Introduction to Databases

What is a Database?

A **Database** is an organized collection of structured information or data, typically stored electronically in a computer system. Its a way to store data in a format that is easily accessible

> Example: Think of a library where books are organized by topic, author, and title – that's essentially what a database does with data.

Why Do We Need Databases?

- To **store**, **manage**, and **retrieve** large amounts of data efficiently.
- To **prevent data duplication** and maintain **data integrity**.
- To allow **multiple users** to access and manipulate data simultaneously.

What is SQL?

SQL (Structured Query Language) is the standard programming language used to communicate with and manipulate databases.

Common operations using SQL:

- `INSERT` – Add new records (CREATE)
- `SELECT` – Retrieve data (READ)
- `UPDATE` – Modify existing data (UPDATE)
- `DELETE` – Remove records (DELETE)

These operations are usually referred to as CRUD Operations. CRUD stands for Create, Read, Update, and Delete — the four basic operations used to manage data in a database.

Comparison with Excel

Databases and Excel may seem similar at first, but they work differently under the hood.

- In Excel, a **sheet** is like a **table** in a database.
- Each **row** in the sheet is similar to a **record** (or entry) in a database table.
- Each **column** in Excel corresponds to a **field** (or attribute) in a table.
- Excel stores all data in one file, whereas a database can contain multiple related tables.
- In databases, you can define strict data types and rules, which Excel doesn't enforce.
- Unlike Excel, databases allow complex querying, relationships between tables, and secure multi-user access.

> Think of a database as a more powerful, structured, and scalable version of Excel for data management.

Relational vs Non-relational Databases

Relational databases store data in structured tables with predefined schemas and relationships between tables (e.g., MySQL, PostgreSQL). Non-relational databases (NoSQL) use flexible formats like documents, key-value pairs, or graphs, and don't require a fixed schema (e.g., MongoDB, Firebase). Relational is ideal for structured data and complex queries, while non-relational is better for scalability and unstructured data.

Feature	Relational (SQL)	Non-relational (NoSQL)
Structure	Tables (rows & cols)	Documents, Key-Value
Language	SQL	Varies (Mongo Query, etc.)

Feature	Relational (SQL)	Non-relational (NoSQL)
Schema	Fixed schema	Flexible schema
Examples	MySQL, PostgreSQL	MongoDB, Firebase

What is DBMS?

A **Database Management System (DBMS)** is software that interacts with users, applications, and the database itself to capture and analyze data. It allows users to create, read, update, and delete data in a structured way.

- **Examples**: MySQL, PostgreSQL, Oracle Database, SQLite.
- **Functions**: Data storage, retrieval, security, backup, and recovery.

What is MySQL?

MySQL is an open-source relational database management system (RDBMS) that uses SQL.

- Widely used in web development
- High performance and reliability
- Powers platforms like WordPress, Facebook (early days), and YouTube

Real-World Use Cases

- **E-commerce websites** to store customer orders and product listings
- **Banking systems** to handle transactions securely
- **Social networks** to manage user data, messages, and posts

Summary

- Databases are essential for structured data storage and retrieval.
- SQL is the language used to interact with relational databases.
- MySQL is a popular and powerful SQL-based database system.
- Understanding databases is a must-have skill for any developer or data analyst.

Creating a Database in MySQL

To start working with MySQL, the first step is to create a database.

Syntax

```
CREATE DATABASE database_name;
```

Example

```
CREATE DATABASE student_db;
```

This command creates a new database named `student_db`.

Tips

- Database names should be **unique**.
- Avoid using spaces or special characters.
- Use lowercase and underscores (`_`) for better readability (e.g., `employee_records`).

Viewing All Databases

To see all available databases:

```
SHOW DATABASES;
```

Switching to a Database

Before working with tables, you must select the database:

```
USE student_db;
```

Dropping a Database

To delete an existing database (this action is irreversible):

```
DROP DATABASE database_name;
```

Example

```
DROP DATABASE student_db;
```

> Be very careful! This will permanently delete all data and tables in the database.

Creating a Table in MySQL

Once you have selected a database, you can create tables to organize and store data.

Syntax

```
CREATE TABLE table_name (
  column1 datatype constraints,
  column2 datatype constraints,
  ...
);
```

Example

```
CREATE TABLE students (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL DEFAULT 'No Name',
  age INT,
  email VARCHAR(100) UNIQUE,
  admission_date DATE
);
```

Explanation

- `id INT AUTO_INCREMENT PRIMARY KEY` – A unique identifier for each student that auto-increments.
- `name VARCHAR(100) NOT NULL` – Name must be provided.
- `age INT` – Stores numeric values for age.
- `email VARCHAR(100) UNIQUE` – Each email must be different.
- `admission_date DATE` – Stores the date of admission.

Commonly Used Data Types

- `INT` – Whole numbers (e.g., age, quantity)
- `VARCHAR(n)` – Variable-length string (e.g., names, emails)
- `TEXT` – Long text strings (e.g., descriptions)
- `DATE` – Stores date values (YYYY-MM-DD)
- `DATETIME` – Stores date and time values
- `BOOLEAN` – Stores `TRUE` or `FALSE`

Common Constraints

- `PRIMARY KEY` – Uniquely identifies each record
- `NOT NULL` – Ensures the column cannot be left empty
- `UNIQUE` – Ensures all values in a column are different
- `AUTO_INCREMENT` – Automatically increases numeric values
- `DEFAULT` – Sets a default value for the column
- `FOREIGN KEY` – Enforces relationships between tables

View All Tables

```
SHOW TABLES;
```

View Table Structure

```
DESCRIBE students;
```

Viewing Table Data

```
SELECT * FROM students;
```

Tables are the backbone of any relational database. A well-structured table leads to efficient data management and fewer issues later on.

Modifying a Table in MySQL

As your application grows or requirements change, you may need to make changes to existing tables. MySQL provides several `ALTER` and related statements for such modifications.

Renaming a Table

Use the `RENAME TABLE` command to change the name of an existing table.

```
RENAME TABLE old_table_name TO new_table_name;
```

Dropping a Table

To permanently delete a table and all of its data:

```
DROP TABLE table_name;
```

Renaming a Column

To rename a column in an existing table:

```
ALTER TABLE table_name RENAME COLUMN old_column_name TO new_column_name;
```

Dropping a Column

To remove a column from a table:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

Adding a Column

To add a new column to an existing table:

```
ALTER TABLE table_name ADD COLUMN column_name datatype constraints;
```

Example:

```
ALTER TABLE students ADD COLUMN gender VARCHAR(10);
```

Modifying a Column

To change the data type or constraints of an existing column:

```
ALTER TABLE table_name MODIFY COLUMN column_name new_datatype new_constraints;
```

Example:

```
ALTER TABLE students MODIFY COLUMN name VARCHAR(150) NOT NULL;
```

Changing the Order of Columns

To change the order of columns in a table, you can use the `MODIFY` command with the `AFTER` keyword:

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype AFTER
another_column_name;
```

Always review changes on production databases carefully. Use tools like `DESCRIBE table_name` to verify structure before and after modifications.

How to Insert Rows into a Table in MySQL

Inserting data into a MySQL table can be done in two ways: inserting one row at a time or inserting multiple rows at once. Below are the steps to create a new database, create a table, and insert data into it.

1. Create a New Database

```
CREATE DATABASE schooldb;
```

2. Select the Database

```
USE schooldb;
```

3. Create the `student` Table

```
CREATE TABLE student (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    grade VARCHAR(10),
    date_of_birth DATE
);
```

4. Insert Data into the Table

Insert One Row at a Time

```
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (1, 'Ayesha Khan', 16, '10th', '2007-05-15');
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (2, 'Ravi Sharma', 17, '11th', '2006-03-22');
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (3, 'Meena Joshi', 15, '9th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (4, 'Arjun Verma', 18, '12th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (5, 'Sara Ali', 16, '10th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (6, 'Karan Mehta', 17, '11th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (7, 'Tanya Roy', 15, '9th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (8, 'Vikram Singh', 18, '12th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (9, 'Anjali Desai', 16, '10th', NULL);
INSERT INTO student (id, name, age, grade, date_of_birth) VALUES (10, 'Farhan Zaidi', 17, '11th', NULL);
```

Insert All Rows at Once

```
INSERT INTO student (id, name, age, grade) VALUES
(15, 'Ayesha Khan', 16, '10th'),
(25, 'Ravi Sharma', 17, '11th'),
(35, 'Meena Joshi', 15, '9th'),
(45, 'Arjun Verma', 18, '12th'),
(55, 'Sara Ali', 16, '10th'),
(65, 'Karan Mehta', 17, '11th'),
(75, 'Tanya Roy', 15, '9th'),
(85, 'Vikram Singh', 18, '12th'),
(95, 'Anjali Desai', 16, '10th'),
(105, 'Farhan Zaidi', 17, '11th');
```

5. Verify the Inserted Records

To check all the data in the table:

```
SELECT * FROM student;
```

This will display all the records in the `student` table.

How to Select Data in MySQL

Lets now understand how to query data from a table in MySQL using the `SELECT` statement, filter results using the `WHERE` clause, and apply different comparison operators including how to work with `NULL` values.

---

1. Basic `SELECT` Statement

To retrieve all data from the `student` table:

```
SELECT * FROM student;
```

To retrieve specific columns (e.g., only `name` and `grade`):

```
SELECT name, grade FROM student;
```

---

2. Using the `WHERE` Clause

The `WHERE` clause is used to filter rows based on a condition.

Example: Students in 10th grade

```
SELECT * FROM student WHERE grade = '10th';
```

Example: Students older than 16

```
SELECT * FROM student WHERE age > 16;
```

3. Comparison Operators in MySQL

Operator	Description	Example
`=`	Equals	`WHERE age = 16`
`!=`	Not equal to	`WHERE grade != '12th'`
`<>`	Not equal to (alternative)	`WHERE grade <> '12th'`
`>`	Greater than	`WHERE age > 16`
`<`	Less than	`WHERE age < 17`
`>=`	Greater than or equal to	`WHERE age >= 16`
`<=`	Less than or equal to	`WHERE age <= 18`
`BETWEEN`	Within a range (inclusive)	`WHERE age BETWEEN 15 AND 17`
`IN`	Matches any in a list	`WHERE grade IN ('10th', '12th')`
`NOT IN`	Excludes list items	`WHERE grade NOT IN ('9th', '11th')`
`LIKE`	Pattern matching	`WHERE name LIKE 'A%'` (names starting with A)
`NOT LIKE`	Pattern not matching	`WHERE name NOT LIKE '%a'` (names not ending in a)

4. Handling `NULL` Values

What is `NULL`?

`NULL` represents missing or unknown values. It is **not** equal to `0`, empty string, or any other value.

Common Mistake (Incorrect):

```
-- This will not work as expected
SELECT * FROM student WHERE grade = NULL;
```

Correct Ways to Handle NULL

Condition	Correct Syntax
Is NULL	`WHERE grade IS NULL`
Is NOT NULL	`WHERE grade IS NOT NULL`

Example: Select students with no grade assigned

```
SELECT * FROM student WHERE grade IS NULL;
```

Example: Select students who have a grade

```
SELECT * FROM student WHERE grade IS NOT NULL;
```

---

5. Combining Conditions

You can use `AND`, `OR`, and parentheses to combine conditions.

Example: Students in 10th grade and older than 16

```sql
SELECT * FROM student WHERE grade = '10th' AND age > 16;
```

Example: Students in 9th or 12th grade

```sql
SELECT * FROM student WHERE grade = '9th' OR grade = '12th';
```

Example: Complex conditions

```sql
SELECT * FROM student
WHERE (grade = '10th' OR grade = '11th') AND age >= 16;
```

---

6. Sorting Results with `ORDER BY`

Sort by age in ascending order:

```sql
SELECT * FROM student ORDER BY age ASC;
```

Sort by name in descending order:

```sql
SELECT * FROM student ORDER BY name DESC;
```

---

7. Limiting Results with `LIMIT`

Get only 5 rows:

```sql
SELECT * FROM student LIMIT 5;
```

Get 5 rows starting from the 3rd (offset 2):

```
SELECT * FROM student LIMIT 2, 5;
```

You’re right! The `_` wildcard is very handy for matching **dates**, especially when you’re looking for values at a **specific position** (like a specific day or month). Let’s expand the section accordingly:

---

8. Using Wildcards with `LIKE`

Wildcards are used with the `LIKE` operator to search for patterns. They’re helpful when you’re not exactly sure about the full value, or you want to match based on structure or partial content.

There are two wildcards in MySQL:

- `%` – Matches **zero or more characters**
- `_` – Matches **exactly one character**

Example: Names starting with `'A'`

```
SELECT * FROM students
WHERE name LIKE 'A%';
```

This finds any name that **starts with ‘A’**, like `Aakash`, `Ananya`, `Aryan`.

Matching Dates with `_` Wildcard

The `_` wildcard is useful for matching specific patterns in **date strings**, especially in `YYYY-MM-DD` format.

Let’s say you want to find records from the **5th day of any month**:

```
SELECT * FROM attendance
WHERE date LIKE '____-__-05';
```

Explanation:

- `____` matches any year (4 characters)
- `__` matches any month (2 characters)
- `05` is the 5th day

You're basically telling MySQL:

> "Give me all rows where the date ends with `-05` — which means the 5th of any month, any year."

More Date Pattern Examples

Pattern to be Matched	Matches
`'2025-05-%'`	Any day in May 2025
`'2024-12-__'`	All 2-digit days in December 2024
`'____-01-01'`	1st January of any year
`'202_-__-__'`	Any date in the 2020s decade
`'____-__-3_'`	All dates from day 30 to 39 (not valid, but works syntactically)

Quick Recap: `LIKE` Wildcard Matching

Pattern	Meaning
`'A%'`	Starts with A
`'%sh'`	Ends with sh

Pattern	Meaning
`'%ar%'`	Contains “ar”
`'R____'`	5-letter name starting with R
`'____-__-05'`	Dates with day = 05

How to Update Data in a MySQL Table

Today we will learn about how to modify existing data in a table using the `UPDATE` statement. We will see how to use the `SET` keyword and how to use the `WHERE` clause to target specific rows.

1. Basic Syntax

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

- `UPDATE` : Specifies the table you want to modify.
- `SET` : Assigns new values to columns.
- `WHERE` : Filters which rows should be updated. **Always include a WHERE clause unless you want to update all rows.**

2. Example Table

Assume the following `student` table:

id	name	age	grade
1	Ayesha Khan	16	10th
2	Ravi Sharma	17	11th
3	Meena Joshi	15	9th

3. Update a Single Row

Example: Change the grade of student with `id = 2` to `12th`

```
UPDATE student
SET grade = '12th'
WHERE id = 2;
```

4. Update Multiple Columns

Example: Change `age` to 17 and `grade` to ‘10th’ for `id = 3`

```
UPDATE student
SET age = 17, grade = '10th'
WHERE id = 3;
```

5. Update All Rows

Be careful when updating without a `WHERE` clause.

Example: Set all students to age 18

```
UPDATE student
SET age = 18;
```

Warning: This will modify **every row** in the table.

6. Conditional Update with Comparison Operators

Example: Promote all students in 9th grade to 10th grade

```
UPDATE student
SET grade = '10th'
WHERE grade = '9th';
```

Example: Increase age by 1 for students younger than 18

```
UPDATE student
SET age = age + 1
WHERE age < 18;
```

7. Update Using `IS NULL`

Example: Set default grade to ‘Unknown’ where grade is NULL

```
UPDATE student
SET grade = 'Unknown'
WHERE grade IS NULL;
```

8. Verify the Update

To check the results:

```
SELECT * FROM student;
```

How to Delete Data in a MySQL Table

This guide explains how to remove records from a table using the `DELETE` statement. It covers deleting specific rows using the `WHERE` clause and the consequences of deleting all rows.

---

1. Basic Syntax

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

- `DELETE FROM` : Specifies the table from which to remove rows.
- `WHERE` : Filters which rows should be deleted.

Important: If you omit the `WHERE` clause, **all rows** in the table will be deleted.

---

2. Example Table

Assume the following `student` table:

id	name	age	grade
1	Ayesha Khan	16	10th
2	Ravi Sharma	17	12th
3	Meena Joshi	15	9th

---

3. Delete a Specific Row

Example: Delete student with `id = 2`

```
DELETE FROM student
WHERE id = 2;
```

---

4. Delete Rows Based on a Condition

Example: Delete all students in 9th grade

```
DELETE FROM student
WHERE grade = '9th';
```

---

5. Delete Rows Using Comparison Operators

Example: Delete all students younger than 16

```
DELETE FROM student
WHERE age < 16;
```

---

6. Delete Rows Where a Column is NULL

Example: Delete students with no grade assigned

```
DELETE FROM student
WHERE grade IS NULL;
```

7. Delete All Rows (Use with Caution)

Example: Remove all data from the `student` table

```
DELETE FROM student;
```

This deletes all rows but **retains the table structure**.

8. Completely Remove the Table

To delete the table itself (not just the data), use:

```
DROP TABLE student;
```

This removes both the data and the table structure.

9. Verify After Deletion

Check the contents of the table:

```
SELECT * FROM student;
```

MySQL Tutorial: AUTOCOMMIT, COMMIT, and ROLLBACK

Now, we will explore how MySQL handles transactions using the `AUTOCOMMIT`, `COMMIT`, and `ROLLBACK` statements. Understanding these is essential for maintaining data integrity in your databases, especially in data science workflows where large and complex data operations are common.

What is a Transaction?

A **transaction** is a sequence of one or more SQL statements that are executed as a single unit. A transaction has four key properties, known as ACID:

- **Atomicity**: All or nothing.
- **Consistency**: Valid state before and after.
- **Isolation**: Transactions do not interfere.
- **Durability**: Changes persist after commit.

AUTOCOMMIT

By default, MySQL runs in **autocommit mode**. This means that every SQL statement is treated as a separate transaction and is committed automatically right after it is executed.

Check Autocommit Status

```
SELECT @@autocommit;
```

Disable Autocommit

```
SET autocommit = 0;
```

This allows you to group multiple statements into a transaction manually.

Enable Autocommit

```
SET autocommit = 1;
```

COMMIT

The `COMMIT` statement is used to **permanently save** all the changes made in the current transaction.

Example

```
START TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;

COMMIT;
```

Once committed, the changes are visible to other sessions and are stored permanently in the database.

ROLLBACK

The `ROLLBACK` statement is used to **undo** changes made in the current transaction. It is useful if something goes wrong or a condition is not met.

Example

```
START TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE id = 1;

-- An error or condition check fails here
ROLLBACK;
```

After a rollback, all changes since the start of the transaction are discarded.

Summary Table

Statement	Description
AUTOCOMMIT	Automatically commits every query
SET autocommit = 0	Disables autocommit mode
COMMIT	Saves all changes in a transaction
ROLLBACK	Reverts all changes in a transaction

Best Practices

- Always use transactions when performing multiple related operations.
- Disable autocommit when working with critical data updates.
- Rollback if any step in your transaction fails.
- Test your transactions thoroughly before running them on production data.

MySQL Tutorial: Getting Current Date and Time

MySQL provides built-in functions to retrieve the **current date**, **time**, and **timestamp**. These are commonly used in data logging, time-based queries, and tracking records in data science workflows.

---

1. `CURRENT_DATE`

Returns the **current date** in `YYYY-MM-DD` format.

```
SELECT CURRENT_DATE;
```

Example Output:

```
2025-05-02
```

---

2. `CURRENT_TIME`

Returns the **current time** in `HH:MM:SS` format.

```
SELECT CURRENT_TIME;
```

Example Output:

```
14:23:45
```

3. `CURRENT_TIMESTAMP` (or `NOW()`)

Returns the **current date and time**.

```
SELECT CURRENT_TIMESTAMP;
-- or
SELECT NOW();
```

Example Output:

```
2025-05-02 14:23:45
```

This is especially useful for storing creation or update times in records.

4. `LOCALTIME` and `LOCALTIMESTAMP`

These are synonyms for `NOW()` and return the current date and time.

```
SELECT LOCALTIME;
SELECT LOCALTIMESTAMP;
```

These functions return the local date and time of the MySQL server, not the client's time zone.

Important Clarification:

The "local" in LOCALTIME refers to the time zone configured on the MySQL server, not the user's system.

You can check the current server time zone using:

5. Using in Table Inserts

You can use `NOW()` or `CURRENT_TIMESTAMP` to auto-fill date-time columns.

```
INSERT INTO logs (event, created_at)
VALUES ('data_import', NOW());
```

6. Date and Time Functions Recap

Function	Returns	Example Output
`CURRENT_DATE`	Date only	`2025-05-02`
`CURRENT_TIME`	Time only	`14:23:45`
`NOW()`	Date and time	`2025-05-02 14:23:45`
`CURRENT_TIMESTAMP`	Date and time	`2025-05-02 14:23:45`
`LOCALTIME`	Date and time	`2025-05-02 14:23:45`

Best Practices

- Use `CURRENT_TIMESTAMP` for record timestamps.
- Use `NOW()` in queries to filter records by current time.
- Avoid relying on system time for business logic; prefer database time for consistency.

SQL Tutorial: Deep Dive into Constraints

Constraints in SQL are rules applied to table columns to enforce data integrity, consistency, and validity. They restrict the type of data that can be inserted into a table and help prevent invalid or duplicate entries.

Why Use Constraints?

- Ensure **data quality** and **reliability**
- Prevent **invalid**, **duplicate**, or **null** data
- Maintain **business rules** directly in the database layer

---

1. `NOT NULL` Constraint

Ensures that a column cannot contain `NULL` values.

```sql
CREATE TABLE employees (
    id INT NOT NULL,
    name VARCHAR(100) NOT NULL
);
```

Use Case:

Make sure critical fields like `id`, `name`, or `email` are always filled.

---

2. `UNIQUE` Constraint

Ensures that all values in a column are **distinct** (no duplicates).

```
CREATE TABLE users (
    username VARCHAR(50) UNIQUE,
    email VARCHAR(100) UNIQUE
);
```

Use Case:

Prevent duplicate usernames or email addresses.

> Note: A table can have **multiple UNIQUE constraints**, but only **one PRIMARY KEY**.

---

3. `DEFAULT` Constraint

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

```
CREATE TABLE products (
    name VARCHAR(100),
    status VARCHAR(20) DEFAULT 'in_stock'
);
```

Use Case:

Auto-fill common values to reduce data entry effort and prevent missing data.

---

4. `CHECK` Constraint

Validates that values in a column meet a specific condition.

```
CREATE TABLE accounts (
    id INT,
    balance DECIMAL(10,2) CHECK (balance >= 0)
);
```

Use Case:

Enforce business rules such as non-negative balances or valid age ranges.

> Note: MySQL versions before 8.0 parsed `CHECK` but did **not enforce** it. From MySQL 8.0 onwards, `CHECK` constraints are **enforced**.

---

5. Naming Constraints

You can give **explicit names** to constraints. This makes them easier to reference, especially when altering or dropping them later.

```
CREATE TABLE students (
    roll_no INT PRIMARY KEY,
    age INT CONSTRAINT chk_age CHECK (age >= 5),
    email VARCHAR(100) UNIQUE
);
```

Benefits of Named Constraints:

- Improves clarity and debugging
- Useful when using `ALTER TABLE` to drop constraints

---

6. Constraint Recap Table

Constraint	Purpose	Enforced By MySQL	Custom Name Support
`NOT NULL`	Disallow null values	Yes	Yes
`UNIQUE`	Disallow duplicate values	Yes	Yes
`DEFAULT`	Set default value if none given	Yes	No
`CHECK`	Enforce value conditions	Yes (MySQL 8.0+)	Yes

Best Practices

- Use constraints to **enforce critical rules** in the database layer.
- Always name important constraints for easier maintenance.
- Prefer constraints over application-side validation for core rules.
- Test `CHECK` constraints carefully to ensure compatibility and enforcement in your MySQL version.

MySQL Foreign Key Tutorial

This guide demonstrates how to create a database, define tables, and use **foreign keys** to establish relationships between them.

1. Create a Database

```
CREATE DATABASE school;
USE school;
```

2. Create Tables

We'll create two tables:

- `students`
- `classes`

Each student will belong to a class, creating a **one-to-many** relationship (one class has many students).

Create `classes` Table

```
CREATE TABLE classes (
    class_id INT AUTO_INCREMENT PRIMARY KEY,
    class_name VARCHAR(50) NOT NULL
);
```

Create `students` Table

```
CREATE TABLE students (
    student_id INT AUTO_INCREMENT PRIMARY KEY,
    student_name VARCHAR(100) NOT NULL,
    class_id INT,
    FOREIGN KEY (class_id) REFERENCES classes(class_id)
        ON UPDATE CASCADE
        ON DELETE SET NULL
);
```

---

3. Insert Sample Data

Insert into `classes`

```
INSERT INTO classes (class_name) VALUES ('Mathematics'), ('Science'), ('History');
```

Insert into `students`

```
INSERT INTO students (student_name, class_id) VALUES
('Alice', 1),
('Bob', 2),
('Charlie', 1);
```

---

4. Explanation of Foreign Key Behavior

In the `students` table:

- `class_id` is a **foreign key**.
- It references `class_id` in the `classes` table.

- `ON DELETE SET NULL` : If a class is deleted, the related students will have `class_id` set to `NULL` .
- `ON UPDATE CASCADE` : If a class ID changes, it will update automatically in the `students` table.

---

5. View the Relationships

To check the foreign key constraints:

```
SHOW CREATE TABLE students;
```

To see all foreign keys in the current database:

```
SELECT
    table_name,
    column_name,
    constraint_name,
    referenced_table_name,
    referenced_column_name
FROM
    information_schema.key_column_usage
WHERE
    referenced_table_name IS NOT NULL
    AND table_schema = 'school';
```

Understanding `ON UPDATE CASCADE` and `ON DELETE SET NULL`

When you define a **foreign key** in MySQL, you can specify what should happen to the child table when the parent table is **updated** or **deleted**. These are called **referential actions**.

---

1. `ON UPDATE CASCADE`

Definition: If the value in the parent table (i.e., the referenced column) is **updated**, the corresponding foreign key value in the child table is **automatically updated** to match.

Example: Suppose we update a `class_id` in the `classes` table:

```
UPDATE classes SET class_id = 10 WHERE class_id = 1;
```

Then all students in the `students` table whose `class_id` was `1` will automatically be updated to `10`.

---

2. `ON DELETE SET NULL`

Definition: If a row in the parent table is **deleted**, the foreign key in the child table will be set to **NULL** for all matching rows.

Example: If we delete a class from the `classes` table:

```
DELETE FROM classes WHERE class_id = 2;
```

Then all students in the `students` table who were in class `2` will have their `class_id` set to `NULL`, indicating that they are no longer assigned to a class.

Why Use These Options?

- `ON UPDATE CASCADE` is useful when the primary key of the parent table might change (rare but possible).
- `ON DELETE SET NULL` is helpful when you want to **preserve child records** but indicate that the relationship has been broken.

Alternatives

- `ON DELETE CASCADE` : Deletes the child rows when the parent row is deleted.
- `ON DELETE RESTRICT` : Prevents deletion if any child rows exist.
- `ON DELETE NO ACTION` : Same as RESTRICT in MySQL.
- `ON DELETE SET DEFAULT` : Not supported in MySQL (but available in some other DBMSs).

MySQL Joins – A Simple Guide

When we have data split across multiple tables, we use **joins** to combine that data and get the full picture.

Let’s say we have two tables:

`students`

id	name
1	Alice
2	Bob

`marks`

student_id	subject	score
1	Math	95
2	Math	88
2	Science	90

Now let’s see the most common types of joins.

1. INNER JOIN

We are telling MySQL to include only the rows that have matching values in both tables.

```
SELECT students.name, marks.subject, marks.score
FROM students
INNER JOIN marks ON students.id = marks.student_id;
```

This will show only students who have marks recorded. If a student has no marks, they will not appear in the result.

2. LEFT JOIN (or LEFT OUTER JOIN)

We are telling MySQL to include **all students**, even if they don’t have any marks. If there’s no match in the `marks` table, it will show `NULL`.

```
SELECT students.name, marks.subject, marks.score
FROM students
LEFT JOIN marks ON students.id = marks.student_id;
```

This is useful when we want to list all students, and show marks only if available.

3. RIGHT JOIN (or RIGHT OUTER JOIN)

We are telling MySQL to include **all rows from the right table (`marks`)**, even if the student is missing from the `students` table.

```
SELECT students.name, marks.subject, marks.score
FROM students
RIGHT JOIN marks ON students.id = marks.student_id;
```

This is rarely used unless we expect some marks that don’t have a student record.

5. CROSS JOIN

We are telling MySQL to combine every row in the first table with every row in the second table.

```
SELECT students.name, marks.subject
FROM students
CROSS JOIN marks;
```

Use this only when you really want **all combinations** – it can produce a lot of rows.

Summary

Join Type	**What it does**
INNER JOIN	Only rows with a match in both tables
LEFT JOIN	All rows from the left table, with matched data if any
RIGHT JOIN	All rows from the right table, with matched data if any
CROSS JOIN	All combinations of rows from both tables

Using `UNION` in MySQL

The `UNION` operator is used to **combine the result sets of two or more `SELECT` statements** into a single result.

It helps when:

- You're pulling **similar data** from different tables.
- You want to **merge results** from multiple queries into one list.

When `UNION` Works

1. **Same number of columns** in all `SELECT` statements.
2. **Compatible data types** in corresponding columns.
3. Columns will be matched **by position**, not by name.

```
SELECT name, city FROM customers
UNION
SELECT name, city FROM vendors;
```

This combines names and cities from both tables into a single result.

When `UNION` Doesn't Work

- If the number of columns is different:

```
-- This will throw an error
SELECT name, city FROM customers
```

```
UNION
SELECT name FROM vendors;
```

- If the data types don’t match:

```
-- Error if 'age' is an integer and 'name' is a string
SELECT age FROM users
UNION
SELECT name FROM students;
```

MySQL will complain that the columns can’t be matched due to type mismatch.

`UNION` vs `UNION ALL`

By default, `UNION` removes **duplicate rows**. If you want to **keep duplicates**, use `UNION ALL`:

```
SELECT name FROM students
UNION ALL
SELECT name FROM alumni;
```

Use `UNION` if:

- You want a **clean list** without duplicates.

Use `UNION ALL` if:

- You want **performance** and don’t care about duplicates.
- Or you expect **duplicate values** and want to preserve them.

Practical Use Case Example

Let’s say you have two tables: `students_2023` and `students_2024`.

```
SELECT name, batch FROM students_2023
UNION
SELECT name, batch FROM students_2024;
```

This gives a combined list of all students across both years, without duplicates.

Sorting the Combined Result

You can sort the final output using `ORDER BY` at the end:

```
SELECT name FROM students_2023
UNION
SELECT name FROM students_2024
ORDER BY name;
```

MySQL Functions

Functions in MySQL are like built-in tools that help you work with data — whether you're manipulating text, doing math, or working with dates.

Let's explore some commonly used functions with simple examples.

1. `CONCAT()` – Join strings together

We are telling MySQL to combine two or more strings.

```
SELECT CONCAT('Hello', ' ', 'World') AS greeting;
-- Output: Hello World
```

You can also use it with columns:

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM users;
```

2. `NOW()` – Get the current date and time

We are telling MySQL to give us the current date and time.

```
SELECT NOW();
-- Output: 2025-05-03 14:20:45 (example)
```

3. `LENGTH()` – Find length of a string (in bytes)

```sql
SELECT LENGTH('Harry');
-- Output: 5
```

Useful for validations or checking string size.

---

4. `ROUND()` – Round numbers to a specific number of decimal places

```sql
SELECT ROUND(12.6789, 2);
-- Output: 12.68
```

---

5. `DATEDIFF()` – Difference between two dates (in days)

```sql
SELECT DATEDIFF('2025-06-01', '2025-05-01');
-- Output: 31
```

Comprehensive List of Useful MySQL Functions

Function	Description	Example Usage
`CONCAT()`	Combine multiple strings	`CONCAT('A', 'B')` → `'AB'`
`LENGTH()`	Length of a string (in bytes)	`LENGTH('Hi') → 2`

Function	Description	Example Usage
`CHAR_LENGTH()`	Number of characters in a string	`CHAR_LENGTH('हिंदी')` → `5`
`LOWER()`	Convert string to lowercase	`LOWER('MySQL')` → `mysql`
`UPPER()`	Convert string to uppercase	`UPPER('hello')` → `HELLO`
`REPLACE()`	Replace part of a string	`REPLACE('abc', 'b', 'x')` → `axc`
`TRIM()`	Remove leading/trailing spaces	`TRIM('  hello  ')` → `hello`
`NOW()`	Current date and time	`NOW()`
`CURDATE()`	Current date only	`CURDATE()`
`CURTIME()`	Current time only	`CURTIME()`
`DATE()`	Extract date from datetime	`DATE(NOW())`
`MONTHNAME()`	Get month name from date	`MONTHNAME('2025-05-03')` → `May`
`YEAR()`	Extract year from date	`YEAR(NOW())`
`DAY()`	Extract day of month	`DAY('2025-05-03')` → `3`
`DATEDIFF()`	Days between two dates	`DATEDIFF('2025-06-01', '2025-05-01')`
`ROUND()`	Round to decimal places	`ROUND(5.678, 2)` → `5.68`
`FLOOR()`	Round down to nearest whole number	`FLOOR(5.9)` → `5`
`CEIL()`	Round up to nearest whole number	`CEIL(5.1)` → `6`

Function	Description	Example Usage
`ABS()`	Absolute value	`ABS(-10)` → `10`
`MOD()`	Get remainder	`MOD(10, 3)` → `1`
`RAND()`	Random decimal between 0 and 1	`RAND()`
`IFNULL()`	Replace `NULL` with a default value	`IFNULL(NULL, 'N/A')` → `N/A`
`COALESCE()`	Return first non-NULL value in a list	`COALESCE(NULL, '', 'Hello')` → `''`
`COUNT()`	Count rows	`COUNT(*)`
`AVG()`	Average of a numeric column	`AVG(score)`
`SUM()`	Total sum of values	`SUM(score)`
`MIN()`	Smallest value	`MIN(score)`
`MAX()`	Largest value	`MAX(score)`

MySQL Views

A **View** in MySQL is like a **virtual table**. It doesn’t store data by itself but instead **shows data from one or more tables** through a saved SQL query.

You can use a view just like a regular table: `SELECT` from it, filter it, join it, etc.

Why Use Views?

- To **simplify complex queries** by giving them a name.
- To **hide sensitive columns** from users.
- To show **only specific rows/columns** from a table.
- To **reuse** common query logic across your app or reports.

Creating a View

Let’s say you have an `employees` table with lots of details, but you only want to show public employee info (name, department, and salary).

```
CREATE VIEW public_employees AS
SELECT name, department, salary
FROM employees;
```

You’re telling MySQL:

> “Create a view called `public_employees` that shows only name, department, and salary from the `employees` table.”

Using a View

Now you can query it like a normal table:

```
SELECT * FROM public_employees;
```

Or even apply filters:

```
SELECT * FROM public_employees
WHERE department = 'IT';
```

Updating a View

You can **replace** a view like this:

```
CREATE OR REPLACE VIEW public_employees AS
SELECT name, department
FROM employees;
```

Dropping (Deleting) a View

```
DROP VIEW public_employees;
```

This removes the view from the database.

Notes

- Views don’t store data. If the underlying table changes, the view reflects that automatically.

- **Not all views are updatable.** Simple views usually are (like those selecting from one table without grouping or joins), but complex ones may not allow `INSERT`, `UPDATE`, or `DELETE`.
- Views can make your queries **cleaner and easier to maintain**.

Example Use Case

You have this query used 5 times across your app:

```
SELECT customer_id, name, total_orders, status
FROM customers
WHERE status = 'active' AND total_orders > 5;
```

Instead of repeating it, just create a view:

```
CREATE VIEW top_customers AS
SELECT customer_id, name, total_orders, status
FROM customers
WHERE status = 'active' AND total_orders > 5;
```

Now just do:

```
SELECT * FROM top_customers;
```

MySQL Indexes

An **index** in MySQL is a data structure that makes **data retrieval faster**—especially when you're using `WHERE`, `JOIN`, `ORDER BY`, or searching large tables.

Think of an index like the index in a book: instead of reading every page, MySQL uses the index to jump straight to the relevant row(s).

Why Use Indexes?

- **Speed up queries** that search, filter, or sort data.
- **Improve performance** for frequent lookups or joins.
- **Enhance scalability** of your database over time.

How to Create an Index

1. Single Column Index

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

You're telling MySQL:

> "Create a quick lookup structure for the `email` column in the `users` table."

2. Multi-column (Composite) Index

```
CREATE INDEX idx_name_city ON users(name, city);
```

This is useful when your query filters on both `name` and `city` in that specific order.

How to Delete (Drop) an Index

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

You’re saying:

> “Remove the index named `idx_email` from the `users` table.”

When to Use Indexes

Use indexes when:

- A column is often used in `WHERE`, `JOIN`, or `ORDER BY` clauses.
- You’re searching by unique fields like `email`, `username`, or `ID`.
- You’re filtering large tables for specific values regularly.
- You want to improve performance of lookups and joins.

When Not to Use Indexes

Avoid adding indexes when:

- The table is **small** (MySQL can scan it quickly anyway).
- The column is rarely used in searches or filtering.
- You’re indexing a column with **very few unique values** (like a `gender` field with just `'M'` and `'F'`).
- You’re inserting or updating **very frequently**—indexes can slow down writes because they also need to be updated.

Viewing Existing Indexes

To list all indexes on a table:

```
SHOW INDEX FROM users;
```

Summary

Action	Syntax Example
Create index	`CREATE INDEX idx_name ON table(column);`
Delete index	`DROP INDEX idx_name ON table;`
List indexes	`SHOW INDEX FROM table;`

Indexes are essential for performance, but overusing them or indexing the wrong columns can actually hurt performance. Use them wisely based on how your data is queried.

Subqueries in MySQL

A **subquery** is a query nested inside another SQL query.
It helps you perform complex filtering, calculations, or temporary data shaping by breaking down the logic into smaller steps.

You can use subqueries in `SELECT`, `FROM`, or `WHERE` clauses.

What is a Subquery?

A subquery is enclosed in parentheses and returns data to be used by the outer query.
It can return: - A single value (scalar) - A row - A full table

---

Subquery in the `WHERE` Clause

Example: Employees who earn more than average

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

We are telling MySQL: **“First calculate the average salary, then return employees with salaries greater than that.”**

Subquery in the `FROM` Clause

Example: Department-wise average salary above 50,000

```
SELECT department, avg_salary
FROM (
    SELECT department, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department
) AS dept_avg
WHERE avg_salary > 50000;
```

We are telling MySQL: “**Create a temporary table of average salaries by department, then filter departments where the average is above 50,000.**”

Subquery in the `SELECT` Clause

Example: Count of projects per employee

```
SELECT name,
       (SELECT COUNT(*) FROM projects WHERE projects.employee_id = employees.id) AS
project_count
FROM employees;
```

This gives each employee along with the number of projects they are assigned to.

Correlated Subqueries

A **correlated subquery** depends on the outer query. It runs once for **each row** in the outer query.

Example: Employee earning more than department’s average

```
SELECT name, department, salary
FROM employees e
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department = e.department
);
```

We are telling MySQL: “**For each employee, compare their salary with the average salary of their department.**”

Types of Subqueries

Type	Description
Scalar Subquery	Returns a single value
Row Subquery	Returns one row with multiple columns
Table Subquery	Returns multiple rows and columns
Correlated Subquery	Refers to the outer query inside the subquery

---

When to Use Subqueries

- When your logic depends on **calculated values** (like averages or counts)
- When you need to **filter based on dynamic conditions**
- When you’re **breaking down complex queries** for readability

---

When to Avoid Subqueries

- When the same result can be achieved with a **JOIN**, which is often faster
- When the subquery is being **executed repeatedly** for every row (correlated subqueries on large tables)

Summary

Clause	Use Case
`WHERE`	Filter based on the result of a subquery
`FROM`	Use a subquery as a derived table
`SELECT`	Add related calculations inline

Subqueries are powerful for solving multi-step problems and isolating logic, but be mindful of performance when working with large data sets.

GROUP BY in MySQL

The `GROUP BY` clause is used when you want to **group rows that have the same values in specified columns**.
It's usually combined with **aggregate functions** like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, or `MIN()`.

We are telling MySQL:
"Group these rows together by this column, and then apply an aggregate function to each group."

Example: Count of employees in each department

```
SELECT department, COUNT(*) AS total_employees
FROM employees
GROUP BY department;
```

Here, we're grouping all employees **by their department** and counting how many are in each group.

Example: Average salary per department

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department;
```

We are telling MySQL: "**Group the data by department, then calculate the average salary for each group.**"

Using `GROUP BY` with Multiple Columns

You can group by more than one column to get more detailed groupings.

Example: Count by department and job title

```
SELECT department, job_title, COUNT(*) AS count
FROM employees
GROUP BY department, job_title;
```

This will count how many employees hold each job title within each department.

The `HAVING` Clause

Once you've grouped data using `GROUP BY`, you might want to **filter the groups themselves** based on the result of an aggregate function. This is where `HAVING` comes in.

> `HAVING` is like `WHERE`, but it works **after** the grouping is done.

Example: Departments with more than 5 employees

```
SELECT department, COUNT(*) AS total
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;
```

We are telling MySQL: "**First group employees by department, then only show those departments where the total number is greater than 5.**"

Difference Between `WHERE` and `HAVING`

Clause	Used For	Example Use
`WHERE`	Filters **rows before** grouping	`WHERE salary > 50000`
`HAVING`	Filters **groups after** grouping	`HAVING AVG(salary) > 60000`

You can also use both together:

```sql
SELECT department, AVG(salary) AS avg_salary
FROM employees
WHERE status = 'active'
GROUP BY department
HAVING AVG(salary) > 60000;
```

Here's what's happening:

1. `WHERE` filters only the active employees.
2. `GROUP BY` groups them by department.
3. `HAVING` filters out departments with low average salary.

Using `WITH ROLLUP` in MySQL

The `WITH ROLLUP` clause in MySQL is used with `GROUP BY` to add **summary rows** (totals and subtotals) to your result set.

Summary

Keyword	Role
`GROUP BY`	Groups rows with same values into summary rows
`HAVING`	Filters groups based on aggregate results

Use `GROUP BY` when you want to **aggregate** data. Use `HAVING` when you want to **filter those aggregated groups**.

Stored Procedures in MySQL

A **Stored Procedure** is a saved block of SQL code that you can execute later by calling its name.
It allows you to group SQL statements and reuse them—just like a function in programming.

Why Use Stored Procedures?

- To avoid repeating the same SQL logic in multiple places
- To improve performance by reducing network traffic
- To encapsulate complex business logic inside the database

Creating a Stored Procedure

When you create a stored procedure, you need to temporarily change the SQL statement delimiter from `;` to something else like `//` or `$$`.

Why change the `DELIMITER`?

MySQL ends a command at the first `;`.
Since stored procedures contain multiple SQL statements (each ending in `;`), we need to tell MySQL **not to end the procedure too early**.
So we temporarily change the delimiter to something else—then switch it back.

Example: Simple Procedure to List All Employees

```
DELIMITER //

CREATE PROCEDURE list_employees()
BEGIN
    SELECT * FROM employees;
END //

DELIMITER ;
```

This creates a procedure named `list_employees`.

Calling a Stored Procedure

You use the `CALL` statement:

```
CALL list_employees();
```

Stored Procedure with Parameters

You can pass values into procedures using the `IN` keyword.

Example: Get details of an employee by ID

```
DELIMITER //

CREATE PROCEDURE get_employee_by_id(IN emp_id INT)
BEGIN
    SELECT * FROM employees WHERE id = emp_id;
END //
```

```
DELIMITER ;
```

Here, `IN emp_id INT` means:

> “Take an integer input called `emp_id` when this procedure is called.”

Call it like this:

```
CALL get_employee_by_id(3);
```

Dropping a Stored Procedure

To delete a stored procedure:

```
DROP PROCEDURE IF EXISTS get_employee_by_id;
```

This ensures it doesn’t throw an error if the procedure doesn’t exist.

Summary

Task	SQL Command
Create Procedure	`CREATE PROCEDURE`
Change Delimiter	`DELIMITER //` (or any unique symbol)
Call a Procedure	`CALL procedure_name();`
With Input Parameter	`IN param_name data_type`
Drop a Procedure	`DROP PROCEDURE IF EXISTS procedure_name;`

Best Practices

- Always give clear names to procedures.
- Use `IN`, `OUT`, or `INOUT` for flexible parameter handling.
- Keep business logic in the database only if it improves clarity or performance.