ULTIMATE MySQL HANDBOOK

CodeWithHarry

Installing MySQL

What is MySQL workbench?

MySQL Workbench is a visual tool for database architects, developers, and DBAs. It provides data modeling, SQL development, and comprehensive administration tools for server configuration, user administration, backup, and much more.

What is a Database Management System (DBMS)?

A Database Management System (DBMS) is software that interacts with end users, applications, and the database itself to capture and analyze data. It allows for the creation, retrieval, updating, and management of data in databases. If you know one DBMS, you can easily transition to another, as they share similar concepts and functionalities.

Windows / macOS Installation

1. Download from: https://dev.mysql.com/downloads/installer/
2. Run the installer and choose **Developer Default**.
3. Set a root password when prompted.
4. Install MySQL Workbench (optional but helpful GUI).

Linux (Ubuntu) Installation

Follow these steps to install MySQL and create a user:

Step 1: Update Package Index

```
sudo apt update
```

Step 2: Install MySQL Server

```
sudo apt install mysql-server
```

Step 3: Secure the Installation

```
sudo mysql_secure_installation
```

Choose your options (yes to most).

Step 4: Create a User `'harry'@'localhost'`

Log into MySQL:

```
sudo mysql
```

Run the following SQL commands:

```
CREATE USER 'harry'@'localhost' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON *.* TO 'harry'@'localhost' WITH GRANT OPTION;
FLUSH PRIVILEGES;
EXIT;
```

Step 5: Test Login

```
mysql -u harry -p
```

Enter the password: `password`

Make sure to replace `'password'` with a secure password of your choice in production environments.

Getting Started with MySQL

What is a Database?

A **database** is a container that stores related data in an organized way. In MySQL, a database holds one or more **tables**.

Think of it like:

- **Folder analogy**:
  - A **database** is like a folder.
  - Each **table** is a file inside that folder.
  - The **rows** in the table are like the content inside each file.
- **Excel analogy**:
  - A **database** is like an Excel workbook.
  - Each **table** is a separate sheet inside that workbook.
  - Each **row** in the table is like a row in Excel.

Step 1: Create a Database

```
CREATE DATABASE startersql;
```

After creating the database, either:

- Right-click it in MySQL Workbench and select "**Set as Default Schema**", or
- Use this SQL command:

```
USE startersql;
```

Step 2: Create a Table

Now we'll create a simple `users` table:

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    gender ENUM('Male', 'Female', 'Other'),
    date_of_birth DATE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

This table will store basic user info.

Step 3: Drop the Database

You can delete the entire database (and all its tables) using:

```
DROP DATABASE startersql;
```

> Be careful — this will delete everything in that database.

Data Types Explained

- `INT` : Integer type, used for whole numbers.
- `VARCHAR(100)` : Variable-length string, up to 100 characters.
- `ENUM` : A string object with a value chosen from a list of permitted values. eg. `gender ENUM('Male', 'Female', 'Other')`
- `DATE` : Stores date values. eg `date_of_birth DATE`

- `TIMESTAMP` : Stores date and time, automatically set to the current timestamp when a row is created.
- `BOOLEAN` : Stores TRUE or FALSE values, often used for flags like `is_active` .

- ## `DECIMAL(10, 2)` : Stores exact numeric data values, useful for financial data. The first number is the total number of digits, and the second is the number of digits after the decimal point.

Constraints Explained

- `AUTO_INCREMENT` : Automatically generates a unique number for each row.
- `PRIMARY KEY` : Uniquely identifies each row in the table.
- `NOT NULL` : Ensures a column cannot have a NULL value.
- `UNIQUE` : Ensures all values in a column are different.
- `DEFAULT` : Sets a default value for a column if no value is provided. eg. `created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP` , `is_active BOOLEAN DEFAULT TRUE`

Working with Tables in MySQL

Selecting Data from a Table

Select All Columns

```
SELECT * FROM users;
```

This fetches every column and every row from the `users` table.

Select Specific Columns

```
SELECT name, email FROM users;
```

This only fetches the `name` and `email` columns from all rows.

Renaming a Table

To rename an existing table:

```
RENAME TABLE users TO customers;
```

To rename it back:

```
RENAME TABLE customers TO users;
```

Altering a Table

You can use `ALTER TABLE` to modify an existing table.

Add a Column

```
ALTER TABLE users ADD COLUMN is_active BOOLEAN DEFAULT TRUE;
```

Drop a Column

```
ALTER TABLE users DROP COLUMN is_active;
```

Modify a Column Type

```
ALTER TABLE users MODIFY COLUMN name VARCHAR(150);
```

Move a Column to the First Position

To move a column (e.g., `email`) to the first position:

```
ALTER TABLE users MODIFY COLUMN email VARCHAR(100) FIRST;
```

To move a column after another column (e.g., move `gender` after `name`):

```
ALTER TABLE users MODIFY COLUMN gender ENUM('Male', 'Female', 'Other') AFTER name;
```

Inserting Data into MySQL Tables

To add data into a table, we use the `INSERT INTO` statement.

Insert Without Specifying Column Names (Full Row Insert)

This method **requires** you to provide values for **all columns in order**, except columns with default values or `AUTO_INCREMENT`.

```
INSERT INTO users VALUES
(1, 'Alice', 'alice@example.com', 'Female', '1995-05-14', DEFAULT);
```

> Not recommended if your table structure might change (e.g., new columns added later).

Insert by Specifying Column Names (Best Practice)

This method is safer and more readable. You only insert into specific columns.

```
INSERT INTO users (name, email, gender, date_of_birth) VALUES
('Bob', 'bob@example.com', 'Male', '1990-11-23');
```

or for multiple rows:

```
INSERT INTO users (name, email, gender, date_of_birth) VALUES
('Bob', 'bob@example.com', 'Male', '1990-11-23'),
('Charlie', 'charlie@example.com', 'Other', '1988-02-17');
```

The remaining columns like `id` (which is `AUTO_INCREMENT`) and `created_at` (which has a default) are automatically handled by MySQL.

Insert Multiple Rows at Once

```
INSERT INTO users (name, email, gender, date_of_birth) VALUES
('Charlie', 'charlie@example.com', 'Other', '1988-02-17'),
('David', 'david@example.com', 'Male', '2000-08-09'),
('Eva', 'eva@example.com', 'Female', '1993-12-30');
```

This is more efficient than inserting rows one by one.

Querying Data in MySQL using `SELECT`

The `SELECT` statement is used to query data from a table.

Basic Syntax

```
SELECT column1, column2 FROM table_name;
```

To select all columns:

```
SELECT * FROM users;
```

Filtering Rows with `WHERE`

Equal To

```
SELECT * FROM users WHERE gender = 'Male';
```

Not Equal To

```
SELECT * FROM users WHERE gender != 'Female';
-- or
SELECT * FROM users WHERE gender <> 'Female';
```

Greater Than / Less Than

```
SELECT * FROM users WHERE date_of_birth < '1995-01-01';
SELECT * FROM users WHERE id > 10;
```

Greater Than or Equal / Less Than or Equal

```
SELECT * FROM users WHERE id >= 5;
SELECT * FROM users WHERE id <= 20;
```

Working with NULL

IS NULL

```
SELECT * FROM users WHERE date_of_birth IS NULL;
```

IS NOT NULL

```
SELECT * FROM users WHERE date_of_birth IS NOT NULL;
```

BETWEEN

```
SELECT * FROM users WHERE date_of_birth BETWEEN '1990-01-01' AND '2000-12-31';
```

IN

```
SELECT * FROM users WHERE gender IN ('Male', 'Other');
```

LIKE (Pattern Matching)

```
SELECT * FROM users WHERE name LIKE 'A%';  -- Starts with A
SELECT * FROM users WHERE name LIKE '%a';  -- Ends with a
SELECT * FROM users WHERE name LIKE '%li%'; -- Contains 'li'
```

AND / OR

```
SELECT * FROM users WHERE gender = 'Female' AND date_of_birth > '1990-01-01';

SELECT * FROM users WHERE gender = 'Male' OR gender = 'Other';
```

ORDER BY

```
SELECT * FROM users ORDER BY date_of_birth ASC;
SELECT * FROM users ORDER BY name DESC;
```

LIMIT

```
SELECT * FROM users LIMIT 5;  -- Top 5 rows
SELECT * FROM users LIMIT 10 OFFSET 5;  -- Skip first 5 rows, then get next 10
SELECT * FROM users LIMIT 5, 10;  -- Get 10 rows starting from the 6th row (Same as
```

```
above)
SELECT * FROM users ORDER BY created_at DESC LIMIT 10;
```

Quick Quiz

What does the following queries do?

```
SELECT * FROM users WHERE salary > 60000 ORDER BY created_at DESC LIMIT 5;
```

```
SELECT * FROM users ORDER BY salary DESC;
```

```
SELECT * FROM users WHERE salary BETWEEN 50000 AND 70000;
```

UPDATE - Modifying Existing Data

The `UPDATE` statement is used to change values in one or more rows.

Basic Syntax

```
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE condition;
```

Example: Update One Column

```
UPDATE users
SET name = 'Alicia'
WHERE id = 1;
```

This changes the name of the user with `id = 1` to “Alicia”.

Example: Update Multiple Columns

```
UPDATE users
SET name = 'Robert', email = 'robert@example.com'
WHERE id = 2;
```

Without WHERE Clause (Warning)

```
UPDATE users
SET gender = 'Other';
```

> This updates **every row** in the table. Be very careful when omitting the `WHERE` clause.

Quick Quiz: Practice Your `UPDATE` Skills

Try answering or running these queries based on your `users` table.

1. Update the salary of user with `id = 5` to ₹70,000.

```
UPDATE users
SET salary = 70000
WHERE id = 5;
```

---

2. Change the name of the user with email `aisha@example.com` to `Aisha Khan`.

```
UPDATE users
SET name = 'Aisha Khan'
WHERE email = 'aisha@example.com';
```

---

3. Increase salary by ₹10,000 for all users whose salary is less than ₹60,000.

```
UPDATE users
SET salary = salary + 10000
WHERE salary < 60000;
```

---

4. Set the gender of user `Ishaan` to `Other`.

```
UPDATE users
SET gender = 'Other'
WHERE name = 'Ishaan';
```

---

5. Reset salary of all users to ₹50,000 (Careful - affects all rows).

```
UPDATE users
SET salary = 50000;
```

> Note: This query will overwrite salary for **every** user. Use with caution!

DELETE - Removing Data from a Table

The `DELETE` statement removes rows from a table.

Basic Syntax

```
DELETE FROM table_name
WHERE condition;
```

Example: Delete One Row

```
DELETE FROM users
WHERE id = 3;
```

Delete Multiple Rows

```
DELETE FROM users
WHERE gender = 'Other';
```

Delete All Rows (but keep table structure)

```
DELETE FROM users;
```

Drop the Entire Table (use with caution)

```
DROP TABLE users;
```

> This removes the table structure **and all data** permanently.

Best Practices

- Always use `WHERE` unless you're intentionally updating/deleting everything.
- Consider running a `SELECT` with the same `WHERE` clause first to confirm what will be affected:

```
SELECT * FROM users WHERE id = 3;
```

- Always back up important data before performing destructive operations.

Quick Quiz: Practice Your `DELETE` Skills

what will happen if you run these queries?

```
DELETE FROM users
WHERE salary < 50000;
```

```
DELETE FROM users
WHERE salary IS NULL;
```

MySQL Constraints

Constraints in MySQL are rules applied to table columns to ensure the **accuracy**, **validity**, and **integrity** of the data.

---

1. `UNIQUE` Constraint

Ensures that all values in a column are **different**.

Example (during table creation):

```
CREATE TABLE users (
    id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE
);
```

Add `UNIQUE` using `ALTER TABLE`:

```
ALTER TABLE users
ADD CONSTRAINT unique_email UNIQUE (email);
```

---

2. `NOT NULL` Constraint

Ensures that a column **cannot contain NULL** values.

Example:

```
CREATE TABLE users (
    id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL
);
```

Change an existing column to NOT NULL:

```
ALTER TABLE users
MODIFY COLUMN name VARCHAR(100) NOT NULL;
```

Make a column nullable again:

```
ALTER TABLE users
MODIFY COLUMN name VARCHAR(100) NULL;
```

---

3. `CHECK` Constraint

Ensures that values in a column satisfy a **specific condition**.

Example: Allow only dates of birth after Jan 1, 2000

```
ALTER TABLE users
ADD CONSTRAINT chk_dob CHECK (date_of_birth > '2000-01-01');
```

Naming the constraint (`chk_dob`) helps if you want to drop it later.

---

4. `DEFAULT` Constraint

Sets a **default value** for a column if none is provided during insert.

Example:

```
CREATE TABLE users (
    id INT PRIMARY KEY,
    is_active BOOLEAN DEFAULT TRUE
);
```

Add DEFAULT using `ALTER TABLE`:

```
ALTER TABLE users
ALTER COLUMN is_active SET DEFAULT TRUE;
```

---

5. `PRIMARY KEY` Constraint

Uniquely identifies each row. Must be **NOT NULL** and **UNIQUE**.

Example:

```
CREATE TABLE users (
    id INT PRIMARY KEY,
    name VARCHAR(100)
);
```

Add later with `ALTER TABLE`:

```
ALTER TABLE users
ADD PRIMARY KEY (id);
```

---

6. `AUTO_INCREMENT`

Used with `PRIMARY KEY` to automatically assign the next number.

Example:

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100)
);
```

Each new row gets the next available integer value in `id`.

Summary Table

Constraint	Purpose
`UNIQUE`	Prevents duplicate values
`NOT NULL`	Ensures value is not NULL
`CHECK`	Restricts values using a condition
`DEFAULT`	Sets a default value
`PRIMARY KEY`	Uniquely identifies each row
`AUTO_INCREMENT`	Automatically generates unique numbers

SQL Functions (MySQL)

SQL functions help you **analyze**, **transform**, or **summarize** data in your tables.

We'll use the `users` table which includes:

- `id`, `name`, `email`, `gender`, `date_of_birth`, `salary`, `created_at`

---

1. Aggregate Functions

These return a **single value** from a set of rows.

`COUNT()`

Count total number of users:

```
SELECT COUNT(*) FROM users;
```

Count users who are Female:

```
SELECT COUNT(*) FROM users WHERE gender = 'Female';
```

`MIN()` and `MAX()`

Get the minimum and maximum salary:

```
SELECT MIN(salary) AS min_salary, MAX(salary) AS max_salary FROM users;
```

`SUM()`

Calculate total salary payout:

```
SELECT SUM(salary) AS total_payroll FROM users;
```

`AVG()`

Find average salary:

```
SELECT AVG(salary) AS avg_salary FROM users;
```

Grouping with `GROUP BY`

Average salary by gender:

```
SELECT gender, AVG(salary) AS avg_salary
FROM users
GROUP BY gender;
```

---

2. String Functions

`LENGTH()`

Length of user names:

```
SELECT name, LENGTH(name) AS name_length FROM users;
```

`LOWER()` and `UPPER()`

Convert names to lowercase or uppercase:

```
SELECT name, LOWER(name) AS lowercase_name FROM users;
SELECT name, UPPER(name) AS uppercase_name FROM users;
```

`CONCAT()`

Combine name and email:

```
SELECT CONCAT(name, ' <', email, '>') AS user_contact FROM users;
```

---

3. Date Functions

`NOW()`

Current date and time:

```
SELECT NOW();
```

`YEAR()`, `MONTH()`, `DAY()`

Extract parts of `date_of_birth`:

```
SELECT name, YEAR(date_of_birth) AS birth_year FROM users;
```

`DATEDIFF()`

Find number of days between today and birthdate:

```
SELECT name, DATEDIFF(CURDATE(), date_of_birth) AS days_lived FROM users;
```

`TIMESTAMPDIFF()`

Calculate age in years:

```
SELECT name, TIMESTAMPDIFF(YEAR, date_of_birth, CURDATE()) AS age FROM users;
```

---

4. Mathematical Functions

`ROUND()`, `FLOOR()`, `CEIL()`

```
SELECT salary,
       ROUND(salary) AS rounded,
       FLOOR(salary) AS floored,
       CEIL(salary) AS ceiled
FROM users;
```

`MOD()`

Find even or odd user IDs:

```
SELECT id, MOD(id, 2) AS remainder FROM users;
```

---

5. Conditional Functions

`IF()`

```
SELECT name, gender,
       IF(gender = 'Female', 'Yes', 'No') AS is_female
FROM users;
```

Summary Table

Function	Purpose
`COUNT()`	Count rows
`SUM()`	Total of a column
`AVG()`	Average of values
`MIN()` / `MAX()`	Lowest / highest value
`LENGTH()`	String length
`CONCAT()`	Merge strings
`YEAR()` / `DATEDIFF()`	Date breakdown / age
`ROUND()`	Rounding numbers
`IF()`	Conditional logic

MySQL Transactions and AutoCommit

By default, MySQL operates in **AutoCommit** mode. This means that every SQL statement is treated as a **transaction** and is committed automatically. However, for more control over when changes are saved, you can turn **AutoCommit** off and manage transactions manually.

---

1. Disabling AutoCommit

When **AutoCommit** is off, you can explicitly control when to commit or rollback changes.

To disable AutoCommit:

```
SET autocommit = 0;
```

This turns off AutoCommit, meaning that changes you make won't be saved to the database unless you explicitly tell MySQL to commit them.

> **Important**: Until you execute a `COMMIT`, your changes are not permanent.

---

2. COMMIT — Save Changes to the Database

Once you've made changes and you're confident that everything is correct, you can use the `COMMIT` command to save those changes.

To commit a transaction:

```
COMMIT;
```

This saves all the changes made since the last `COMMIT` or `ROLLBACK`. After this point, the changes become permanent.

---

3. ROLLBACK — Revert Changes to the Last Safe Point

If you make an error or decide you don't want to save your changes, you can **rollback** the transaction to its previous state.

To rollback a transaction:

```
ROLLBACK;
```

This undoes all changes since the last `COMMIT` or `ROLLBACK`.

Example Workflow

Here's a simple example of using `COMMIT` and `ROLLBACK` in a transaction:

1. Turn off AutoCommit:

   ```
   SET autocommit = 0;
   ```

2. Make some changes (e.g., updating a salary):

   ```
   UPDATE users SET salary = 80000 WHERE id = 5;
   ```

3. Decide whether to commit or rollback:

   1. If you're happy with the changes, run:

      ```
      COMMIT;
      ```

   2. If you're not happy and want to revert the changes, run:

```
ROLLBACK;
```

---

4. Enabling AutoCommit Again

If you want to turn AutoCommit back on (so that every statement is automatically committed), you can do so with:

```
SET autocommit = 1;
```

Best Practices

- **Use** `COMMIT` when you want to make changes permanent.
- **Use** `ROLLBACK` to discard changes if something goes wrong.
- Consider **disabling AutoCommit** when performing complex updates to avoid saving partial or incorrect data.

Understanding `PRIMARY KEY` in MySQL

A `PRIMARY KEY` is a constraint in SQL that **uniquely identifies each row** in a table. It is one of the most important concepts in database design.

What is a Primary Key?

- A `PRIMARY KEY`:
    - Must be **unique**
    - Cannot be **NULL**
    - Is used to identify rows in a table
    - Can be a single column or a combination of columns
    - Each table can have **only one** primary key

Example:

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100)
);
```

How Is `PRIMARY KEY` Different from `UNIQUE`?

At first glance, `PRIMARY KEY` and `UNIQUE` might seem similar since both prevent duplicate values. But there are important differences:

Feature	`PRIMARY KEY`	`UNIQUE`
Must be unique	Yes	Yes
Allows NULL values	No	Yes (one or more NULLs allowed)
How many allowed	Only one per table	Can have multiple
Required by table	Recommended, often required	Optional
Dropping	Cannot be easily dropped	Can be dropped anytime

Example with UNIQUE

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(100) UNIQUE,
    name VARCHAR(100)
);
```

In this example:

- `id` is the unique identifier for each row.
- `email` must be unique, but is not the primary key.

Can I Drop a PRIMARY KEY?

Yes, but it is more restricted than dropping a `UNIQUE` constraint.

```
ALTER TABLE users DROP PRIMARY KEY;
```

> This may fail if the primary key is being used elsewhere (like in a foreign key or auto_increment column).

To drop a `UNIQUE` constraint:

```
ALTER TABLE users DROP INDEX email;
```

Auto Increment

In MySQL, a `PRIMARY KEY` is often used with the `AUTO_INCREMENT` attribute to automatically generate unique values for new rows.

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100)
);
```

This means that every time you insert a new row, MySQL will automatically assign a unique value to the `id` column. You can change the starting value of `AUTO_INCREMENT` using:

```
ALTER TABLE users AUTO_INCREMENT = 1000;
```

Key Takeaways

- Use `PRIMARY KEY` for the **main identifier** of a row.
- Use `UNIQUE` for enforcing **non-duplicate values** in other columns (like email or phone).
- You can have **only one primary key**, but you can have **many unique constraints**.

Foreign Keys in MySQL

A **foreign key** is a column that creates a **link between two tables**. It ensures that the value in one table **must match** a value in another table.

This is used to maintain **data integrity** between related data.

Why Use Foreign Keys?

Imagine this scenario:

You have a `users` table. Now you want to store each user's address. Instead of putting address columns inside the `users` table, you create a separate `addresses` table, and link it to `users` using a **foreign key**.

Creating a Table with a Foreign Key

Let's create an `addresses` table where each address belongs to a user.

```
CREATE TABLE addresses (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    street VARCHAR(255),
    city VARCHAR(100),
    state VARCHAR(100),
    pincode VARCHAR(10),
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Explanation:

- `user_id` is a **foreign key**.
- It references the `id` column in the `users` table.
- This ensures that every address must be linked to a valid user.

Dropping a Foreign Key

To drop a foreign key, you need to know its **constraint name**. MySQL auto-generates it if you don't specify one, or you can name it yourself:

```
CREATE TABLE addresses (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    CONSTRAINT fk_user FOREIGN KEY (user_id) REFERENCES users(id)
);
```

To drop it:

```
ALTER TABLE addresses
DROP FOREIGN KEY fk_user;
```

Adding a Foreign Key Later (Using ALTER)

Suppose the foreign key was **not defined during table creation**. You can add it later using `ALTER TABLE`:

```
ALTER TABLE addresses
ADD CONSTRAINT fk_user FOREIGN KEY (user_id) REFERENCES users(id);
```

Adding `ON DELETE` Action

By default, if you delete a user that has related addresses, MySQL will throw an error. You can control this behavior with `ON DELETE`.

Example with `ON DELETE CASCADE`:

If you want addresses to be automatically deleted when the user is deleted:

```
CREATE TABLE addresses (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    street VARCHAR(255),
    city VARCHAR(100),
    state VARCHAR(100),
    pincode VARCHAR(10),
    CONSTRAINT fk_user FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

Or alter it later:

```
ALTER TABLE addresses
ADD CONSTRAINT fk_user FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE
CASCADE;
```

Other ON DELETE Options

ON DELETE Option	Behavior
`CASCADE`	Deletes all related rows in child table
`SET NULL`	Sets the foreign key to NULL in the child table
`RESTRICT`	Prevents deletion of parent if child exists (default)

Summary

- Foreign keys **connect tables** and enforce **valid references**.
- You can create them inline or with `ALTER TABLE`.
- You can drop them by name.
- Use `ON DELETE` to control what happens when the parent row is deleted.

SQL JOINs in MySQL

In SQL, **JOINs** are used to combine rows from **two or more tables** based on related columns — usually a **foreign key** in one table referencing a **primary key** in another.

We'll use the following two tables:

`users` table

id	name
1	Aarav
2	Sneha
3	Raj

`addresses` table

id	user_id	city
1	1	Mumbai
2	2	Kolkata
3	4	Delhi

Note: `user_id` is a **foreign key** that references `users.id`.

---

1. INNER JOIN

Returns **only the matching rows** from both tables.

```sql
SELECT users.name, addresses.city
FROM users
INNER JOIN addresses ON users.id = addresses.user_id;
```

Output:

name	city
Aarav	Mumbai
Sneha	Kolkata

> `Raj` is excluded because there is no matching address. `Delhi` is excluded because its `user_id` (4) is not in `users`.

Visual Representation:

```
users          addresses
 -----           ------
|  1  |         |  1   |
|  2  |         |  2   |
|     |         |      |
 => only matching pairs
```

---

2. LEFT JOIN

Returns **all rows from the left table** (`users`), and matching rows from the right table (`addresses`). If no match is found, NULLs are returned.

```sql
SELECT users.name, addresses.city
FROM users
LEFT JOIN addresses ON users.id = addresses.user_id;
```

Output:

name	city
Aarav	Mumbai
Sneha	Kolkata
Raj	NULL

> `Raj` is shown even though he doesn't have an address.

Visual Representation:

```
users          addresses
 -----           ------
|  1  |         |  1   |
|  2  |         |  2   |
|  3  |         |      |
 => all users + matched addresses (or NULL)
```

---

3. RIGHT JOIN

Returns **all rows from the right table** (`addresses`), and matching rows from the left table (`users`). If no match is found, NULLs are returned.

```
SELECT users.name, addresses.city
FROM users
RIGHT JOIN addresses ON users.id = addresses.user_id;
```

Output:

name	city
Aarav	Mumbai
Sneha	Kolkata

name	city
NULL	Delhi

> `Delhi` is shown even though it points to a `user_id` that doesn't exist.

Visual Representation:

```
users          addresses
 -----           ------
|  1  |         |  1   |
|  2  |         |  2   |
|     |         |  4   |
 => all addresses + matched users (or NULL)
```

Summary Table

JOIN Type	Description
`INNER JOIN`	Only matching rows from both tables
`LEFT JOIN`	All rows from left table + matching from right
`RIGHT JOIN`	All rows from right table + matching from left

SQL `UNION` and `UNION ALL`

The `UNION` operator in SQL is used to **combine the result sets of two or more `SELECT` statements**. It removes duplicates by default.

If you want to include all rows including duplicates, use `UNION ALL`.

Example Scenario

You already have a `users` table for active users. Now, we'll create an `admin_users` table to store users who are administrators or have special roles. We will then combine the names from both tables using `UNION`.

Step 1: Create the `admin_users` Table

```
CREATE TABLE admin_users (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100),
    gender ENUM('Male', 'Female', 'Other'),
    date_of_birth DATE,
    salary INT
);
```

Step 2: Insert Sample Data into `admin_users`

```
INSERT INTO admin_users (id, name, email, gender, date_of_birth, salary) VALUES
(101, 'Anil Kumar', 'anil@example.com', 'Male', '1985-04-12', 60000),
(102, 'Pooja Sharma', 'pooja@example.com', 'Female', '1992-09-20', 58000),
(103, 'Rakesh Yadav', 'rakesh@example.com', 'Male', '1989-11-05', 54000),
(104, 'Fatima Begum', 'fatima@example.com', 'Female', '1990-06-30', 62000);
```

Step 3: Use `UNION` to Combine Data

Let's combine the active and admin user names.

```
SELECT name FROM users
UNION
SELECT name FROM admin_users;
```

This returns a **single list of unique names** from both tables.

`UNION ALL` Example

If you want to keep duplicate names (if any), use `UNION ALL`.

```
SELECT name FROM users
UNION ALL
SELECT name FROM admin_users;
```

Using More Than One Column

You can also select multiple columns as long as both `SELECT` queries return the **same number of columns with compatible types**.

```sql
SELECT name, salary FROM users
UNION
SELECT name, salary FROM admin_users;
```

Adding separate roles

```sql
SELECT name, 'User' AS role FROM users
UNION
SELECT name, 'Admin' AS role FROM admin_users;
```

Using Order By with `UNION`

```sql
SELECT name FROM users
UNION
SELECT name FROM admin_users
ORDER BY name;
```

Rules of `UNION`

1. The number of columns and their data types must match in all `SELECT` statements.
2. `UNION` removes duplicates by default.
3. `UNION ALL` keeps duplicates.

When to Use `UNION`

- When you have **two similar tables** (like current and archived data).
- When you need to **combine filtered results** (e.g., high-salary users from two sources).
- When performing **cross-category reporting**.

Summary

Operator	Behavior
`UNION`	Combines results, removes duplicates
`UNION ALL`	Combines results, keeps duplicates

Self JOIN in MySQL

A **Self JOIN** is a regular join, but the table is joined **with itself**.

This is useful when rows in the same table are related to each other. For example, when users refer other users, and we store the ID of the person who referred them in the same `users` table.

Step 1: Add a `referred_by_id` Column

We'll extend the existing `users` table to include a column called `referred_by_id`, which holds the `id` of the user who referred them.

```
ALTER TABLE users
ADD COLUMN referred_by_id INT;
```

This column:

- Will be **NULL** for users who were not referred.
- Will contain the **`id` of another user** who referred them.

Step 2: Insert Referral Data (Optional)

Assuming `id = 1` is the first user, and referred others:

```
UPDATE users SET referred_by_id = 1 WHERE id IN (2, 3); -- User 1 referred Users 2
and 3
UPDATE users SET referred_by_id = 2 WHERE id = 4;       -- User 2 referred User 4
```

Step 3: Use a Self JOIN to Get Referrer Names

We want to get each user's name **along with the name of the person who referred them**.

```
SELECT
  a.id,
  a.name AS user_name,
  b.name AS referred_by
FROM users a
INNER JOIN users b ON a.referred_by_id = b.id;
```

Explanation:

- `a` refers to the user being queried.
- `b` refers to the user who referred them.
- `LEFT JOIN` is used so that users with `NULL` in `referred_by_id` are also included.

Sample Output:

id	user_name	referred_by
1	Aarav	NULL
2	Sneha	Aarav
3	Raj	Aarav
4	Fatima	Sneha

Summary

- Use **Self JOIN** when you need to **join a table with itself**.
- In referral-based relationships, store the referrer's `id` in the same table.

- Use aliases like `a` and `b` to differentiate the two instances of the same table.

MySQL Views

A **view** in MySQL is a **virtual table** based on the result of a `SELECT` query. It does not store data itself — it always reflects the **current data** in the base tables.

Views are useful when:

- You want to simplify complex queries
- You want to reuse logic
- You want to hide certain columns from users
- You want a "live snapshot" of filtered data

Creating a View

Suppose we want a view that lists all users earning more than ₹70,000.

```
CREATE VIEW high_salary_users AS
SELECT id, name, salary
FROM users
WHERE salary > 70000;
```

Querying the View

```
SELECT * FROM high_salary_users;
```

This will return all users from the `users` table where salary is above ₹70,000.

Demonstrating That a View is Always Up-To-Date

Let's see what happens when the underlying data changes.

Step 1: View before update

```
SELECT * FROM high_salary_users;
```

Output:

id	name	salary
2	Sneha	75000
5	Fatima	80000

Step 2: Update a user's salary

```
UPDATE users
SET salary = 72000
WHERE name = 'Raj';
```

Step 3: Query the view again

```
SELECT * FROM high_salary_users;
```

New Output:

id	name	salary
2	Sneha	75000
5	Fatima	80000
3	Raj	72000

> Notice how **Raj** is now included in the view — without updating the view itself. That's because views **always reflect live data** from the original table.

Dropping a View

To remove a view:

```
DROP VIEW high_salary_users;
```

Summary

- Views act like **saved SELECT queries**
- Views are **not duplicated data**
- Changes to base tables are **reflected automatically**
- Great for simplifying complex queries or creating filtered access

MySQL Indexes

Indexes in MySQL are used to **speed up data retrieval**. They work like the index of a book — helping the database engine find rows faster, especially for **searches, filters, and joins**.

Viewing Indexes on a Table

To see the indexes on a table, use:

```
SHOW INDEXES FROM users;
```

This shows all the indexes currently defined on the `users` table, including the automatically created **primary key** index.

Creating a Single-Column Index

Suppose you're frequently searching users by their `email`. You can speed this up by indexing the `email` column.

```
CREATE INDEX idx_email ON users(email);
```

What this does:

- Creates an index named `idx_email`
- Improves performance of queries like:

```
SELECT * FROM users WHERE email = 'example@example.com';
```

Important Notes

- Indexes **consume extra disk space**
- Indexes **slow down** `INSERT`, `UPDATE`, **and** `DELETE` operations slightly (because the index must be updated)
- Use indexes **only when needed** (i.e., for columns used in `WHERE`, `JOIN`, `ORDER BY`)

Creating a Multi-Column Index

If you often query users using **both** `gender` **and** `salary`, a multi-column index is more efficient than separate indexes.

```
CREATE INDEX idx_gender_salary ON users(gender, salary);
```

Usage Example:

```
SELECT * FROM users
WHERE gender = 'Female' AND salary > 70000;
```

This query can take advantage of the combined index on `gender` and `salary`.

Index Order Matters

For a multi-column index on `(gender, salary)`:

- This works efficiently:

```
WHERE gender = 'Female' AND salary > 70000
```

- But this **may not** use the index effectively:

```
WHERE salary > 70000
```

Because the first column in the index (`gender`) is missing in the filter.

Dropping an Index

To delete an index:

```
DROP INDEX idx_email ON users;
```

Summary

Feature	Description
`SHOW INDEXES`	View current indexes on a table
`CREATE INDEX`	Create single or multi-column indexes
`DROP INDEX`	Remove an index
Use when	Query performance on large tables is a concern
Avoid on	Columns that are rarely queried or always unique

Subqueries in MySQL

A **subquery** is a query nested inside another query. Subqueries are useful for breaking down complex problems into smaller parts.

They can be used in:

- `SELECT` statements
- `WHERE` clauses
- `FROM` clauses

Example Scenario: Salary Comparison

Suppose we want to find all users who **earn more than the average salary** of all users.

Scalar Subquery Example

This subquery returns **a single value** — the average salary — and we compare each user's salary against it.

```
SELECT id, name, salary
FROM users
WHERE salary > (
    SELECT AVG(salary) FROM users
);
```

Explanation:

- The inner query: `SELECT AVG(salary) FROM users` returns the average salary.

- The outer query selects all users with a salary **greater than that average**.

Subquery with `IN`

Now let's say we want to find users who have been referred by someone who earns more than ₹75,000.

```
SELECT id, name, referred_by_id
FROM users
WHERE referred_by_id IN (
    SELECT id FROM users WHERE salary > 75000
);
```

Explanation:

- The inner query: `SELECT id FROM users WHERE salary > 75000` returns a list of user IDs (referrers) who earn more than ₹75,000.
- The outer query selects users whose `referred_by_id` is in that list.

Other Places Subqueries Are Used

You can also use subqueries:

- Inside `SELECT` columns (called scalar subqueries)
- In the `FROM` clause to create derived tables

Example in `SELECT`:

```
SELECT name, salary,
  (SELECT AVG(salary) FROM users) AS average_salary
FROM users;
```

This shows each user's salary along with the overall average.

Summary

Subquery Type	Use Case
Scalar Subquery	Returns one value (e.g. AVG, MAX)
Subquery with `IN`	Returns multiple values
Subquery in `SELECT`	Shows related calculated value
Subquery in `FROM`	Acts as a virtual table

Subqueries are powerful tools when filtering based on computed or dynamic conditions.

GROUP BY and HAVING in MySQL

The `GROUP BY` clause is used to **group rows that have the same values** in specified columns. It is typically used with **aggregate functions** like `COUNT`, `SUM`, `AVG`, `MIN`, or `MAX`.

The `HAVING` clause is used to **filter groups** after aggregation — similar to how `WHERE` filters individual rows.

Example Table: `users`

Assume this is your `users` table:

id	name	gender	salary	referred_by_id
1	Aarav	Male	80000	NULL
2	Sneha	Female	75000	1
3	Raj	Male	72000	1
4	Fatima	Female	85000	2
5	Priya	Female	70000	NULL

GROUP BY Example: Average Salary by Gender

```
SELECT gender, AVG(salary) AS average_salary
FROM users
GROUP BY gender;
```

Explanation:

- This groups users by gender.
- Then calculates the average salary for each group.

GROUP BY with COUNT

Find how many users were referred by each user:

```
SELECT referred_by_id, COUNT(*) AS total_referred
FROM users
WHERE referred_by_id IS NOT NULL
GROUP BY referred_by_id;
```

Output:

referred_by_id	total_referred
1	2
2	1

HAVING Clause: Filtering Groups

Let's say we only want to show genders where the **average salary is greater than ₹75,000**.

```
SELECT gender, AVG(salary) AS avg_salary
FROM users
GROUP BY gender
HAVING AVG(salary) > 75000;
```

Why not `WHERE`?

- `WHERE` is used **before** grouping.
- `HAVING` is used **after** groups are formed — it's the only way to filter aggregated values.

Another Example: Groups with More Than 1 Referral

```
SELECT referred_by_id, COUNT(*) AS total_referred
FROM users
WHERE referred_by_id IS NOT NULL
GROUP BY referred_by_id
HAVING COUNT(*) > 1;
```

Summary

Clause	Purpose	Can use aggregates?
`WHERE`	Filters rows **before** grouping	No
`GROUP BY`	Groups rows based on column values	N/A
`HAVING`	Filters groups **after** aggregation	Yes

Use `GROUP BY` to organize data, and `HAVING` to filter those groups based on aggregate conditions.

ROLLUP

To get subtotals and grand totals, you can use `ROLLUP`:

```
SELECT gender, COUNT(*) AS total_users
FROM users
GROUP BY gender WITH ROLLUP;
```

Explanation:

- This will give you a count of users by gender, along with a grand total for all users.

Stored Procedures in MySQL

A **stored procedure** is a saved SQL block that can be executed later. It's useful when you want to encapsulate logic that can be reused multiple times — like queries, updates, or conditional operations.

Why Change the Delimiter?

By default, MySQL uses `;` to end SQL statements.

But when defining stored procedures, we use `;` **inside** the procedure as well. This can confuse MySQL. To avoid this, we **temporarily change the delimiter** (e.g. to `$$` or `//`) while creating the procedure.

Syntax for Creating a Stored Procedure

```
DELIMITER $$

CREATE PROCEDURE procedure_name()
BEGIN
    -- SQL statements go here
END$$

DELIMITER ;
```

After the procedure is created, we reset the delimiter back to `;`.

Creating a Procedure with Input Parameters

Let's say you want to create a stored procedure that inserts a new user into the `users` table.

Example:

```
DELIMITER $$

CREATE PROCEDURE AddUser(
    IN p_name VARCHAR(100),
    IN p_email VARCHAR(100),
    IN p_gender ENUM('Male', 'Female', 'Other'),
    IN p_dob DATE,
    IN p_salary INT
)
BEGIN
    INSERT INTO users (name, email, gender, date_of_birth, salary)
    VALUES (p_name, p_email, p_gender, p_dob, p_salary);
END$$

DELIMITER ;
```

This creates a procedure named `AddUser` that accepts five input parameters.

Calling the Procedure

You can call the procedure using:

```
CALL AddUser('Kiran Sharma', 'kiran@example.com', 'Female', '1994-06-15', 72000);
```

This will insert the new user into the `users` table.

Notes

- Input parameters are declared using the `IN` keyword.
- Stored procedures are stored in the database and can be reused.

Viewing Stored Procedures

```
SHOW PROCEDURE STATUS WHERE Db = 'startersql';
```

Dropping a Stored Procedure

```
DROP PROCEDURE IF EXISTS AddUser;
```

Summary

Command	Purpose
`DELIMITER $$`	Temporarily change statement delimiter
`CREATE PROCEDURE`	Defines a new stored procedure
`CALL procedure_name(...)`	Executes a stored procedure
`DROP PROCEDURE`	Removes an existing procedure

Triggers in MySQL

A **trigger** is a special kind of stored program that is **automatically executed (triggered)** when a specific event occurs in a table — such as `INSERT`, `UPDATE`, or `DELETE`.

Triggers are commonly used for:

- Logging changes
- Enforcing additional business rules
- Automatically updating related data

Basic Trigger Structure

```
CREATE TRIGGER trigger_name
AFTER INSERT ON table_name
FOR EACH ROW
BEGIN
    -- statements to execute
END;
```

Triggers can be fired:

- `BEFORE` or `AFTER` an event
- On `INSERT`, `UPDATE`, or `DELETE`

Scenario: Log Every New User Insertion

Suppose we want to log every time a new user is inserted into the `users` table. We'll create a separate table called `user_log` to store log entries.

Step 1: Create the Log Table

```
CREATE TABLE user_log (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    name VARCHAR(100),
    created_on TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 2: Create the Trigger

We now define a trigger that runs **after a new user is added**.

```
DELIMITER $$

CREATE TRIGGER after_user_insert
AFTER INSERT ON users
FOR EACH ROW
BEGIN
    INSERT INTO user_log (user_id, name)
    VALUES (NEW.id, NEW.name);
END$$

DELIMITER ;
```

Explanation:

- `AFTER INSERT` means the trigger fires after the user is inserted.
- `NEW` refers to the new row being added to the `users` table.
- We insert the new user's ID and name into the `user_log` table.

Step 3: Test the Trigger

```
CALL AddUser('Ritika Jain', 'ritika@example.com', 'Female', '1996-03-12', 74000);
```

Now check the `user_log` table:

```
SELECT * FROM user_log;
```

You should see Ritika's info automatically logged.

Dropping a Trigger

If you need to remove a trigger:

```
DROP TRIGGER IF EXISTS after_user_insert;
```

Summary

Trigger Component	Description
`BEFORE` / `AFTER`	When the trigger runs
`INSERT` / `UPDATE` / `DELETE`	What kind of action triggers it
`NEW.column`	Refers to the new row (for `INSERT`, `UPDATE`)
`OLD.column`	Refers to the old row (for `UPDATE`, `DELETE`)
`FOR EACH ROW`	Executes for each affected row

More on MySQL

This section covers some essential MySQL features and operators that help you write more powerful and flexible queries.

---

1. Logical Operators

Logical operators are used to combine multiple conditions in a `WHERE` clause.

Operator	Description	Example
`AND`	All conditions must be true	`salary > 50000 AND gender = 'Male'`
`OR`	At least one condition is true	`gender = 'Male' OR gender = 'Other'`
`NOT`	Reverses a condition	`NOT gender = 'Female'`

---

2. Add a Column to an Existing Table

Use `ALTER TABLE` to add a column:

```
ALTER TABLE users
ADD COLUMN city VARCHAR(100);
```

This adds a new column named `city` to the `users` table.

---

3. Wildcard Operators

Wildcards are used with the `LIKE` operator for pattern matching in text.

Wildcard	Description	Example
`%`	Matches any sequence	`WHERE name LIKE 'A%'` (starts with A)
_	Matches a single character	`WHERE name LIKE '_a%'` (second letter is 'a')

---

4. LIMIT with OFFSET

`LIMIT` is used to limit the number of rows returned. `OFFSET` skips a number of rows before starting to return rows.

```
SELECT * FROM users
ORDER BY id
LIMIT 5 OFFSET 10;
```

This skips the first 10 rows and returns the next 5.

Alternative syntax:

```
SELECT * FROM users
ORDER BY id
LIMIT 10, 5;
```

This also skips 10 and returns 5 (syntax: `LIMIT offset, count`).

---

5. DISTINCT Keyword

`DISTINCT` is used to return **only unique values**.

```
SELECT DISTINCT gender FROM users;
```

Returns a list of unique gender values from the `users` table.

---

6. TRUNCATE Keyword

`TRUNCATE` removes **all rows** from a table, but **keeps the table structure**.

```
TRUNCATE TABLE users;
```

- Faster than `DELETE FROM users`
- Cannot be rolled back (unless in a transaction-safe environment)

---

7. CHANGE vs MODIFY Column

Both `CHANGE` and `MODIFY` are used to alter existing columns in a table, but they work slightly differently.

CHANGE: Rename and change datatype

```
ALTER TABLE users
CHANGE COLUMN city location VARCHAR(150);
```

This renames `city` to `location` and changes its type.

MODIFY: Only change datatype

```
ALTER TABLE users
MODIFY COLUMN salary BIGINT;
```

This changes only the datatype of `salary`.