# ****Detailed Notes on TCL (Transaction Control Language) in MySQL****

### ****📌 What is TCL?****

TCL (Transaction Control Language) in MySQL is used to **manage transactions**, ensuring **data consistency** and **integrity** in databases. It allows you to commit or roll back changes made during a transaction.

## ****🚀 TCL Commands in MySQL****

| **Command** | **Description** |
| --- | --- |
| START TRANSACTION | Begins a transaction. |
| COMMIT | Saves all changes made in the transaction permanently. |
| ROLLBACK | Undoes all changes made in the transaction. |
| SAVEPOINT | Creates a temporary save point to roll back part of a transaction. |
| RELEASE SAVEPOINT | Deletes a savepoint. |
| SET AUTOCOMMIT | Enables or disables automatic commits. |

## ****1. START TRANSACTION – Begin a Transaction****

A transaction starts when you execute START TRANSACTION. It **groups multiple queries** into a single unit, so they are executed together.

### ****Syntax:****

START TRANSACTION;

### ****Example:****

START TRANSACTION;

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

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

COMMIT; -- Saves changes permanently

**Without** COMMIT**, changes will not be saved.**

## ****2. COMMIT – Save Changes Permanently****

After making multiple changes in a transaction, use COMMIT to **permanently save** them.

### ****Syntax:****

COMMIT;

### ****Example:****

START TRANSACTION;

UPDATE employees SET salary = salary + 1000 WHERE emp\_id = 101;

UPDATE employees SET salary = salary + 500 WHERE emp\_id = 102;

COMMIT; -- Confirms and saves both updates

👉 Once committed, **changes cannot be undone**.

## ****3. ROLLBACK – Undo Changes****

If you make an error, ROLLBACK cancels all changes since the last START TRANSACTION.

### ****Syntax:****

ROLLBACK;

### ****Example:****

START TRANSACTION;

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

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

ROLLBACK; -- Undo all changes

👉 This **restores the original state** before the transaction started.

## ****4. SAVEPOINT – Create a Checkpoint****

SAVEPOINT lets you create **a temporary point** within a transaction. If something goes wrong, you can **rollback to a savepoint** instead of rolling back the entire transaction.

### ****Syntax:****

SAVEPOINT savepoint\_name;

### ****Example:****

START TRANSACTION;

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

SAVEPOINT sp1; -- Create a checkpoint

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

SAVEPOINT sp2; -- Create another checkpoint

ROLLBACK TO sp1; -- Undo only the second update

COMMIT; -- Save the first update permanently

👉 ROLLBACK TO savepoint\_name; will undo only changes after that savepoint.

## ****5. RELEASE SAVEPOINT – Delete a Savepoint****

Deletes a savepoint, so it can no longer be used for rollback.

### ****Syntax:****

RELEASE SAVEPOINT savepoint\_name;

### ****Example:****

RELEASE SAVEPOINT sp1;

## ****6. SET AUTOCOMMIT – Enable or Disable Auto-Commit****

By default, MySQL automatically commits each query. You can disable this using SET AUTOCOMMIT.

### ****Syntax:****

SET AUTOCOMMIT = {0 | 1};

* SET AUTOCOMMIT = 0; → Disables auto-commit (manual commit needed).
* SET AUTOCOMMIT = 1; → Enables auto-commit (default).

### ****Example:****

SET AUTOCOMMIT = 0; -- Disable auto-commit

START TRANSACTION;

UPDATE products SET stock = stock - 10 WHERE product\_id = 5;

COMMIT; -- Manually commit

SET AUTOCOMMIT = 1; -- Re-enable auto-commit

## ****🚀 Example Scenario: Bank Transaction****

### ****Scenario:****

A bank transfers **$200** from **Account A** to **Account B**. If something goes wrong (e.g., network failure), the transaction should be **rolled back** to avoid inconsistent balances.

START TRANSACTION;

UPDATE accounts SET balance = balance - 200 WHERE account\_id = 1;

UPDATE accounts SET balance = balance + 200 WHERE account\_id = 2;

-- If everything is fine, save changes

COMMIT;

-- If an error occurs, rollback changes

ROLLBACK;

## ****🔹 Best Practices for Using TCL****

✅ Always **use** COMMIT **carefully**—once committed, changes **cannot** be undone.  
✅ Use SAVEPOINT for **complex transactions** to avoid full rollback.  
✅ **Disable** AUTOCOMMIT before making multiple changes that should be grouped.  
✅ **Test transactions** in a development environment before applying them to live data.

## ****🚀 Summary****

| **Command** | **Description** |
| --- | --- |
| START TRANSACTION | Begins a transaction. |
| COMMIT | Saves all changes permanently. |
| ROLLBACK | Undoes all changes. |
| SAVEPOINT | Creates a checkpoint in a transaction. |
| RELEASE SAVEPOINT | Deletes a savepoint. |
| SET AUTOCOMMIT | Enables/disables auto-commit. |

Would you like me to help you design a TCL-based transaction for your project? 🚀😊

# ****Real-World Example of TCL in MySQL (Step-by-Step)****

### ****💼 Scenario: Bank Transaction Management System****

We will simulate a real-world **banking transaction** where money is transferred between two accounts. The system must ensure that:

1. If all operations succeed, the transaction **commits**.
2. If something goes wrong, it **rolls back** to avoid incorrect balances.

## ****📝 Step 1: Create a New Database****

CREATE DATABASE BankDB;

USE BankDB;

## ****📝 Step 2: Create Tables****

### ****Accounts Table****

Each account has an ID, holder name, and balance.

CREATE TABLE accounts (

account\_id INT PRIMARY KEY AUTO\_INCREMENT,

holder\_name VARCHAR(100),

balance DECIMAL(10,2) NOT NULL

);

### ****Transactions Table****

Logs all money transfers.

CREATE TABLE transactions (

transaction\_id INT PRIMARY KEY AUTO\_INCREMENT,

from\_account INT,

to\_account INT,

amount DECIMAL(10,2),

transaction\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (from\_account) REFERENCES accounts(account\_id),

FOREIGN KEY (to\_account) REFERENCES accounts(account\_id)

);

## ****📝 Step 3: Insert Sample Data****

INSERT INTO accounts (holder\_name, balance) VALUES ('Alice', 5000);

INSERT INTO accounts (holder\_name, balance) VALUES ('Bob', 3000);

SELECT \* FROM accounts;

📌 Now we have two accounts:

+------------+-------------+---------+

| account\_id | holder\_name | balance |

+------------+-------------+---------+

| 1 | Alice | 5000.00 |

| 2 | Bob | 3000.00 |

+------------+-------------+---------+

## ****📝 Step 4: Implement a Money Transfer Transaction****

Alice wants to **transfer $1000** to Bob.  
We will use **TCL commands** to ensure that:

* If both updates succeed, COMMIT the transaction.
* If an error occurs, ROLLBACK to cancel the transaction.

START TRANSACTION;

-- Deduct $1000 from Alice's account

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

-- Add $1000 to Bob's account

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

-- Log the transaction

INSERT INTO transactions (from\_account, to\_account, amount)

VALUES (1, 2, 1000);

-- If everything is fine, save changes

COMMIT;

👉 After execution, **Alice's balance = $4000** and **Bob's balance = $4000**.

## ****📝 Step 5: Simulate a Failure with ROLLBACK****

Now, let’s assume Alice tries to transfer **$6000** to Bob, but she **doesn’t have enough balance**.  
We must **rollback** to prevent her from having a negative balance.

START TRANSACTION;

-- Deduct $6000 from Alice's account

UPDATE accounts SET balance = balance - 6000 WHERE account\_id = 1;

-- Check if Alice has enough balance before proceeding

SELECT balance FROM accounts WHERE account\_id = 1;

-- If balance is negative, rollback the transaction

ROLLBACK;

### ****After rollback, balances remain the same:****

+------------+-------------+---------+

| account\_id | holder\_name | balance |

+------------+-------------+---------+

| 1 | Alice | 4000.00 |

| 2 | Bob | 4000.00 |

+------------+-------------+---------+

## ****📝 Step 6: Using SAVEPOINT****

Let’s say Alice is making multiple transactions, but we want the ability to **rollback specific parts**.

START TRANSACTION;

-- First transfer of $500 from Alice to Bob

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

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

SAVEPOINT transfer1;

-- Second transfer of $1500 (possible issue)

UPDATE accounts SET balance = balance - 1500 WHERE account\_id = 1;

UPDATE accounts SET balance = balance + 1500 WHERE account\_id = 2;

SAVEPOINT transfer2;

-- Oops! Alice's balance is too low! Rollback only the second transfer.

ROLLBACK TO transfer1;

-- Commit only the first transfer

COMMIT;

👉 Now, **only the first transfer of $500 is successful**, and the second transfer is canceled.

## ****📌 Final Thoughts****

| **Command** | **Purpose** |
| --- | --- |
| START TRANSACTION | Begins a transaction. |
| COMMIT | Saves changes permanently. |
| ROLLBACK | Undoes all changes in a transaction. |
| SAVEPOINT | Creates a checkpoint within a transaction. |
| ROLLBACK TO SAVEPOINT | Rolls back to a specific savepoint. |

🔹 **Using TCL commands ensures that money transfers are safe, preventing data inconsistencies.**

# ****Real-World Example of TCL in MySQL (Step-by-Step)****

### ****🏪 Scenario: Inventory Management System****

A retail store manages its inventory using a **MySQL database**.

* When a customer **buys a product**, the stock should **decrease**.
* If stock **runs out** or an error occurs, the transaction **must roll back** to prevent inconsistencies.
* If everything is successful, the transaction **commits** to save changes permanently.

## ****📝 Step 1: Create a New Database****

CREATE DATABASE RetailStore;

USE RetailStore;

## ****📝 Step 2: Create Tables****

### ****Products Table****

Stores details of each product in the inventory.

CREATE TABLE products (

product\_id INT PRIMARY KEY AUTO\_INCREMENT,

product\_name VARCHAR(100),

stock\_quantity INT NOT NULL,

price DECIMAL(10,2) NOT NULL

);

### ****Orders Table****

Stores customer orders.

CREATE TABLE orders (

order\_id INT PRIMARY KEY AUTO\_INCREMENT,

product\_id INT,

quantity INT NOT NULL,

order\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

);

## ****📝 Step 3: Insert Sample Data****

INSERT INTO products (product\_name, stock\_quantity, price)

VALUES

('Laptop', 10, 75000),

('Smartphone', 20, 25000),

('Headphones', 30, 2000);

SELECT \* FROM products;

📌 Now we have some products in stock:

+------------+-------------+---------------+---------+

| product\_id | product\_name | stock\_quantity | price |

+------------+-------------+---------------+---------+

| 1 | Laptop | 10 | 75000.00 |

| 2 | Smartphone | 20 | 25000.00 |

| 3 | Headphones | 30 | 2000.00 |

+------------+-------------+---------------+---------+

## ****📝 Step 4: Implementing an Order Transaction****

A customer **buys 3 smartphones**. We must:

1. Check if stock is available.
2. Deduct from stock.
3. Insert an order record.
4. If successful, commit the transaction. If stock is insufficient, rollback.

START TRANSACTION;

-- Check stock availability

SELECT stock\_quantity FROM products WHERE product\_id = 2;

-- Reduce stock by 3

UPDATE products SET stock\_quantity = stock\_quantity - 3 WHERE product\_id = 2;

-- Insert order details

INSERT INTO orders (product\_id, quantity) VALUES (2, 3);

-- Everything is fine, save changes

COMMIT;

**After the transaction:**

+------------+-------------+---------------+

| product\_id | product\_name | stock\_quantity |

+------------+-------------+---------------+

| 1 | Laptop | 10 |

| 2 | Smartphone | 17 | <-- Updated

| 3 | Headphones | 30 |

+------------+-------------+---------------+

📌 Stock for **Smartphone** is now **17**.

## ****📝 Step 5: Handling Insufficient Stock with ROLLBACK****

Now, a customer tries to **buy 25 smartphones**, but only **17 are available**.  
To **prevent negative stock**, we must **rollback**.

START TRANSACTION;

-- Check stock before proceeding

SELECT stock\_quantity FROM products WHERE product\_id = 2;

-- Deduct 25 from stock (This will cause an issue)

UPDATE products SET stock\_quantity = stock\_quantity - 25 WHERE product\_id = 2;

-- Insert order (Will be removed if rollback occurs)

INSERT INTO orders (product\_id, quantity) VALUES (2, 25);

-- If stock goes negative, rollback

ROLLBACK;

✅ **Transaction is rolled back.** No stock is deducted, and no order is placed.

## ****📝 Step 6: Using SAVEPOINT for Partial Order Processing****

A customer buys:

* **1 Laptop**
* **10 Headphones**
* But **tries to buy 15 Smartphones**, which fails due to insufficient stock.

Instead of **rolling back everything**, we **rollback only the smartphone order** and commit the rest.

START TRANSACTION;

-- Reduce Laptop stock by 1

UPDATE products SET stock\_quantity = stock\_quantity - 1 WHERE product\_id = 1;

SAVEPOINT laptop\_order;

-- Reduce Headphones stock by 10

UPDATE products SET stock\_quantity = stock\_quantity - 10 WHERE product\_id = 3;

SAVEPOINT headphones\_order;

-- Attempt to reduce Smartphones stock by 15 (Not enough stock)

UPDATE products SET stock\_quantity = stock\_quantity - 15 WHERE product\_id = 2;

SAVEPOINT smartphone\_order;

-- If the smartphone stock is insufficient, rollback that part only

ROLLBACK TO smartphone\_order;

-- Commit successful changes (Laptop & Headphones)

COMMIT;

📌 Only **Laptop & Headphones orders** are processed, and the **Smartphone order is canceled**.

## ****📌 Final Summary****

| **Command** | **Usage** |
| --- | --- |
| START TRANSACTION | Begin a transaction. |
| COMMIT | Save changes permanently. |
| ROLLBACK | Undo all changes in a transaction. |
| SAVEPOINT | Set a checkpoint inside a transaction. |
| ROLLBACK TO SAVEPOINT | Undo changes after a savepoint. |

✅ **Ensures no negative stock, and failed orders don't affect the whole system!**

## ****🚀 Real-World Application Benefits****

✔ **Prevents data inconsistencies** (e.g., stock going negative).  
✔ **Ensures reliability** in critical systems like e-commerce & banking.  
✔ **Optimizes performance** by using partial rollbacks (SAVEPOINT).

Would you like another real-world scenario, such as **hotel booking or employee payroll**? 😊