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;
```
