Creating a Database and Table

Imagine:

You're the **Principal** of a brand new school.

You want to:

1. Create a new **school database**
2. Create a **student table**
3. Add student information using code, not clicks

CREATE DATABASE

This command creates a **new database**. Think of it like making a **new folder** to keep files.

SQL:

```
CREATE DATABASE school_database;
```

Says: "Hey MySQL, please make a new folder called `school_database` to store my data."

Tip: The name should have **no spaces**. Use `_` instead.

USE DATABASE

This tells MySQL:
👉 "I want to work **inside this database** now."

SQL:

```
USE school_database;
```

Says: “Switch to my school database now.”

Without this, MySQL won’t know **where to store your table**.

What Are Data Types in MySQL?

Think of a **column** like a **container or basket**.

You need to choose **what type of thing** you’ll put inside.

- 🍎 Some baskets are for **numbers**
- ✏️ Some are for **words**
- 📅 Some are for **dates**
- ✅ Some are for **true/false answers**

This choice is called the **data type**.

Common MySQL Data Types

Data Type	Use For	Example Value	Simple Meaning
`INT`	Numbers (no decimal)	`85`, `200`, `1`	Use for things like marks, age, quantity
`FLOAT`	Numbers with decimal	`89.5`, `3.14`	Use when decimals are needed (like prices)
`VARCHAR(n)`	Text/words (limit n)	`'Meena'`	Use for names, classes, cities, etc.
`DATE`	Dates	`2025-07-08`	Use for DOB, joining date

`BOOLEAN`	Yes/No (True/False)	`TRUE`, `FALSE`	Use for active/inactive, passed/failed, etc.
`TEXT`	Long text	`'This is a note.'`	Use for comments, descriptions, etc.
`AUTO_INCREMENT`	Grows automatically	1, 2, 3...	Use with `INT` to make IDs increase by itself

---

Some Friendly Rules to Remember:

- Use `VARCHAR(100)` if you're unsure — it's safe for most text.
- Use `INT` for anything that's a whole number (like marks or quantity).
- Use `FLOAT` only when you **need decimal** values (like `99.99`).
- Always use `AUTO_INCREMENT` + `PRIMARY KEY` for your ID column.

Example: Student Table With Correct Data Types

```
CREATE TABLE students (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  class VARCHAR(10),
  marks INT,
  date_of_birth DATE,
  passed BOOLEAN
);
```

Here's what each column is storing:

- `id` → number, grows by itself
- `name` → text (like "Kavita")

- `class` → text (like “5A”)
- `marks` → number (like 87)
- `date_of_birth` → date (like 2012-06-24)
- `passed` → true or false (like passed exam? Yes/No)

Real-Life Analogy:

- **INT** → Number of pencils in your box
- **VARCHAR** → Your name on the box
- **BOOLEAN** → Is the box open? (Yes/No)
- **DATE** → Date you bought the box
- **AUTO_INCREMENT** → Sticker number on each new box

🧠 Imagine: You run a school...

You have:

- A **register** for students
- A **register** for classes

You want to make sure:

1. No two students have the same ID

2. A student belongs to a valid class

Let's learn how MySQL helps us do this.

`PRIMARY KEY` – One & Only One

Think of a **Primary Key** as a **Roll Number** in school:

- It is **unique** (no two students have same roll no.)
- It is **not empty** (every student must have a roll no.)
- It **identifies** each row (just like roll no. identifies student)

Example:

```
id INT PRIMARY KEY
```

"This column will be used to **uniquely identify** each row in the table."

Commonly used on: `id` column
Often paired with: `AUTO_INCREMENT`

`UNIQUE` – No Duplicates Allowed

Use `UNIQUE` when you want a column to have **no repeated values**, but it's **not the main key**.

Example: Student's **Aadhaar number** or **email**

```
email VARCHAR(100) UNIQUE
```

"No two students can have the same email."

Unlike `PRIMARY KEY`, `UNIQUE` can be **empty/null** (unless you say NOT NULL)

`FOREIGN KEY` – Link Between Tables

Think of `FOREIGN KEY` like a **reference**.

Let's say:

- You have a `students` table.
- You also have a `classes` table with valid class codes.

You want to **make sure** that every student is in a **valid class**.

We do that using `FOREIGN KEY`.

Example Tables:

`classes` table:

```
CREATE TABLE classes (
  id INT PRIMARY KEY,
  class_name VARCHAR(10)
);
```

`students` table:

```
CREATE TABLE students (
  id INT PRIMARY KEY,
  name VARCHAR(100),
  class_id INT,
  FOREIGN KEY (class_id) REFERENCES classes(id)
);
```

This means: "The `class_id` in students must match the `id` in classes."

If you try to insert a student with a fake class ID, MySQL will stop you ❌

`CREATE TABLE`

Let's say we want a **students** table.

Basic Syntax:

```
CREATE TABLE students (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  class VARCHAR(10),
  marks INT
);
```

How to Add Data Using `INSERT INTO`

Imagine:

You've created an **empty register** (table) called `students`.

Now you want to **fill in the details** — like writing a student's name, class, and marks.

For that, we use the **`INSERT INTO`** command.

Basic Format:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

Example:

We have this table called `students`:

id	name	class	marks

Now let's **add Meena** (class 5A, marks 85).

```
INSERT INTO students (name, class, marks)
VALUES ('Meena', '5A', 85);
```

Says:
"Hey MySQL, in the `students` table, add a new student —
her name is Meena, she's in class 5A, and got 85 marks."

We **don't write `id`** because it's **AUTO_INCREMENT** — MySQL will fill it automatically.

Add More Students:

```
INSERT INTO students (name, class, marks)
VALUES
  ('Raj', '6B', 90),
  ('Kavita', '5A', 78),
  ('Aryan', '6A', 88);
```

This will add 3 students in one go!

Format Tips:

- Always put **text** in **single quotes**: `'Meena'`
- Numbers don't need quotes: `85`
- Order of values **must match** the columns

Insert All Columns (including ID – not common):

```
INSERT INTO students (id, name, class, marks)
VALUES (10, 'Rita', '7A', 95);
```

🛑 This only works **if you are manually adding the ID**.
Not needed if `id` is `AUTO_INCREMENT`.

Insert into Other Tables

Let's say we have a `teachers` table:

```
CREATE TABLE teachers (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  subject VARCHAR(50),
  email VARCHAR(100)
);
```

You can insert like this:

```
INSERT INTO teachers (name, subject, email)
VALUES ('Mr. Sharma', 'Math', 'sharma@example.com');
```

Insert Multiple Rows in One Go (Multi-Value INSERT)

🧠 Imagine:

You're the teacher. You want to write **3 students** into the register at the same time, instead of one by one.

This is what **multi-row insert** does in SQL.

Syntax (Very Easy!):

```
INSERT INTO table_name (column1, column2, ...)
VALUES
  (value1a, value2a, ...),
  (value1b, value2b, ...),
  (value1c, value2c, ...);
```

Example 1: Students Table

Let's say this is your table:

```
CREATE TABLE students (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  class VARCHAR(10),
  marks INT
);
```

Now insert 3 students at once:

```
INSERT INTO students (name, class, marks)
VALUES
  ('Rani', '5B', 81),
  ('Ayaan', '6A', 93),
  ('Simran', '7C', 76);
```

This tells MySQL:
“Please add Rani, Ayaan, and Simran in one go.”

MySQL will give them `id` numbers automatically like:

- Rani → ID 1
- Ayaan → ID 2
- Simran → ID 3

Why Use Multi-Value Insert?

Why?	Benefit
Saves time	Only one query instead of three
Faster for big data	Better performance
Cleaner code	Easy to read and manage

Important Rules

- Keep the **column order** and **value order** the same
- Every row is inside `()` and separated by `,`
- Text = single quotes `'text'`
- Numbers = no quotes `123`

Example 2: Teachers Table

```sql
CREATE TABLE teachers (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  subject VARCHAR(50),
  email VARCHAR(100)
);


INSERT INTO teachers (name, subject, email)
VALUES
  ('Mr. Sharma', 'Math', 'sharma@example.com'),
  ('Ms. Gupta', 'Science', 'gupta@example.com'),
  ('Mr. Khan', 'English', 'khan@example.com');
```

UPDATE Command in MySQL (Fix or Change Data)

Imagine:

You wrote a student's marks wrong in the register.
Now you want to **correct it**.

Use: `UPDATE`
It's like saying:

> "Go to the student row where name is 'Rani' and change her marks to 91."

Syntax of UPDATE:

```
UPDATE table_name
SET column_name = new_value
WHERE condition;
```

Simple Meaning:

- **UPDATE** → Which table to change
- **SET** → What to change and what to set it to
- **WHERE** → Which row(s) to update

Example 1: Update Marks of One Student

```
UPDATE students
SET marks = 91
WHERE name = 'Rani';
```

Says:

> "In the `students` table, find the student whose name is `Rani`, and set her marks to 91."

After update:

id	name	class	marks
1	Rani	5B	91

WARNING: Never forget the `WHERE` clause!

If you run:

```
UPDATE students SET marks = 0;
```

This will change **marks for ALL students** to 0.

Always use `WHERE` to tell **which row** to update.

Example 2: Change Class of a Student by ID

```
UPDATE students
SET class = '6A'
WHERE id = 3;
```

"Find the student with ID 3 and change their class to 6A."

Example 3: Update Multiple Columns

```
UPDATE students
SET name = 'Simran Kaur', marks = 84
WHERE name = 'Simran';
```

"Find Simran and change her name AND her marks."

Example 4: Update Teachers Table

```
UPDATE teachers
SET subject = 'Computer Science'
WHERE name = 'Ms. Gupta';
```

Ms. Gupta now teaches Computer Science!

DELETE Command in MySQL (Remove Data)

Why use `DELETE`?

Sometimes:

- A student was added by mistake ❌
- A teacher left the school 👋
- You want to clear old or wrong data 🧹

You can **delete one row** or **many rows**, depending on your need.

Basic Syntax:

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

Meaning:

- `DELETE FROM` → Which table
- `WHERE` → Which row(s) to delete

Very Important:

> ❌ If you **forget the `WHERE`**, it will delete **ALL rows** from the table!

Example 1: Delete One Student by Name

```
DELETE FROM students
WHERE name = 'Rani';
```

Says:

> "Go to the `students` table and remove the row where the name is Rani."

Example 2: Delete Student by ID

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

Delete the student whose ID is 3.

Example 3: Delete Multiple Rows (All from Class 6A)

```
DELETE FROM students
WHERE class = '6A';
```

This deletes **all students** from class 6A.

Example 4: Delete All Rows (Be Very Careful!)

```
DELETE FROM students;
```

This will delete **every student** in the table — use only when you're sure!

What is TRUNCATE?

- `TRUNCATE` is used to **remove all rows** from a table — **instantly**.
- It's like saying:

> “Throw away everything inside the table, but keep the table ready to reuse.”

Syntax:

```
TRUNCATE TABLE table_name;
```

Simple Example:

You created a `students` table and added 10 rows.

Now you want to:

- Keep the table
- But remove **all student records**
- And **reset** the auto-increment ID back to 1

Use this:

```
TRUNCATE TABLE students;
```

It will:

- Delete all rows
- Reset `AUTO_INCREMENT` ID to 1
- But keep the **table structure** (columns stay)

TRUNCATE vs DELETE (What's the difference?)

Feature	DELETE	TRUNCATE
Deletes some rows?	✅ Yes (with `WHERE`)	❌ No (deletes **all** only)
Deletes all rows?	✅ Yes (but row by row)	✅ Yes (very fast!)
Can use `WHERE`?	✅ Yes	❌ No
Resets auto-increment ID?	❌ No	✅ Yes (back to 1)
Fast?	❌ Slower (deletes one by one)	✅ Very fast (clears directly)
Can be undone (with rollback)?	✅ Sometimes (if in transaction)	❌ No (permanent!)

Think of it like this:

Command	Real-Life Example
`DELETE`	Erase some lines in your notebook
`TRUNCATE`	Tear out all the pages — but keep the notebook
`DROP`	Throw the whole notebook into the trash

Be Careful:

- `TRUNCATE` cannot delete **just one or a few rows**.
- You **can't undo it** — once it's gone, it's gone!
- It **resets IDs** — so if you had a student with ID 10, and you truncate, the next added student will get ID 1.

What does `SELECT *` do?

It means:

> "Hey MySQL, show me **all the columns** and **all the data** from this table."

It's like saying:
"Open the whole register and let me read everything!"

Syntax:

```
SELECT * FROM table_name;
```

Example 1: See All Students

Let's say you have a table called `students`.

Just run:

```
SELECT * FROM students;
```

This will show something like:

id	name	class	marks
1	Rani	5B	91
2	Aryan	6A	95
3	Simran	7C	88

What does * mean?

The * means **"all columns"** — so you'll see:

- Every row (record)
- Every column (id, name, class, marks, etc.)

When should you use SELECT *?

Use it when...
You want to **see everything** in a table
You are just **testing** or exploring the data
You are **learning SQL** and want simple output

When NOT to use SELECT *

If your table has **too many columns**, or you want only a few, it's better to be specific:

```
SELECT name, class FROM students;
```

This only shows:

name	class
Rani	5B
Aryan	6A
Simran	7C