

Delete vs truncate vs drop

In the context of databases, especially in SQL, **DELETE**, **TRUNCATE**, and **DROP** are commands used to remove data or database objects, but they serve distinct purposes:

1. DELETE:

- Used to remove specific rows from a table based on a condition.
- It's a **DML (Data Manipulation Language)** command.
- The table structure and data remain, but the specified rows are deleted.
- It supports a **WHERE** clause to specify conditions.
- Example: DELETE FROM table_name WHERE condition;
- Can be rolled back if wrapped in a transaction.

EX:

```
-- Step 1: Start a transaction
START TRANSACTION;

-- Step 2: Delete a record
DELETE FROM EmployeeDetail WHERE EmployeeID = 1;

-- Step 3: Try to fetch the deleted record (It should return 0 rows)
SELECT * FROM EmployeeDetail WHERE EmployeeID = 1;

-- Step 4: Rollback the transaction (undo the delete)
ROLLBACK;

-- Step 5: Check if the record is restored (It should return the original row)
SELECT * FROM EmployeeDetail WHERE EmployeeID = 1;
```

```

35 • START TRANSACTION ;
36 • delete from EmployeeDetail where EmployeeID=2;
37 • SELECT * FROM EmployeeDetail WHERE EmployeeID = 2;
38
39
40

```

Result Grid

Filter Rows:

Edit:

Export/Import:

Wrap Cell Content:

	EmployeeID	FirstName	LastName	Salary	JoiningDate	Department	Gender
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

```

35 • START TRANSACTION ;
36 • delete from EmployeeDetail where EmployeeID=2;
37 • SELECT * FROM EmployeeDetail WHERE EmployeeID = 2;
38 • rollback;
39 • SELECT * FROM EmployeeDetail WHERE EmployeeID = 2;
40

```

Result Grid

Filter Rows:

Edit:

Export/Import:

Wrap Cell Content:

	EmployeeID	FirstName	LastName	Salary	JoiningDate	Department	Gender
▶	2	Nikita	Jain	530000.00	2014-01-09 17:31:08	HR	Female

✓ **ROLLBACK** only works if your table uses the **InnoDB** engine

- ♦ Run this to check your storage engine:

```

sql
SHOW TABLE STATUS WHERE Name = 'EmployeeDetail';

```

- ♦ If the engine is **MyISAM**, change it to **InnoDB**:

```

sql
ALTER TABLE EmployeeDetail ENGINE=InnoDB;

```

2. TRUNCATE:

- Removes **all rows** from a table, effectively clearing it.
- It's a **DDL (Data Definition Language)** command.
- It's faster than DELETE since it doesn't log individual row deletions.
- It resets any auto-increment counters.
- Cannot include a **WHERE** clause.
- Example: TRUNCATE TABLE table_name;

- Cannot be rolled back in most databases.

Testing TRUNCATE Behavior

Unlike DELETE, TRUNCATE cannot be rolled back even if used inside a transaction.

Example:

```
sql                                                                    Copy Edit

START TRANSACTION;

TRUNCATE TABLE EmployeeDetail;

ROLLBACK; -- This won't restore data

SELECT * FROM EmployeeDetail; -- No records will be restored!
```

43 • `SELECT * FROM EmployeeDetail;`
44
45
46

Result Grid

Filter Rows:

Edit:

Export/Import:

Wrap Cell Content:

	EmployeeID	FirstName	LastName	Salary	JoiningDate	Department	Gender
▶	2	Nikita	Jain	530000.00	2014-01-09 17:31:08	HR	Female
	3	Ashish	Kumar	1000000.00	2014-01-09 10:05:08	IT	Male
	4	Nikhil	Sharma	480000.00	2014-01-09 09:00:08	HR	Male
	5	Anish	Kadian	500000.00	2014-01-09 09:31:08	Payroll	Male
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

```

44 • Start transaction;
45 • truncate EmployeeDetail;
46 • SELECT * FROM EmployeeDetail;

```

EmployeeID	FirstName	LastName	Salary	JoiningDate	Department	Gender
NULL	NULL	NULL	NULL	NULL	NULL	NULL

```

47 • rollback;
48 • SELECT * FROM EmployeeDetail;

```

EmployeeID	FirstName	LastName	Salary	JoiningDate	Department	Gender
NULL	NULL	NULL	NULL	NULL	NULL	NULL

DELETE is **safer** and supports rollback if using InnoDB.

TRUNCATE is **faster** but **permanent**—it cannot be undone.

3. DROP:

- Completely removes a table or database from the system.
- It's a **DDL command** and destroys both the data and the table structure.
- Example: DROP TABLE table_name; or DROP DATABASE database_name;
- Irreversible; once executed, the data and structure are gone

```

50 • DROP TABLE EmployeeDetail;
51 • SELECT * FROM EmployeeDetail;
52

```

#	Time	Action	Message	Du
16	23:25:23	truncate EmployeeDetail	0 row(s) affected	0.0
17	23:25:35	SELECT * FROM EmployeeDetail LIMIT 0, 500	0 row(s) returned	0.0
18	23:26:12	rollback	0 row(s) affected	0.0
19	23:26:26	SELECT * FROM EmployeeDetail LIMIT 0, 500	0 row(s) returned	0.0
20	23:29:04	DROP TABLE EmployeeDetail	0 row(s) affected	0.0
21	23:29:15	SELECT * FROM EmployeeDetail LIMIT 0, 500	Error Code: 1146. Table 'employee.db.employeeDetail' doesn't exist	0.0

Difference Between DELETE , TRUNCATE , and DROP in MySQL

Feature	DELETE	TRUNCATE	DROP
Can be rolled back?	✔ Yes (if InnoDB)	✘ No	✘ No
Removes specific rows?	✔ Yes (with WHERE)	✘ No (removes all rows)	✘ No (removes entire table)
Resets AUTO_INCREMENT ?	✘ No	✔ Yes	✘ Not applicable (table is deleted)
Affects table structure?	✘ No	✘ No	✔ Yes (removes table)
Faster?	✘ Slow (row-by-row delete)	✔ Faster (drops and recreates table)	✔ Fastest (removes entire table)
Affects triggers?	✔ Yes	✘ No	✘ No
Locks the table?	✔ Yes (row-level lock in InnoDB)	✔ Yes (table-level lock)	✔ Yes (locks table before dropping)

ROLLBACK and COMMIT

What are COMMIT and ROLLBACK ?

COMMIT and ROLLBACK are transaction control commands used in MySQL to manage changes in a database.


Command	What it does?
START TRANSACTION;	Begins a transaction (keeps changes temporary)
COMMIT;	Saves all changes permanently
ROLLBACK;	Undoes all changes made since the transaction started

1 COMMIT – Save Changes Permanently

- When you use `COMMIT`, all the operations inside the transaction become **permanent**.
- You **cannot undo** a committed transaction.

Example:

```
sql Copy Edit  
  
START TRANSACTION; -- Start transaction  
  
UPDATE EmployeeDetail SET Salary = 700000 WHERE EmployeeID = 1; -- Update salary  
  
COMMIT; -- Save changes permanently
```


 **Effect:** The salary is permanently updated.

2 ROLLBACK – Undo Changes

- If you use `ROLLBACK`, all changes made after `START TRANSACTION` will be **undone**.
- The database **returns to its previous state**.

Example:

```
sql Copy Edit  
  
START TRANSACTION; -- Start transaction  
  
DELETE FROM EmployeeDetail WHERE EmployeeID = 1; -- Delete an employee  
  
ROLLBACK; -- Undo the delete operation
```

 **Effect:** The deleted employee is **restored** because `ROLLBACK` was used.

What Happens If You Use `ROLLBACK` After `COMMIT` ?

Once you use `COMMIT` , all changes are **permanently saved** in the database.

🚫 You **CANNOT** undo a committed transaction using `ROLLBACK` .

Example Scenario

```
sql                                                                    Copy Edit

START TRANSACTION;  -- Begin transaction

DELETE FROM EmployeeDetail WHERE EmployeeID = 1;  -- Delete an employee

COMMIT;  -- Save changes permanently

ROLLBACK;  -- Try to undo (WON'T WORK)
```

🚀 Effect:

- The employee record is **permanently deleted** after `COMMIT` .
- `ROLLBACK` **does nothing** because there's nothing left to undo.

Once `COMMIT` is Executed, Can You Change the Data?

❌ No, you cannot use `ROLLBACK` after `COMMIT` .

✅ But you can manually insert/update the data again.

Understanding COMMIT Finality

When you run `COMMIT`, all the changes are **permanently saved** in the database.

- You **cannot undo** them with `ROLLBACK`.
 - The only way to restore data is **manual reinsertion** or **database backup recovery**.
-

Example Scenario

sql

Copy

Edit

```
START TRANSACTION;
DELETE FROM EmployeeDetail WHERE EmployeeID = 1; -- Delete an employee

COMMIT; -- Save changes permanently

ROLLBACK; -- This will NOT restore the deleted record!
```

 **Effect:**

- The employee record is **gone permanently** after `COMMIT`.
- `ROLLBACK` **does nothing** since there's nothing left to undo.

How to Fix Data After COMMIT ?

1 Manually Insert the Data Again

If data is lost due to `COMMIT`, you can reinsert it manually:

```
sql                                                                    Copy Edit

INSERT INTO EmployeeDetail (EmployeeID, FirstName, LastName, Salary, JoiningDate, Department,
VALUES (1, 'Vikas', 'Ahlawat', 600000.00, '2013-02-15 11:16:28', 'IT', 'Male');
```

2 Restore from a Backup (If Available)

If you have database backups, you can **restore data** from a backup file.

```
sql                                                                    Copy Edit

mysql -u root -p EmployeeDB < backup.sql
```

Effect:

- This restores the **entire database** to a previous state.

Storage Engine Used by Your MySQL Database

A storage engine in MySQL is the software component that manages how data is stored, retrieved, and manipulated within tables. Different storage engines offer different features like transaction support, performance optimization, and indexing strategies.

```
52
53 • SHOW ENGINES;
54
```

Engine	Support	Comment	Transactions	XA	Savepoints
MEMORY	YES	Hash based, stored in memory, useful for temp...	NO	NO	NO
MRG_MYISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
FEDERATED	NO	Federated MySQL storage engine	NULL	NULL	NULL
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO
MyISAM	YES	MyISAM storage engine	NO	NO	NO
InnoDB	DEFAULT	Supports transactions, row-level locking, and fo...	YES	YES	YES
BLACKHOLE	YES	/dev/null storage engine (anything you write to ...	NO	NO	NO
ARCHIVE	YES	Archive storage engine	NO	NO	NO

AUTO COMMIT Mode

- MySQL by default **auto-commits** every SQL statement.
- To disable auto-commit, use:

sql

 Copy

 Edit

```
SET AUTOCOMMIT = 0;
```

- To enable it again:

sql


 Copy

 Edit

```
SET AUTOCOMMIT = 1;
```

Basic MySQL Questions


❶ What is MySQL?

 MySQL is an **open-source relational database management system (RDBMS)** based on **SQL (Structured Query Language)**.

❷ What are the different MySQL storage engines?

- ✓ **InnoDB** → Supports transactions, ACID compliance, foreign keys. (Default)
- ✓ **MyISAM** → Fast read-heavy operations, but no transactions.
- ✓ **Memory** → Stores data in RAM, super-fast but volatile.
- ✓ **CSV** → Stores data in plain-text format (CSV files).
- ✓ **Archive** → Compresses data for logging and historical storage.

❸ What is the difference between MySQL and SQL?

 **SQL (Structured Query Language)** is a **language**, while **MySQL** is a **database management system (DBMS)** that uses SQL to interact with data

MySQL Joins Explained with Examples

In MySQL, **joins** are used to retrieve data from multiple tables based on a related column. There are four main types of joins:

- ① **INNER JOIN** → Returns only matching rows.
- ② **LEFT JOIN** → Returns all rows from the left table and matching rows from the right.
- ③ **RIGHT JOIN** → Returns all rows from the right table and matching rows from the left.
- ④ **FULL OUTER JOIN** → Returns all rows from both tables (not supported in MySQL directly).

```
1 create database jointsexample;
2 CREATE TABLE Employees (
3     EmployeeID INT PRIMARY KEY,
4     Name VARCHAR(50),
5     DepartmentID INT
6 );
7
8 CREATE TABLE Departments (
9     DepartmentID INT PRIMARY KEY,
10    DepartmentName VARCHAR(50)
11 );
12
13 INSERT INTO Employees (EmployeeID, Name, DepartmentID) VALUES
14 (1, 'Alice', 101),
15 (2, 'Bob', 102),
16 (3, 'Charlie', NULL),
17 (4, 'David', 103);
18
19 INSERT INTO Departments (DepartmentID, DepartmentName) VALUES
20 (101, 'IT'),
21 (102, 'HR'),
22 (104, 'Finance');
23
24
25 select * from Employees;
26 select * from Departments;
27
```

Tables Before Join:

24

25 • `select * from Employees;`

	EmployeeID	Name	DepartmentID
▶	1	Alice	101
	2	Bob	102
	3	Charlie	NULL
	4	David	103
	NULL	NULL	NULL

26 • `select * from Departments;`

Result Grid | Filter Rows: | Edit:

	DepartmentID	DepartmentName
▶	101	IT
	102	HR
	104	Finance
✱	NULL	NULL

INNER JOIN (Matching Rows Only)

✓ Returns only records that have a matching DepartmentID in both tables.

29 • `SELECT Employees.Name, Departments.DepartmentName`
 30 `FROM Employees`
 31 `INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;`
 32

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

	Name	DepartmentName
▶	Alice	IT
	Bob	HR

Explanation:

- Only Alice and Bob are displayed because they have matching **DepartmentID** in both tables.
- Charlie (NULL) and David (103) are **excluded** because there's no matching department.

LEFT JOIN (All from Left, Matching from Right)

✓ Returns all records from the Employees table and matches from Departments.

```

35 • SELECT Employees.Name, Departments.DepartmentName
36 FROM Employees
37 LEFT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
38
39
40

```

Name	DepartmentName
Alice	IT
Bob	HR
Charlie	NULL
David	NULL

Explanation:

- Alice & Bob match and show department names.
- Charlie has **NULL** because they have **no department assigned**.
- David's department (103) **doesn't exist** in the **Departments** table, so it's also **NULL**.

RIGHT JOIN (All from Right, Matching from Left)

✓ Returns all records from the Departments table and matches from Employees.

```

41 • SELECT Employees.Name, Departments.DepartmentName
42 FROM Employees
43 RIGHT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
44
45
46

```

Name	DepartmentName
Alice	IT
Bob	HR
NULL	Finance

FULL OUTER JOIN (All Rows from Both Tables)

✓ Returns all records from both tables. If no match, fills with NULL.

⚠ MySQL does NOT support **FULL OUTER JOIN** directly. You can **simulate it** using **UNION**.

```
46 • SELECT Employees.Name, Departments.DepartmentName
47 FROM Employees
48 LEFT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID
49 UNION
50 SELECT Employees.Name, Departments.DepartmentName
51 FROM Employees
52 RIGHT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	Name	DepartmentName
▶	Alice	IT
	Bob	HR
	Charlie	NULL
	David	NULL
	NULL	Finance

What is an Index in MySQL?

An **index** in MySQL is a **data structure** that improves the speed of data retrieval operations on a database table **at the cost of additional storage** and **slower writes** (INSERT, UPDATE, DELETE).

Think of an **index** like the **table of contents in a book**—instead of searching page-by-page, you can jump directly to the relevant section.

Why is Indexing Used?

Indexes help optimize queries by:

- ✓ Speeding up SELECT queries
- ✓ Reducing the number of scanned rows
- ✓ Enabling efficient sorting and filtering
- ✓ Improving JOIN performance

⚠ But be careful:

- ✗ Indexes use extra storage
- ✗ Too many indexes slow down INSERT/UPDATE/DELETE operations

Types of Indexes in MySQL

PRIMARY INDEX (Clustered Index)

- **Automatically created** on **PRIMARY KEY** columns.
- Ensures **uniqueness** of each row.
- **One per table.**

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY, -- PRIMARY INDEX on EmployeeID
    Name VARCHAR(50),
    Department VARCHAR(50)
);
```

54
55 • `SHOW INDEX FROM Employees;`
56

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Express
employees	0	PRIMARY	1	EmployeeID	A	4				BTREE			YES	

UNIQUE INDEX

- Ensures values in a column are **unique**.
- Unlike **PRIMARY KEY**, a table **can have multiple UNIQUE indexes**.

```
CREATE UNIQUE INDEX idx_employee_name ON Employees(Name);
```

OR

```
ALTER TABLE Employees ADD UNIQUE (Name);
```

EX;

Using select query

```
57 • SELECT * FROM Employees WHERE EmployeeID = 1;
58
```

Result Grid			
	EmployeeID	Name	DepartmentID
▶	1	Alice	101
•	NULL	NULL	NULL

Query Optimization Strategies in MySQL

Ex:

	EmployeeID	Name	DepartmentID
▶	1	Alice	101
	2	Bob	102
	3	Charlie	NULL
	4	David	103
•	NULL	NULL	NULL

EXPLAIN helps analyze query performance before execution.

It shows **how MySQL processes the query**, including:

- ✓ Whether an **index** is used
- ✓ **Number of rows scanned**
- ✓ Query execution plan

```
59 • EXPLAIN SELECT * FROM Employees WHERE Name = 'Alice';
60
61
62
```

Result Grid											
	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
▶	1	SIMPLE	Employees	NULL	ALL	NULL	NULL	NULL	NULL	4	25.00

Extra Using where

What this means:

- type = **ALL** → Full table scan (❌ No index used → slow).
- key = **NULL under possible_keys** → No index exists on **Name**.
- **4 rows** → MySQL scans all 4 rows.

Optimizing the Query

✅ Create an index on **Name** to speed up search:

```
61 • CREATE INDEX idx_name ON Employees(Name);
62 • SHOW INDEX FROM Employees;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Express
employees	0	PRIMARY	1	EmployeeID	A	4				BTREE			YES	
employees	1	idx_emp	1	EmployeeID	A	4				BTREE			YES	
employees	1	idx_name	1	Name	A	4			YES	BTREE			YES	

Running same query again

```
63
64 • EXPLAIN SELECT * FROM Employees WHERE Name = 'Alice';
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	Employees		ref	idx_name	idx_name	203	const	1	100.00	

What changed?

- type = **ref** instead of **ALL** → MySQL uses the index! ✅
- key = **idx_name under key** → The index is applied.
- **1 row** instead of **4** → Only 1 row scanned. ✅

```

64 • EXPLAIN SELECT * FROM Employees WHERE Name = 'Alice';
65
66 • INSERT INTO Employees (EmployeeID, Name, DepartmentID) VALUES
67 (5, 'Alice', 101)

```

Result Grid
Filter Rows:
Export:
Wrap Cell Content:

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	Employees	HULL	ref	idx_name	idx_name	203	const	2	100.00	HULL

Overview
Way1

Optimized Query for Fetching Data

1 Without Optimization (Slow for Large Tables)

SELECT * FROM Employees WHERE Name = 'Alice';

👉 If **Name** is not indexed, MySQL performs a **full table scan** (slower).

2 Optimized Query (Using Index)

CREATE INDEX idx_name ON Employees(Name);
SELECT * FROM Employees WHERE Name = 'Alice';

👉 If **idx_name** exists, MySQL uses the **index** and fetches results **much faster**.

♦ How to Check If MySQL Uses the Index?

EXPLAIN SELECT * FROM Employees WHERE Name = 'Alice';

👉 If MySQL uses the **key = idx_name**, then **the index is working**, and your query is optimized.

Using **LIMIT** to Reduce Scanning

❌ Fetching All Data (Slow)

```
SELECT * FROM Employees WHERE Department = 'IT';
```

👉 This fetches **all matching rows**, which can be slow.

✅ Fetching Only Needed Rows

```
SELECT * FROM Employees WHERE Department = 'IT' LIMIT 10;
```

👉 **Faster** because MySQL stops searching after **finding 10 rows**.

Avoid **SELECT ***, Fetch Only Required Columns

❌ Fetching All Columns (Slow)

```
SELECT * FROM Employees WHERE EmployeeID = 5;
```

👉 **Unnecessary data** is retrieved, increasing query time.

✅ Fetching Only Required Columns

```
SELECT EmployeeID, Name FROM Employees WHERE EmployeeID = 5;
```

👉 **Faster** because only needed data is retrieved.

Data types

sql

Copy

Edit

```
CREATE TABLE Employees (  
    EmployeeID INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
    FirstName VARCHAR(50) NOT NULL,  
    LastName VARCHAR(50) NOT NULL,  
    Email VARCHAR(100) UNIQUE NOT NULL,  
    PhoneNumber VARCHAR(15),  
    Salary DECIMAL(10,2) NOT NULL,  
    Department ENUM('IT', 'HR', 'Finance', 'Sales') NOT NULL,  
    HireDate DATE NOT NULL,  
    LastLogin DATETIME DEFAULT CURRENT_TIMESTAMP,  
    IsActive TINYINT(1) DEFAULT 1  
);
```

Why These Choices?

1. `INT UNSIGNED AUTO_INCREMENT` for IDs → Saves space, avoids negative values.
2. `VARCHAR` instead of `CHAR` → Saves space for variable-length data.
3. `DECIMAL(10,2)` instead of `FLOAT` → Prevents rounding issues in salaries.
4. `ENUM` for Department → Reduces storage instead of using `VARCHAR`.
5. `TINYINT(1)` for Boolean fields → Efficient storage instead of `BOOLEAN`.