levels**

Isolation levels determine how transactions interact when running simultaneously.

| **Isolation Level** | **Behavior** |
| --- | --- |
| **READ UNCOMMITTED** | Can read uncommitted data (dirty reads). |
| **READ COMMITTED** | Only committed data is visible to transactions. |
| **REPEATABLE READ** (Default) | Ensures that the same data remains unchanged during a transaction. |
| **SERIALIZABLE** | Fully isolated, no two transactions can overlap. |

**Example: Setting Isolation Level**

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

**Avoiding Deadlocks**

🔹 A **deadlock** happens when two transactions wait for each other to release resources.

**Example of Deadlock**

-- Transaction 1

START TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE account\_id = 1;

UPDATE accounts SET balance = balance + 100 WHERE account\_id = 2;

COMMIT;

-- Transaction 2 (executed at the same time)

START TRANSACTION;

UPDATE accounts SET balance = balance + 100 WHERE account\_id = 2;

UPDATE accounts SET balance = balance - 100 WHERE account\_id = 1;

COMMIT;

**Solution:** Avoid locking rows in different orders.

**Best Practices for Transactions**

✔ **Use transactions for critical operations** like banking, inventory, etc.  
✔ **Keep transactions short** to reduce lock time.  
✔ **Use SAVEPOINT to allow partial rollbacks.**  
✔ **Always handle errors and rollback when needed.**  
✔ **Choose an appropriate isolation level** based on the requirement.

**Practical Exercises**

✅ **Exercise 1:** Simulate a bank transfer using transactions.  
✅ **Exercise 2:** Implement ROLLBACK when insufficient balance exists.  
✅ **Exercise 3:** Use SAVEPOINT to cancel part of a transaction.  
✅ **Exercise 4:** Demonstrate the effect of different isolation levels.