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.
- o 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 θ rows)

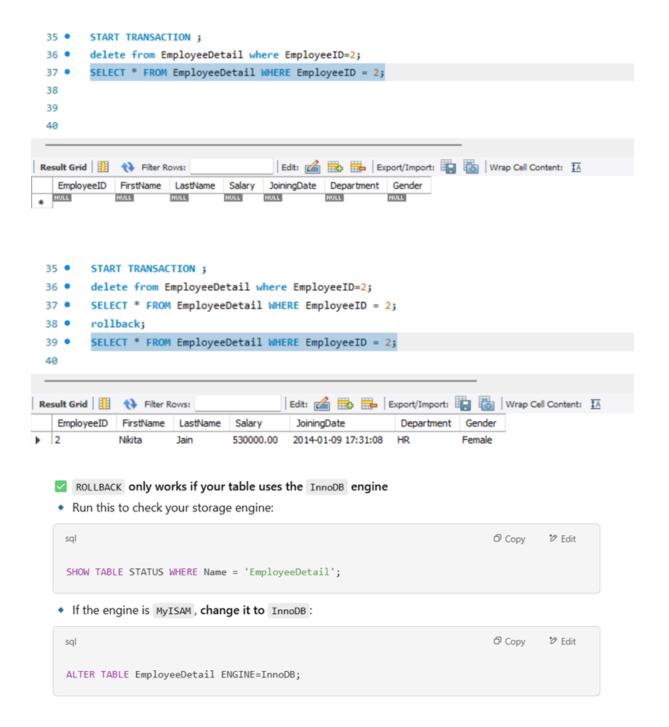
SELECT * FROM EmployeeDetail WHERE EmployeeID = 1;

-- 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;
```



2. TRUNCATE:

- Removes all rows from a table, effectively clearing it.
- It's a **DDL** (**Data Definition Language**) command.
- o 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;

o Cannot be rolled back in most databases.

Testing TRUNCATE Behavior

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

Example:

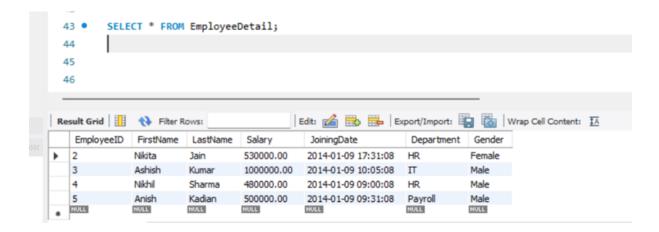
```
sql

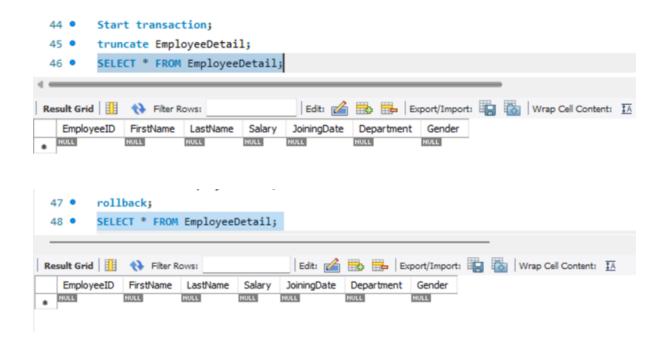
START TRANSACTION;

TRUNCATE TABLE EmployeeDetail;

ROLLBACK; -- This won't restore data

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





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;
- o Irreversible; once executed, the data and structure are gone



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)	X No (removes all rows)	X No (removes entire table)
Resets AUTO_INCREMENT ?	X No	✓ Yes	X Not applicable (table is deleted)
Affects table structure?	X No	× No	Yes (removes table)
Faster?	X 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:

```
START TRANSACTION; -- Start transaction

UPDATE EmployeeDetail SET Salary = 700000 WHERE EmployeeID = 1; -- Update salary

COMMIT; -- Save changes permanently
```

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

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

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?

X No, you cannot use ROLLBACK after COMMIT.

W 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

```
START TRANSACTION;

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

COMMIT; -- Save changes permanently

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

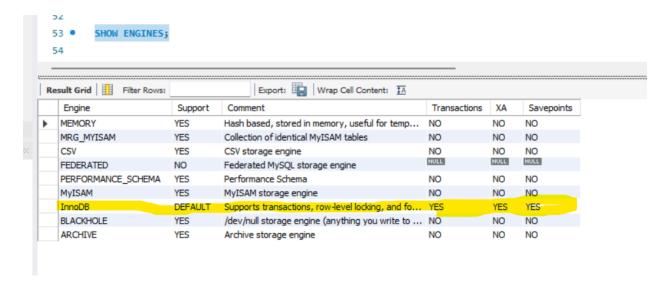
Effect:

- The employee record is gone permanently after COMMIT.
- ROLLBACK does nothing since there's nothing $| f^{r_k} |$ to undo.

Storage Engine Used by Your MySQL Database

• This restores the entire database to a previous state.

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.



- 4 AUTO COMMIT Mode
- MySQL by default auto-commits every SQL statement.
- To disable auto-commit, use:

```
sql

SET AUTOCOMMIT = 0;

• To enable it again:

sql

SET AUTOCOMMIT = 1;
```

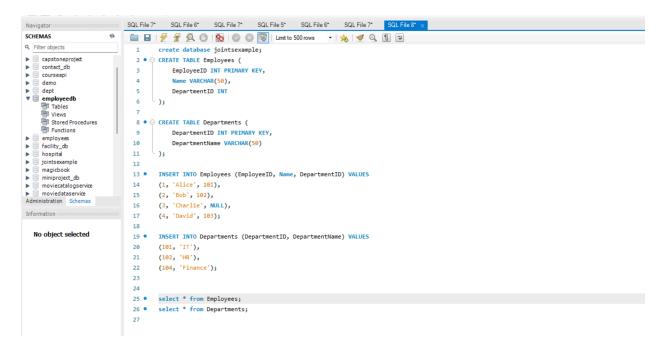
Basic MySQL Questions

- **What is MySQL?**
- MySQL is an open-source relational database management system (RDBMS) based on SQL (Structured Query Language).
- 2 What are the different MySQL storage engines?
 - **InnoDB** → Supports transactions, ACID compliance, foreign keys. (Default)
 - \mathbf{W} MyISAM \rightarrow Fast read-heavy operations, but no transactions.
 - \bigvee Memory \rightarrow Stores data in RAM, super-fast but volatile.
 - \overline{V} CSV \rightarrow Stores data in plain-text format (CSV files).
 - \bigvee Archive \rightarrow Compresses data for logging and historical storage.
- 3 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

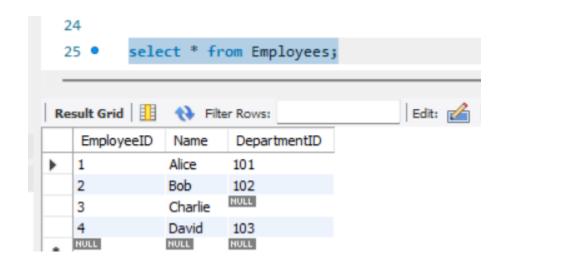
MySQL Joins Explained with Examples

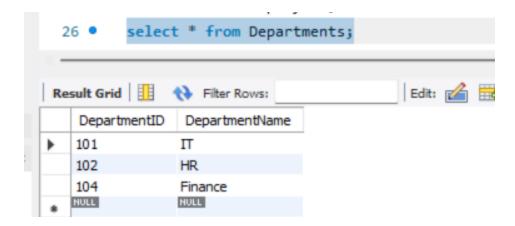
In MySQL, **joins** are used to retrieve data from multiple tables based on a related column. There are four main types of joins:

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



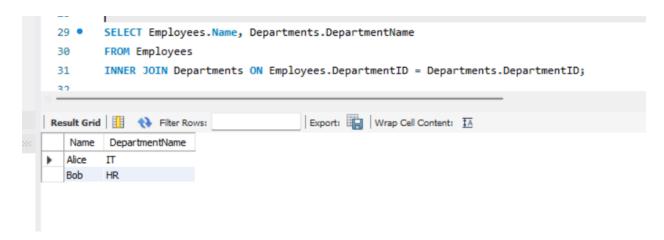
Tables Before Join:





INNER JOIN (Matching Rows Only)

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

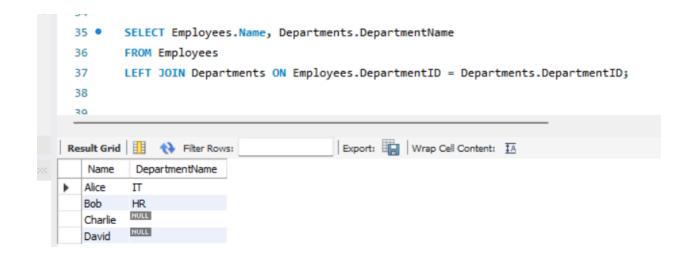


Explanation:

- 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)

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

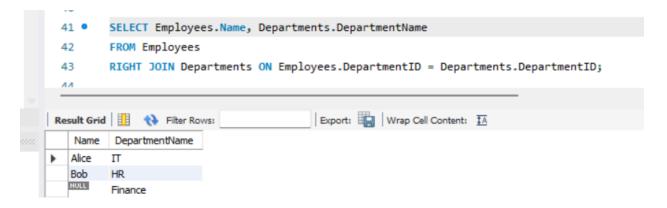


Explanation:

- 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)

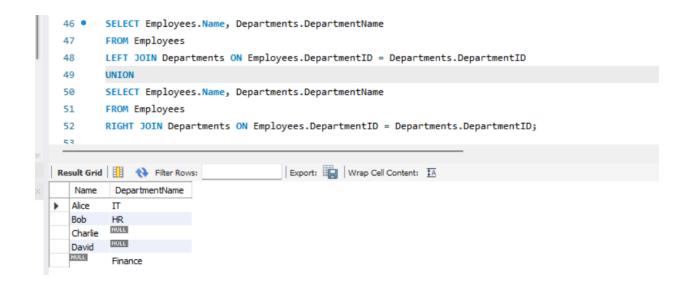
W Returns all records from the Departments table and matches from Employees.



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.



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
- **W** Reducing the number of scanned rows
- **Enabling efficient sorting and filtering**
- **✓** Improving JOIN performance

A But be careful:

X Indexes use extra storage

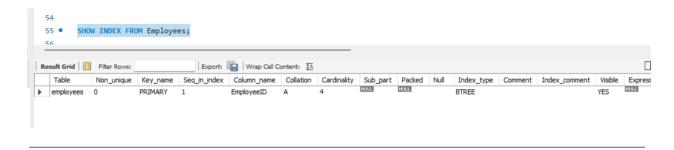
X Too many indexes slow down INSERT/UPDATE/DELETE operations

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)
);
```



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);

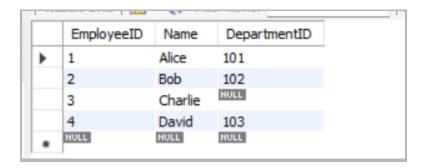
EX;

Using select query



Query Optimization Strategies in MySQL

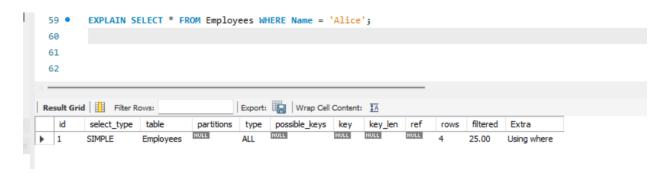
Ex:



EXPLAIN helps analyze query performance before execution.

It shows how MySQL processes the query, including:

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

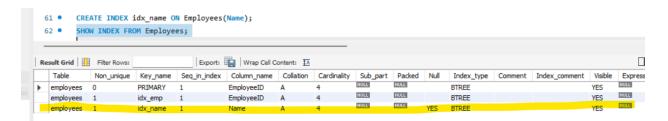


What this means:

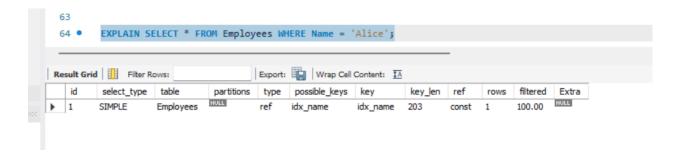
- type = $ALL \rightarrow Full table scan (X No index used \rightarrow slow).$
- key = NULL under possible keys \rightarrow No index exists on Name.
- 4 rows → MySQL scans all 4 rows.

Optimizing the Query

Create an index on Name to speed up search:

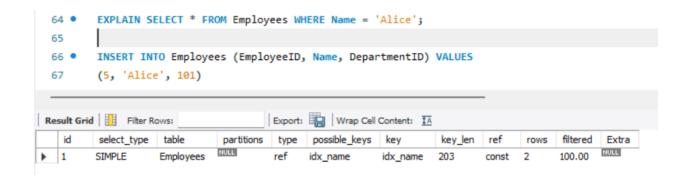


Running same query again



What changed?

- type = ref instead of ALL \rightarrow MySQL uses the index! \checkmark
- $key = idx_name under key \rightarrow The index is applied.$
- 1 row instead of $4 \rightarrow$ Only 1 row scanned.



Overview

Way1

Optimized Query for Fetching Data

Without Optimization (Slow for Large Tables)

SELECT * FROM Employees WHERE Name = 'Alice';

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

2Optimized Query (Using Index)

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

• How to Check If MySQL Uses the Index?

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

Using LIMIT to Reduce Scanning

X Fetching All Data (Slow)

SELECT * FROM Employees WHERE Department = 'IT';

- **t** This fetches **all matching rows**, which can be slow.
- **V** 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

X Fetching All Columns (Slow)

SELECT * FROM Employees WHERE EmployeeID = 5;

- **Unnecessary data** is retrieved, increasing query time.
- **V** Fetching Only Required Columns

SELECT EmployeeID, Name FROM Employees WHERE EmployeeID = 5;

Faster because only needed data is retrieved.

Data types

```
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?

- 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 \rightarrow Reduces storage instead of using VARCHAR.
- 5. TINYINT(1) for Boolean fields \rightarrow Efficient storage instead of BOOLEAN.