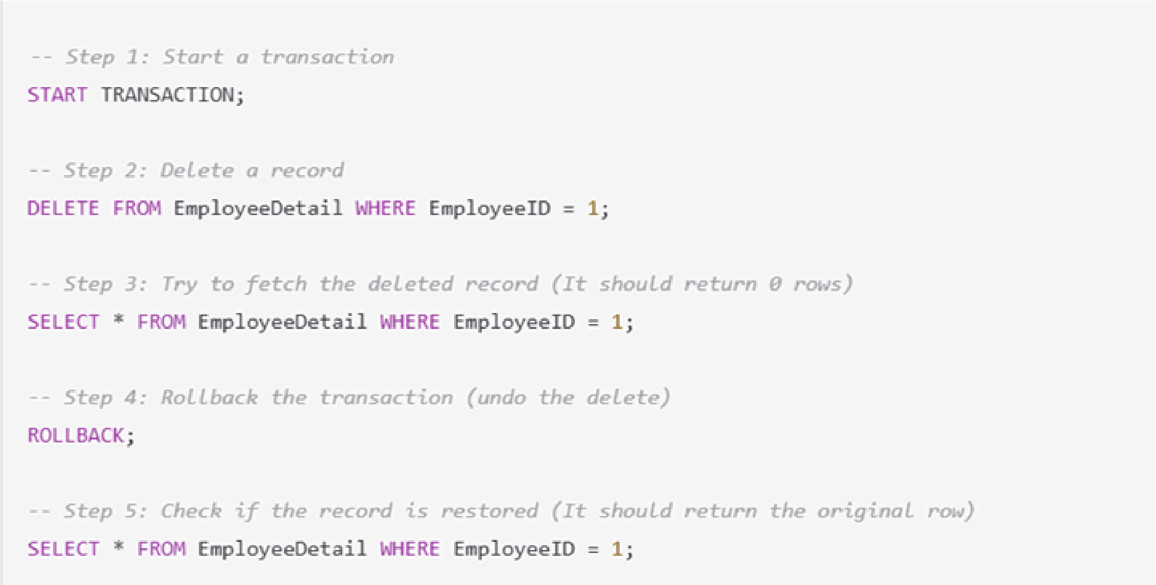
Delete vs truncate vs drop

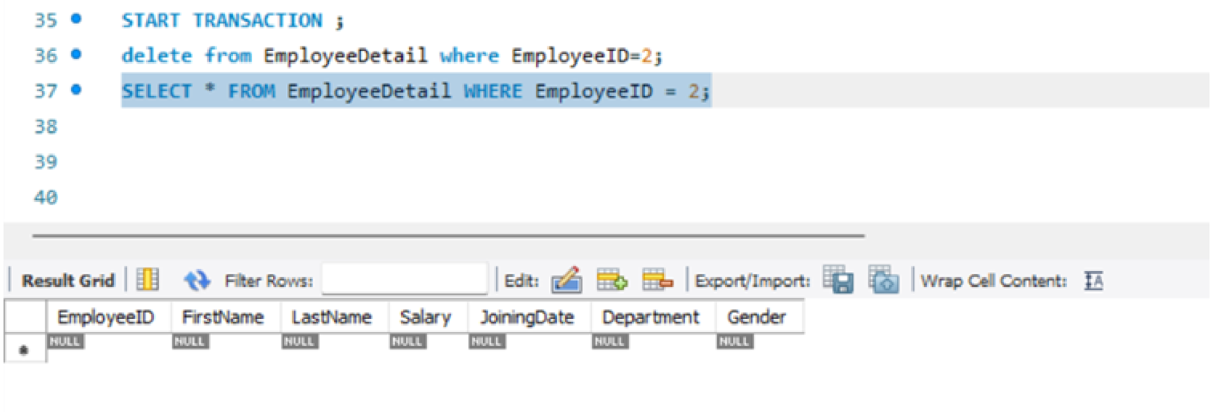
In the context of databases, especially in SQL, **DELETE**, **TRUNCATE**, and **DROP** are commands used to remove data or database objects, but they serve distinct purposes:

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:

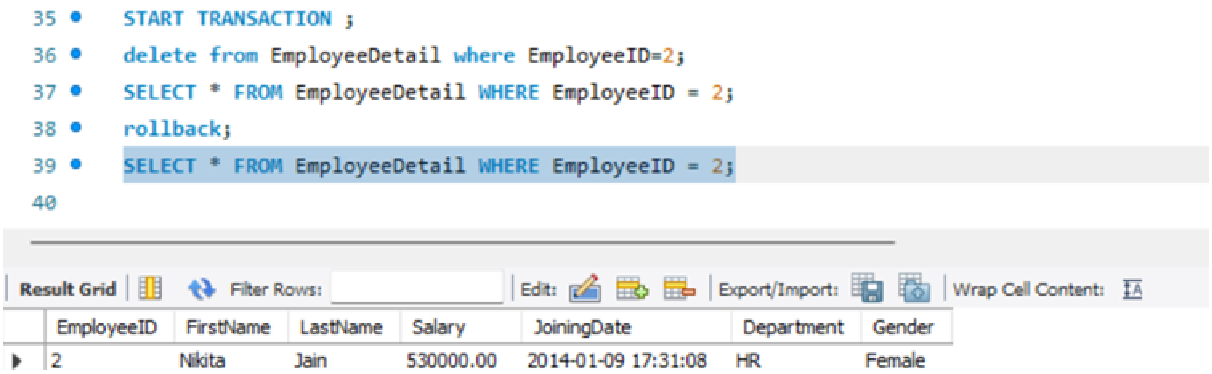


```
-- Step 1: Start a transaction
START TRANSACTION;

-- 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;
```



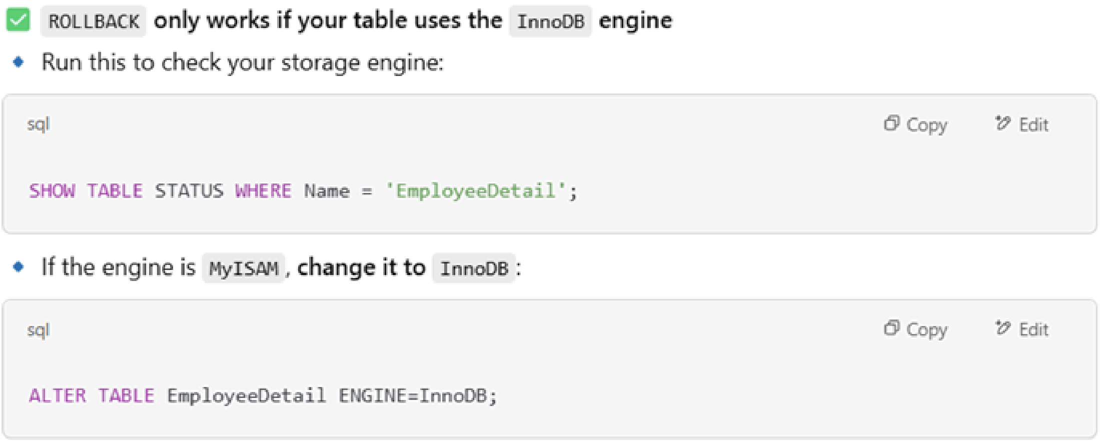
```sql
START TRANSACTION ;
delete from EmployeeDetail where EmployeeID=2;
SELECT * FROM EmployeeDetail WHERE EmployeeID = 2;
```

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



```sql
START TRANSACTION ;
delete from EmployeeDetail where EmployeeID=2;
SELECT * FROM EmployeeDetail WHERE EmployeeID = 2;
rollback;
SELECT * FROM EmployeeDetail WHERE EmployeeID = 2;
```

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:

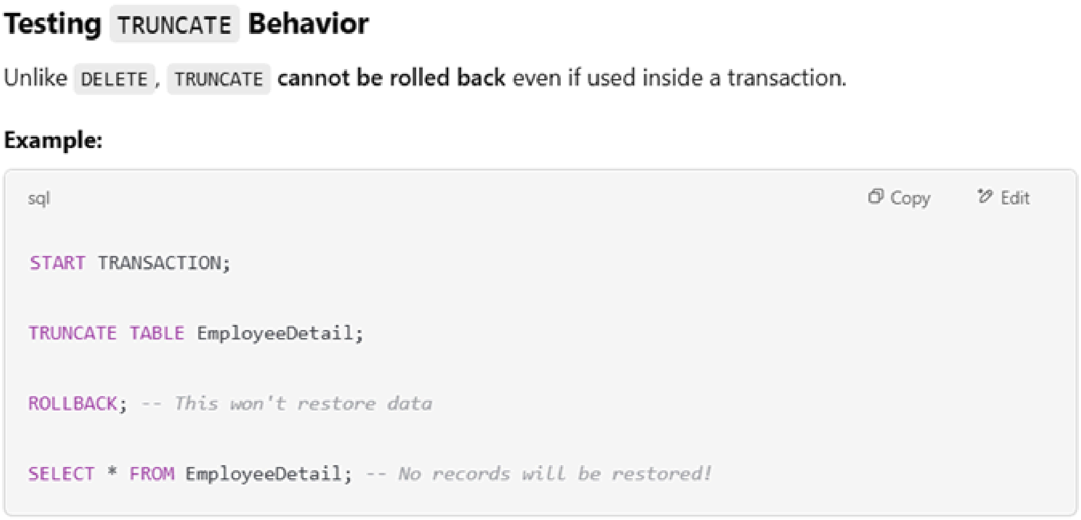
```sql
SHOW TABLE STATUS WHERE Name = 'EmployeeDetail';
```

- 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.

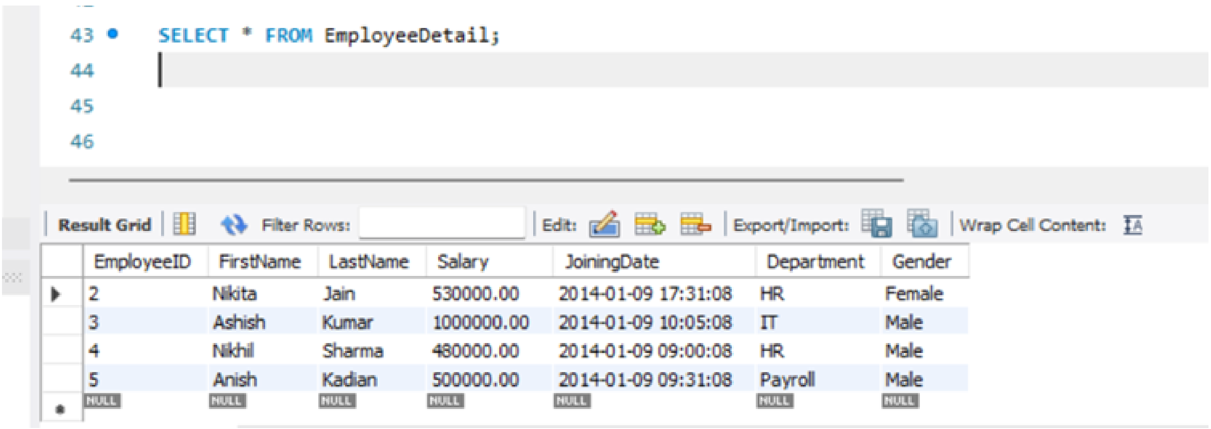


Testing `TRUNCATE` Behavior

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

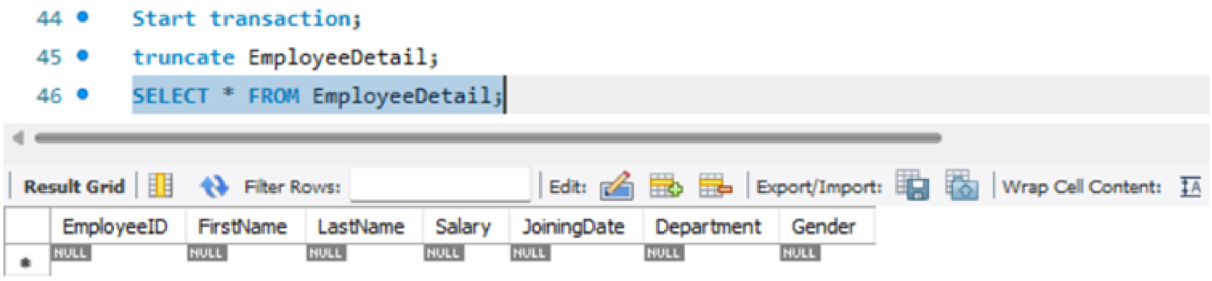
Example:

```sql
START TRANSACTION;

TRUNCATE TABLE EmployeeDetail;

ROLLBACK; -- This won't restore data

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



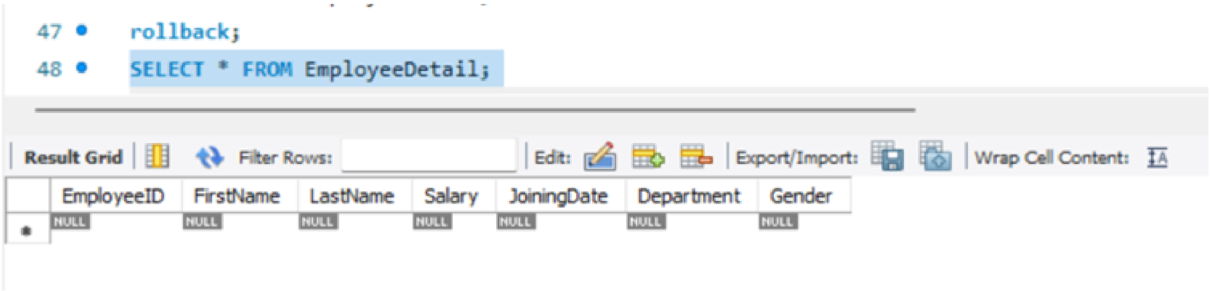
```sql
SELECT * FROM EmployeeDetail;
```

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



```
Start transaction;
truncate EmployeeDetail;
SELECT * FROM EmployeeDetail;
```

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



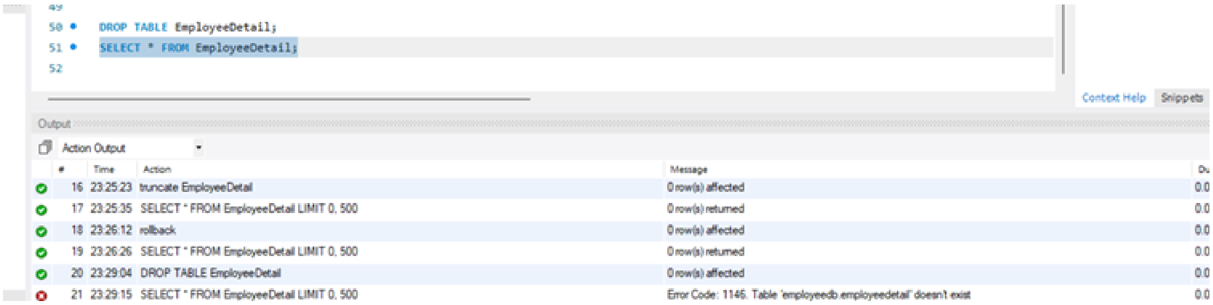
```
rollback;
SELECT * FROM EmployeeDetail;
```

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

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

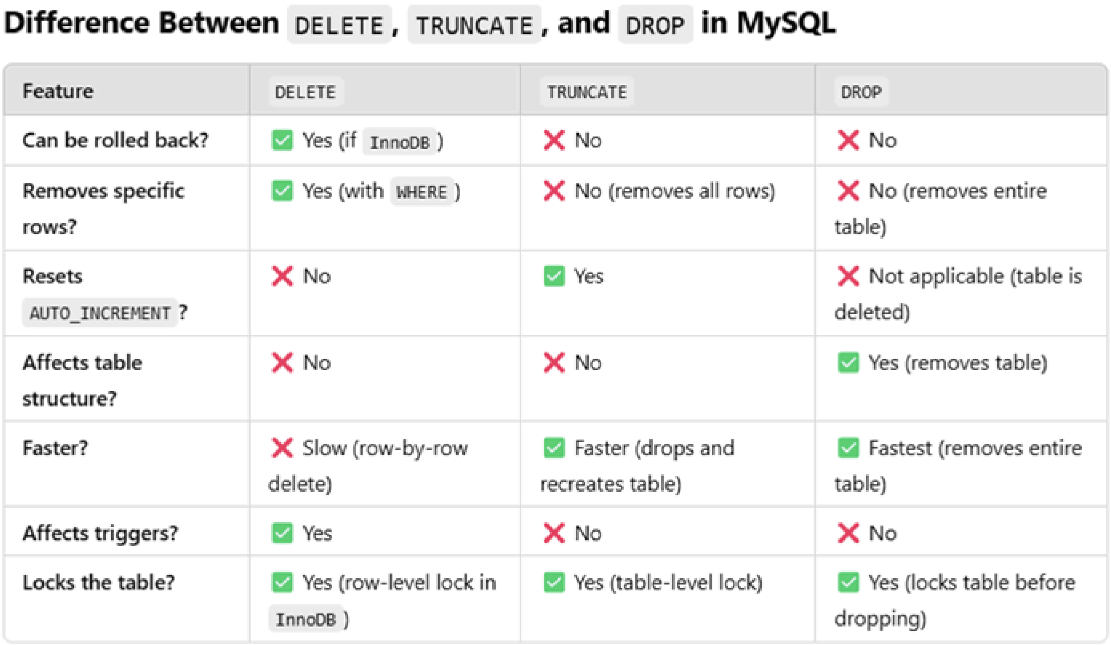
TRUNCATE is **faster** but **permanent**—it cannot be undone.

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



```
DROP TABLE EmployeeDetail;
SELECT * FROM EmployeeDetail;
```

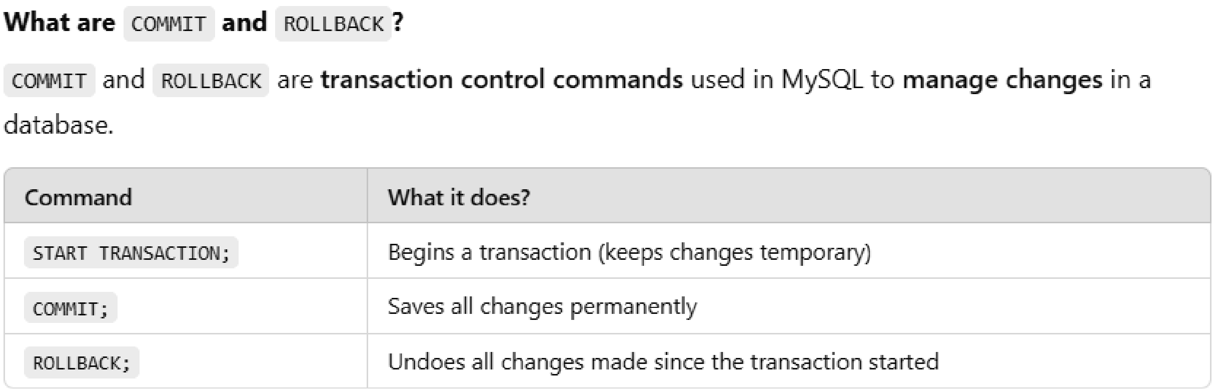
#	Time	Action	Message	Du
16	23:25:23	truncate EmployeeDetail	0 row(s) affected	0.0
17	23:25:35	SELECT * FROM EmployeeDetail LIMIT 0, 500	0 row(s) returned	0.0
18	23:26:12	rollback	0 row(s) affected	0.0
19	23:26:26	SELECT * FROM EmployeeDetail LIMIT 0, 500	0 row(s) returned	0.0
20	23:29:04	DROP TABLE EmployeeDetail	0 row(s) affected	0.0
21	23:29:15	SELECT * FROM EmployeeDetail LIMIT 0, 500	Error Code: 1146. Table 'employeedb.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)

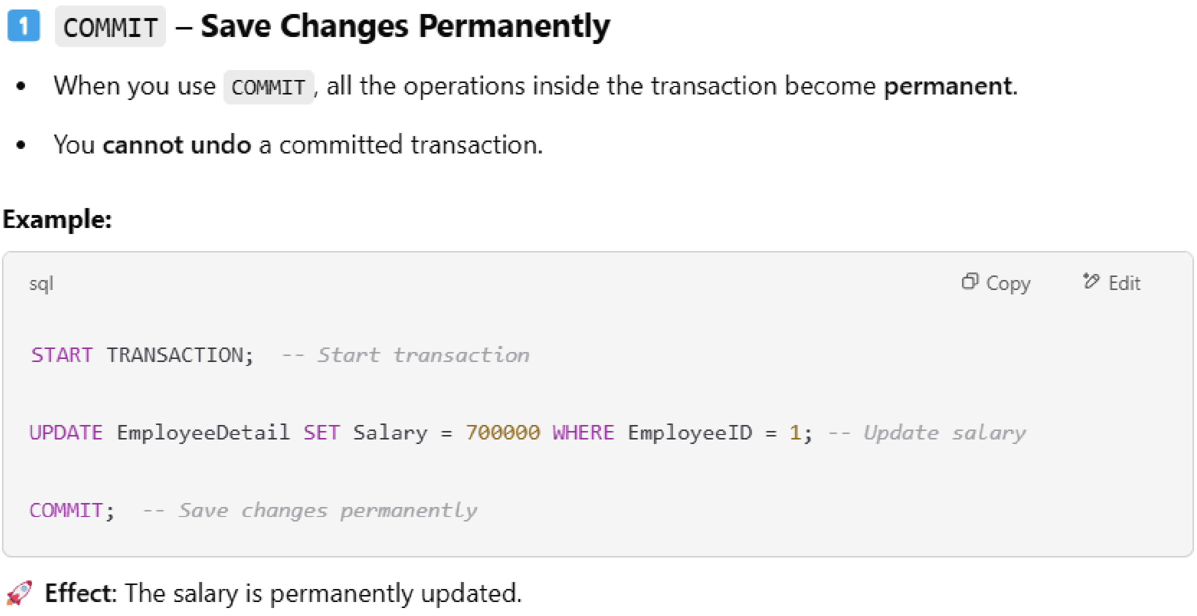
ROLLBACK and COMMIT



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



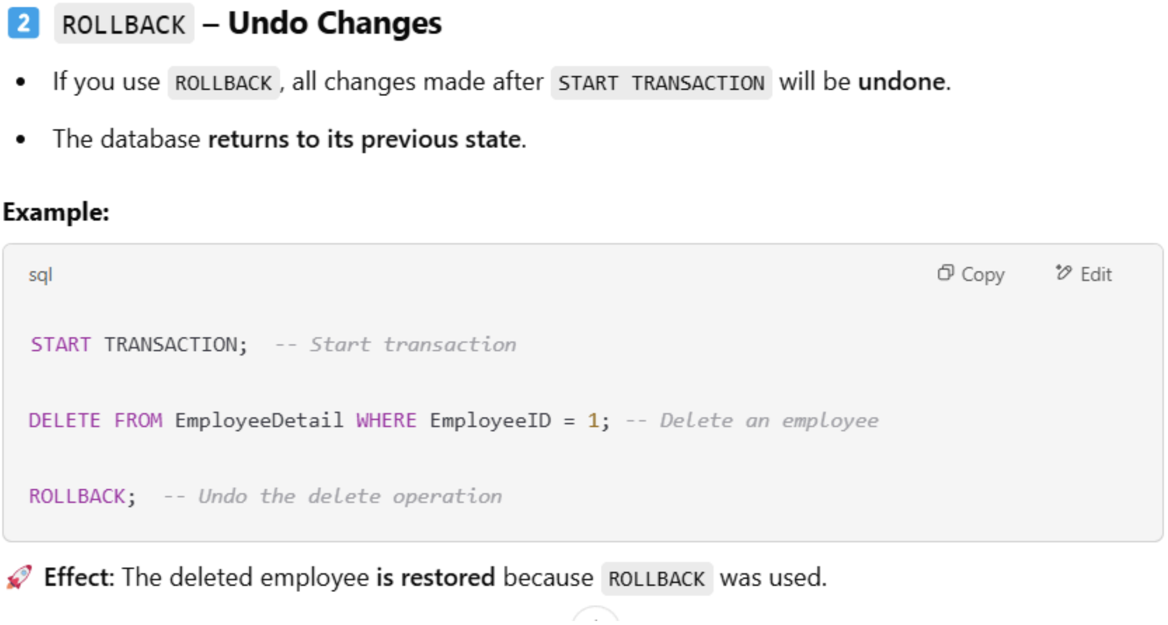
1️⃣ `COMMIT` – Save Changes Permanently

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

Example:

```sql
START TRANSACTION;  -- Start transaction

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

COMMIT;  -- Save changes permanently
```

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



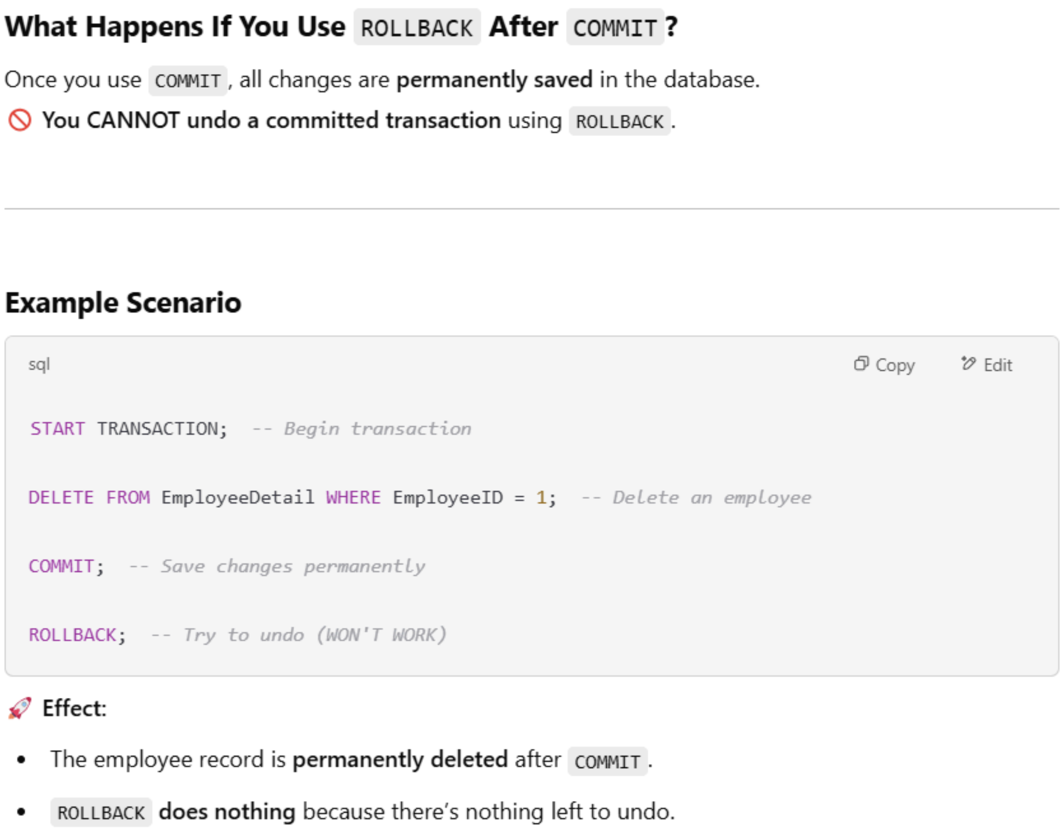
2️⃣ `ROLLBACK` – Undo Changes

- 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?

❌ 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
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
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.

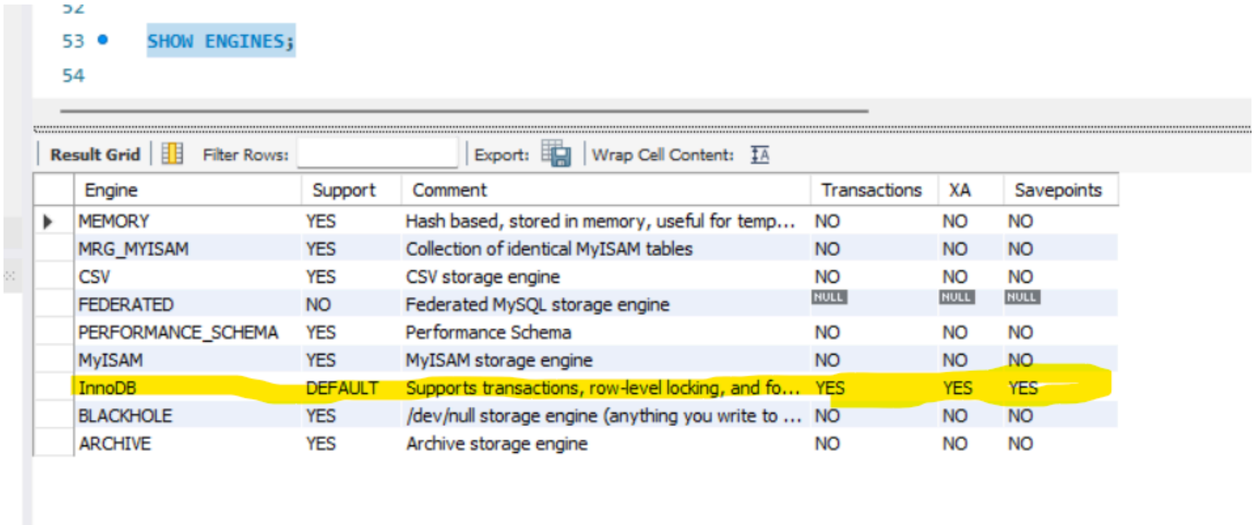
```sql
mysql -u root -p EmployeeDB < backup.sql
```

🚀 **Effect:**

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

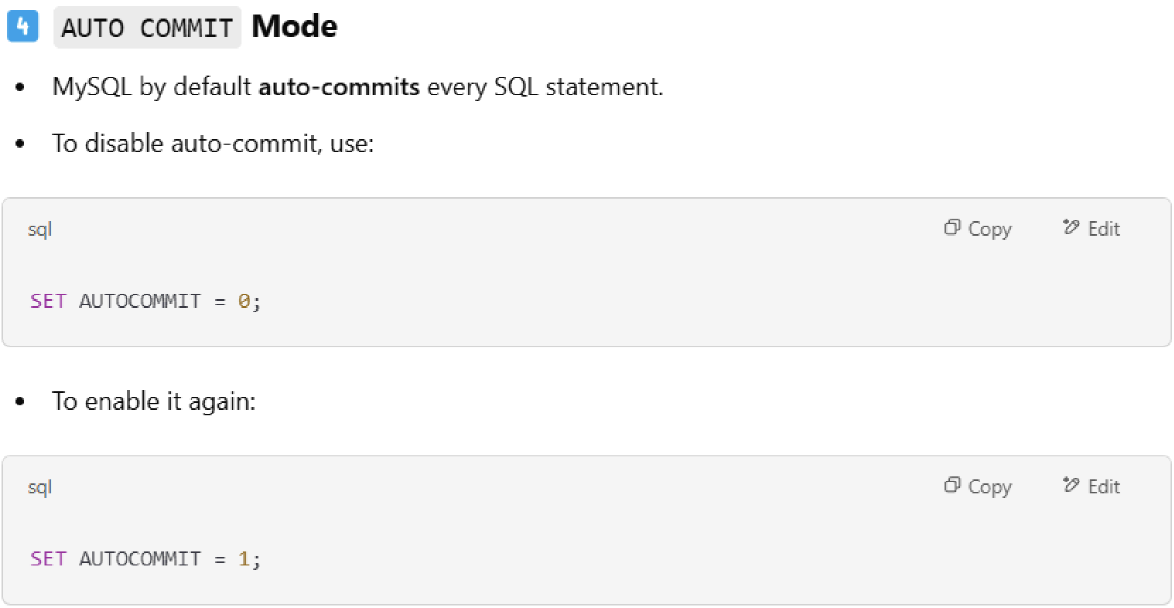
Storage Engine Used by Your MySQL Database

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.



```
SHOW ENGINES;
```

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



4️⃣ `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

1️⃣ What is MySQL?

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

2️⃣ 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.

3️⃣ 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

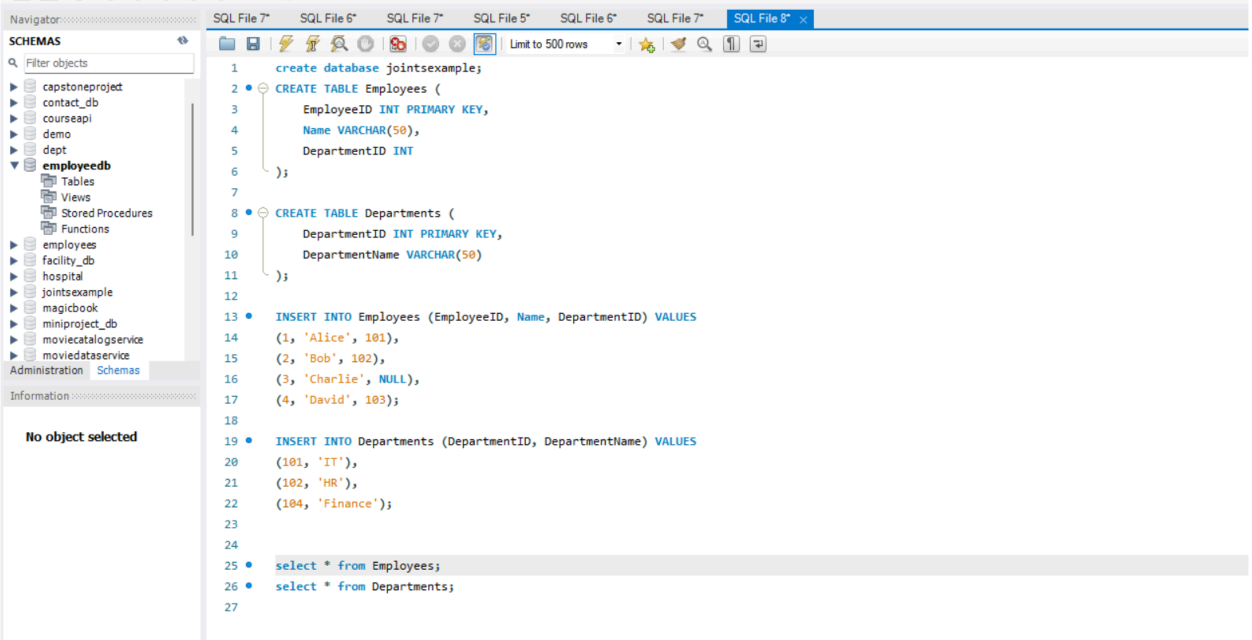
In MySQL, **joins** are used to retrieve data from multiple tables based on a related column. There are four main types of joins:

1️⃣ **INNER JOIN** → Returns only matching rows.

2️⃣ **LEFT JOIN** → Returns all rows from the left table and matching rows from the right.

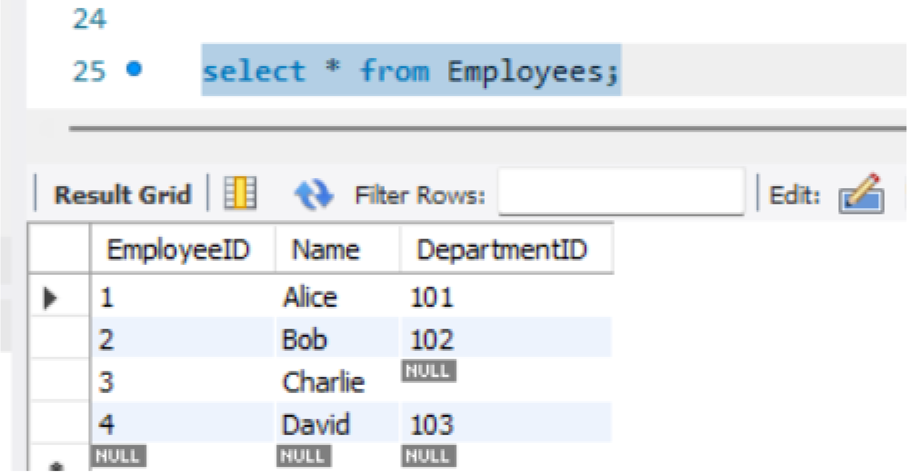
3️⃣ **RIGHT JOIN** → Returns all rows from the right table and matching rows from the left.

4️⃣ **FULL OUTER JOIN** → Returns all rows from both tables (not supported in MySQL directly).



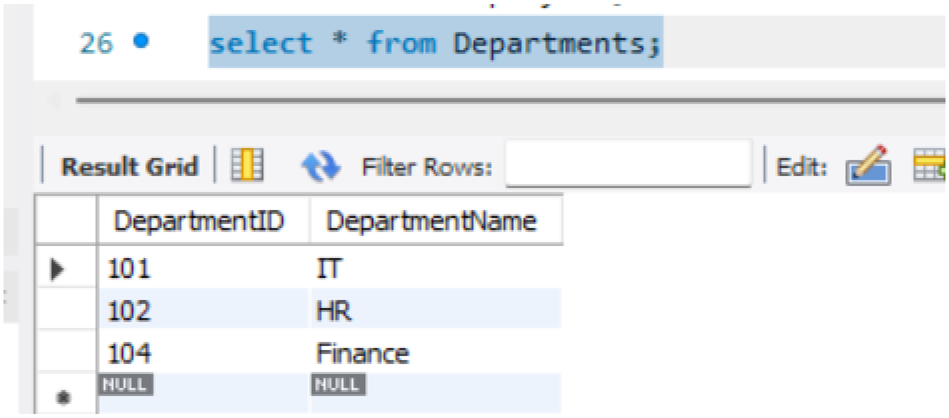
```sql
create database jointsexample;
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(50),
    DepartmentID INT
);

CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(50)
);

INSERT INTO Employees (EmployeeID, Name, DepartmentID) VALUES
(1, 'Alice', 101),
(2, 'Bob', 102),
(3, 'Charlie', NULL),
(4, 'David', 103);

INSERT INTO Departments (DepartmentID, DepartmentName) VALUES
(101, 'IT'),
(102, 'HR'),
(104, 'Finance');


select * from Employees;
select * from Departments;
```

Tables Before Join:



```sql
select * from Employees;
```

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

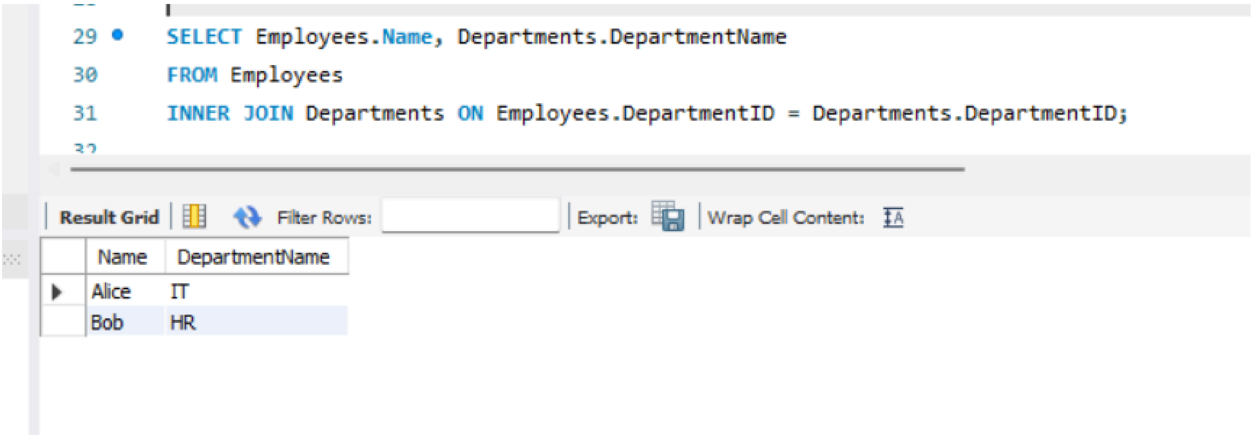


```sql
select * from Departments;
```

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.**



```sql
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

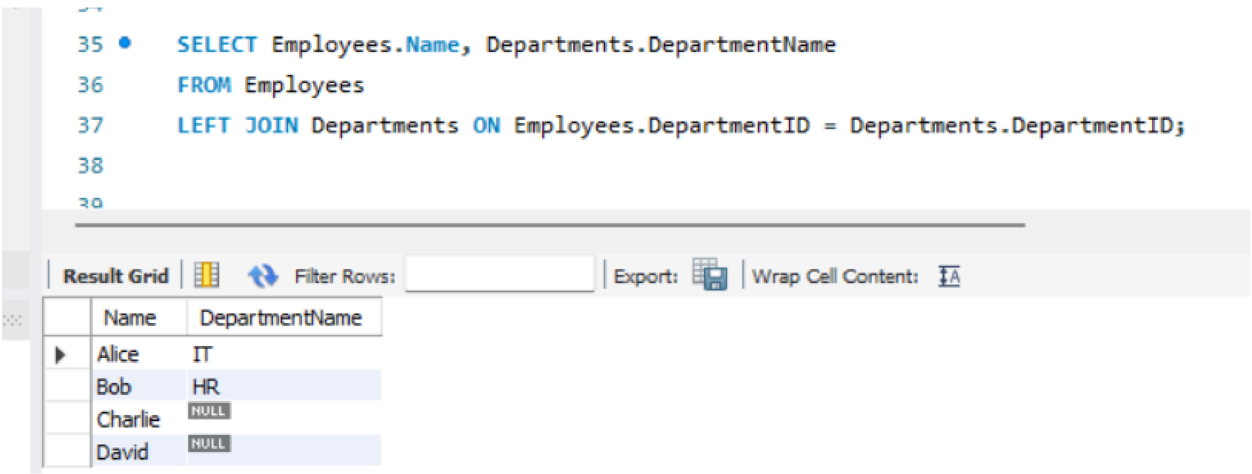
Name	DepartmentName
Alice	IT
Bob	HR

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.**



```sql
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
LEFT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

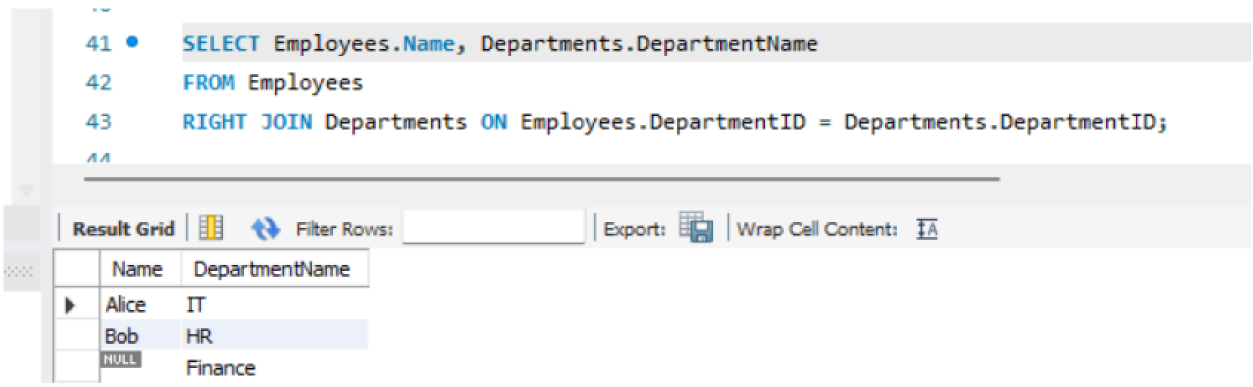
Name	DepartmentName
Alice	IT
Bob	HR
Charlie	NULL
David	NULL

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.**



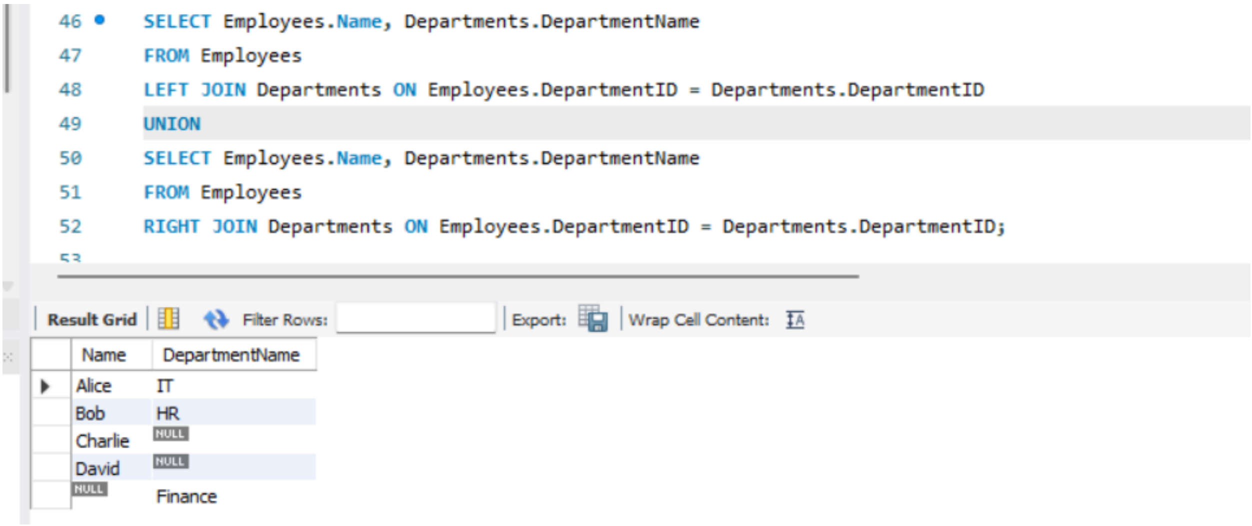
```sql
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
RIGHT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

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.



```sql
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
LEFT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID
UNION
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
RIGHT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

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**

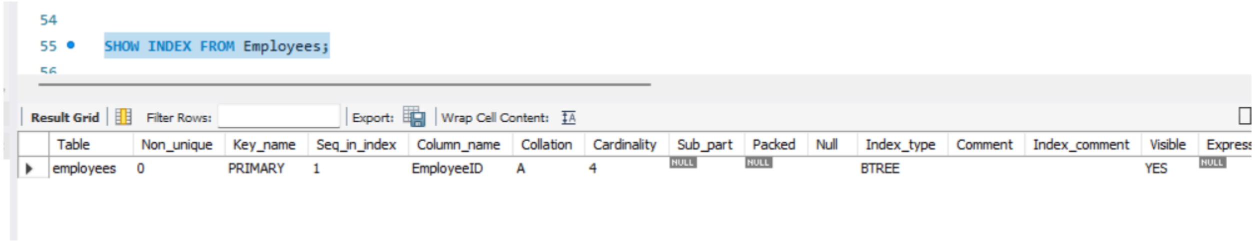
⚠️ **But be careful:**
❌ **Indexes use extra storage**
❌ **Too many indexes slow down INSERT/UPDATE/DELETE operations**

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)
);
```



```
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	NULL	NULL		BTREE			YES	NULL

UNIQUE INDEX

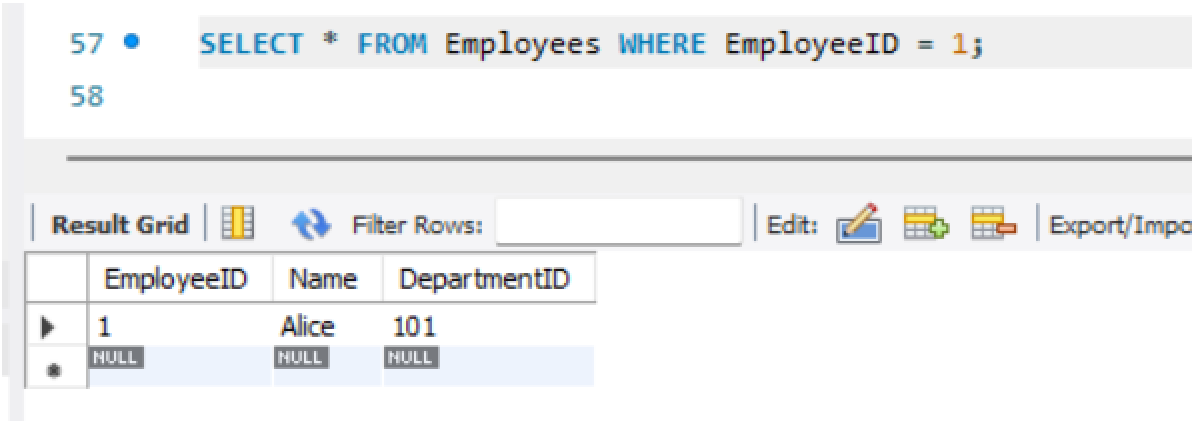
- Ensures values in a column are **unique**.
- Unlike PRIMARY KEY, a table **can have multiple UNIQUE indexes**.

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

OR

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

EX;
 Using select query

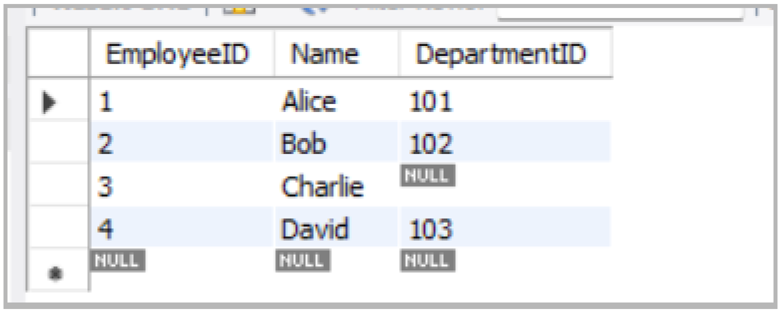


```sql
SELECT * FROM Employees WHERE EmployeeID = 1;
```

	EmployeeID	Name	DepartmentID
▶	1	Alice	101
*	NULL	NULL	NULL

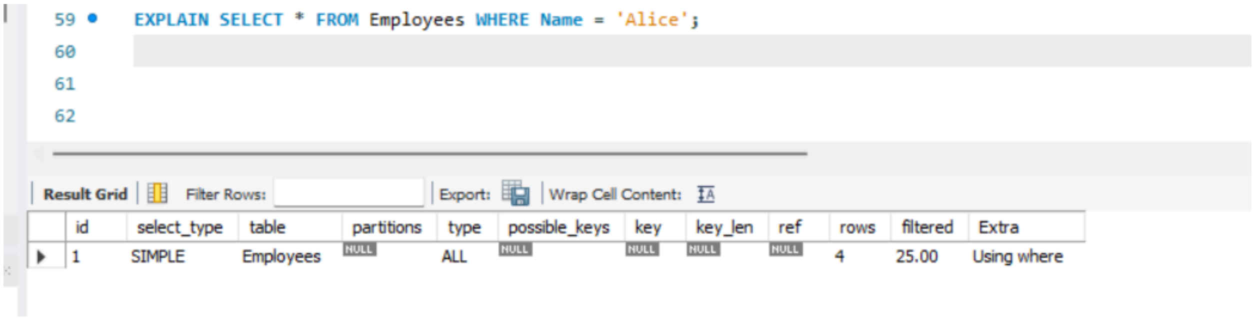
Query Optimization Strategies in MySQL

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



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

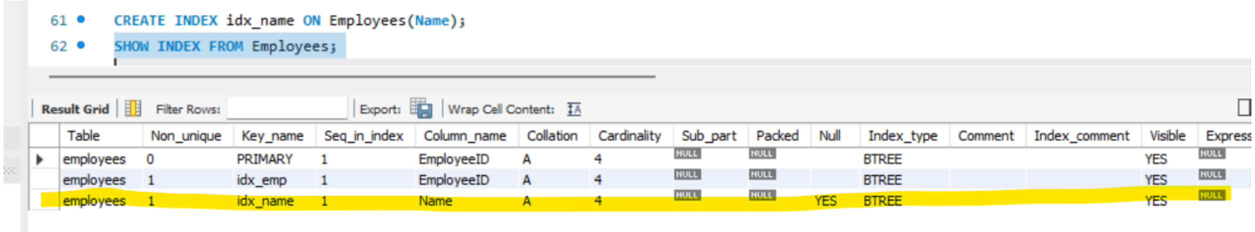
	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	Employees	NULL	ALL	NULL	NULL	NULL	NULL	4	25.00	Using where

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

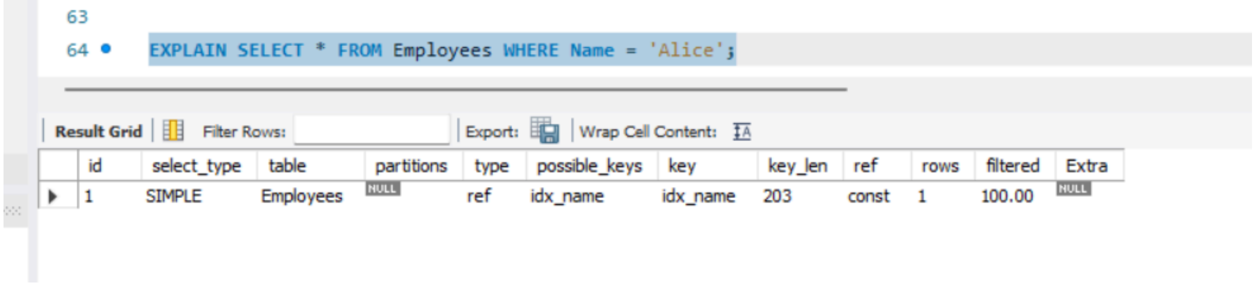
✅ **Create an index on Name** to speed up search:



```
CREATE INDEX idx_name ON Employees(Name);
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	NULL	NULL		BTREE			YES	NULL
employees	1	idx_emp	1	EmployeeID	A	4	NULL	NULL		BTREE			YES	NULL
employees	1	idx_name	1	Name	A	4	NULL	NULL	YES	BTREE			YES	NULL

Running same query again

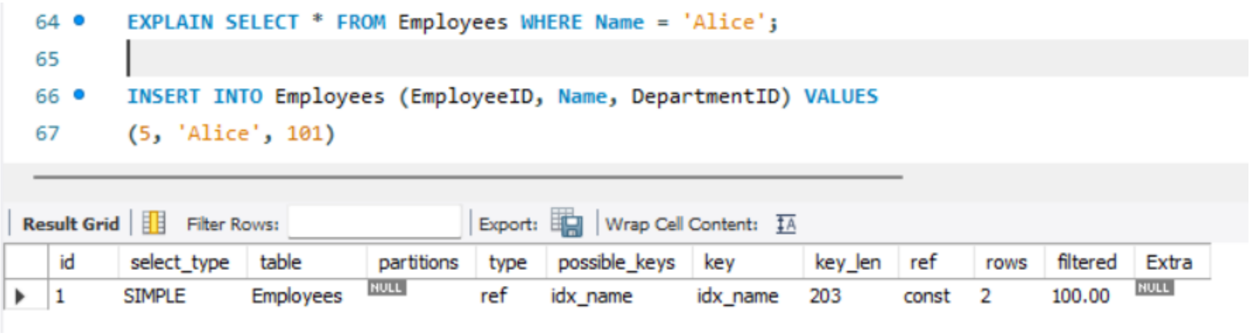


```
EXPLAIN SELECT * FROM Employees WHERE Name = 'Alice';
```

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

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. ✅



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

INSERT INTO Employees (EmployeeID, Name, DepartmentID) VALUES
(5, 'Alice', 101)
```

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

Overview
Way1

Optimized Query for Fetching Data

1️⃣Without Optimization (Slow for Large Tables)

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

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

2️⃣Optimized 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;
```

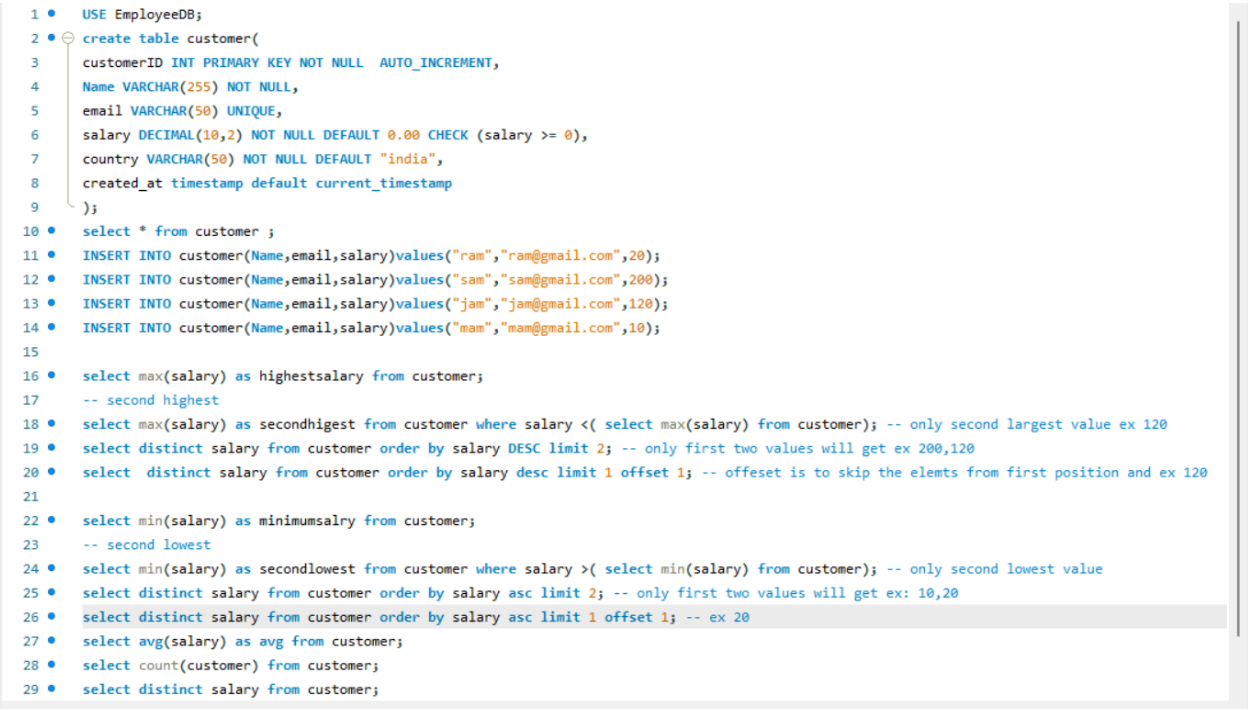
👉 **Faster** because only needed data is retrieved.

Data types

```sql
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`.



```sql
USE EmployeeDB;
create table customer(
customerID INT PRIMARY KEY NOT NULL  AUTO_INCREMENT,
Name VARCHAR(255) NOT NULL,
email VARCHAR(50) UNIQUE,
salary DECIMAL(10,2) NOT NULL DEFAULT 0.00 CHECK (salary >= 0),
country VARCHAR(50) NOT NULL DEFAULT "india",
created_at timestamp default current_timestamp
);
select * from customer ;
INSERT INTO customer(Name,email,salary)values("ram","ram@gmail.com",20);
INSERT INTO customer(Name,email,salary)values("sam","sam@gmail.com",200);
INSERT INTO customer(Name,email,salary)values("jam","jam@gmail.com",120);
INSERT INTO customer(Name,email,salary)values("mam","mam@gmail.com",10);

select max(salary) as highestsalary from customer;
-- second highest
select max(salary) as secondhigest from customer where salary <( select max(salary) from customer); -- only second largest value ex 120
select distinct salary from customer order by salary DESC limit 2; -- only first two values will get ex 200,120
select  distinct salary from customer order by salary desc limit 1 offset 1; -- offeset is to skip the elemts from first position and ex 120

select min(salary) as minimumsalry from customer;
-- second lowest
select min(salary) as secondlowest from customer where salary >( select min(salary) from customer); -- only second lowest value
select distinct salary from customer order by salary asc limit 2; -- only first two values will get ex: 10,20
select distinct salary from customer order by salary asc limit 1 offset 1; -- ex 20
select avg(salary) as avg from customer;
select count(customer) from customer;
select distinct salary from customer;
```

ACID Properties in Database (with Examples)

In databases, **ACID properties** ensure the reliability, consistency, and integrity of transactions. The acronym **ACID** stands for:

1. **Atomicity** – "All or nothing"
2. **Consistency** – "Valid before, valid after"
3. **Isolation** – "No interference"
4. **Durability** – "Permanent changes"

---

1. Atomicity ("All or Nothing")

- Ensures that a transaction is **fully completed or not executed at all**.
- If any part of a transaction fails, **the entire transaction is rolled back**.

Real-World Example: Bank Transfer

Imagine transferring ₹500 from **Account A** to **Account B**. The transaction involves:

1. Deducting ₹500 from Account A.
2. Adding ₹500 to Account B.

✅ **Success Case:**

- Both operations succeed → Transaction is **committed**.

❌ **Failure Case (Power Failure After Step 1):**

- ₹500 is deducted from A, but B does not receive it.
- **Atomicity ensures rollback**, so A’s balance remains unchanged.

SQL Example:

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 500 WHERE account_id = 1; -- Deduct ₹500
UPDATE accounts SET balance = balance + 500 WHERE account_id = 2; -- Add ₹500
COMMIT; -- Commit transaction if both operations succeed
```

If **Step 2 fails**, rollback to the original state:

```
ROLLBACK;
```

💡 **Key Benefit:** Prevents incomplete transactions from corrupting the database.

---

2. Consistency ("Valid Before, Valid After")

- Ensures the database remains **in a valid state** before and after a transaction.
- If a transaction violates database rules, it is **rolled back**.

Real-World Example: Maintaining Bank Rules

- **Business rule**: An account must have a **minimum balance of ₹1000**.
- If a withdrawal violates this rule, the transaction should fail.

SQL Example (Enforcing Consistency)

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 2000 WHERE account_id = 1;
-- If balance goes below 1000, rollback the transaction
COMMIT;
```

If the balance drops below ₹1000, the database **rolls back** the transaction.

💡 **Key Benefit: Data integrity is always maintained**—the database remains in a **valid** state.

---

3. Isolation ("No Interference")

- Ensures that **concurrent transactions** do not interfere with each other.
- Different levels of isolation prevent issues like **dirty reads, non-repeatable reads, and phantom reads**.

Real-World Example: Two Users Checking Same Account Balance

- **User A** reads Account Balance = ₹5000.
- **User B** withdraws ₹1000 and commits.
- If **User A reads again**, they may see an **inconsistent balance**.

✅ **Isolation ensures that User A sees the correct, committed balance.**

SQL Example (Setting Isolation Level)

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
SELECT balance FROM accounts WHERE account_id = 1;
-- Other transactions must wait until this one finishes
COMMIT;
```

---

4. Durability ("Permanent Changes")

- Ensures that once a transaction is committed, **it is permanently stored**, even if a system crash occurs.
- Uses **logs and backups** to recover committed transactions.

Example: Order Processing System

- A user places an order, and the payment is **processed and committed**.
- If the server crashes after payment, **the order must not be lost**.
- Durability ensures the order remains recorded in the database.

or

- A customer **places an order** and **pays online**.
- The order details **must be saved permanently**, even if the server crashes.

```
START TRANSACTION;

INSERT INTO orders (user_id, product_id, amount) VALUES (101, 5, 1500);

COMMIT; -- Ensures order is permanently saved
```

Even if the **database crashes**, the order is **not lost**.

💡 **Key Benefit:** Prevents **data loss** in critical systems like **banking, e-commerce, and healthcare**.

Summary Table of ACID Properties

ACID Property	Ensures	Example
Atomicity	Complete or rollback	Bank transfer fails midway → rollback
Consistency	Valid database state	Prevent negative balance in an account
Isolation	Transactions don't interfere	Prevent dirty reads in concurrent transactions
Durability	Committed changes are saved	Order remains after system crash

Key Takeaways

✅ ACID properties **prevent data corruption and ensure integrity**.
✅ **Atomicity** guarantees transactions are fully completed or not executed at all.
✅ **Consistency** enforces rules so data remains valid.
✅ **Isolation** prevents concurrent transaction conflicts.
✅ **Durability** ensures committed changes are never lost.

✅ **Atomicity** guarantees that a transaction is **fully completed or fully undone**.
✅ **Consistency** ensures the database **remains valid before and after transactions**.
✅ **Isolation** prevents **concurrent transactions from interfering with each other**.
✅ **Durability** ensures that **committed data is never lost**, even after crashes.