INTRODUCTION TO MYSQL

- MySQL is a data base system used for developing web-based software applications.
- MySQL is an open-source, fast, reliable and flexible relational data base management system, typically used with PHP
- MySQL used for both small and large applications.
- MySQL is are Relational Data Base Management System*(RDBMS).*
- MySQL is fast, reliable, and flexible and easy to use.
- MySQL supports standard SQL*(Structured Query Language).*
- MySQL is free to download and use.
- MySQL was developed by Michael Widenius and DavidAxmarkin1994.
- MySQL is presently developed, distributed, and supported by Oracle Corporation.

Features of MYSQL

- MySQL server design is multi-layered with in dependent modules.
- MySQL is fully multi threaded by using kernel threads. It can handle multiple CPU sifthey are available.
- MySQL provides transactional and non-transactional storage engines.
- MySQL has a high-speed thread-based memory allocation system.
- MySQL support sin-memory heap table.
- MySQL Handles large databases.
- MySQL Server work sin client/server or embedded systems.
- MySQL Works on many different platforms.

DATABASE

In MySQL**, a** database **is a structured collection of data that is stored and managed by the MySQL Database Management System (DBMS). It acts as a container for organizing and storing** tables**,** views**,** stored procedures**,** triggers**, and** other database objects**.**

Database Structure

- A **database** contains **tables**.
- **Tables** contain **rows** (records) and **columns** (fields).
- Each column has a specific **data type** (e.g., INT, VARCHAR, DATE, etc.).

RDBMS

RDBMS in MySQL stands for **Relational Database Management System**, which is the foundation of how MySQL manages data.

An **RDBMS** is a type of database management system that stores data in the form of **related tables**. These tables are made up of **rows** and **columns**, like a spreadsheet.

MySQL is one of the most popular RDBMS systems.

RDBMS Terminology

1. **Database**
   A collection of organized data stored electronically. In RDBMS, data is stored in tables.
2. **Table (Relation)**
   A structure to store data in rows and columns. Each table represents one entity (e.g., students, products).
3. **Row (Tuple)**
   A single record in a table. Each row contains data for one item/entity.
4. **Column (Attribute)**
   A field in a table. Each column holds data of a specific type and represents one property of the entity.
5. **Primary Key**
   A column (or combination of columns) that uniquely identifies each row in a table. Must be unique and not null.
6. **Foreign Key**
   A column that creates a relationship between two tables. It refers to the primary key of another table.
7. **Schema**
   The structure of the database, including tables, fields, relationships, views, indexes, etc.
8. **Normalization**
   The process of organizing data to reduce redundancy and improve data integrity.
9. **SQL (Structured Query Language)**
   The standard language used to interact with an RDBMS. It includes commands like SELECT, INSERT, UPDATE, DELETE, etc.
10. **Constraints**
    Rules applied to columns to ensure valid data, like:
    - NOT NULL (no empty values)
    - UNIQUE (no duplicate values)
    - CHECK (validates values against a condition)
11. **Index**
    A data structure that improves the speed of data retrieval operations on a table.
12. **View**
    A virtual table created by a query. It does not store data itself but displays it from other tables.
13. **Join**
    An SQL operation used to combine data from two or more tables based on a related column.
14. **Transaction**
    A set of operations performed as a single unit. A transaction must be **atomic**, **consistent**, **isolated**, and **durable** (ACID properties).

DATA TYPES

MySQL numeric data types

Numeric Types	Description
TINYINT	A very small integer
SMALLINT	A small integer
MEDIUMINT	A medium-size integer
INT	A standard integer
BIGINT	A large integer
DECIMAL	A fixed-point number
FLOAT	A single-precision floating point number
DOUBLE	A double-precision floating point number
BIT	A bit field

MySQL string data types

String Types	Description
CHAR	A fixed-length non binary(character)string
VARCHAR	A variable-length non-binary string
BINARY	A fixed-length binary string
VARBINARY	A variable-length binary string
TINYBLOB	A very small BLOB(binary large object)
BLOB	A small BLOB
MEDIUMBLOB	A medium-sized BLOB
LONGBLOB	A large BLOB
TINYTEXT	A very small non-binary string
TEXT	A small non-binary string
MEDIUMTEXT	A medium-sized non-binary string
LONGTEXT	A large non-binary string

MySQL date and time data types

Date and Time Types	Description
DATE	A date value in CCYY-MM-DD format
TIME	A time value in hh:mm:ss format
DATETIME	A date and time value in CCYY-MM-DDhh:mm:ss format
TIMESTAMP	A times tamp value in CCYY-MM-DDhh:mm:ss format
YEAR	A year value in CCYY or YY format

Types of COMMANDS

1. Data Definition Language

Data Definition Language (DDL) statements are used to define the database structure or schema. Data Definition Language understanding with database schemas and describes how the data should consist in the database, therefore language statements like CREATETABLE or ALTER TABLE belongs to the DDL. DDL is about "metadata".

DDL includes commands such as CREATE, ALTER and DROP statements. DDL is used to CREATE, ALTER OR DROP the database objects (Table, Views, Users).

Data Definition Language (DDL) are used different statements:

- CREATE-to create objects in the database
- ALTER-alters the structure of the database
- DROP –delete objects from the database
- TRUNCATE-removeallrecordsfromatable,includingallspacesallocatedforthe records are removed
- COMMENT-add comments to the data dictionary
- RENAME-rename an object

Note for student: ***write example and output by your own on the basis of syntax***

CREATE TABLE

Syntax: Create table table_name (field_name1 data_type(), field_name2 data_type()...);

Ex: create table student (id int, name varchar(20));

ALTER TABLE

1. ADD
2. MODIFY

ADD

Syntax: alter table table_name ADD (fieldname data type()...);

MODIFY

Syntax: alter table table_name modify (field_name data type()...);

DESCRIBE TABLE

Syntax: DESCRIBE TABLE_NAME

DROP TABLE

Syntax: DROP Table_name;

2. Data Manipulation Language

Data Manipulation Language (DML) statements are used for managing data within schema objects DML deals with data manipulation, and therefore includes most common SQL statements such SELECT, INSERT, etc. DML allows to add / modify / delete data itself.

DML is used to manipulate with the existing data in the database objects (insert, select, update, delete).

DML Commands:

1. INSERT
2. SELECT
3. UPDATE
4. DELETE

***INSERT:**

Syntax: INSERT INTO Table_name values(); Ex: insert into student values(20,"ABC");

***SELECT:**

Syntax: Select*from<table_name>; EX: Select * from student;

***UPDATE:**

Syntax: Update <table_name> set to (calculation); Ex: update student set_name = "XYZ"

where id = 20;

***DELETE:**

Syntax: Delete from<table_name> where condition; EX: delete from student where id=20;

3. Data Control Language

DCL is the abstract of Data Control Language. Data Control Language includes commands such as GRANT, and concerns with rights, permissions and other controls of the database system. DCL is used to grant / revoke permissions on databases and their contents. DCL is simple, but MySQL permissions are a bit complex. DCL is about security. DCL is used to control the database transaction. DCL statement allow you to control who has access to specific object in your database.

GRANT

It provides the user's access privileges to the database. In the MySQL database offers both the administrator and user a great extent of the control options. By the administration side of the process includes the possibility for the administrators to control certain user privileges over the MySQL server by restricting their access to an entire the database or user limiting permissions for a specific table. It Creates an entry in the security system that allows user in the current data base to work with data in the current data base or execute specific statements.

Syntax :

```
    GRANT{ALL statement[,...n]} TO security_account [ ,...n ]
```

Normally, a data base administrator first uses CREATE USER to create an account, then GRANT to define its privileges and characteristics.

For example:

```
    CREATE USER 'arjun'@'localhost' IDENTIFIED BY' mypass'; GRANT ALL ON db1.*

    TO 'arjun'@'localhost';

    GRANT SELECT ON child TO 'arjun'@'localhost';
    GRANT US AGE ON*.* TO 'arjun'@'localhost'WITHMAX_QUERIES_PER_HOUR 90;
```

REVOKE

The REVOKE statement enables system administrators and to revoke the privileges from MySQL accounts.

Syntax :

```
    REVOKE priv_type[(column_list)]
    [,priv_type[(column_list)]]... ON [object_type] priv_level FROM user [, user] ...
```

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM user[,user]...
```

For example:

```
mysql> REVOKE INSERT ON*.*FROM'arjun'@'localhost';
```

COMMAND CLAUSES

WHERE

The WHERE clause is used to filter records. The WHERE clause is used to extract only those records that fulfill a specified condition.

WHERE Syntax

```
SELECTcolumn1,column2,...
FROM table_name
WHERE condition;
```

Example: (**Notes for student**: While giving example-take one table and extract values from those table)

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

ORDER BY

The ORDER BY key word is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

Example:

```
SELECT*FROM Students ORDER BY name;
```

GROUP BY

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT,MAX,MIN, SUM, AVG) to group the result-set by one or more columns.

Syntax

```
SELECT column_name(s) FROM table_name WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Example:

```
SELECT COUNT(studID),name FROM Students
GROUPBY name
```

HAVING

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

Syntax:

```
SELECT column_name(s)
FROM table_name WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDERBY column_name(s)
```

Example

```
SELECT COUNT(studID), name
FROM Student
GROUP BY name
HAVING COUNT(studID)>5;
```

LIKE

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the LIKE operator:

- % - The percent sign represents zero, one, or multiple characters
- _ - The underscore represents a single character

Syntax

```
SELECTcolumn1,column2,...
FROM table_name
WHERE column LIKE pattern;
```

Example:

The following SQL statement selects all customers with a Customer Name starting with "a":

```
SELECT * FROM Student
WHERE name LIKE 'a%';
```

BETWEE N

The BETWEEN operator selects values with in a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Example

```
SELECT * FROM Employee
WHERE salary BETWEEN 10000 AND 20000;
```

My SQL Operations on Crime Database

Project Overview

This project titled **“MySql Operations on Crime Database”** is designed to simulate and manage crime-related data using structured and relational database systems. The primary objective is to provide a real-world, hands-on learning environment for mastering MySQL queries and database operations, using data relevant to crime reporting, investigation, and suspect profiling.

The dataset consists of **two interrelated tables**:

- **Crimes Table:** Records details of each reported crime such as the type, location, date and time, status, description, case number, and links to the suspect involved.
- **Suspects Table:** Maintains suspect-related information like name, age, gender, contact details, prior records, and threat levels.

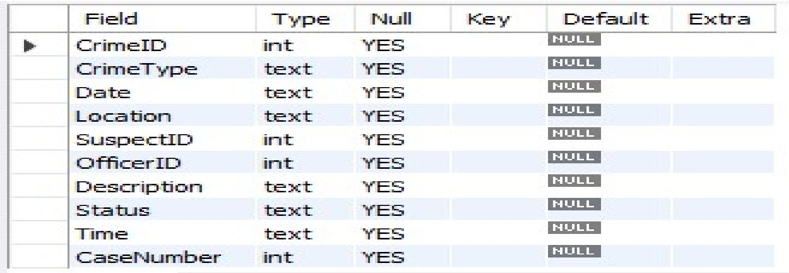
The data has been synthetically generated to reflect 300+ realistic entries in each table with full relational integrity, allowing for the application of a wide variety of SQL concepts.

This project serves as a capstone-style exercise for learners and practitioners of MySQL and relational databases. It offers a complete environment to practice, implement, and understand how data can be modeled and manipulated effectively in a real-world use case—crime tracking and investigation systems.

In practical terms, this system could act as a prototype for use by law enforcement agencies, forensic teams, or judicial organizations for storing and analyzing criminal data systematically. For educational purposes, the depth of operations included makes this project suitable for interviews, coursework, or database certification preparation.

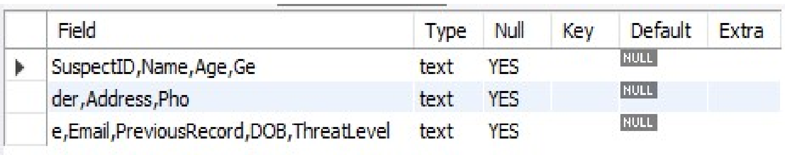
Description of Tables

Crimes Table



	Field	Type	Null	Key	Default	Extra
▶	CrimeID	int	YES		NULL	
	CrimeType	text	YES		NULL	
	Date	text	YES		NULL	
	Location	text	YES		NULL	
	SuspectID	int	YES		NULL	
	OfficerID	int	YES		NULL	
	Description	text	YES		NULL	
	Status	text	YES		NULL	
	Time	text	YES		NULL	
	CaseNumber	int	YES		NULL	

Suspects Table:



	Field	Type	Null	Key	Default	Extra
▶	SuspectID,Name,Age,Ge	text	YES		NULL	
	der,Address,Pho	text	YES		NULL	
	e,Email,PreviousRecord,DOB,ThreatLevel	text	YES		NULL	

Database Name: Crime

Tables Name: Crimes, Suspect

Practice Questions and its Queries:

# Q-1: Show all crimes that happened after 2022-01-01.

```
SELECT * FROM Crimes
WHERE Date > '2022-01-01';
```

# Q-2: Retrieve the suspect details where threat level is 'High'.

```
SELECT * FROM suspects
WHERE ThreatLevel = 'High';
```

# Q-3: Find the number of crimes that are currently "Open".

```
SELECT COUNT(*) AS OpenCrimes
FROM Crimes
WHERE Status = 'Open';
```

# Q-4: Find all crimes not located in 'Chicago'.

```
SELECT * FROM Crimes
WHERE Location != 'Chicago';
```

# Q-5: Display crimes that are 'Closed' and of type 'Homicide'.

```
SELECT * FROM Crimes
WHERE Status = 'Closed' AND CrimeType = 'Homicide';
```

# Q-6: List all cities where crimes of type 'Assault' occurred.

```
SELECT DISTINCT Location FROM Crimes
WHERE CrimeType = 'Assault';
```

# Q-7: Count crimes grouped by status.

```
SELECT Status, COUNT(*) AS Count
FROM Crimes
GROUP BY Status;
```

# Q-8: Find the average number of crimes per day.

```
SELECT AVG(CrimeCount)
FROM (SELECT COUNT(*) AS CrimeCount
FROM Crimes GROUP BY Date) AS DailyCounts;
```

# Q-9: List suspect IDs with more than 3 crimes associated.

```
SELECT SuspectID FROM Crimes
GROUP BY SuspectID
HAVING COUNT(*) > 3;
```

# Q-10: Get crime details along with the suspect’s name and age (INNER JOIN).

```
SELECT c.*, s.Name, s.Age FROM Crimes c
INNER JOIN Suspects s
ON c.SuspectID = s.SuspectID;
```

# Q-11: Show all crimes and matching suspect info (LEFT JOIN).

```
SELECT c.*, s.* FROM Crimes c

LEFT JOIN Suspects s

ON c.SuspectID = s.SuspectID;
```

# Q-12: Show all suspects and any crimes they are associated with (RIGHT JOIN).

```
SELECT s.*, c.* FROM Suspects s

RIGHT JOIN Crimes c

ON s.SuspectID = c.SuspectID;
```

# Q-13: Show all suspects and all crimes (FULL OUTER JOIN simulation using UNION).

```
SELECT * FROM Crimes c

LEFT JOIN Suspects s

ON c.SuspectID = s.SuspectID

UNION

SELECT * FROM Suspects s

RIGHT JOIN Crimes c

ON s.SuspectID = c.SuspectID;
```

# Q-14: List all crimes with no associated suspect (LEFT JOIN + WHERE IS NULL).

```
SELECT * FROM Crimes c

LEFT JOIN Suspects s

ON c.SuspectID = s.SuspectID

WHERE s.SuspectID IS NULL;
```

# Q-15: Display suspects who have not committed any crimes (RIGHT JOIN + WHERE IS NULL).

```sql
SELECT * FROM Crimes c
right JOIN Suspects s
ON c.SuspectID = s.SuspectID
WHERE s.SuspectID IS NULL;
```

# Q-16: Show the number of crimes each suspect was involved in (GROUP BY with JOIN).

```sql
SELECT s.Name, COUNT(c.CrimeID) AS CrimeCount
FROM Suspects s
JOIN Crimes c ON s.SuspectID = c.SuspectID
GROUP BY s.Name;
```

# Q-17: Find pairs of suspects with same age but different names.

```sql
SELECT a.Name, b.Name
FROM Suspects a
JOIN Suspects b ON a.Age = b.Age AND a.Name != b.Name;
```

# Q-18: Show pairs of suspects who have same threat level but different emails.

```sql
SELECT a.Name AS Suspect1, b.Name AS Suspect2
FROM Suspects a
JOIN Suspects b ON a.ThreatLevel = b.ThreatLevel AND a.Email != b.Email;
```

# Q-19: Get age of each suspect based on DOB (use YEAR(CURDATE()) - YEAR(DOB)).

```sql
SELECT Name, YEAR(CURDATE()) - YEAR(DOB) AS CalculatedAge
FROM Suspects;
```

# Q-20: Get age of each suspect based on DOB (use YEAR(CURDATE()) - YEAR(DOB)).

```
SELECT Location, COUNT(*) AS CrimeCount
FROM Crimes
GROUP BY Location
ORDER BY CrimeCount
DESC LIMIT 3;
```

# Q-21: Show crimes where suspect's age is above average age of all suspects.

```
SELECT * FROM Crimes
WHERE SuspectID
IN(SELECT SuspectID
FROM Suspects
WHERE Age > (SELECT AVG(Age)
FROM Suspects));
```

# Q-22: Create a view of all high threat level suspects and their crimes.

```
CREATE VIEW HighThreatView AS SELECT * FROM Suspects
WHERE ThreatLevel = 'High';
```

# Q-23: Create an alias for Suspect.Name as SuspectFullName and display it.

```
SELECT Name AS SuspectFullName
FROM Suspects;
```

# Q-24: Use CASE to label suspects as 'Senior' if age > 50, else 'Junior'.

```
SELECT Name, Age, CASE
WHEN Age > 50 THEN 'Senior' ELSE 'Junior' END AS AgeCategory FROM Suspects;
```

# Q-25: Create an index on the CrimeType column.

```
CREATE INDEX idx_crimetype

ON Crimes(CrimeType);
```

# Q-26: Write a cursor to iterate over all suspects with ThreatLevel = 'High' and insert their names into a separate audit table HighThreatAudit with timestamp of insertion.

```
DELIMITER //

CREATE PROCEDURE AuditHighThreat()

BEGIN

  DECLARE done INT DEFAULT FALSE;

  DECLARE s_name VARCHAR(255);

  DECLARE cur CURSOR FOR SELECT Name FROM Suspects WHERE ThreatLevel = 'High';

  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN cur;

  read_loop: LOOP

    FETCH cur INTO s_name;

    IF done THEN

      LEAVE read_loop;

    END IF;

    INSERT INTO HighThreatAudit(Name, Timestamp) VALUES (s_name, NOW());

  END LOOP;    CLOSE cur;    END //
```

# Q-27: Create a cursor that goes through each crime with status 'Open', and for each, prints (or logs) the CrimeType, Location, and SuspectID.

```
DELIMITER //

CREATE PROCEDURE PrintOpenCrimes()

BEGIN

  DECLARE done INT DEFAULT FALSE;
  DECLARE ct VARCHAR(50);

  DECLARE loc VARCHAR(100);

  DECLARE sid INT;

  DECLARE cur CURSOR FOR SELECT CrimeType, Location, SuspectID FROM Crimes WHERE Status = 'Open';

  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN cur;

  read_loop: LOOP

    FETCH cur INTO ct, loc, sid;

    IF done THEN

      LEAVE read_loop;

    END IF;

    SELECT ct, loc, sid;

  END LOOP;

  CLOSE cur;

END //
```

# Q-28: Develop a cursor to calculate and store the number of crimes associated with each suspect in a summary table SuspectCrimeCount.

```
DELIMITER //

CREATE PROCEDURE GenerateCrimeSummary()

BEGIN

  DECLARE done INT DEFAULT FALSE;

  DECLARE sid INT;

  DECLARE crime_count INT;

  DECLARE cur CURSOR FOR SELECT DISTINCT SuspectID FROM Crimes;

  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
  OPEN cur;
  read_loop: LOOP
    FETCH cur INTO sid;
    IF done THEN
      LEAVE read_loop;
    END IF;
    SELECT COUNT(*) INTO crime_count FROM Crimes WHERE SuspectID = sid;
    INSERT INTO SuspectCrimeCount(SuspectID, CrimeCount) VALUES (sid, crime_count);
  END LOOP;
  CLOSE cur;
END //
```

# Q-29: Create a stored procedure that accepts a SuspectID and returns all crimes involving that suspect.

```
DELIMITER //
CREATE PROCEDURE GetCrimesBySuspect(IN sid INT)
BEGIN
  SELECT * FROM Crimes WHERE SuspectID = sid;
END //
```

# Q-30: Write a stored procedure that takes a CrimeType and Status as input and returns the total number of crimes matching both.

```
DELIMITER //
CREATE PROCEDURE CountCrimeByTypeStatus(IN ctype VARCHAR(50), IN cstatus VARCHAR(50))
BEGIN
  SELECT COUNT(*) FROM Crimes WHERE CrimeType = ctype AND Status = cstatus;
END //
```

# Q-31: Develop a stored procedure that updates the Status of all crimes older than 3 years to 'Archived'.

```
DELIMITER //
CREATE PROCEDURE ArchiveOldCrimes()
BEGIN
  UPDATE Crimes SET Status = 'Archived' WHERE Date < CURDATE() - INTERVAL 3 YEAR;
END //
```

# Q-32: Create a stored procedure that accepts a ThreatLevel and returns a list of suspects with that level, sorted by age.

```
DELIMITER //
CREATE PROCEDURE ListSuspctsByThreat(IN level VARCHAR(50))
BEGIN
  SELECT * FROM Suspects WHERE ThreatLevel = level ORDER BY Age;
END //
```

# Q-33: Create a trigger that automatically updates a LastUpdated column in the Crimes table every time a row is updated.

```
DELIMITER //
CREATE TRIGGER UpdateTimestamp
BEFORE UPDATE ON Crimes
FOR EACH ROW
BEGIN
  SET NEW.LastUpdated = NOW();
END //
```

# Q-34: Write a trigger on the Suspects table that prevents insertion of a suspect whose age is below 18.

```
DELIMITER //
CREATE TRIGGER PreventUnderageSuspects
BEFORE INSERT ON Suspects
FOR EACH ROW
BEGIN
  IF NEW.Age < 18 THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Age must be at least 18.';
  END IF;
END //
```

# Q-35: Develop a trigger that logs any deletion from the Crimes table into an ArchiveCrimes table for backup purposes.

```
DELIMITER //
CREATE TRIGGER ArchiveDeletedCrimes
BEFORE DELETE ON Crimes
FOR EACH ROW
BEGIN
  INSERT INTO ArchiveCrimes SELECT * FROM Crimes WHERE CrimeID = OLD.CrimeID;
END //
```