Contents

Introduction to SQL

Introduction to MySQL

MySQL is an open-source, relational database management system (RDBMS) that uses Structured Query Language (SQL) to manage and manipulate data. It is one of the most popular database systems used in web applications, known for its speed, reliability, and ease of use. MySQL is commonly used in conjunction with programming languages such as PHP, Java, and Python to build dynamic websites and applications.

- It is developed by Oracle Corporation.
- It supports multiple platforms like Windows, Linux, and macOS.
- It is widely used by developers for its scalability, data security features, and extensive community support.

Key Features in MySQL
MySQL is a popular choice for managing relational databases for several reasons:

- Open-Source: MySQL is free and open-source, allowing modification and distribution.
- High Performance: It offers fast data retrieval and processing for large datasets.
- ACID Compliance: Ensures data integrity and reliability, especially with InnoDB storage.
- Scalability: Supports large databases and high traffic with features like partitioning and clustering.
- Multiple Storage Engines: Offers different storage engines (e.g., InnoDB, MyISAM) for flexible use.
- Replication: Supports master-slave replication for data redundancy and high availability.
- Security Features: Provides user authentication, SSL encryption, and secure data storage options.

MySQL supports the ACID properties for a transaction-safe Relational Database Management System.

- Atomic – All statements execute successfully or are canceled as a unit.
- Consistent – A database that is in a consistent state when a transaction begins is left in a consistent state by the transaction.
- Isolated – One transaction does not affect another.
- Durable – All changes made by transactions that complete successfully are recorded properly in the database. Changes are not lost.

Connecting to a MySQL instance

From CMD (MySQL Instance Running on default port)

```
mysql -u [username] -p -h [hostname] -P [port] [database_name]
```

Parameter	Description
-u	MySQL username
-p	Prompts for password
-h	Hostname (use localhost for local DB)
-P	Port number (default: 3306)
[database_name]	Optional: directly enter the database

Connect to Local MySQL Server

```
mysql -u root -p
```

*This will prompt you for the root password.

Connect to Remote MySQL Server

```
mysql -u admin_user -p -h 192.168.1.10 -P 3306
```

*This connects to a remote MySQL server at 192.168.1.10 on port 3306.

Connect and Select a Specific Database

```
mysql -u hr_user -p -h localhost -P 3306 company_db
```

Check MySQL Version

```
--From Windows CMD or terminal:
mysql -V

-- From inside the MySQL prompt:
SELECT VERSION();
```

Check MySQL is Installed and in PATH
If mysql command is not recognized:

- ✓ Ensure MySQL client tools are installed.
- ✓ Add the MySQL bin directory to your system PATH:

Useful Commands Once Connected

Command	**Action**
SHOW DATABASES;	Lists available databases
USE db_name;	Selects a database
SHOW TABLES;	Lists tables in the selected DB
DESCRIBE table_name;	Show columns of a table
EXIT; or QUIT;	Exit MySQL prompt

Special Administration Notes:

Set or Change MySQL Root Password
For MySQL 5.7 and 8.0+

1. Login as root (if no password yet)

   ```
   mysql -u root
   ```

2. Set or change the password

   ```
   ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'YourNewPassword';
   FLUSH PRIVILEGES;
   ```

3. Exit MySQL

   ```
   EXIT;
   ```

4. Now connect with password

```
mysql -u root -p
```

Enable Remote Access for Root or Another User
Enabling remote root access is not recommended for production. It's better to create a separate user with limited privileges.

1. Edit MySQL Configuration File

On Linux (e.g., Ubuntu, RHEL, etc.):
Edit my.cnf or mysqld.cnf (common paths):

```
sudo nano /etc/mysql/mysql.conf.d/mysqld.cnf
```

Find this line:

```
bind-address = 127.0.0.1
```

Change to:

```
bind-address = 0.0.0.0
```

Then restart MySQL:

```
sudo systemctl restart mysql
```

2. Create a Remote-Access User

```
CREATE USER 'remote_user'@'%' IDENTIFIED BY 'UserPassword123!';
GRANT ALL PRIVILEGES ON *.* TO 'remote_user'@'%' WITH GRANT OPTION;
FLUSH PRIVILEGES;
```

'%' allows access from any host. You can specify a specific IP instead, like '192.168.1.100'.

3. Allow MySQL Port (3306) in Firewall

On Ubuntu / Debian:

```
sudo ufw allow 3306/tcp
```

On RHEL / CentOS / Oracle Linux:

```
sudo firewall-cmd --permanent --add-port=3306/tcp
sudo firewall-cmd --reload
```

4. Test Remote Connection
From a remote client machine:

```
mysql -u remote_user -p -h your_mysql_server_ip
```

Tables in MySQL

A table in MySQL is a database object that stores data in rows and columns. Each column has a specific data type, and each row represents a record.

```
CREATE TABLE employees (
  id INT AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(50),
  last_name VARCHAR(50),
  department VARCHAR(50),
  salary DECIMAL(10, 2)
);
```

Viewing tables:

```
SHOW TABLES;
```

Describing a table:

```
DESCRIBE employees;
```

Modifying a table (e.g., add column):

```
ALTER TABLE employees ADD hire_date DATE;
```

Dropping a table:

```
DROP TABLE employees;
```

Relationships in MySQL

Relationships define how tables are connected. In MySQL, this is implemented via foreign keys. Types:

- One-to-One
- One-to-Many
- Many-to-Many (via junction table)

One-to-Many Relationship

Tables: departments and employees

```
CREATE TABLE departments (
  department_id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100)
);
```

```
CREATE TABLE employees (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  department_id INT,
  FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

*This ensures every employee belongs to a valid department.

Checking constraints:

```
SELECT *
FROM information_schema.KEY_COLUMN_USAGE
WHERE TABLE_NAME = 'employees';
```

Adding a foreign key later:

```
ALTER TABLE employees
ADD CONSTRAINT fk_dept
FOREIGN KEY (department_id) REFERENCES departments(department_id);
```

More Examples:
One-to-One Relationship
Scenario: Each employee has one profile, and each profile belongs to one employee.

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

```
CREATE TABLE employee_profiles (
  profile_id INT PRIMARY KEY AUTO_INCREMENT,
  employee_id INT UNIQUE,
  bio TEXT,
  photo_url VARCHAR(255),
  FOREIGN KEY (employee_id) REFERENCES employees(employee_id)
);
```

Explanation:

- employee_id in employee_profiles is unique, ensuring one-to-one mapping.
- An employee can only have one profile, and a profile belongs to one employee.

One-to-Many Relationship
Scenario: A department can have many employees, but each employee belongs to only one department.

```
CREATE TABLE departments (
  department_id INT PRIMARY KEY AUTO_INCREMENT,
  department_name VARCHAR(100)
);
```

```
CREATE TABLE employees (
  employee_id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(100),
  department_id INT,
  FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

Explanation:

- Each employee has one department_id.
- Multiple employees can reference the same department_id.

Many-to-Many Relationship (via Junction Table)

Scenario: Students can enroll in many courses, and each course can have many students.

```
CREATE TABLE students (
  student_id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(100)
);
```

```
CREATE TABLE courses (
  course_id INT PRIMARY KEY AUTO_INCREMENT,
  course_name VARCHAR(100)
);
```

```
CREATE TABLE enrollments (
  student_id INT,
  course_id INT,
  enrollment_date DATE,
  PRIMARY KEY (student_id, course_id),
  FOREIGN KEY (student_id) REFERENCES students(student_id),
  FOREIGN KEY (course_id) REFERENCES courses(course_id)
);
```

Explanation:

- enrollments is a junction table with a composite primary key (student_id, course_id).
- This supports many-to-many relationships between students and courses.

How to Set Relationships in MySQL Workbench (EER Diagram)

1. Open or Create a Model
    a. Launch MySQL Workbench
    b. Go to File > New Model
    c. Under the EER Diagram section, click Add Diagram
    d. You'll see an empty canvas with table tools

2. Create Tables for Relationship
   Example: departments (one) ←→ employees (many)
    a. Drag the "Place a New Table" tool onto the canvas twice
    b. Name the first table departments, the second employees
    c. Double-click each table to open its Properties

For departments table:
Column Name: department_id
Datatype: INT
PK: (yes)

For employees table:
Column Name: employee_id → INT, PK
Column Name: name → VARCHAR(100)
Column Name: department_id → INT
Click Apply after editing each table.

3. Add a Relationship (Foreign Key)
    a. Select the "1:n" (one-to-many) Relationship tool from the toolbar (crow's foot icon)
    b. Click the parent table (departments)
    c. Then click the child table (employees)

    This will:
    - Automatically create a foreign key in employees(department_id)
    - Show the line in the diagram with crow's foot notation

4. Edit Relationship Properties (Optional)
    Double-click the relationship line
    You can:
    - Change the FK name
    - Set ON DELETE / ON UPDATE actions like CASCADE, RESTRICT, SET NULL

5. Forward Engineer the Schema to SQL
    When done:
    - Go to File > Export > Forward Engineer SQL CREATE Script
    - Save the script
    - You can run it in SQL Editor or export to a running MySQL instance

6. Preview the SQL Script

Example output:

```
CREATE TABLE departments (
  department_id INT PRIMARY KEY
);
```

```
CREATE TABLE employees (
  employee_id INT PRIMARY KEY,
  name VARCHAR(100),
  department_id INT,
  CONSTRAINT fk_department
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

Quick Tips

Task	Shortcut
Add table	Drag “table” icon onto canvas
Add 1:n relation	Click 1:n icon → parent → child
Add n:n relation	Click n:m icon → table A → table B (junction table auto-created)
Edit FK settings	Double-click relationship line
Rename table or column	Double-click table, use "Columns" tab

Views in MySQL

A view is a virtual table based on a query. It doesn’t store data itself but presents it from one or more tables.

Creating and Using a View

```
CREATE VIEW high_salary_employees AS
SELECT id, name, salary
FROM employees
WHERE salary > 50000;
```

Querying the view:

```
SELECT * FROM high_salary_employees;
```

Show existing views:

```
SHOW FULL TABLES IN your_database WHERE TABLE_TYPE LIKE 'VIEW';
```

Update an existing view:

```
CREATE OR REPLACE VIEW high_salary_employees AS
SELECT id, name, salary
FROM employees
WHERE salary > 60000;
```

Drop a view:

```
DROP VIEW high_salary_employees;
```

Table VS View

Criteria	Table	View
Stores Data	Yes	No
Updatable	Yes	Sometimes (under specific conditions)
Security Control	Low	High (selective data exposure)
Based on Query	No	Yes
Performance	Fast for I/O	Slower for complex views
Use Case	Core data storage	Simplify query logic, limit access, present data

Database Normalization and Entity-Relationship (ER) Model

Entity-Relationship (ER) Model

An ER model visually represents the database structure: entities, attributes, and relationships.

CMD / SQL:

```
CREATE TABLE department (
  id INT PRIMARY KEY,
  name VARCHAR(50)
);
```

```
CREATE TABLE employee (
  id INT PRIMARY KEY,
  name VARCHAR(100),
  department_id INT,
  FOREIGN KEY (department_id) REFERENCES department(id)
);
```

phpMyAdmin:

1. Go to your database > "Designer" tab.
2. Drag tables onto canvas to see ER layout.

MySQL Workbench:

1. Use File > New Model.
2. Add Tables > Relationships, then export to SQL.

Components of ERDiagram

Component	**Description**	**Example**
Entity	Table or object	Student, Course
Attributes	Columns of entity	name, age
Relationship	Link between entities	Enrolled_In
Primary Key	Unique identifier	student_id
Foreign Key	Connects tables	course_id in enrollment

Attributes and its types

Attribute Type	Description	Example
Simple	Atomic, indivisible	name
Composite	Divided into sub-parts	full_name = first + last
Derived	Computed from others	age from dob
Multi-valued	More than one value	phone_numbers (not directly supported in 1NF)

SQL Example:

```
CREATE TABLE student (
  id INT PRIMARY KEY,
  first_name VARCHAR(50),
  last_name VARCHAR(50),
  dob DATE
);
```

Relationship Sets

A relationship set is a collection of relationships of the same type between entity sets.

Example: All students enrolled in courses form an Enrollment relationship set.

```
CREATE TABLE enrollment (
  student_id INT,
  course_id INT,
  PRIMARY KEY(student_id, course_id),
  FOREIGN KEY (student_id) REFERENCES student(id),
  FOREIGN KEY (course_id) REFERENCES course(id)
);
```

Relationship Degree

Degree	**Description**	**Example**
Unary	Entity relates to itself	Employee manages another Employee
Binary	Between two entities	Student - Course
Ternary	Among three entities	Doctor - Patient - Treatment

Relationship Types

Type	**Description**
One-to-One	Each entity in A maps to one in B
One-to-Many	One in A maps to many in B
Many-to-Many	Entities in A map to many in B and vice versa

Mapping Cardinalities

Describes number of instances involved in a relationship.

Type	**Meaning**
1:1	One to one
1:N	One to many
M:N	Many to many

Cardinalities Notations of ER Diagram

Notations (used in Workbench and diagrams):

Symbol	Meaning
–	One
–<	Many
(0,N)	Zero to many

In MySQL Workbench:

- Use crow's foot notation by default.
- Define relationships between tables in the EER Diagram interface.

Database Normalization

Process of organizing data to reduce redundancy and improve data integrity.

Example:

A bad table:

```
-- Redundant and unnormalized
CREATE TABLE orders (
 order_id INT,
 customer_name VARCHAR(100),
 customer_address VARCHAR(255),
 product1 VARCHAR(100),
 product2 VARCHAR(100),
 ...
);
```

Normalized:

- customers(customer_id, name, address)
- orders(order_id, customer_id)
- order_items(order_id, product_id)

Types of Anomalies

Type	Description	Example
Insertion	Cannot insert without unrelated data	Can't add customer unless they place an order
Update	Redundant data requires multiple updates	Customer name repeated in multiple rows
Deletion	Deleting info removes unintended data	Deleting last order deletes customer info

Types of Normalization

Normal Form	Goal	Rules
1NF	Eliminate repeating groups	Atomic values only
2NF	Remove partial dependencies	Table depends on whole primary key
3NF	Remove transitive dependencies	No non-key attribute depends on another non-key
BCNF	Stronger form of 3NF	Every determinant is a candidate key
4NF/5NF	Rarely used	For multi-valued and join dependencies

Database normalization is a manual process of restructuring your schema based on design principles. MySQL does not provide automatic normalization tools, but several visual tools can assist in the process.

How to Normalize a Database
unnormalized table example:

```
CREATE TABLE orders (
  order_id INT,
  customer_name VARCHAR(100),
  customer_address VARCHAR(255),
  product1_name VARCHAR(100),
  product2_name VARCHAR(100)
);
```

1. Convert to 1NF (First Normal Form)
   - ✓ Remove repeating groups
   - ✓ Make sure fields contain atomic values

Fix:
Split product columns into separate rows in a new table:

```
CREATE TABLE customers (
  customer_id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(100),
  address VARCHAR(255)
);
```

```
CREATE TABLE orders (
  order_id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

```
CREATE TABLE products (
  product_id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(100)
);
```

```
CREATE TABLE order_items (
  order_id INT,
  product_id INT,
  PRIMARY KEY (order_id, product_id),
  FOREIGN KEY (order_id) REFERENCES orders(order_id),
  FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

2. Convert to 2NF
   - ✓ Remove partial dependencies (only needed if composite PK exists)
   - ✓ Ensure non-key columns fully depend on the primary key
   - ✓ In our example, we already did this because:
     - o order_items depends on the full PK (order_id, product_id)
     - o orders depends on customer_id

3. Convert to 3NF
   - ✓ Remove transitive dependencies
   - ✓ Ensure non-key attributes depend only on primary key
   - ✓ Already done: no non-key attribute depends on another non-key attribute.

Tools to Help Normalize (Manual Assistance)

1. MySQL Workbench (Recommended)

Best for visual normalization

- ▪ Use EER Diagrams:
- ▪ Create initial unnormalized schema
- ▪ Add new tables, primary keys, and relationships
- ▪ Use foreign key connectors to enforce structure
- ▪ Easy to refactor and visually inspect for normalization

2. phpMyAdmin (Limited)

Go to the Designer tab inside a database

- ▪ View foreign key links
- ▪ Split tables manually
- ▪ Refactor fields
- ▪ Not as powerful as Workbench for design

Installation and Setup

Downloading MySQL Community Setup

1. Visit: https://dev.mysql.com/downloads/
2. Click MySQL Community Edition.
3. Choose the MySQL Installer for Windows (Full or Web version).
4. Full version includes all MySQL tools (~400MB)
5. Web version downloads components on-demand (~2.5MB)

Download link (direct):
https://dev.mysql.com/downloads/installer/

Installing MySQL Community

1. Run the downloaded .msi installer.
2. Choose "Developer Default" to install:
    a. MySQL Server
    b. MySQL Workbench
    c. MySQL Shell
    d. MySQL Connectors
3. Click Next > resolve any missing dependencies > Execute.

Configuring MySQL Community

- Choose Config Type: Standalone / Classic MySQL Server
- Port: Leave default 3306
- Authentication Method: Choose Use Legacy Authentication (for compatibility)
- Set root password and optionally create a user
- Start MySQL as a Windows Service

After installation:

```
mysql -u root -p
```

Configuring MySQL Workbench

1. Open MySQL Workbench
2. Under MySQL Connections, click +
3. Set:
    a. Connection Name: Local MySQL
    b. Hostname: localhost
    c. Port: 3306
    d. Username: root
4. Click Test Connection > Enter password

Connecting to MySQL Server

Command Line:

```
mysql -u root -p
```

Workbench:

1. Click the saved connection (e.g., Local MySQL)
2. You will see the SQL editor and schema list

phpMyAdmin (in XAMPP):

1. Go to http://localhost/phpmyadmin/
2. Use root and leave password blank (unless you configured it)

Downloading Sample MySQL Database

Popular Sample: Sakila, World, or Employees
Download from: https://dev.mysql.com/doc/index-other.html

Example:

- sakila-schema.sql: defines tables and relationships
- sakila-data.sql: inserts sample data

Loading Sample MySQL Database in MySQL Workbench

1. Open MySQL Workbench
2. Connect to your local MySQL server
3. Open sakila-schema.sql:
4. Go to File > Open SQL Script
5. Click Execute (⚡) or F5 to run
6. Repeat for sakila-data.sql to populate tables.

Check the database:

```
USE sakila;
SHOW TABLES;
SELECT * FROM film LIMIT 5;
```

Example Confirmation

```
SELECT title, release_year FROM film WHERE rating = 'PG-13' LIMIT 3;
```

Working with Database and Tables

Database Manipulation in MySQL

Database-Level Manipulation

Action	SQL Example
Create a database	CREATE DATABASE school;
Select a database	USE school;
Rename database*	(not supported directly; export/import instead)
Delete database	DROP DATABASE school;
View all databases	SHOW DATABASES;

Table-Level Manipulation

Action	SQL Example
Create table	CREATE TABLE students (id INT, name VARCHAR(100));
Alter table	ALTER TABLE students ADD email VARCHAR(100);
Rename table	RENAME TABLE students TO learners;
Drop table	DROP TABLE learners;
View table structure	DESCRIBE students;
View all tables	SHOW TABLES;

Data-Level Manipulation (CRUD)

```
--Insert Data
INSERT INTO students (id, name) VALUES (1, 'Alice');

--Read Data
SELECT * FROM students;
SELECT name FROM students WHERE id = 1;

--Update Data
UPDATE students SET name = 'Bob' WHERE id = 1;

--Delete Data
DELETE FROM students WHERE id = 1;
```

Query-Based Manipulation

Task	SQL Example
Filtering rows	SELECT * FROM students WHERE name = 'Alice';
Sorting rows	SELECT * FROM students ORDER BY name;
Limiting rows	SELECT * FROM students LIMIT 5;
Grouping rows	SELECT age, COUNT(*) FROM students GROUP BY age;

Joining tables SELECT * FROM students JOIN courses ON students.course_id = courses.id;

Index and Key Manipulation

Task	SQL Example
Add primary key	ALTER TABLE students ADD PRIMARY KEY (id);
Add foreign key	ALTER TABLE students ADD CONSTRAINT fk_course FOREIGN KEY (course_id) REFERENCES courses(id);
Create index	CREATE INDEX idx_name ON students(name);
Drop index	DROP INDEX idx_name ON students;

User and Permission Manipulation

Task	SQL Example
Create user	CREATE USER 'john'@'localhost' IDENTIFIED BY 'pass123';
Grant privileges	GRANT SELECT, INSERT ON school.* TO 'john'@'localhost';
Revoke privileges	REVOKE INSERT ON school.* FROM 'john'@'localhost';
Delete user	DROP USER 'john'@'localhost';

Storage Engine Types

Storage engines define how data is stored, indexed, and processed.

View storage engines:

```
SHOW ENGINES;
```

Storage Engine Types

Engine	**Features**
InnoDB	Default. Supports transactions, foreign keys
MyISAM	Fast reads, no transaction support
MEMORY	Stores data in RAM, volatile
CSV	Saves data in CSV format

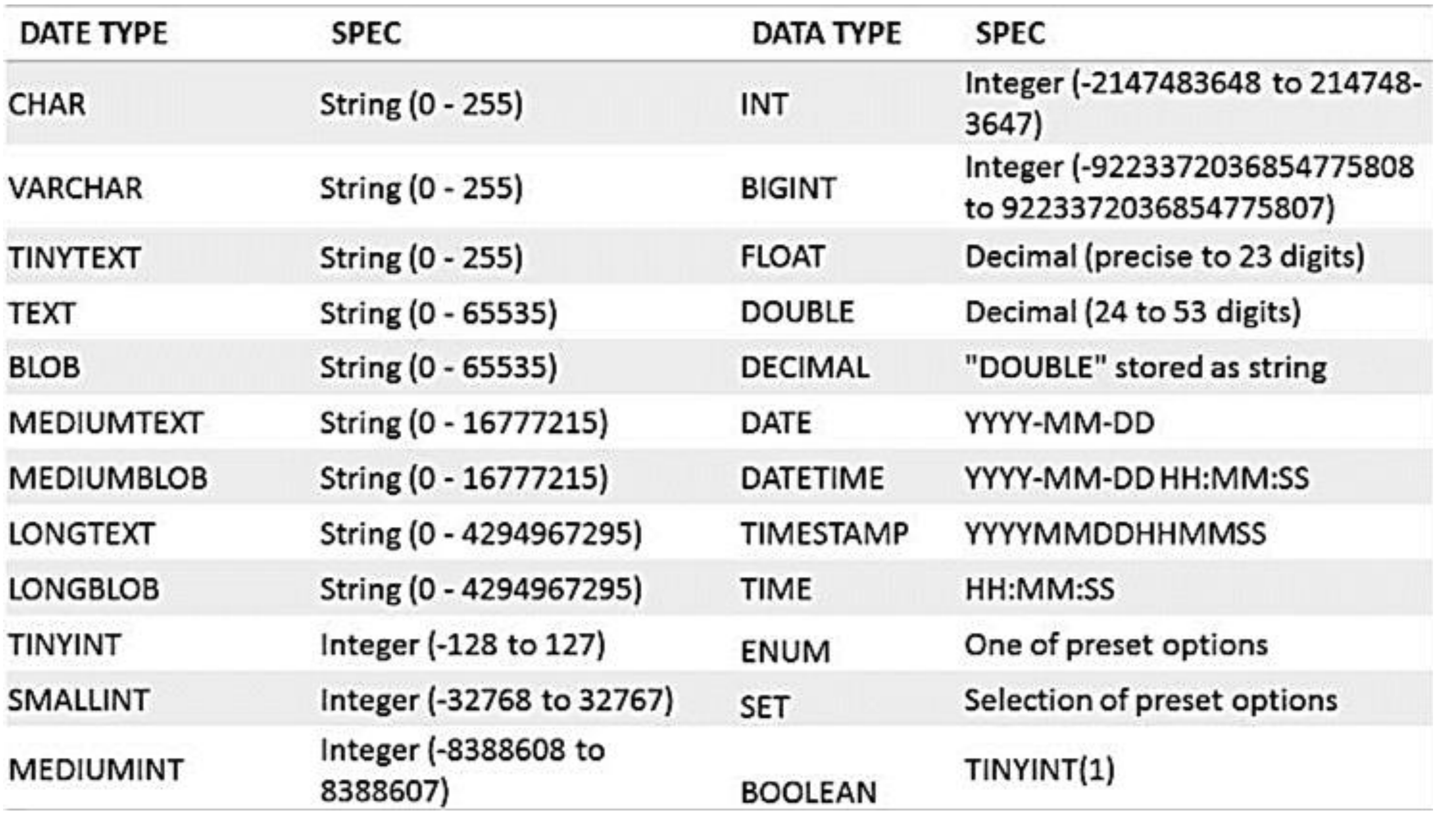
Storage Engine Setup in MySQL

```
CREATE TABLE logs (
  id INT,
  message TEXT
) ENGINE = MyISAM;
```

To check engine:

```
SHOW TABLE STATUS LIKE 'logs';
```

Data Types



DATE TYPE	SPEC	DATA TYPE	SPEC
CHAR	String (0 - 255)	INT	Integer (-2147483648 to 214748-3647)
VARCHAR	String (0 - 255)	BIGINT	Integer (-9223372036854775808 to 9223372036854775807)
TINYTEXT	String (0 - 255)	FLOAT	Decimal (precise to 23 digits)
TEXT	String (0 - 65535)	DOUBLE	Decimal (24 to 53 digits)
BLOB	String (0 - 65535)	DECIMAL	"DOUBLE" stored as string
MEDIUMTEXT	String (0 - 16777215)	DATE	YYYY-MM-DD
MEDIUMBLOB	String (0 - 16777215)	DATETIME	YYYY-MM-DD HH:MM:SS
LONGTEXT	String (0 - 4294967295)	TIMESTAMP	YYYYMMDDHHMMSS
LONGBLOB	String (0 - 4294967295)	TIME	HH:MM:SS
TINYINT	Integer (-128 to 127)	ENUM	One of preset options
SMALLINT	Integer (-32768 to 32767)	SET	Selection of preset options
MEDIUMINT	Integer (-8388608 to 8388607)	BOOLEAN	TINYINT(1)

Creating and Managing Tables in MySQL

Creating Columns (When Creating a Table)

```
CREATE TABLE employees (
  id INT PRIMARY KEY,
  name VARCHAR(100),
  salary DECIMAL(10, 2),
  hire_date DATE
);
```

Adding a Column to an Existing Table

```
ALTER TABLE employees ADD COLUMN email VARCHAR(100);
```

Add at a specific position:

```
ALTER TABLE employees ADD COLUMN department VARCHAR(50) AFTER name;
```

Change Data Type of Column

```
ALTER TABLE employees MODIFY salary FLOAT;
```

Rename Column (MySQL 8+)

```
ALTER TABLE employees RENAME COLUMN email TO work_email;
```

Changing Default Value

```
ALTER TABLE employees ALTER COLUMN department SET DEFAULT 'HR';
```

To remove default:

```
ALTER TABLE employees ALTER COLUMN department DROP DEFAULT;
```

Dropping a Column

```
ALTER TABLE employees DROP COLUMN department;
```

*This is irreversible—use with caution.

Making a Column NOT NULL or NULL

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

Adding AUTO_INCREMENT to a Column
Only works with INT types and requires it to be PRIMARY KEY or UNIQUE:

```
ALTER TABLE employees MODIFY id INT AUTO_INCREMENT;
```

Viewing Table Columns

```
DESCRIBE employees;
-- or
SHOW COLUMNS FROM employees;
```

Inserting Data in Tables

```
INSERT INTO students (id, name, age, email) VALUES (1, 'Alice', 20, 'alice@example.com');
```

Querying Table Data

```
SELECT * FROM students;
SELECT name, age FROM students;
```

Filtering Data From Tables

```
SELECT * FROM students WHERE age > 18;
```

WHERE Clause

```
SELECT * FROM students WHERE name = 'Alice';
```

DISTINCT Clause

```
SELECT DISTINCT age FROM students;
```

AND Operator

```
SELECT * FROM students WHERE age > 18 AND name = 'Alice';
```

OR Operator

```
SELECT * FROM students WHERE age < 18 OR name = 'Bob';
```

IN Operator

```
SELECT * FROM students WHERE age IN (18, 20, 22);
```

NOT IN Operator

```
SELECT * FROM students WHERE age NOT IN (18, 20);
```

BETWEEN Operator

```
SELECT * FROM students WHERE age BETWEEN 18 AND 25;
```

LIKE Operator and Wildcards

Wildcard	Meaning
%	Zero or more characters
_	Single character

```
SELECT * FROM students WHERE name LIKE 'A%'; -- Starts with A
SELECT * FROM students WHERE name LIKE '_l%'; -- Second letter is l
```

LIMIT Operator

```
SELECT * FROM students LIMIT 5;
SELECT * FROM students LIMIT 5 OFFSET 5;
```

IS NULL Operator

```
SELECT * FROM students WHERE email IS NULL;
```

IS NOT NULL Operator

```
SELECT * FROM students WHERE email IS NOT NULL;
```

Sorting Table Data

```
SELECT * FROM students ORDER BY age ASC;
SELECT * FROM students ORDER BY name DESC;
```

Grouping Table Data

```
SELECT age, COUNT(*) FROM students GROUP BY age;
```

ROLLUP

Adds subtotals to GROUP BY.

```
SELECT age, COUNT(*) FROM students GROUP BY age WITH ROLLUP;
```

Grouping Sets

MySQL does not support GROUPING SETS directly, but you can use UNION ALL.

```
SELECT age, COUNT(*) FROM students GROUP BY age
UNION ALL
SELECT NULL, COUNT(*) FROM students;
```

Comments in MySQL

```
-- This is a single-line comment

# Another single-line comment

/*
This is a
multi-line comment
*/
```

Working with Operators, Constraints, and Data Types

MySQL Operators

Operators in MySQL are symbols or keywords used to perform operations on values (e.g., comparisons, math, logic).

Arithmetic Operators

Operator	Description	Example
+	Addition	SELECT 5 + 3;
-	Subtraction	SELECT 10 - 4;
*	Multiplication	SELECT 2 * 6;
/	Division	SELECT 8 / 2;

Comparison Operators

Operator	Description	Example
=	Equal	SELECT * FROM employees WHERE salary = 50000;
!= or <>	Not equal	WHERE name != 'John'
> <	Greater/Less Than	WHERE age > 25
>= <=	Greater/Less Than or Equal	WHERE age <= 30

Logical Operators

Operator	**Description**	**Example**
AND	Both conditions true	WHERE age > 20 AND salary > 30000
OR	At least one condition true	WHERE age < 18 OR city = 'Manila'
NOT	Reverses condition	WHERE NOT (status = 'inactive')

Special Operators

Operator	**Example**
BETWEEN	WHERE age BETWEEN 20 AND 30
IN	WHERE department IN ('HR', 'IT')
LIKE	WHERE name LIKE 'A%'
IS NULL / IS NOT NULL	WHERE email IS NULL

Indexing in MySQL

An index is a data structure that improves the speed of data retrieval operations on a table at the cost of additional storage and slower writes.

Types of Indexes:

Type	Purpose
PRIMARY KEY	Unique identifier, automatically indexed
UNIQUE	Prevents duplicate values
INDEX (NON-UNIQUE)	Improves search/sort speed
FULLTEXT	For text searching
SPATIAL	For geolocation data (GIS)

Example:

```
-- Create a normal index
CREATE INDEX idx_lastname ON employees(last_name);

-- Create a unique index
CREATE UNIQUE INDEX idx_email ON users(email);

-- Drop an index
DROP INDEX idx_lastname ON employees;
```

Check indexes:

```
SHOW INDEX FROM employees;
```

Notes:

- Indexes improve SELECT, but can slow down INSERT/UPDATE/DELETE.
- Use indexes on columns often used in WHERE, JOIN, ORDER BY, GROUP BY.

Level of Data in SQL

SQL organizes data in a hierarchy, from high-level to low-level:

Level	Description	Example
Database	Collection of schemas	CREATE DATABASE school;
Schema	Logical structure in DB (in MySQL = database)	USE school;
Table	Stores related data	CREATE TABLE students (...);
Row (Record)	Single data entry	One student
Column (Field)	Data attribute	name, age, email
Cell (Value)	Intersection of row and column	'Alice', 20

MySQL Constraints

SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Prevents actions that would destroy links between tables
- CHECK - Ensures that the values in a column satisfies a specific condition
- DEFAULT - Sets a default value for a column if no value is specified
- CREATE INDEX - Used to create and retrieve data from the database very quickly

NOT NULL
The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

```
CREATE TABLE Persons (
  ID int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255) NOT NULL,
  Age int
);
```

To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

```
ALTER TABLE Persons
MODIFY Age int NOT NULL;
```

UNIQUE

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

```
CREATE TABLE Persons (
  ID int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Age int,
  UNIQUE (ID)
);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Persons (
  ID int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Age int,
  CONSTRAINT UC_Person UNIQUE (ID,LastName)
);
```

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons
ADD UNIQUE (ID);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

```
ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);
```

To drop a UNIQUE constraint, use the following SQL:

```
ALTER TABLE Persons
DROP INDEX UC_Person;
```

PRIMARY KEY

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

```
CREATE TABLE Persons (
  ID int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Age int,
  PRIMARY KEY (ID)
);
```

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

Note: In the example above there is only ONE PRIMARY KEY (PK_Person). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
```

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

```
ALTER TABLE Persons
ADD CONSTRAINT PK_Person PRIMARY KEY (ID,LastName);
```

*If you use ALTER TABLE to add a primary key, the primary key column(s) must have been declared to not contain NULL values (when the table was first created).

To drop a PRIMARY KEY constraint, use the following SQL:

```
ALTER TABLE Persons
DROP PRIMARY KEY;
```

CHECK

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   CHECK (Age>=18)
);
```

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   City varchar(255),
   CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
);
```

To create a CHECK constraint on the "Age" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons
ADD CHECK (Age>=18);
```

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

```
ALTER TABLE Persons
ADD CONSTRAINT CHK_PersonAge CHECK (Age>=18 AND City='Sandnes');
```

To drop a CHECK constraint, use the following SQL:

```
ALTER TABLE Persons
DROP CHECK CHK_PersonAge;
```

DEFAULT

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   City varchar(255) DEFAULT 'Sandnes'
);
```

The DEFAULT constraint can also be used to insert system values, by using functions like CURRENT_DATE():

```
CREATE TABLE Orders (
   ID int NOT NULL,
   OrderNumber int NOT NULL,
   OrderDate date DEFAULT CURRENT_DATE()
);
```

To create a DEFAULT constraint on the "City" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons
ALTER City SET DEFAULT 'Sandnes';
```

To drop a DEFAULT constraint, use the following SQL:

```
ALTER TABLE Persons
ALTER City DROP DEFAULT;
```

INDEXES

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

Note: Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

Creates an index on a table. Duplicate values are allowed:

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
CREATE UNIQUE INDEX Syntax
```

Creates a unique index on a table. Duplicate values are not allowed:

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

The SQL statement below creates an index named "idx_lastname" on the "LastName" column in the "Persons" table:

```
CREATE INDEX idx_lastname
ON Persons (LastName);
```

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

```
CREATE INDEX idx_pname
ON Persons (LastName, FirstName);
```

The DROP INDEX statement is used to delete an index in a table.

```
ALTER TABLE table_name
DROP INDEX index_name;
```

Functions in SQL

Understanding SQL Functions

SQL functions are built-in routines that take input, process it, and return a value. They are used in SELECT, WHERE, ORDER BY, and other clauses.

Categories:

- Aggregate Functions → Return a single result for a group
- Scalar Functions → Operate on individual values (row-wise)

Aggregate Functions

Used with GROUP BY to summarize data.

Function	**Description**	**Example**
COUNT()	Count rows	SELECT COUNT(*) FROM students;
SUM()	Total	SELECT SUM(salary) FROM employees;
AVG()	Average	SELECT AVG(score) FROM exams;
MIN()	Lowest value	SELECT MIN(age) FROM students;
MAX()	Highest value	SELECT MAX(salary) FROM employees;

Scalar Functions

Return a value per row

UPPER() – Convert to uppercase

```
SELECT name, UPPER(name) AS name_upper FROM students;
```

ROUND() – Round numbers

```
SELECT salary, ROUND(salary, 0) AS rounded_salary FROM employees;
```

LENGTH() – Character length

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

NOW() – Current date and time

```
SELECT NOW() AS current_time;
```

CONCAT() – Combine strings

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

String Functions

Function	Purpose	Example
CONCAT()	Join strings	SELECT CONCAT(first_name, ' ', last_name) FROM users;
UPPER()	To uppercase	SELECT UPPER(name) FROM users;
LOWER()	To lowercase	SELECT LOWER(email) FROM users;
SUBSTRING()	Extract part of string	SELECT SUBSTRING(name, 1, 3) FROM users;
LENGTH()	Length in bytes	SELECT LENGTH(name) FROM users;
REPLACE()	Replace text	SELECT REPLACE(name, 'A', 'X') FROM users;

Advanced (Miscellaneous) Functions

Function	Purpose	Example
IF()	Conditional check	IF(age > 18, 'Adult', 'Minor')
IFNULL()	Replace NULL	IFNULL(email, 'N/A')
COALESCE()	First non-null	COALESCE(email, alt_email, 'N/A')
NULLIF()	Returns NULL if equal	NULLIF(score, 0)
CASE	Multi-condition logic	CASE WHEN ... THEN ...
GREATEST()	Max of several columns	GREATEST(a, b, c)
FORMAT()	Add commas/decimal formatting	FORMAT(salary, 2)
RAND()	Generate random float	RAND()
UUID()	Generate unique ID	UUID()
CAST()	Change data type	CAST(score AS CHAR)

Numeric (Mathematical) Functions

Function	Description	Example
ABS()	Absolute value	SELECT ABS(-5); → 5
ROUND()	Round number	SELECT ROUND(123.456, 2); → 123.46
FLOOR()	Round down	SELECT FLOOR(4.7); → 4
CEIL() / CEILING()	Round up	SELECT CEIL(4.1); → 5
MOD()	Modulus	SELECT MOD(10, 3); → 1

Date and Time Functions

Function	Description	Example
NOW()	Current datetime	SELECT NOW();
CURDATE()	Current date	SELECT CURDATE();
CURTIME()	Current time	SELECT CURTIME();
DATE_ADD()	Add interval	SELECT DATE_ADD(NOW(), INTERVAL 7 DAY);
DATEDIFF()	Difference in days	SELECT DATEDIFF('2025-12-31', '2025-12-01');
YEAR(), MONTH()	Extract parts	SELECT YEAR(birthdate) FROM students;

Handling duplicate records

Find Duplicates (e.g., same name):

```
SELECT name, COUNT(*)
FROM students
GROUP BY name
HAVING COUNT(*) > 1;
```

Delete Duplicates (Keep One):

```
DELETE s1 FROM students s1
JOIN students s2
ON s1.name = s2.name
AND s1.id > s2.id;
```

Subqueries, Operators, and Derived Tables in SQL

Introduction to Alias

An alias is a temporary name for a column or table used to make query results more readable.

Column Alias:

```
SELECT name AS full_name, salary AS monthly_income
FROM employees;
```

Table Alias:

```
SELECT e.name, d.department_name
FROM employees AS e
JOIN departments AS d ON e.department_id = d.id;
```

Introduction to JOINS in MySQL

A JOIN combines rows from two or more tables based on a related column (usually a foreign key).

INNER JOIN (only matching records)

```
SELECT e.name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.id;
```

LEFT JOIN (all from left table, even if no match)

```
SELECT e.name, d.department_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.id;
```

RIGHT JOIN (opposite of LEFT JOIN)

```
SELECT e.name, d.department_name
FROM employees e
RIGHT JOIN departments d ON e.department_id = d.id;
```

Subquery in SQL

A subquery is a query inside another query, enclosed in parentheses.

Example: Subquery in WHERE

```
SELECT name FROM employees
WHERE department_id = (
  SELECT id FROM departments WHERE department_name = 'HR'
);
```

Subqueries with Statements and Operators

Usage	Example
IN	WHERE id IN (SELECT employee_id FROM project_members)
NOT IN	WHERE id NOT IN (...)
=, >, <	WHERE salary > (SELECT AVG(salary) FROM employees)
ANY	WHERE score > ANY (SELECT score FROM exams)
ALL	WHERE salary > ALL (SELECT salary FROM developers)

Derived Tables in SQL

A derived table is a subquery that acts like a virtual table in the FROM clause.

Example:

```
SELECT dept, total
FROM (
  SELECT department_id AS dept, SUM(salary) AS total
  FROM employees
  GROUP BY department_id
) AS salary_summary
WHERE total > 50000;
```

EXISTS Operator

EXISTS checks if a subquery returns any row. It stops at the first match (faster than IN for large datasets).

Example:

```
SELECT name FROM employees e
WHERE EXISTS (
  SELECT 1 FROM departments d
  WHERE d.id = e.department_id AND d.location = 'Manila'
);
```

EXISTS vs. IN Operators

Feature	**EXISTS**	**IN**
Checks for existence	yes	Yes
Returns value	No	yes (matches values)
Performance (large subquery)	Faster (stops early)	Slower (evaluates all)
Null-safe	yes	no (NULLs may cause unexpected results)

IN Example:

```
SELECT name FROM employees
WHERE department_id IN (
  SELECT id FROM departments WHERE location = 'Manila'
);
```

EXISTS Example:

```
SELECT name FROM employees e
WHERE EXISTS (
  SELECT 1 FROM departments d
  WHERE d.id = e.department_id AND d.location = 'Manila'
);
```

Windows Functions in SQL

Introduction to Window Function

A window function performs a calculation across a set of table rows that are somehow related to the current row, without collapsing them into a single result.

- ✓ Uses OVER() clause to define the “window” (range of rows)
- ✓ Does not group rows like GROUP BY — each row remains visible

Syntax:

```
function_name(...) OVER (
  PARTITION BY column
  ORDER BY column
  ROWS BETWEEN ... AND ...
)
```

Aggregate Window Functions

These perform aggregate calculations over a "window" of rows.

SUM() OVER a department

```
SELECT name, department_id, salary,
     SUM(salary) OVER (PARTITION BY department_id) AS dept_total_salary
FROM employees;
```

AVG(), COUNT(), MAX(), MIN() (also supported)

```
SELECT name, salary,
     AVG(salary) OVER () AS avg_salary,
     COUNT(*) OVER () AS total_employees
FROM employees;
```

Ranking Window Functions

Used to assign ranks or row numbers within partitions or the whole dataset.

ROW_NUMBER() – Unique row number

```
SELECT name, department_id, salary,
     ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) AS row_num
FROM employees;
```

RANK() – Same rank for ties, gaps in sequence

```
SELECT name, salary,
     RANK() OVER (ORDER BY salary DESC) AS rank_position
FROM employees;
```

DENSE_RANK() – No gaps in rank

```
SELECT name, salary,
     DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank
FROM employees;
```

Miscellaneous Window Functions

LEAD() – Get value from next row

```
SELECT name, salary,
    LEAD(salary) OVER (ORDER BY salary) AS next_salary
FROM employees;
```

LAG() – Get value from previous row

```
SELECT name, salary,
    LAG(salary) OVER (ORDER BY salary) AS previous_salary
FROM employees;
```

FIRST_VALUE() – First value in partition

```
SELECT name, department_id, salary,
    FIRST_VALUE(salary) OVER (PARTITION BY department_id ORDER BY salary DESC) AS top_salary
FROM employees;
```

LAST_VALUE() – Last value in partition

```
SELECT name, department_id, salary,
    LAST_VALUE(salary) OVER (PARTITION BY department_id ORDER BY salary DESC
                    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
bottom_salary
FROM employees;
```

Stored Procedures and Triggers in SQL

Advantages of Stored Procedures

A stored procedure is a set of SQL statements saved in the database and executed as a single unit using a name.

Benefit	**Description**
Modularity	Logic is centralized — change in one place affects all uses
Reusability	Call the same procedure from multiple apps/scripts
Performance	Executes faster after first compile (stored in DB)
Security	Permissions can be granted to run a procedure without exposing the underlying tables
Reduced Network Traffic	Only a procedure call is sent, not full SQL statements
Simplifies Complex Logic	You can use variables, conditions, loops inside procedures

Working with Stored Procedures

Creating a Stored Procedure

Basic example: a procedure to fetch employees by department

```
DELIMITER $$

CREATE PROCEDURE GetEmployeesByDept(IN dept_id INT)
BEGIN
  SELECT name, salary
  FROM employees
  WHERE department_id = dept_id;
END$$

DELIMITER ;
```

Calling a Stored Procedure

```
CALL GetEmployeesByDept(2);
```

Stored Procedure with Output Parameter

```
DELIMITER $$

CREATE PROCEDURE GetTotalSalary(OUT total DECIMAL(10,2))
BEGIN
  SELECT SUM(salary) INTO total FROM employees;
END$$

DELIMITER ;
```

```
-- Call and show output
CALL GetTotalSalary(@result);
SELECT @result;
```

Viewing Stored Procedures

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

Drop/Delete a Stored Procedure

```
DROP PROCEDURE IF EXISTS GetEmployeesByDept;
```

Procedure With Multiple Statements and Logic

```
DELIMITER $$

CREATE PROCEDURE GiveBonus(IN percent DECIMAL(5,2))
BEGIN
  UPDATE employees
```

```
   SET salary = salary + (salary * percent / 100);
END$$

DELIMITER ;
```

```
CALL GiveBonus(10);  -- Increase all salaries by 10%
```

Compound Statements

Compound statements are blocks of multiple SQL statements grouped using BEGIN ... END. They are essential in stored procedures, triggers, loops, and conditionals.

Example:

```
DELIMITER $$

CREATE PROCEDURE SayHello()
BEGIN
   DECLARE msg VARCHAR(50);
   SET msg = 'Hello from Stored Procedure!';
   SELECT msg;
END$$

DELIMITER ;
```

*Useful when you need multiple operations to run in a single block.

Conditional Statements

Used for decision-making logic within procedures using IF, CASE, etc.

IF ... THEN ... ELSE

```
DELIMITER $$

CREATE PROCEDURE CheckAge(IN age INT)
BEGIN
   IF age >= 18 THEN
      SELECT 'Adult' AS result;
   ELSE
      SELECT 'Minor' AS result;
   END IF;
END$$

DELIMITER ;
```

CASE Statement

```
DELIMITER $$

CREATE PROCEDURE GradeRemark(IN grade CHAR(1))
BEGIN
  CASE grade
    WHEN 'A' THEN SELECT 'Excellent';
    WHEN 'B' THEN SELECT 'Good';
    WHEN 'C' THEN SELECT 'Fair';
    ELSE SELECT 'Fail';
  END CASE;
END$$

DELIMITER ;
```

Loops in Stored Procedures

Type	**Use Case**
LOOP	Basic loop with manual EXIT
WHILE	Loop while condition is true
REPEAT	Loop until condition is true (like do-while)

LOOP + LEAVE

```
DELIMITER $$

CREATE PROCEDURE PrintNumbers()
BEGIN
  DECLARE i INT DEFAULT 1;

  number_loop: LOOP
    IF i > 5 THEN
      LEAVE number_loop;
    END IF;
    SELECT i;
    SET i = i + 1;
  END LOOP number_loop;
END$$

DELIMITER ;
```

WHILE Loop

```
DELIMITER $$

CREATE PROCEDURE WhileLoopDemo()
BEGIN
  DECLARE i INT DEFAULT 1;

  WHILE i <= 3 DO
    SELECT CONCAT('Row ', i);
    SET i = i + 1;
  END WHILE;
END$$

DELIMITER ;
```

REPEAT Loop

```
DELIMITER $$

CREATE PROCEDURE RepeatLoopDemo()
BEGIN
  DECLARE i INT DEFAULT 1;

  REPEAT
    SELECT CONCAT('Count: ', i);
    SET i = i + 1;
  UNTIL i > 3
  END REPEAT;
END$$

DELIMITER ;
```

Error Handling in Stored Procedures

Use DECLARE ... HANDLER to catch and respond to errors or warnings.

Example: Handling SQL error

```
DELIMITER $$

CREATE PROCEDURE SafeInsert()
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN
      SELECT 'An error occurred!' AS error_message;
    END;
  INSERT INTO employees (id, name) VALUES (1, 'Alice'); -- Assume 1 already exists
END$$

DELIMITER ;
```

```
CALL SafeInsert();
```

Types of Handlers:

Handler Type	Description
CONTINUE	Continue after error
EXIT	Exit the block immediately
UNDO	Not supported in MySQL

Cursors in Stored Procedures

A cursor is a database pointer used to iterate row by row over a result set.

Basic Cursor Template

```
DECLARE cursor_name CURSOR FOR SELECT_statement;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done_flag = TRUE;

OPEN cursor_name;
FETCH cursor_name INTO variable;
CLOSE cursor_name;
```

Example: Loop through rows

```
DELIMITER $$

CREATE PROCEDURE ListEmployeeNames()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE emp_name VARCHAR(100);
  DECLARE emp_cursor CURSOR FOR SELECT name FROM employees;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN emp_cursor;

  read_loop: LOOP
    FETCH emp_cursor INTO emp_name;
    IF done THEN
      LEAVE read_loop;
    END IF;
    SELECT emp_name AS employee_name;
  END LOOP;
  CLOSE emp_cursor;
END$$

DELIMITER ;
```

```
CALL ListEmployeeNames();
```

Example 2: Print all student names

```
DELIMITER $$

CREATE PROCEDURE ListStudentNames()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE student_name VARCHAR(100);

  DECLARE student_cursor CURSOR FOR
    SELECT name FROM students;

  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN student_cursor;

  read_loop: LOOP
    FETCH student_cursor INTO student_name;
    IF done THEN
      LEAVE read_loop;
    END IF;

    SELECT student_name AS 'Student Name';
  END LOOP;

  CLOSE student_cursor;
END$$

DELIMITER ;
```

```
CALL ListStudentNames();
```

Example 2: Sum salaries by department using cursor

```
DELIMITER $$

CREATE PROCEDURE SumSalariesByDept()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE dept_id INT;
  DECLARE dept_salary DECIMAL(10,2);

  DECLARE dept_cursor CURSOR FOR
    SELECT DISTINCT department_id FROM employees;

  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN dept_cursor;

  dept_loop: LOOP
    FETCH dept_cursor INTO dept_id;
    IF done THEN
      LEAVE dept_loop;
    END IF;

    SELECT department_id, SUM(salary) AS total_salary
    FROM employees
    WHERE department_id = dept_id
    GROUP BY department_id;
  END LOOP;

  CLOSE dept_cursor;
END$$

DELIMITER ;
```

```
CALL SumSalariesByDept();
```

Example 3: Update bonus column using cursor

```
DELIMITER $$

CREATE PROCEDURE UpdateEmployeeBonus()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE emp_id INT;
  DECLARE emp_salary DECIMAL(10,2);
  DECLARE bonus_rate DECIMAL(5,2) DEFAULT 0.10;
  DECLARE emp_cursor CURSOR FOR
    SELECT id, salary FROM employees;

  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN emp_cursor;

  loop_bonus: LOOP
    FETCH emp_cursor INTO emp_id, emp_salary;
    IF done THEN
      LEAVE loop_bonus;
    END IF;

    UPDATE employees
    SET bonus = emp_salary * bonus_rate
    WHERE id = emp_id;
  END LOOP;

  CLOSE emp_cursor;
END$$

DELIMITER ;
```

```
CALL UpdateEmployeeBonus();
```

Example 4: Count and list low-score students using cursor and IF

```
DELIMITER $$

CREATE PROCEDURE FlagLowScores()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE sid INT;
  DECLARE score INT;

  DECLARE cur CURSOR FOR
    SELECT student_id, exam_score FROM exam_results;

  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN cur;

  loop_scores: LOOP
    FETCH cur INTO sid, score;
    IF done THEN
      LEAVE loop_scores;
    END IF;

    IF score < 50 THEN
      SELECT CONCAT('Student ID ', sid, ' needs help') AS notice;
    END IF;
  END LOOP;

  CLOSE cur;
END$$

DELIMITER ;
```

```
CALL FlagLowScores();
```

Cursor Best Practices

Tip	**Why**
Always use a NOT FOUND handler	Prevent infinite loop
Always CLOSE the cursor	Free up resources
Use aliases in SELECT	Avoid confusion inside loops
Keep logic simple inside loops	For readability and performance

Stored Functions in Stored Procedures

A stored function returns a single value and can be used inside SQL expressions. It can be called from stored procedures, SELECT queries, etc.

Create a Stored Function:

```
DELIMITER $$

CREATE FUNCTION TaxAmount(salary DECIMAL(10,2)) RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
  RETURN salary * 0.1;
END$$

DELIMITER ;
```

Use Function in Procedure:

```
DELIMITER $$

CREATE PROCEDURE CalculateTax()
BEGIN
  SELECT name, salary, TaxAmount(salary) AS tax
  FROM employees;
```

```
END$$

DELIMITER ;
```

```
CALL CalculateTax();
```

Example of CRUD (Create, Read, Update, Delete) stored procedures in MySQL using a simple students table.

1. Create the Table

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

2. CREATE Procedure – Add a New Student

```
DELIMITER $$

CREATE PROCEDURE AddStudent(
   IN p_name VARCHAR(100),
   IN p_age INT,
   IN p_email VARCHAR(100)
)
BEGIN
   INSERT INTO students(name, age, email)
   VALUES (p_name, p_age, p_email);
END$$

DELIMITER ;
```

```
CALL AddStudent('Alice', 20, 'alice@example.com');
```

3. READ Procedure – Get All Students

```
DELIMITER $$

CREATE PROCEDURE GetAllStudents()
BEGIN
   SELECT * FROM students;
END$$

DELIMITER ;
CALL GetAllStudents();
```

4. UPDATE Procedure – Update a Student by ID

```
DELIMITER $$

CREATE PROCEDURE UpdateStudent(
  IN p_id INT,
  IN p_name VARCHAR(100),
  IN p_age INT,
  IN p_email VARCHAR(100)
)
BEGIN
  UPDATE students
  SET name = p_name,
    age = p_age,
    email = p_email
  WHERE id = p_id;
END$$

DELIMITER ;
```

```
CALL UpdateStudent(1, 'Alice Smith', 21, 'alice.smith@example.com');
```

5. DELETE Procedure – Remove a Student by ID

```
DELIMITER $$

CREATE PROCEDURE DeleteStudent(IN p_id INT)
BEGIN
  DELETE FROM students WHERE id = p_id;
END$$

DELIMITER ;
```

```
CALL DeleteStudent(1);
```

Stored Program Security

Stored Program Security refers to who can create, alter, execute, or manage stored programs (procedures, functions, triggers, events) and how execution privileges are controlled.

Security Models:

A. Definer vs Invoker Rights
   - DEFINER (default): Executes with the privileges of the user who created the stored program.
   - INVOKER: Executes with the privileges of the user who calls the program.

Example:

```
CREATE DEFINER='admin'@'localhost' PROCEDURE ViewSensitiveData()
BEGIN
  SELECT * FROM payroll;
END;
```

*Only admin needs access to payroll, but others can CALL it.

Granting EXECUTE Privileges

Let another user run a procedure without direct access to tables:

```
GRANT EXECUTE ON PROCEDURE ViewSensitiveData TO 'user1'@'localhost';
```

View Security Information

```
SHOW CREATE PROCEDURE ViewSensitiveData;
```

SQL Trigger

A trigger is a stored program that automatically executes in response to an event on a table (e.g., INSERT, UPDATE, DELETE).

Example 1: Audit log on INSERT

```
-- Create audit table
CREATE TABLE student_audit (
  id INT AUTO_INCREMENT PRIMARY KEY,
  student_id INT,
  name VARCHAR(100),
  action_type VARCHAR(10),
  action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
-- Create the trigger
DELIMITER $$

CREATE TRIGGER after_student_insert
AFTER INSERT ON students
FOR EACH ROW
BEGIN
  INSERT INTO student_audit(student_id, name, action_type)
  VALUES (NEW.id, NEW.name, 'INSERT');
END$$

DELIMITER ;
```

Behavior: After a new row is inserted into students, an entry is created in student_audit.

Example 2: Auto-set Timestamp on Insert
Automatically populate a created_at field when a new row is added.

Table:

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

Trigger:

```
DELIMITER $$

CREATE TRIGGER before_users_insert
BEFORE INSERT ON users
FOR EACH ROW
BEGIN
   SET NEW.created_at = NOW();
END$$

DELIMITER ;
```

Example 3: Audit Log on DELETE
Track deleted records in a backup table.

Tables:

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

```
CREATE TABLE employee_deletions (
   emp_id INT,
   emp_name VARCHAR(100),
   deleted_at DATETIME
);
```

Trigger:

```
DELIMITER $$

CREATE TRIGGER after_employee_delete
AFTER DELETE ON employees
```

```
FOR EACH ROW
BEGIN
  INSERT INTO employee_deletions(emp_id, emp_name, deleted_at)
  VALUES (OLD.id, OLD.name, NOW());
END$$

DELIMITER ;
```

Example 4: Prevent Negative Stock (Validation)
Block any update that would set stock below 0.

Table:

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

Trigger:

```
DELIMITER $$

CREATE TRIGGER before_stock_update
BEFORE UPDATE ON products
FOR EACH ROW
BEGIN
  IF NEW.stock < 0 THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Stock cannot be negative';
  END IF;
END$$

DELIMITER ;
```

Example 5: Auto-calculate Total Price on INSERT
Update total price automatically from quantity × unit price.

Table:

```
CREATE TABLE orders (
  id INT PRIMARY KEY AUTO_INCREMENT,
  product_name VARCHAR(100),
  quantity INT,
  unit_price DECIMAL(10,2),
  total_price DECIMAL(10,2)
);
```

Trigger:

```
DELIMITER $$

CREATE TRIGGER before_order_insert
BEFORE INSERT ON orders
FOR EACH ROW
BEGIN
  SET NEW.total_price = NEW.quantity * NEW.unit_price;
END$$

DELIMITER ;
```

Example 6: Cascading Update
Update related records in another table (simulate foreign key ON UPDATE CASCADE).

Tables:

```
CREATE TABLE departments (
  dept_id INT PRIMARY KEY,
  dept_name VARCHAR(100)
);
```

```
CREATE TABLE employees (
  emp_id INT PRIMARY KEY,
  emp_name VARCHAR(100),
  dept_id INT
);
```

Trigger:

```
DELIMITER $$

CREATE TRIGGER after_department_update
AFTER UPDATE ON departments
FOR EACH ROW
BEGIN
  UPDATE employees
  SET dept_id = NEW.dept_id
  WHERE dept_id = OLD.dept_id;
END$$

DELIMITER ;
```

Note: For referential integrity, foreign keys with ON UPDATE CASCADE are preferred if possible.

How to View and Drop Triggers

View:

```
SHOW TRIGGERS;
```

Drop:

```
DROP TRIGGER IF EXISTS before_stock_update;
```

Other Trigger Types

Timing	**Event**	**Syntax Example**
BEFORE INSERT	Set default values	BEFORE INSERT ON tablename
AFTER UPDATE	Audit updates	AFTER UPDATE ON tablename
BEFORE DELETE	Log delete requests	BEFORE DELETE ON tablename

Accessing OLD and NEW values

Value Type	**Use in Trigger**
NEW.column_name	Value after change (INSERT/UPDATE)
OLD.column_name	Value before change (UPDATE/DELETE)

Drop a Trigger

```
DROP TRIGGER IF EXISTS after_student_insert;
```

Performance Optimization and Best Practices in SQL

Execution Plan in SQL

An execution plan shows how MySQL executes a query: which indexes are used, join types, row estimates, and table access strategies.

View the Execution Plan:

```
EXPLAIN SELECT name FROM employees WHERE department_id = 2;
```

Important Columns:

Column	**Meaning**
type	Join type (e.g., ALL, index, ref, const)
key	Index used
rows	Estimated number of rows to scan
Extra	Notes like Using index, Using where, Using temporary

*Try using EXPLAIN ANALYZE in MySQL 8.0+ for more detailed cost insights.

Identifying the differences between Char, Varchar, and NVarchar

Type	Description	Storage	Use Case
CHAR(n)	Fixed-length	Always uses n bytes	Fixed ID codes (e.g., CHAR(2) for country code)
VARCHAR(n)	Variable-length	Uses length of data + 1 byte	Names, emails, etc.
NVARCHAR(n)	Not in MySQL (used in SQL Server)	Supports Unicode	Use CHARACTER SET utf8mb4 in MySQL instead

Example:

```
CREATE TABLE example (
  code CHAR(3),
  name VARCHAR(100),
  comment VARCHAR(255) CHARACTER SET utf8mb4
);
```

Clustered Indexes in MySQL

A clustered index determines how rows are physically stored. In MySQL InnoDB, the primary key is always the clustered index.

- ✓ Only one clustered index per table
- ✓ Improves performance on range queries and primary key lookups

Example:

```
CREATE TABLE books (
  book_id INT PRIMARY KEY,  -- clustered index
  title VARCHAR(200),
  published_year INT
);
```

See Index Info:

```
SHOW INDEX FROM books;
```

Common Table Expressions

A CTE is a named temporary result set that can be referenced within a query. It improves readability, recursion, and modular logic.

Syntax:

```
WITH recent_orders AS (
  SELECT * FROM orders WHERE order_date >= CURDATE() - INTERVAL 7 DAY
)
SELECT * FROM recent_orders WHERE total_amount > 100;
```

Recursive CTE:

```
WITH RECURSIVE counter AS (
  SELECT 1 AS n
  UNION ALL
  SELECT n + 1 FROM counter WHERE n < 5
)
SELECT * FROM counter;
```

Example 1: CTE for Filtering and Reuse

Reuse filtered result of orders placed in the last 30 days

```
WITH recent_orders AS (
 SELECT * FROM orders
 WHERE order_date >= CURDATE() - INTERVAL 30 DAY
)
SELECT customer_id, COUNT(*) AS total_orders
FROM recent_orders
GROUP BY customer_id;
```

Example 2: CTE with JOIN

List employees and their department names

```
WITH emp_dept AS (
 SELECT e.id, e.name, d.department_name
 FROM employees e
 JOIN departments d ON e.department_id = d.id
)
SELECT * FROM emp_dept
WHERE department_name = 'Sales';
```

Example 3: CTE with Aggregation

Get the average salary per department and filter those > 50,000

```
WITH avg_salary_per_dept AS (
 SELECT department_id, AVG(salary) AS avg_salary
 FROM employees
 GROUP BY department_id
)
SELECT department_id, avg_salary
FROM avg_salary_per_dept
WHERE avg_salary > 50000;
```

Example 4: Recursive CTE – Generate a Number Sequence

Create a sequence of numbers 1 to 10

```
WITH RECURSIVE counter(n) AS (
  SELECT 1
  UNION ALL
  SELECT n + 1 FROM counter WHERE n < 10
)
SELECT * FROM counter;
```

Example 5: Recursive CTE – Hierarchical Data (Org Chart)

Tables:

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

CTE: Get All Subordinates of a Manager (e.g., id = 1)

```
WITH RECURSIVE org_chart AS (
  SELECT id, name, manager_id
  FROM employees
  WHERE manager_id IS NULL   -- start from top-level manager (CEO)

  UNION ALL

  SELECT e.id, e.name, e.manager_id
  FROM employees e
  INNER JOIN org_chart o ON e.manager_id = o.id
)
SELECT * FROM org_chart;
```

Example 6: Multiple CTEs in One Query

```
WITH dept_counts AS (
  SELECT department_id, COUNT(*) AS emp_count
  FROM employees
  GROUP BY department_id
),
top_departments AS (
  SELECT department_id FROM dept_counts WHERE emp_count >= 5
)
SELECT e.name, e.department_id
FROM employees e
JOIN top_departments t ON e.department_id = t.department_id;
```

Backup and Restore

Backup Using Command Line (mysqldump)

```
mysqldump -u [username] -p [database_name] > backup_file.sql
```

Example:

```
mysqldump -u root -p school_db > school_backup.sql
```

*You'll be prompted for the password. The backup is saved as a .sql file.

Restore Using Command Line

```
mysql -u [username] -p [database_name] < backup_file.sql
```

Example:

```
mysql -u root -p school_db < school_backup.sql
```

*The database (school_db) must already exist. If not, create it first:

```
CREATE DATABASE school_db;
```

Backup Using MySQL Workbench

1. Open MySQL Workbench
2. Go to Server > Data Export > Choose the database(s)
3. Choose Export to Self-Contained File (e.g., backup.sql)
4. Click Start Export

Restore Using MySQL Workbench

1. Open Server > Data Import
2. Choose Import from Self-Contained File
3. Select the .sql file
4. Choose database to import into (or create new)
5. Click Start Import

Backup Using phpMyAdmin

1. Go to http://localhost/phpmyadmin
2. Select the database
3. Click Export > Choose Quick or Custom method > Choose format: SQL
4. Click Go → It will download a .sql file

Restore Using phpMyAdmin

1. Go to http://localhost/phpmyadmin
2. Create or select the target database
3. Click Import
4. Browse and upload your .sql file
5. Click Go → It will run the SQL commands in the file

MySQL Best Practices

Area	**Best Practice**
Indexing	Use indexes on columns in WHERE, JOIN, ORDER BY
Use EXPLAIN	Optimize queries using EXPLAIN to understand bottlenecks
Limit Columns	Only select the columns you need (SELECT name not SELECT *)
Normalize	Avoid data duplication via 3NF (3rd Normal Form)
Avoid Subquery in WHERE IN	Prefer JOIN or EXISTS when possible
Backups	Use mysqldump or mysqlpump regularly
Security	Don’t allow root access from outside; use least privilege users
Use UTF-8 (utf8mb4)	Supports full Unicode (emoji, accents, Asian characters)
Stored Procedures	Encapsulate business logic in reusable procedures
Monitor Slow Queries	Enable slow_query_log for query performance tracking