

Chapter 18: MySQL Transactions

18.1 Introduction to Transactions

- * A transaction is a sequence of one or more SQL operations performed as a **single logical unit of work**.
- * Transactions ensure **data integrity** and **consistency**.
- * Either **all** operations in a transaction are completed successfully, or **none** of them are applied.
- * MySQL uses **ACID** properties to manage transactions.



TRANSACTIONS

ACID Properties :

- ✓ **Atomicity** - All operations are treated as a single unit.
- ✓ **Consistency** - Data remains consistent before and after the transaction.
- ✓ **Isolation** - Transactions are independent of each other.
- ✓ **Durability** - Once committed, changes are permanent.

Note :

Transactions help prevent data loss and maintain database reliability.

Example Scenario :

Transfer ₹1000 from Account A to Account B.

Operations involved:

- Subtract ₹1000 from Account A
- Add ₹1000 to Account B

These two operations must be executed together. If one fails, the other should not be applied.

Without Transactions :

Step 1
₹1000 debited from A



Step 2 (Fails)
₹1000 not credited to B

Result : Money lost



With Transactions :

Step 1
₹1000 debited from A



Step 2
₹1000 credited to B

Result : Transfer successful



Key Takeaway :



Transactions make sure that either **all** changes are saved or **none** are saved.

18.2 START TRANSACTION

- * The **START TRANSACTION** statement is used to begin a new transaction.
- * Once a transaction is started, all subsequent SQL statements are executed as part of that transaction.
- * The changes are not permanently saved until **COMMIT** is executed.



START TRANSACTION

Syntax :

```
START TRANSACTION;
```

Example :

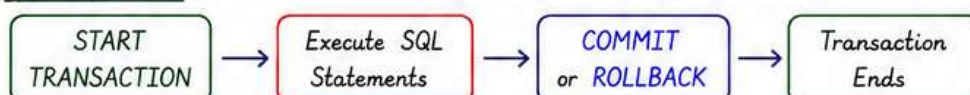
```
START TRANSACTION;  
UPDATE accounts SET balance = balance - 1000  
WHERE account_id = 1;  
UPDATE accounts SET balance = balance + 1000  
WHERE account_id = 2;
```



Note :

If an error occurs before **COMMIT**, you can **ROLLBACK** to undo changes.

How It Works :



Key Takeaway :

- ✓ Use **START TRANSACTION** to begin a new transaction.
- ✓ Make changes using SQL statements.
- ✓ Finish with **COMMIT** to save or **ROLLBACK** to undo.

18.3 COMMIT

- * **COMMIT** permanently saves all changes made in a transaction.
- * After **COMMIT**, changes become permanent and cannot be rolled back.
- * It marks the **end** of a transaction.
- * Changes are written to the database **permanently**.



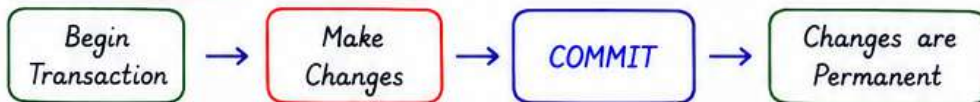
Syntax :

```
COMMIT;
```

Example :

```
START TRANSACTION;  
UPDATE employees SET salary = 60000 WHERE id = 1;  
COMMIT;
```

How It Works :



Note :
COMMIT saves changes permanently in the database.

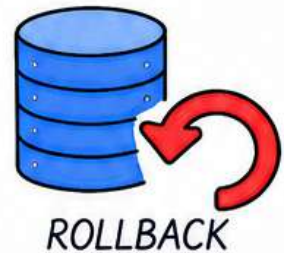
Key Takeaway :



Use **COMMIT** when you want to make all changes permanent and save them to the database.

18.4 ROLLBACK

- * **ROLLBACK** undoes all changes made in a transaction.
- * It restores the database to the state **before** the transaction started.
- * It is used when an **error** occurs or when we do not want to keep the changes.
- * Changes made after the transaction start are **discarded**.



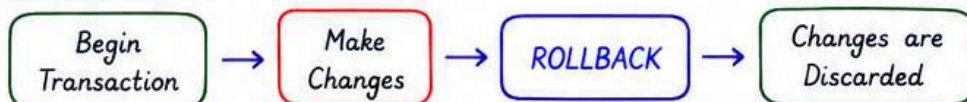
Syntax :

```
ROLLBACK;
```

Example :

```
START TRANSACTION;  
UPDATE employees SET salary = 70000 WHERE id = 2;  
ROLLBACK;
```

How It Works :



Note :
ROLLBACK undoes changes and restores the previous state.

COMMIT vs ROLLBACK :

Feature	COMMIT	ROLLBACK
Purpose	Save changes permanently	Undo changes
Effect	Changes are permanent	Changes are discarded
Use When	Transaction is successful	Transaction has an error
Result	Data is saved in DB	Data is restored

Key Takeaway :



Use **ROLLBACK** when you want to undo the changes and keep the database in the previous state.



18.5 SAVEPOINT



SAVEPOINT

- * **SAVEPOINT** is used to set a point within a transaction.
- * It allows you to **divide** a transaction into smaller parts.
- * You can **rollback** to a savepoint if needed.
- * It **does not** end the transaction.

Example :

Syntax :

SAVEPOINT savepoint_name;

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 1000
WHERE account_id = 1;
SAVEPOINT after_debit;
UPDATE accounts SET balance = balance + 1000
WHERE account_id = 2;
COMMIT;
```

How **SAVEPOINT** Works :



Example Scenario :

Transfer ₹1000 from Account A to Account B in two steps.

Steps :

1. ₹1000 debited from A
2. **SAVEPOINT** after_debit created
3. ₹1000 credited to B
4. **COMMIT** executed

If any error occurs after step 2, we can rollback to the savepoint.

Result After **COMMIT** :

account_id	balance
1 (Account A)	2000
2 (Account B)	6000

Note :

SAVEPOINT sets a checkpoint. It helps in partial rollback without ending the transaction.

Key Takeaway :



- ✓ **SAVEPOINT** creates a checkpoint within a transaction.
- ✓ It is useful for handling errors in complex transactions.
- ✓ The transaction continues until **COMMIT** or **ROLLBACK**.



18.6 ROLLBACK TO SAVEPOINT

- * **ROLLBACK TO SAVEPOINT** is used to **undo** changes made after a specific savepoint.
- * It **does not** cancel the entire transaction.
- * You can **continue** the transaction after rolling back to the savepoint.



ROLLBACK TO SAVEPOINT

Example :

Syntax :

ROLLBACK TO SAVEPOINT savepoint_name;

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 1000
WHERE account_id = 1;
SAVEPOINT after_debit;
UPDATE accounts SET balance = balance + 1000
WHERE account_id = 2;
ROLLBACK TO SAVEPOINT after_debit;
COMMIT;
```

How It Works :



Example Scenario :

Transfer ₹1000 from Account A to Account B.

Steps :

1. ₹1000 debited from A
2. **SAVEPOINT** after_debit created
3. ₹1000 credited to B
4. **ROLLBACK TO SAVEPOINT** after_debit
5. **COMMIT** executed

After rollback, credit to B is undone, but debit from A remains.

Result After **ROLLBACK TO SAVEPOINT** and **COMMIT** :

account_id	balance
1 (Account A)	2000
2 (Account B)	5000

Note :

Only changes made after the savepoint are undone.



Key Takeaway :

- ✓ **ROLLBACK TO SAVEPOINT** undoes changes made after the savepoint.
- ✓ The transaction remains active after rolling back to a savepoint.
- ✓ You can set multiple savepoints and rollback to any of them.



18.7 RELEASE SAVEPOINT



RELEASE SAVEPOINT

- * **RELEASE SAVEPOINT** is used to remove a savepoint that is no longer needed.
- * Once a savepoint is released, you cannot rollback to it.
- * It **does not** end the transaction.
- * It is **optional** and helps in managing savepoints.

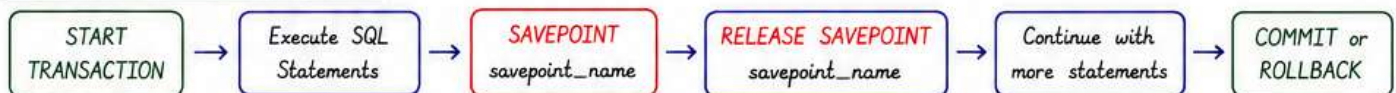
Example :

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 1000
WHERE account_id = 1;
SAVEPOINT after_debit;
UPDATE accounts SET balance = balance + 1000
WHERE account_id = 2;
RELEASE SAVEPOINT after_debit;
COMMIT;
```

Syntax :

```
RELEASE SAVEPOINT savepoint_name;
```

How RELEASE SAVEPOINT Works :



Example Scenario :

Transfer ₹1000 from Account A to Account B in two steps.

Steps :

1. ₹1000 debited from A
2. **SAVEPOINT** after_debit created
3. ₹1000 credited to B
4. **RELEASE SAVEPOINT** after_debit
5. **COMMIT** executed

After step 4, the savepoint is removed. We cannot rollback to after_debit again.

Result After COMMIT :

account_id	balance
1 (Account A)	2000
2 (Account B)	6000

Note :

RELEASE SAVEPOINT deletes the savepoint name from the transaction.



Key Takeaway :

- ✓ Use **RELEASE SAVEPOINT** to remove a savepoint when it is no longer required.
- ✓ Once released, the savepoint cannot be used for rollback.
- ✓ The transaction continues normally after releasing a savepoint.



18.8 AUTOCOMMIT



AUTOCOMMIT

- * **AUTOCOMMIT** is a mode in MySQL where each SQL statement is **committed automatically**.
- * When **AUTOCOMMIT** is **ON**, every statement is committed as soon as it is executed.
- * When **AUTOCOMMIT** is **OFF**, you must use **COMMIT** or **ROLLBACK** manually.
- * **AUTOCOMMIT** is **ON** by default in MySQL.

Syntax :

```
SET AUTOCOMMIT = {0 | 1};
-- 0 = OFF, 1 = ON
```

Example :

```
SET AUTOCOMMIT = 0; -- Turn AUTOCOMMIT OFF
SET AUTOCOMMIT = 1; -- Turn AUTOCOMMIT ON
```

Note :

Always ensure **AUTOCOMMIT** is **ON** in applications unless transactions are used intentionally.

How AUTOCOMMIT Works :

AUTOCOMMIT Mode	Behavior	Example
ON (1)	Each SQL statement is committed automatically .	INSERT, UPDATE, DELETE are committed immediately.
OFF (0)	Changes are not committed automatically. You must use COMMIT or ROLLBACK .	Multiple statements can be grouped in a transaction.

Example Scenario :

1. UPDATE accounts SET balance = balance - 500 WHERE account_id = 1;
2. UPDATE accounts SET balance = balance + 500 WHERE account_id = 2;

Result Comparison :

Mode	What Happens ?
AUTOCOMMIT = 1 (ON)	Each UPDATE is committed immediately .
AUTOCOMMIT = 0 (OFF)	Both UPDATES are pending until you execute COMMIT or ROLLBACK .



Key Takeaway :

- ✓ **AUTOCOMMIT = 1 (ON)** → Each statement is committed automatically.
- ✓ **AUTOCOMMIT = 0 (OFF)** → You control when changes are committed.
- ✓ Use OFF mode when you need transactions and want to group multiple statements.
- ✓ Always switch back to **AUTOCOMMIT = 1 (ON)** if you are not using transactions.

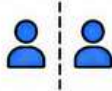


18.9 ACID Properties

- * **ACID** properties ensure reliable processing of transactions in a database system.
- * MySQL follows **ACID** properties to maintain data integrity.



ACID PROPERTIES

Property	Description
A tomicity 	* All operations in a transaction are treated as a single unit. * Either all operations are completed successfully, or none of them are applied.
C onsistency 	* A transaction brings the database from one valid state to another valid state. * It preserves all integrity constraints.
I solation 	* Transactions are executed independently of each other. * Intermediate results of a transaction are not visible to other transactions.
D urability 	* Once a transaction is committed, its changes are permanent and survive system failures. * Data will not be lost even if the system crashes.

Note :

ACID properties work together to ensure that transactions are processed reliably and data remains safe and accurate.

Key Takeaway :



- ✓ ACID properties are essential for processing **reliable** transactions.
- ✓ They maintain data **integrity** and **consistency** in the database.
- ✓ MySQL uses ACID to handle **concurrent** access and **failures** safely.



18.10 Transaction Isolation

- * Isolation ensures that transactions **do not interfere** with each other.
- * It controls how and when changes made by one transaction are visible to other transactions.
- * MySQL provides different isolation levels to handle **concurrency**.



TRANSACTION ISOLATION

Isolation Levels in MySQL :

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read	Description
READ UNCOMMITTED	Possible	Possible	Possible	Lowest isolation level. Allows reading uncommitted changes.
READ COMMITTED	Not Possible	Possible	Possible	Reads only committed data. Non-repeatable reads can occur.
REPEATABLE READ	Not Possible	Not Possible	Possible	Prevents non-repeatable reads. Phantom reads can occur.
SERIALIZABLE	Not Possible	Not Possible	Not Possible	Highest isolation level. Prevents all concurrency issues.

Example Scenario :

Transaction T1 updates a row but does not commit.
Transaction T2 tries to read the same row.
Result depends on the isolation level.

- **READ UNCOMMITTED** → T2 can read uncommitted data.
- **READ COMMITTED** → T2 cannot read uncommitted data.
- **REPEATABLE READ** → T2 sees a consistent snapshot.
- **SERIALIZABLE** → T2 waits until T1 completes.

Setting Isolation Level :

Syntax :

```
SET TRANSACTION ISOLATION LEVEL isolation_level;
```

Example :

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Key Takeaway :



- ✓ Isolation levels control how transactions interact with each other.
- ✓ Higher isolation provides more **consistency** but may reduce **performance**.
- ✓ Choose the appropriate isolation level based on your application needs.

