Department of Information Technology
BABU BANARASI DAS
Institute of Technology & Management, Lucknow
Recognized by AICTE, Govt.of India & Affiliated to Dr. A.P.J Abdul Kalam Technical University,Lucknow,AKTU College Code=054

# LAB MANUAL
# OF
# DATABASE MANAGEMENT SYSTEM

## [BCS – 551]

## B.TECH, 3RD Year, Semester -VII

Dr. A.P.J. Abdul Kalam Tech. University
Uttar Pradesh

## 2025-26

## INFORMATION TECHNOLOGY

**Faculty Name:**

**Mr.Ankit Singh (Assistant Professor)**

**Approved by:**

**(Department Head/Coordinator)**

# MANUAL CONTENTS

This manual is intended for the third year students of Information Technology in the subject of Database Management System Lab. This manual typically contains practical/lab sessions related Database Management covering various aspects related to the subject of enhanced understanding.

Students are advised to thoroughly go through this manual rather than only topics mentioned in the syllabus as practical aspects are the key to understanding and conceptual visualization of theoretical aspects covered in the books.

Good Luck for your Enjoyable Laboratory Sessions

## Vision of the Institute

"To become a leading institute of providing professionally competent and socially responsive technocrats with high moral values."

## Mission of the Institute

M1.To create an eco-system for the dissemination of technical knowledge, to achieve academic excellence.

M2.To develop technocrats with creative skills and leadership qualities, to solve local and global challenges.

M3. To impart human values and ethics in students, to make them socially and eco-friendly responsible.

## Vision of the Department

“To produce globally competent professionals having social values and commitment to serve the global needs with the ability to work in an interdisciplinary environment."

## Mission of the Department

M1."To impart quality education to the students to enhance their ethical , professional and leadership qualities to make them globally competitive."

M2."To create a conducive environment in which students can explore computational problems and analyze them to identify the optimal solutions."

M3."To strive for continual enhancement of technical knowledge & innovation through industry interface to accomplish global needs."

# Program Educational Objectives(PEOs)

**PEO1:**Students must be able to apply software engineering principles to analyze complex computing problems and identify their solutions.

**PEO2:**Students must be able to analyze, design, and implement the latest technology-driven projects.

**PEO3:**Students must be able to work in a collaborative environment and understand the ethical , social, and economic impact of their work.

# Program Outcomes(POs)

PO's An Engineering Graduate of the Department of Computer Science and Engineering Program will be able to demonstrate:

**PO1 ENGINEERING KNOWLEDGE**: Apply the knowledge of mathematics, including discrete mathematics, probability, statistics and fundamentals of various engineering disciplines like computer science and engineering, electronic engineering and Electrical engineering in the core information technologies.

**PO2 PROBLEM ANALYSIS**: Analyze a problem and identify the computing requirements appropriate to its solution.

**PO3 DESIGN/DEVELOPMENT OF SOLUTIONS**: Design and implement hardware and software systems, components, process or program to meet the desired needs within reasonable economic, environmental, social, political, ethical,healthandsafety,manufacturability,andsustainabilityconstraints.

**PO4 CONDUCT INVESTIGATIONS OF COMPLEX PROBLEMS**: Reassess literature and indulge in research to use research-based knowledge and methods to design and conduct new experiments, as well as to organize, analyze and interpret data to produce draw valid Conclusions and recommendations.

**PO5 MODERNTOOLUSAGE**: Use appropriate techniques, resources, and modern engineering and IT tools necessary for computer engineering practice.

**PO6 THE ENGINEER AND SOCIETY**: Show the understanding of local and global impact of computing on individuals, organizations and society.

**PO7 ENVIRONMENT AND SUSTAINABILITY**: Integrate IT-based solutions in environmental contexts, and demonstrate the knowledge of need for sustainable development.

**PO8 ETHICS**: Demonstrate the knowledge of professional and ethical responsibilities along with the norms of the engineering practice.

**PO9 INDIVIDUAL AND TEAM WORK**: Demonstrate leadership and an ability to work as a member with responsibility to function on multi-disciplinary teams to accomplish a common goal.
**PO10 COMMUNICATION**: Demonstrate effectively communicate skills in both oral and written form with a range of audiences.

**PO11 PROJECT MANAGEMENT AND FINANCE**: Apply the knowledge and understanding of engineering and management principles to design, planned budget and propose IT project for an identified need within a specific scope.

**PO12 LIFE-LONG LEARNING**: Developed confidence to acquire new knowledge in the computing discipline and to engage in life-long learning.

# Program Specific Outcomes(PSOs)

**PSO1**: Able to design and implement the data structures and algorithms to deliver quality software products**.**

**PSO2**: Able to apply Artificial Intelligence and Machine Learning concepts to solve society-related needs.

## Course Evaluation Scheme

| Sr No | Subject Code | Subject Name | Periods | | | Evaluation Scheme | | | | Total | Credit |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | L | T | P | Sessional Assessment | | | PE | | |
| | | | | | | CT | TA | PS | | | |
| 1. | BCS551 | Database Management System Lab | 0 | 0 | 2 | - | - | 50 | 50 | 100 | 1 |

## Course Objectives:

*The teacher will explain:*

| | |
|---|---|
| 1. | Able to learn brief knowledge about modeling and Data definition language (DDL) operations on table. |
| 2. | Able to handle Data Base by applying data manipulation operations (DML) operations on table |
| 3. | Able to analyze normalization of data and uses of cursor trigger and views operations on table. |

*Pre- requisite: Fundamentals of Computer Science and Programming,* Basic Algorithms.

## Course Outcomes (COs)

| **Course Outcomes:** The students should be able to: | | **Bloom's Level** |
|---|---|---|
| **CO1** | To implement different types of DDL, DML and DCL queries. | |
| **CO2** | Data Definition Language(DDL) Statements: (Create table, Alter table, Drop table) | |
| **CO3** | Data Manipulation Language(DML) Statements | |
| **CO4** | Data Query Language(DQL) Statements: (Select statement with operations like Where clause, Order by, Logical operators, Scalar functions and Aggregate functions) | |
| **CO5** | To implement programs using different procedures, functions, packages, etc. | |

## CO-PO-PSO Mapping

| | PO1 | PO2 | PO3 | PO4 | PO5 | PO6 | PO7 | PO8 | PO9 | PO10 | PO11 | PO12 | PSO1 | PSO2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CO1 | 2 | 2 | 3 | 2 | 1 | - | - | - | 2 | 1 | 1 | 1 | 2 | 1 |
| CO2 | 3 | 1 | 1 | 1 | 1 | - | - | - | 2 | 3 | 3 | 3 | 3 | 1 |
| CO3 | 3 | 2 | 3 | 2 | 1 | - | - | - | 2 | 1 | 1 | 2 | 2 | 3 |
| CO4 | 1 | 2 | 3 | 2 | - | - | - | - | 3 | 2 | 3 | 3 | 1 | 2 |
| CO5 | 1 | 1 | 3 | 3 | 2 | - | - | - | 1 | 3 | 3 | 1 | 2 | 2 |
| Avg. | 2 | 2 | 3 | 2 | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 |

The extent of mapping is as follows: 1 for low, 2 for moderate, 3 for high &"-" for No correlation between CO & PO.

## List of Programs as per AKTU

1. Installing oracle/ MYSQL
2. Creating Entity-Relationship Diagram using case tools.
3. Writing SQL statements Using ORACLE /MYSQL:
   a) Writing basic SQL SELECT statements.
   b) Restricting and sorting data.
   c) Displaying data from multiple tables.
   d) Aggregating data using group function.
   e) Manipulating data.
   e) Creating and managing tables.
4. Normalization
5. Creating cursor
6. Creating procedure and functions
7. Creating packages and triggers
8. Design and implementation of payroll processing system
9. Design and implementation of Library Information System
10. Design and implementation of Student Information System
11. Automatic Backup of Files and Recovery of Files
12. Mini project (Design & Development of Data and Application) for following:
    a) Inventory Control System.
    b) Material Requirement Processing.
    c) Hospital Management System.
    d) Railway Reservation System.
    e) Personal Information System.
    f) Web Based User Identification System.
    g) Timetable Management System.
    h) Hotel Management System

| |
|---|
| 1.Installing Oracle and creating Entity-Relationship Diagram |
| 2.Write the queries for Data Definition and Data Manipulation Language. |
| 3.Write SQL queries using sql operators and logical operations (=,<,>,etc.) |
| 4.Write SQL query using character, number, date and group functions. |
| 5.a.Write SQL queries for relational algebra.<br>b.Write SQL queries for extracting data from more than one table. |
| 6.Write SQL queries for sub queries, nested queries. |
| 7.Write program by the use of PL/SQL. |
| 8.Concepts for ROLL BACK, COMMIT & CHECK POINTS. |
| 9.Creating Procedure and Functions in Oracle. |
| 10.Creating Cursor and Trigger in Oracle. |

## List of Programs with enhancement of Faculty

| S. No. | Name of the Experiment | CO | BTL | Lab Conduction Date | Evaluated Marks | Faculty Signature with Date |
|---|---|---|---|---|---|---|
| 1. | Installing Oracle and creating Entity-Relationship Diagram | 1 | 4 | | | |
| 2. | Write the queries for Data Definition and Data Manipulation Language. | 2 | 3 | | | |
| 3. | Write SQL queries using sql operators and logical operations (=,<,>,etc) | 2 | 3 | | | |
| 4. | Write SQL query using character, number, date and group functions. | 2 | 3 | | | |
| 5. | a.Write SQL queries for relational algebra.<br>b.Write SQL queries for extracting data from more than one table. | 2 | 4 | | | |
| 6. | Write SQL queries for sub queries, nested queries. | 2 | 5 | | | |
| 7. | Write program by the use of PL/SQL. | 3 | 4 | | | |
| 8. | Concepts for ROLL BACK, COMMIT & CHECK POINTS. | 4 | 4 | | | |
| 9. | Creating Procedure and Functions in Oracle. | 5 | 4 | | | |
| 10. | Creating Cursor and Trigger in Oracle. | 5 | 4 | | | |

# INDEX

## Softwares & Hardwares Required

1. Operating System: Windows, Linux, macOS.
2. Software: MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server.
3. Hardware: **Processor**: At least **Intel i3, RAM**: A minimum of **8 GB RAM,** Storage.

**EXPERIMENT: 1**

## Installation steps for MySQL:

- Download MySQL from https://dev.mysql.com/downloads/mysql/

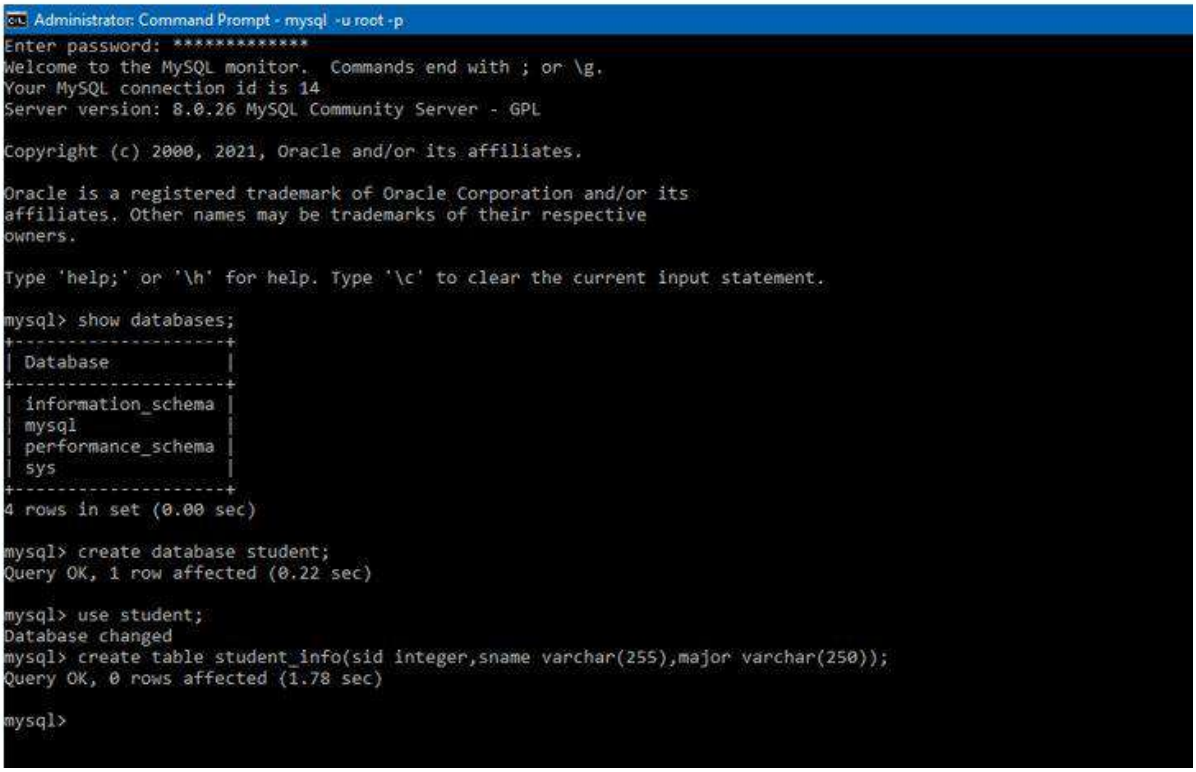
**Installing MySQL on Windows**: Your downloaded MySQL is neatly packaged with an installer. Download the installer package, unzip it anywhere and run setup.exe. By

default, this process will install everything under C:\mysql

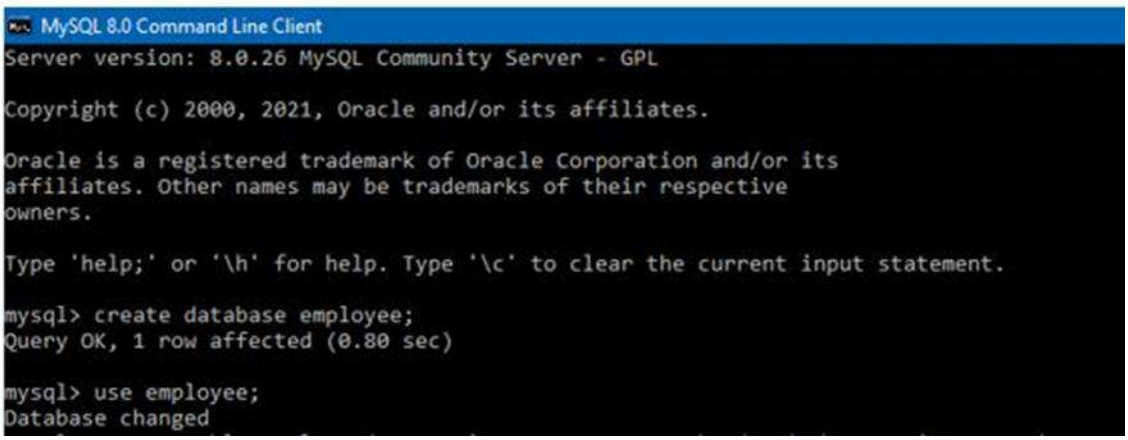
**Verify MySQL installation:**

- **Step- 1 :** Open from start

- **Step-2 :** enter the password you have set during installation .




- **Step-3 :** if you already have created data bases then use them else create new database.



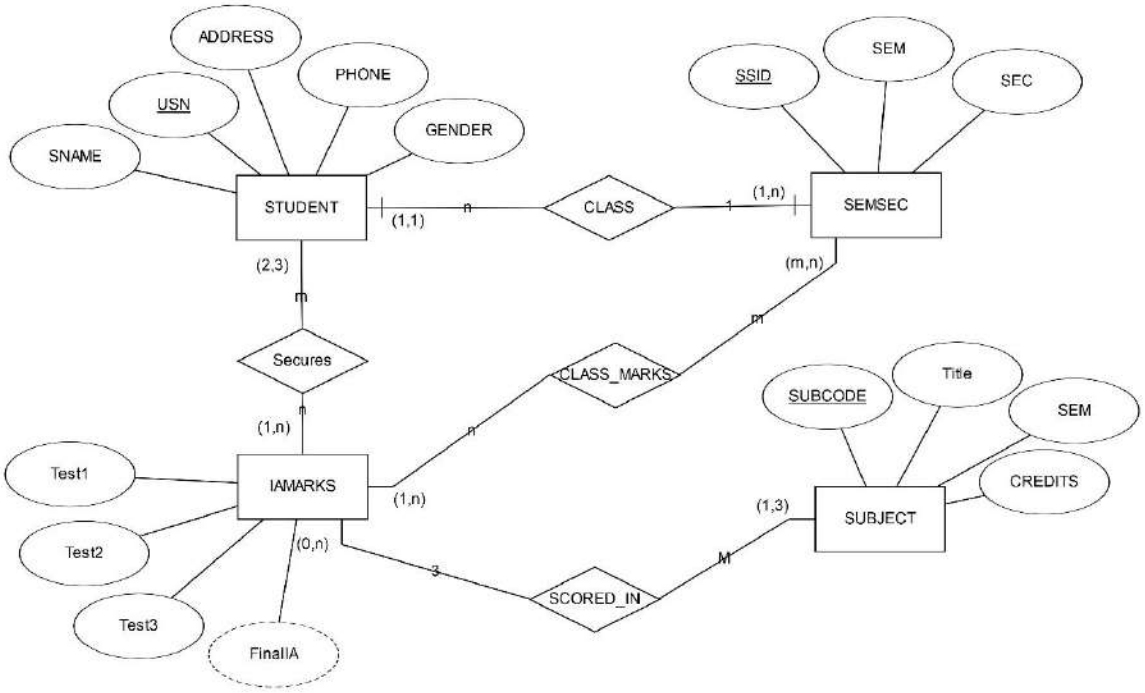
# Creating Entity-Relationship Diagram

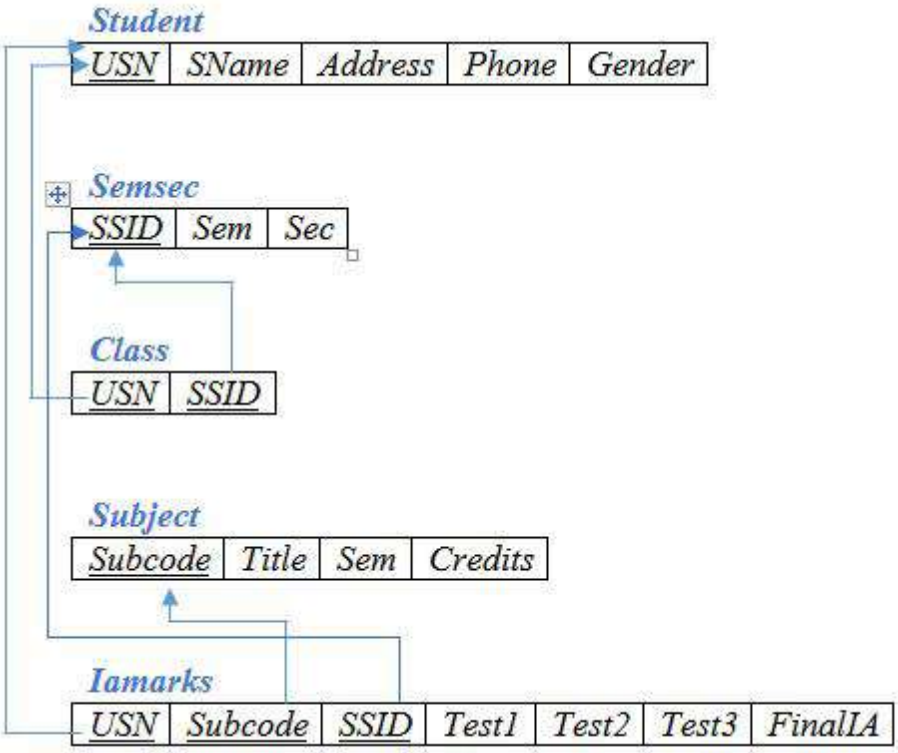
Consider following databases and draw ER diagram and convert entities and relationships to relation table for a given scenario.

1. COLLEGE DATABASE: STUDENT (USN, SName, Address, Phone, Gender) SEMSEC (SSID, Sem, Sec) CLASS (USN, SSID) SUBJECT (Subcode, Title, Sem, Credits) IAMARKS (USN, Subcode, SSID, Test1, Test2, Test3, FinalIA)
2. COMPANY DATABASE: EMPLOYEE (SSN, Name, Address, Sex, Salary, SuperSSN, DNo) DEPARTMENT (DNo, DName, MgrSSN, MgrStartDate) DLOCATION (DNo,DLoc) PROJECT (PNo, PName, PLocation, DNo) WORKS_ON (SSN, PNo, Hours)

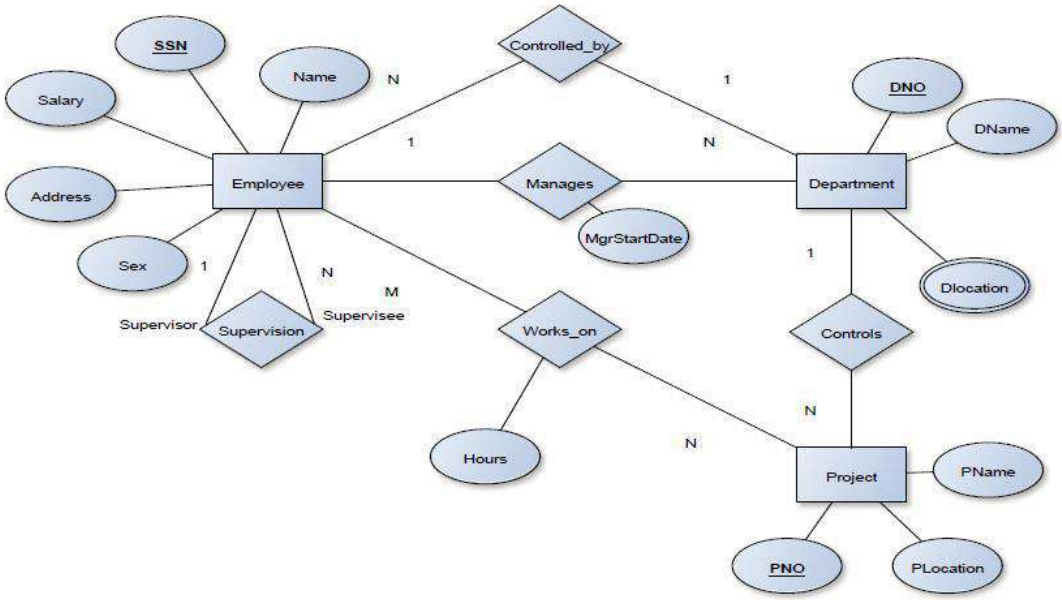
# SOLUTION

**College Database: E-R Diagram**

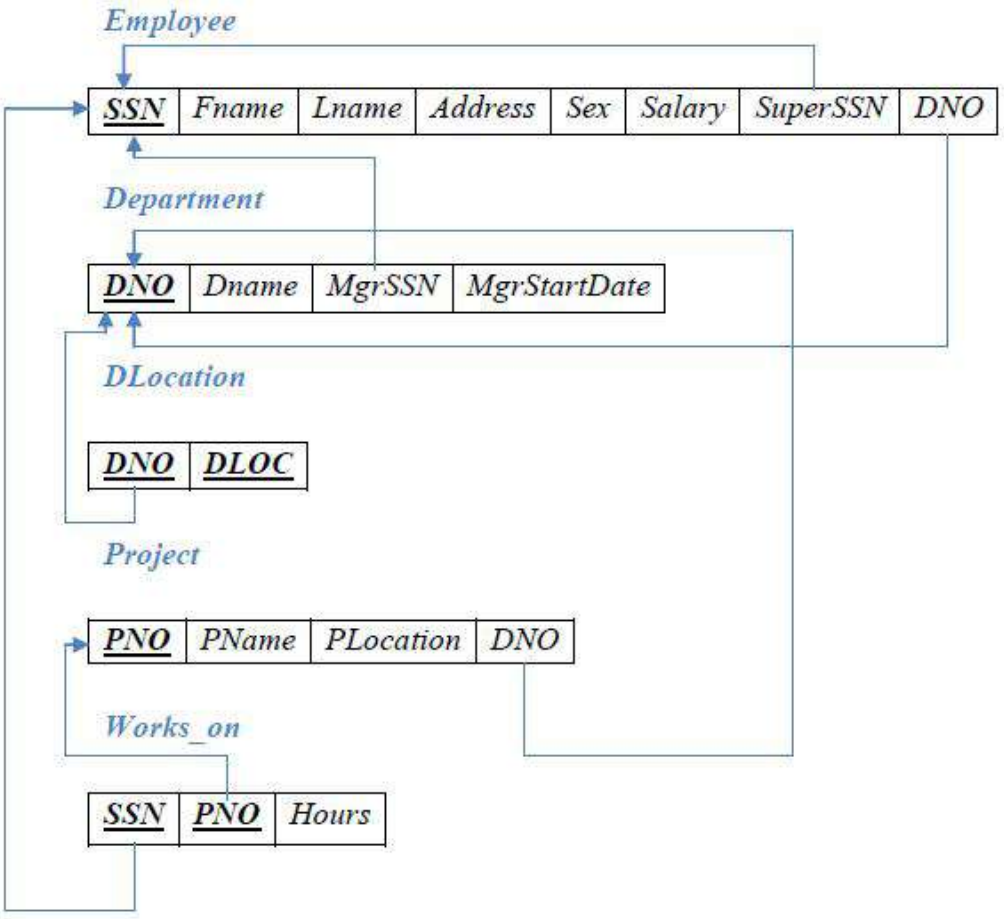





**COMPANY DATABASE:**




**Schema Diagram**




# EXPERIMENT: 2
# Writing SQL statements for Data Definition and Data Manipulation Language Using ORACLE /MYSQL

## SQL Commands

- SQL commands are instructions. It is used to communicate with the database. It is also used to perform specific tasks, functions, and queries of data.
- SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.

### Types of SQL Commands

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.

### 1. Data Definition Language (DDL)

- DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.

- All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

**a. CREATE** It is used to create a new table in the database.

**Syntax:**

```
1. CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);
```

**Example:**

```
1. CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCHAR2(100), DOB DATE);
```

**b. DROP:** It is used to delete both the structure and record stored in the table.

**Syntax**

```
1. DROP TABLE table_name;
```

**Example**

```
1. DROP TABLE EMPLOYEE;
```

**c. ALTER:** It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

**Syntax:**

To add a new column in the table

```
1. ALTER TABLE table_name ADD column_name COLUMN-definition;
```

To modify existing column in the table:

```
1. ALTER TABLE table_name MODIFY(column_definitions....);
```

**EXAMPLE**

1. ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));
2. ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR2(20));

**d. TRUNCATE:** It is used to delete all the rows from the table and free the space containing the table.

**Syntax:**

1. TRUNCATE TABLE table_name;

**Example:**TRUNCATE TABLE EMPLOYEE;

## 2. Data Manipulation Language

- DML commands are used to modify the database. It is responsible for all form of changes in the database.
- The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

- INSERT
- UPDATE
- DELETE

**a. INSERT:** The INSERT statement is a SQL query. It is used to insert data into the row of a table.

**Syntax:**

1. INSERT INTO TABLE_NAME
2. (col1, col2, col3,.... col N)
3. VALUES (value1, value2, value3, .... valueN);

Or

1. INSERT INTO TABLE_NAME
2. VALUES (value1, value2, value3, .... valueN);

**For example:**

1. INSERT INTO javatpoint (Author, Subject) VALUES ("Sonoo", "DBMS");

**b. UPDATE:** This command is used to update or modify the value of a column in the table.

**Syntax:**

1. UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITION]

**For example:**

1. UPDATE students
2. SET User_Name = 'Sonoo'
3. WHERE Student_Id = '3'

**c. DELETE:** It is used to remove one or more row from a table.

**Syntax:**

1. DELETE FROM table_name [WHERE condition];

**For example:**

1. DELETE FROM javatpoint
2. WHERE Author="Sonoo";

## EXPERIMENT: 3

## Writing SQL queries using SQL operators and logical operations (=,<,>,etc)

## SQL Arithmetic Operators:

# SQL Bitwise Operators

| Operator | Description |
|---|---|
| & | Bitwise AND |
| \| | Bitwise OR |

| | |
|---|---|
| ^ | Bitwise exclusive OR |

# SQL Comparison Operators

| Operator | Description |
|---|---|
| = | Equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| <> | Not equal to |

# SQL Logical Operators

| Operator | Description | Example |
|---|---|---|
| ALL | TRUE if all of the subquery values meet the | |

| | |
|---|---|
| | condition |
| AND | TRUE if all the conditions separated by AND is TRUE |
| ANY | TRUE if any of the subquery values meet the condition |
| BETWEEN | TRUE if the operand is within the range of comparisons |
| EXISTS | TRUE if the subquery returns one or more records |
| IN | TRUE if the operand is equal to one of a list of expressions |
| LIKE | TRUE if the operand matches a pattern |
| NOT | Displays a record if the condition(s) is NOT TRUE |
| OR | TRUE if any of the conditions separated by OR is TRUE |
| SOME | TRUE if any of the subquery values meet the condition |

# EXPERIMENT: 4

## Writing SQL queries using Aggregate Functions

## Aggregate Functions

All aggregate functions are deterministic. In other words, aggregate functions return the same value each time that they are called, when called with a specific set of input values. See Deterministic and Nondeterministic Functions for more information about function determinism. The OVER clause may follow all aggregate functions, except the STRING_AGG, GROUPING or GROUPING_ID functions.

Use aggregate functions as expressions only in the following situations:

- The select list of a SELECT statement (either a subquery or an outer query).
- A HAVING clause.

Transact-SQL provides the following aggregate functions:

- APPROX_COUNT_DISTINCT
- AVG
- CHECKSUM_AGG
- COUNT
- COUNT_BIG
- GROUPING
- GROUPING_ID
- MAX
- MIN

1. **Count():**

- ***Count(*):*** Returns total number of records .i.e 6.
  ***Count(salary):*** Return number of Non Null values over the column salary. i.e 5.
  ***Count(Distinct Salary):*** Return number of distinct Non Null values over the column salary .i.e 4

2. **Sum():**

***sum(salary):*** Sum all Non Null values of Column salary i.e., 310
***sum(Distinct salary):*** Sum of all distinct Non-Null values i.e., 250.

3. **Avg():**

***Avg(salary)*** = Sum(salary) / count(salary) = 310/5
***Avg(Distinct salary)*** = sum(Distinct salary) / Count(Distinct Salary) = 250/4

4. **Min():**

***Min(salary):*** Minimum value in the salary column except NULL i.e., 40.

**5. Max():**

***Max(salary):*** Maximum value in the salary i.e., 80.

# EXPERIMENT: 5

# Writing SQL queries using character, number, date and group functions

1. Character Functions

Character functions operate on string data types (e.g., VARCHAR2, CHAR).

sql

CopyEdit

```
SELECT first_name,
    UPPER(last_name) AS upper_last_name,
    LOWER(first_name) AS lower_first_name,
    LENGTH(first_name) AS name_length,
    SUBSTR(last_name, 1, 3) AS first_three_letters
FROM employees;
```

UPPER(): Converts text to uppercase.

LOWER(): Converts text to lowercase.

LENGTH(): Returns the length of a string.

SUBSTR(): Extracts a portion of the string.

---

2. Number Functions

Number functions perform calculations on numeric data.

sql

CopyEdit

```
SELECT employee_id, salary,
    ROUND(salary, -3) AS rounded_salary,
```

```
    TRUNC(salary, -2) AS truncated_salary,
    MOD(salary, 1000) AS remainder_salary
FROM employees;
```

ROUND(): Rounds a number to the specified decimal places.

TRUNC(): Truncates a number to the specified decimal places.

MOD(): Returns the remainder of division.

---

3. Date Functions

Date functions operate on date and time values.

sql

CopyEdit

```
SELECT employee_id, hire_date,
    SYSDATE AS current_date,
    MONTHS_BETWEEN(SYSDATE, hire_date) AS months_since_hire,
    ADD_MONTHS(hire_date, 6) AS six_months_later,
    NEXT_DAY(hire_date, 'FRIDAY') AS next_friday
FROM employees;
```

SYSDATE: Returns the current date.

MONTHS_BETWEEN(): Returns the difference in months between two dates.

ADD_MONTHS(): Adds a specified number of months to a date.

NEXT_DAY(): Finds the next occurrence of a specific weekday.

---

4. Group Functions

Group functions operate on multiple rows and return aggregated values.

sql

CopyEdit

```
SELECT department_id,
    COUNT(*) AS employee_count,
    AVG(salary) AS avg_salary,
    MAX(salary) AS highest_salary,
    MIN(salary) AS lowest_salary,
    SUM(salary) AS total_salary
FROM employees
GROUP BY department_id;
```

COUNT(): Counts the number of rows.

AVG(): Returns the average value.

MAX(): Finds the maximum value.

MIN(): Finds the minimum value.

SUM(): Calculates the sum of a column.

---

These queries cover a variety of SQL functions that are useful for different scenarios.

# EXPERIMENT: 6

# Writing SQL queries for extracting data from more than one table and for sub queries, nested queries

## 1. Extracting Data from More Than One Table Using JOINs

a) INNER JOIN

sql

CopyEdit

```
SELECT e.employee_id, e.first_name, e.last_name, d.department_name
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_id;
```

Retrieves employees along with their department names.

INNER JOIN returns only matching records from both tables.

---

b) LEFT JOIN (LEFT OUTER JOIN)

sql

CopyEdit

SELECT e.employee_id, e.first_name, e.last_name, d.department_name

FROM employees e

LEFT JOIN departments d ON e.department_id = d.department_id;

Returns all employees, even those who do not belong to any department (NULL in department_name).

---

c) RIGHT JOIN (RIGHT OUTER JOIN)

sql

CopyEdit

SELECT e.employee_id, e.first_name, e.last_name, d.department_name

FROM employees e

RIGHT JOIN departments d ON e.department_id = d.department_id;

Retrieves all departments, including those without employees.

---

d) FULL OUTER JOIN

sql

CopyEdit

SELECT e.employee_id, e.first_name, e.last_name, d.department_name

FROM employees e

FULL OUTER JOIN departments d ON e.department_id = d.department_id;

Returns all employees and all departments, showing NULL if there is no match.

---

2. Using Subqueries (Nested Queries)

Subqueries allow you to execute a query inside another query.

a) Simple Subquery (Fetching Employee Details with the Highest Salary)

sql

CopyEdit

```
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE salary = (SELECT MAX(salary) FROM employees);
```

Finds the employee who has the highest salary.

---

b) Correlated Subquery (Fetching Employees Who Earn More Than the Average Salary of Their Department)

sql

CopyEdit

```
SELECT employee_id, first_name, last_name, salary, department_id
FROM employees e1
WHERE salary > (SELECT AVG(salary)
        FROM employees e2
        WHERE e1.department_id = e2.department_id);
```

Compares each employee's salary with the average salary of their department.

---

c) Subquery in FROM Clause (Inline View)

sql

CopyEdit

```
SELECT department_id, avg_salary
FROM (SELECT department_id, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department_id) dept_avg
WHERE avg_salary > 5000;
```

Retrieves departments where the average salary is above 5000.

---

d) EXISTS Clause (Checking if a Department Has Employees)

sql

CopyEdit

```
SELECT department_id, department_name
FROM departments d
WHERE EXISTS (SELECT 1 FROM employees e WHERE e.department_id = d.department_id);
```

Returns departments that have employees.

# EXPERIMENT: 7
# Normalization

**Normalization** is the process of minimizing **redundancy** from a relation or set of relations. Redundancy in relation may cause insertion, deletion and updation anomalies. So, it helps to minimize the redundancy in relations. **Normal forms** are used to eliminate or reduce redundancy in database tables.

## 1. First Normal Form –

If a relation contain composite or multi-valued attribute, it violates first normal form or a relation is in first normal form if it does not contain any composite or multi-valued attribute. A relation is in first normal form if every attribute in that relation is **singled valued attribute**.

- **Example 1 –** Relation STUDENT in table 1 is not in 1NF because of multi-valued attribute STUD_PHONE. Its decomposition into 1NF has been shown in table 2.

| STUD_NO | STUD_NAME | STUD_PHONE | STUD_STATE | STUD_COUNTRY |
|---|---|---|---|---|
| 1 | RAM | 9716271721, 9871717178 | HARYANA | INDIA |
| 2 | RAM | 9898297281 | PUNJAB | INDIA |
| 3 | SURESH | | PUNJAB | INDIA |

Table 1

Conversion to first normal form

| STUD_NO | STUD_NAME | STUD_PHONE | STUD_STATE | STUD_COUNTRY |
|---|---|---|---|---|
| 1 | RAM | 9716271721 | HARYANA | INDIA |
| 1 | RAM | 9871717178 | HARYANA | INDIA |
| 2 | RAM | 9898297281 | PUNJAB | INDIA |
| 3 | SURESH | | PUNJAB | INDIA |

Table 2

- **Example 2 –**

| ID | Name | Course |
|---|---|---|
| 1 | A | C1,C2 |
| 2 | E | C3 |
| 3 | M | C2,C3 |

In the above table Course is a multi valued attribute so it is not in 1NF.

Below Table is in 1NF as there is no multi valued attribute

| ID | Name | Course |
|---|---|---|
| 1 | A | C1 |
| 1 | A | C2 |
| 2 | E | C3 |
| 3 | M | C2 |
| 3 | M | C3 |

## 2. Second Normal Form –

To be in second normal form, a relation must be in first normal form and relation must not contain any partial dependency. A relation is in 2NF if it has **No Partial Dependency,** i.e., no non-prime attribute (attributes which are not part of any candidate key) is dependent on any proper subset of any candidate key of the table.
**Partial Dependency –** If the proper subset of candidate key determines non-prime attribute, it is called partial dependency.

- **Example 1 –** Consider table-3 as following below.

| STUD_NO | COURSE_NO | Course_FEE |
|---|---|---|
| 1 | C1 | 1000 |
| 2 | C2 | 1500 |

| | | |
|---|---|---|
| 1 | C4 | 2000 |
| 4 | C3 | 1000 |
| 4 | C1 | 1000 |
| 2 | C5 | 2000 |

{Note that, there are many courses having the same course fee. }

Here,
COURSE_FEE cannot alone decide the value of COURSE_NO or STUD_NO;
COURSE_FEE together with STUD_NO cannot decide the value of COURSE_NO;
COURSE_FEE together with COURSE_NO cannot decide the value of STUD_NO;
Hence,
COURSE_FEE would be a non-prime attribute, as it does not belong to the one only candidate key {STUD_NO, COURSE_NO} ;
But, COURSE_NO -> COURSE_FEE , i.e., COURSE_FEE is dependent on COURSE_NO, which is a proper subset of the candidate key. Non-prime attribute COURSE_FEE is dependent on a proper subset of the candidate key, which is a partial dependency and so this relation is not in 2NF.

To convert the above relation to 2NF,
we need to split the table into two tables such as :
Table 1: STUD_NO, COURSE_NO
Table 2: COURSE_NO, COURSE_FEE

**TABLE1**

| STUD_NO | COURSE_NO |
|---|---|
| 1 | C1 |
| 2 | C2 |
| 1 | C4 |
| 4 | C3 |
| 4 | C1 |
| 2 | C5 |

**TABLE 2**

| COURSE_NO | COURSE_FEE |
|---|---|
| C1 | 1000 |
| C2 | 1500 |
| C3 | 1000 |
| C4 | 2000 |
| C5 | 2000 |

**NOTE:** 2NF tries to reduce the redundant data getting stored in memory. For instance, if there are 100 students taking C1 course, we dont need to store its Fee as 1000 for all the 100 records, instead once we can store it in the second table as the course fee for C1 is 1000.

- **Example 2 –** Consider following functional dependencies in relation R (A, B , C, D )
- AB -> C [A and B together determine C]

BC -> D [B and C together determine D]

In the above relation, AB is the only candidate key and there is no partial dependency, i.e., any proper subset of AB doesn't determine any non-prime attribute.

### 3. Third Normal Form –

A relation is in third normal form, if there is **no transitive dependency** for non-prime attributes as well as it is in second normal form.
A relation is in 3NF if **at least one of the following condition holds** in every non-trivial function dependency X –> Y

1. X is a super key.
2. Y is a prime attribute (each element of Y is part of some candidate key).

| STUD_NO | STUD_NAME | STUD_STATE | STUD_COUNTRY | STUD_AGE |
|---|---|---|---|---|
| 1 | RAM | HARYANA | INDIA | 20 |
| 2 | RAM | PUNJAB | INDIA | 19 |
| 3 | SURESH | PUNJAB | INDIA | 21 |

Table 4

**Transitive dependency –** If A->B and B->C are two FDs then A->C is called transitive dependency.

- **Example 1 –** In relation STUDENT given in Table 4,
  FD set: {STUD_NO -> STUD_NAME, STUD_NO -> STUD_STATE, STUD_STATE -> STUD_COUNTRY, STUD_NO -> STUD_AGE}
  Candidate Key: {STUD_NO}

  For this relation in table 4, STUD_NO -> STUD_STATE and STUD_STATE -> STUD_COUNTRY are true. So STUD_COUNTRY is transitively dependent on STUD_NO. It violates the third normal form. To convert it in third normal form, we will decompose the relation STUDENT (STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_COUNTRY_STUD_AGE) as:
  STUDENT (STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_AGE)
  STATE_COUNTRY (STATE, COUNTRY)

- **Example 2 –** Consider relation R(A, B, C, D, E)
  A -> BC,
  CD -> E,
  B -> D,
  E -> A
  All possible candidate keys in above relation are {A, E, CD, BC} All attribute are on right sides of all functional dependencies are prime.

### 4. Boyce-Codd Normal Form (BCNF) –

A relation R is in BCNF if R is in Third Normal Form and for every FD, LHS is super key.
A relation is in BCNF iff in every non-trivial functional dependency X –> Y, X is a super key.

- **Example 1 –** Find the highest normal form of a relation R(A,B,C,D,E) with FD set as {BC->D, AC->BE, B->E}
  Step 1. As we can see, (AC)+ ={A,C,B,E,D} but none of its subset can determine all attribute of relation, So AC will be candidate key. A or C can't be derived from any other attribute of the relation, so there will be only 1 candidate key {AC}.
  Step 2. Prime attributes are those attribute which are part of candidate key {A, C} in this example and others will be non-prime {B, D, E} in this example.
  Step 3. The relation R is in 1st normal form as a relational DBMS does not allow multi-valued or composite attribute.
  The relation is in 2nd normal form because BC->D is in 2nd normal form (BC is not a proper subset of candidate key AC) and AC->BE is in 2nd normal form (AC is candidate key) and B->E is in 2nd normal form (B is not a proper subset of candidate key AC).
  The relation is not in 3rd normal form because in BC->D (neither BC is a super key nor D is a prime attribute) and in B->E (neither B is a super key nor E is a prime attribute) but to satisfy 3rd normal for, either LHS of an FD should be super key or RHS should be prime attribute.
  So the highest normal form of relation will be 2nd Normal form.
- **Example 2 –**For example consider relation R(A, B, C)
  A -> BC,
  B ->
  A and B both are super keys so above relation is in BCNF.

.

# EXPERIMENT 8

## Concepts for ROLL BACK, COMMIT & SAVE POINTS

Data Control Language

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

- Grant
- Revoke

**a. Grant:** It is used to give user access privileges to a database.

**Example**

1. GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;

**b. Revoke:** It is used to take back permissions from the user.

**Example**

```
1. REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;
```

## 4. Transaction Control Language

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under TCL:

- COMMIT
- ROLLBACK
- SAVEPOINT

**a. Commit:** Commit command is used to save all the transactions to the database.

**Syntax:**

```
1. COMMIT;
```

**Example:**

```
1. DELETE FROM CUSTOMERS
2. WHERE AGE = 25;
3. COMMIT;
```

**b. Rollback:** Rollback command is used to undo transactions that have not already been saved to the database.

**Syntax:**

```
1. ROLLBACK;
```

**Example:**

```
1. DELETE FROM CUSTOMERS
2. WHERE AGE = 25;
3. ROLLBACK;
```

**c. SAVEPOINT:** It is used to roll the transaction back to a certain point without rolling back the entire transaction.

**Syntax:**

```
1. SAVEPOINT SAVEPOINT_NAME;
```

## 5. Data Query Language

DQL is used to fetch the data from the database.

It uses only one command:

- SELECT

**a. SELECT:** This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.

**Syntax:**

```
1. SELECT expressions
2. FROM TABLES
3. WHERE conditions;
```

**For example:**

```
1. SELECT emp_name
2. FROM employee
3. WHERE age > 20;
```

# EXPERIMENT 9
# Creating Cursor and Triggers

In MySQL, Cursor can also be created. Following are the steps for creating a cursor.

## 1. Declare Cursor

A cursor is a select statement, defined in the declaration section in MySQL.

## Syntax

```
1. DECLARE cursor_name CURSOR FOR
2. Select statement;
```

### Parameter:

**cursor_name:** name of the cursor

**select_statement:** select query associated with the cursor

## 2. Open Cursor

After declaring the cursor the next step is to open the cursor using open statement.

### Syntax

1. **Open** cursor_name;

### Parameter:

**cursor_name:** name of the cursor which is already declared.

## 3. Fetch Cursor

After declaring and opening the cursor, the next step is to fetch the cursor. It is used to fetch the row or the column.

### Syntax

1. **FETCH** [ **NEXT** [ **FROM** ] ] cursor_name **INTO** variable_list;

### Parameter:

**cursor_name:** name of the cursor

**variable_list:** variables, comma separated, etc. is stored in a cursor for the result set

## 4. Close Cursor

The final step is to close the cursor.

### Syntax

1. **Close** cursor_name;

### Parameter:

**Cursor_name:** name of the cursor

### Example for the cursor:

```
MySQL 8.0 Command Line Client
mysql> use test1;
Database changed
mysql> select *from table1;
+------+----------+-------+
| id   | name     | class |
+------+----------+-------+
|    1 | Shristee | MCA   |
|    2 | Ajay     | BCA   |
|    3 | Shweta   | MCA   |
|    4 | Dolly    | BCA   |
|    5 | Heena    | MCA   |
|    6 | Kiran    | BCA   |
|    7 | Sonal    | MCA   |
|    8 | Dimple   | BCA   |
|    9 | Shyam    | MCA   |
|   10 | Mohit    | BCA   |
+------+----------+-------+
10 rows in set (1.24 sec)
```

**Step 1:** Open the database and table.

**Step 2:** Now create the cursor.

**Query:**

```
mysql> DELIMITER $$
mysql> CREATE PROCEDURE list_name (INOUT name_list varchar(4000))
    -> BEGIN
    -> DECLARE is_done INTEGER DEFAULT 0;
    -> DECLARE s_name varchar(100) DEFAULT "";
    -> DECLARE stud_cursor CURSOR FOR
    -> SELECT name FROM table1;
    -> DECLARE CONTINUE HANDLER FOR NOT FOUND SET is_done = 1;
    -> OPEN stud_cursor;
    -> get_list: LOOP
    -> FETCH stud_cursor INTO s_name;
    -> IF is_done = 1 THEN
    -> LEAVE get_list;
    -> END IF;
    -> SET name_list = CONCAT(s_name, ";",name_list);
    -> END LOOP get_list;
    -> CLOSE stud_cursor;
    -> END$$
Query OK, 0 rows affected (0.24 sec)
```

**Step 3:** Now call the cursor.

**Query:**

```
1. SET @name_list ="";
2. CALL list_name(@name_list);
3. SELECT @name_list;
```

```
mysql> SET @name_list ="";
Query OK, 0 rows affected (0.00 sec)

mysql> CALL list_name(@name_list);
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT @name_list;
+-------------------------------------------------------------------+
| @name_list                                                        |
+-------------------------------------------------------------------+
| Mohit;Shyam;Dimple;Sonal;Kiran;Heena;Dolly;Shweta;Ajay;Shristee; |
+-------------------------------------------------------------------+
1 row in set (0.00 sec)
```

## Creating triggers

MySQL triggers apply restrictions to tables when adding, updating, or removing table rows.

Columns in MySQL apply a slight amount of value limitations. For example, setting a column data type as *tiny int* and *not null* requires a small number value input. Still, more restrictions are needed to maintain the integrity of data.

# Prerequisites

A trigger is a named MySQL object that activates when an event occurs in table. Triggers are a particular type of stored procedures associated with a specific table.

Triggers allow access to values from the table for comparison purposes using `NEW` and `OLD`. The availability of the modifiers depends on the trigger event you use:

Checking or modifying a value when trying to insert data makes the `NEW.<column name>` modifier available. This is because a table is updated with new content. In contrast, an `OLD.<column name>` value does not exist for an insert statement because there is no information exists in its place beforehand.

When updating a table row, both modifiers are available. There is `OLD.<column name>` data which we want to update to `NEW.<column name>` data.

Finally, when removing a row of data, the `OLD.<column name>` modifier accesses the removed value. The `NEW.<column name>` does not exist because nothing is replacing the old value upon removal.

## MySQL Trigger Example

As an example of an applied trigger, inserting new values into the table *person* yields a different result than the original input:

```
mysql> INSERT INTO person VALUES ('matthew', 25), ('mark', 20);
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0

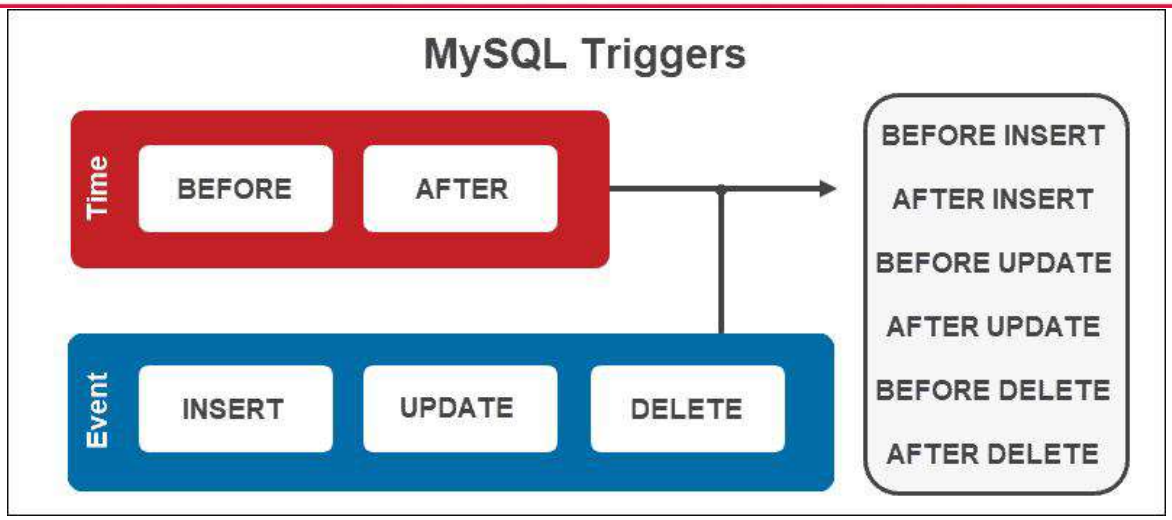
mysql> SELECT * FROM person;
+---------+------+
| name    | age  |
+---------+------+
| Matthew |   25 |
| Mark    |   20 |
+---------+------+
```

Notice the inserted names were initially lowercase. When selecting the table, the first letter shows as capitalized. Even though there is no indication of anything different from a regular insert statement, the trigger fired **before the insert statement** to capitalize the first letter of the name.

## Using MySQL Triggers

Every trigger associated with a table has a unique name and function based on two factors:

1. **Time**. `BEFORE` or `AFTER` a specific row event.
2. **Event**. `INSERT`, `UPDATE` or `DELETE`.




## Create Triggers

Use the `CREATE TRIGGER` statement syntax to create a new trigger:

```
CREATE TRIGGER <trigger name><trigger time ><trigger event>
ON <table name>
FOR EACH ROW
<trigger body>;
```

The best practice is to name the trigger with the following information

`<trigger time>_<table name>_<trigger event>`

For example, if a trigger fires **before insert** on a table named **employee**, the best convention is to call the trigger:

```
before_employee_insert
```

Alternatively, a common practice is to use the following format:

```
<table name>_<first letter of trigger time><first letter of trigger name>
```

The **before insert** trigger name for the table **employee** looks like this:

`employee_bi`

The trigger executes at a specific time of an event on a table defined by **<table name>** for each row affected by the function.

## Delete Triggers

To delete a trigger, use the `DROP TRIGGER` statement:

```
DROP TRIGGER <trigger name>;
```




## Create Example Database

Create a database for the trigger example codes with the following structure:

1. Create a table called *person* with *name* and *age* for columns.

```
CREATE TABLE person (name varchar(45), age int);
```

Insert sample data into the table:

```
INSERT INTO person VALUES ('Matthew', 25), ('Mark', 20);
```

Select the table to see the result:

```
SELECT * FROM person;
```

```
mysql> CREATE TABLE person (name varchar(45), age int);
Query OK, 0 rows affected (0.14 sec)

mysql> INSERT INTO person VALUES ('Matthew', 25), ('Mark', 20);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM person;
+---------+------+
| name    | age  |
+---------+------+
| Matthew |   25 |
| Mark    |   20 |
+---------+------+
2 rows in set (0.00 sec)
```

2. Create a table called *average_age* with a column called *average*:

```
CREATE TABLE average_age (average double);
```

Insert the average age value into the table:

```
INSERT INTO average_age SELECT AVG(age) FROM person;
```

Select the table to see the result:

```
SELECT * FROM average_age;
```

```
mysql> CREATE TABLE average_age (average double);
Query OK, 0 rows affected (0.20 sec)

mysql> INSERT INTO average_age SELECT AVG(age) FROM person;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM average_age;
+---------+
| average |
+---------+
|    22.5 |
+---------+
1 row in set (0.00 sec)
```

# EXPERIMENT 10
# Creating Procedures and Functions

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

## Stored Procedure Syntax

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

## Execute a Stored Procedure

```
EXEC procedure_name;
```

The procedure syntax has the following parameters:

**MySQL procedure parameter has one of three modes:**

**IN parameter**

It is the default mode. It takes a parameter as input, such as an attribute. When we define it, the calling program has to pass an argument to the stored procedure. This parameter's value is always protected.

**OUT parameters**

It is used to pass a parameter as output. Its value can be changed inside the stored procedure, and the changed (new) value is passed back to the calling program. It is noted that a procedure cannot access the OUT parameter's initial value when it starts.

**NOUT parameters**

It is a combination of IN and OUT parameters. It means the calling program can pass the argument,

and the procedure can modify the INOUT parameter, and then passes the new value back to the calling

program.

## Call a stored procedure:

We can use the **CALL statement** to call a stored procedure. This statement returns the values to its caller through its parameters (IN, OUT, or INOUT).

The following syntax is used to call the stored procedure in MySQL:

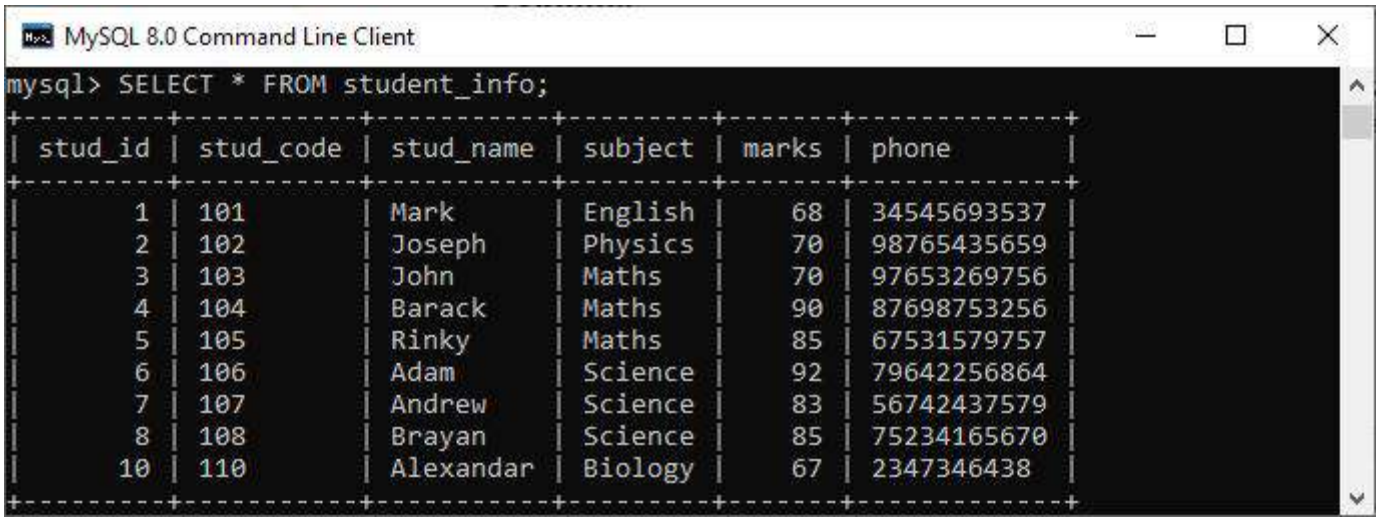
```
CALL procedure_name ( parameter(s))
```

## Example

Let us understand how to create a procedure in MySQL through example. First, we need to select a database that will store the newly created procedure. We can select the database using the below statement:

```
mysql> USE database_name;
```

Suppose this database has a table named **student_info** that contains the following data:



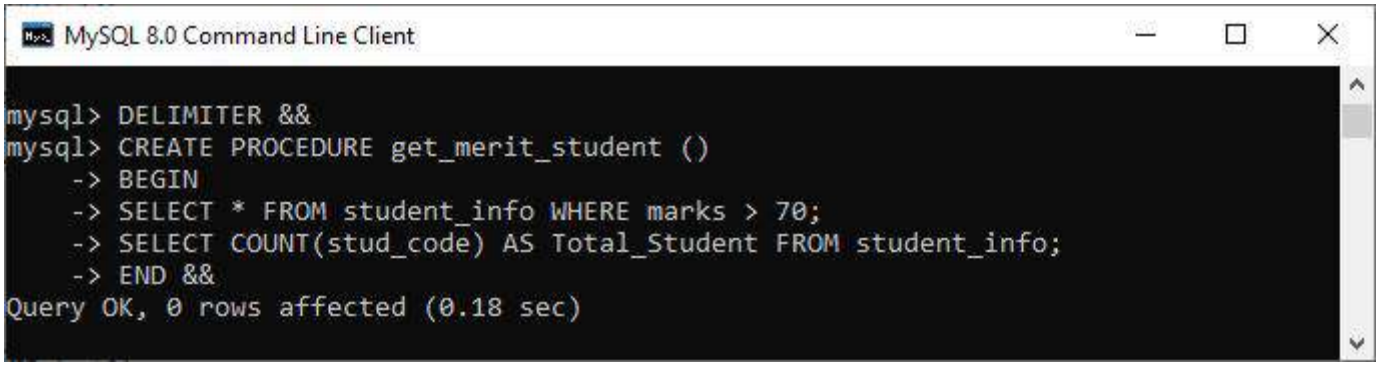

## Procedure without Parameter

Suppose we want **to display all records of this table whose marks are greater than 70** and count all the table rows. The following code creates a procedure named **get_merit_students**:

```
DELIMITER &&
CREATE PROCEDURE get_merit_student ()
BEGIN
```

```
    SELECT * FROM student_info WHERE marks > 70;
    SELECT COUNT(stud_code) AS Total_Student FROM student_info;
END &&
DELIMITER ;
```

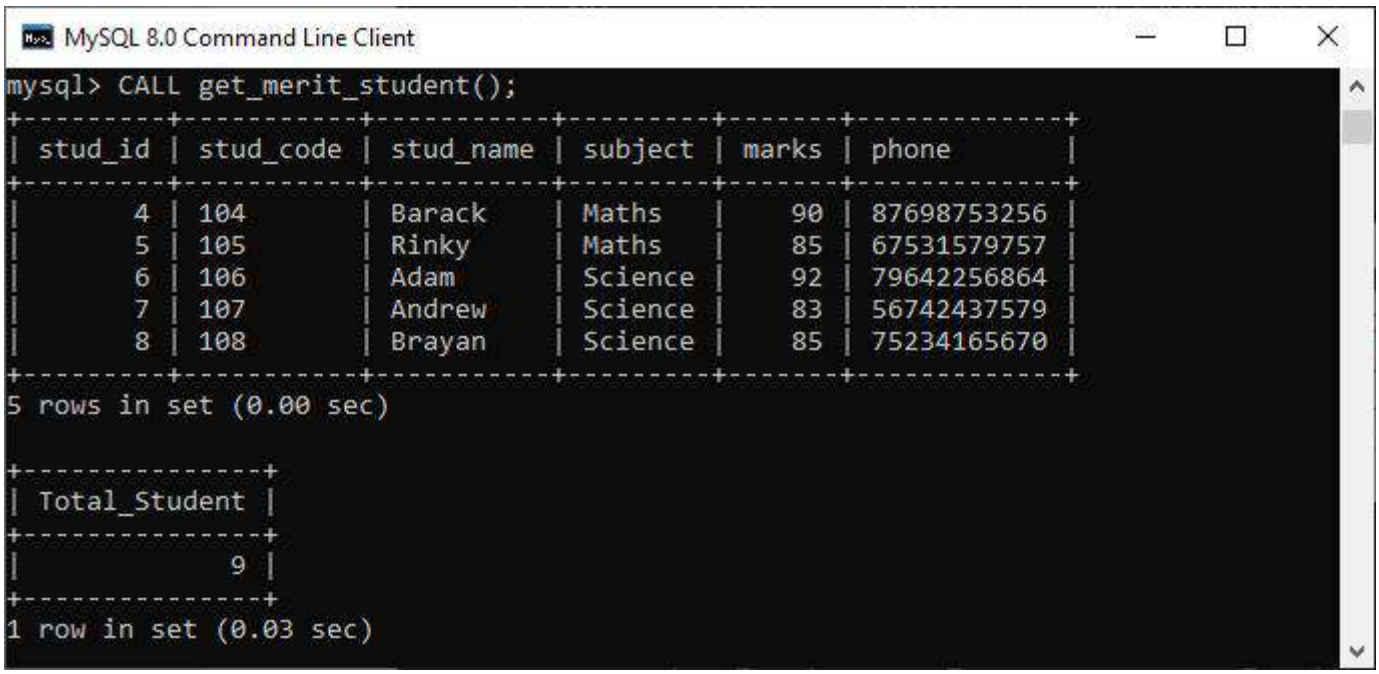
If this code executed successfully, we would get the below output:




Let us call the procedure to verify the output:

```
mysql> CALL get_merit_student();
```

It will give the output as follows:




## Procedures with IN Parameter

In this procedure, we have used the IN parameter as **'var1**' of integer type to accept a number from users. Its body part fetches the records from the table using a **SELECT** statement and returns only those rows that will be supplied by the user. It also returns the total number of rows of the specified table. See the procedure code:

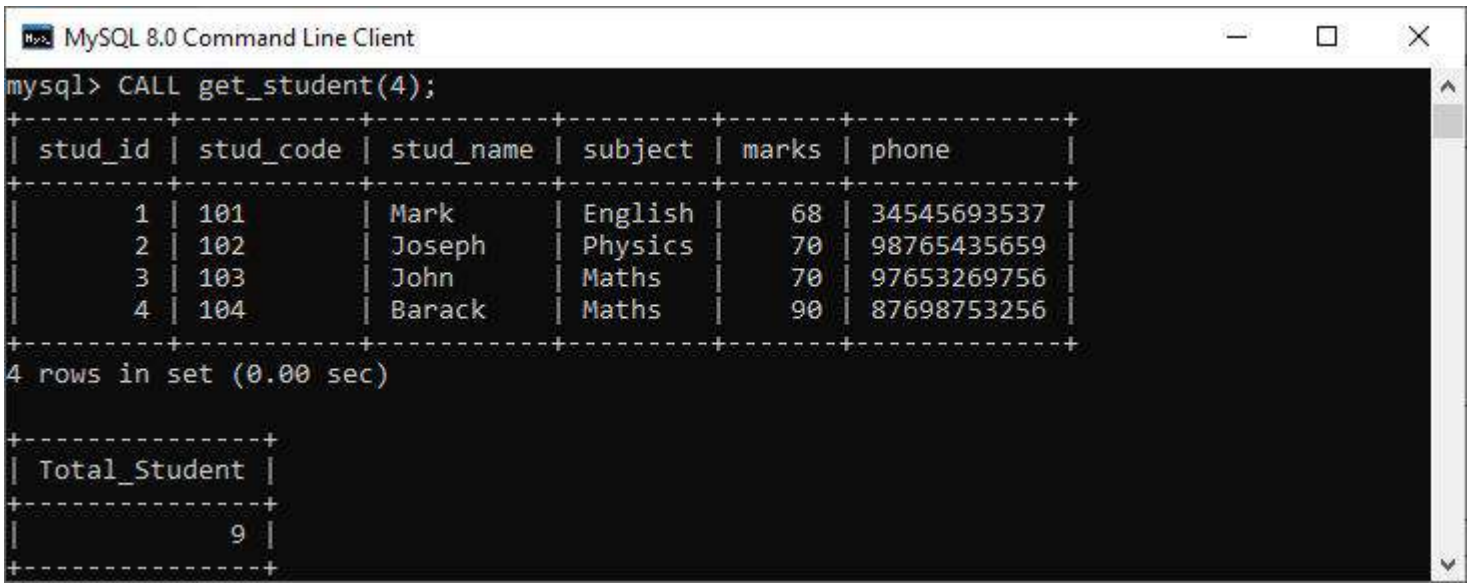
---

```
DELIMITER &&
CREATE PROCEDURE get_student (IN var1 INT)
BEGIN
   SELECT * FROM student_info LIMIT var1;
   SELECT COUNT(stud_code) AS Total_Student FROM student_info;
END &&
DELIMITER ;
```

After successful execution, we can call the procedure as follows:

```
mysql> CALL get_student(4);
```

We will get the below output:




## Procedures with OUT Parameter

In this procedure, we have used the OUT parameter as the **'highestmark'** of integer type. Its body part fe

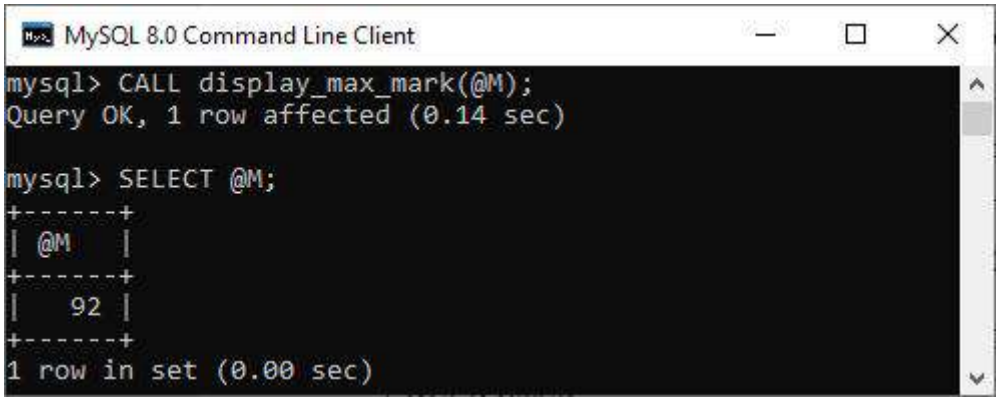
maximum marks from the table using a **MAX() function**. See the procedure code:

```
DELIMITER &&
CREATE PROCEDURE display_max_mark (OUT highestmark INT)
BEGIN
   SELECT MAX(marks) INTO highestmark FROM student_info;
END &&
DELIMITER ;
```

This procedure's parameter will get the highest marks from the **student_info** table. When we call the procedure, the OUT parameter tells the database systems that its value goes out from the procedures. Now, we will pass its value to a session variable **@M** in the CALL statement as follows:

```
mysql> CALL display_max_mark(@M);
mysql> SELECT @M;
```

Here is the output:




## Procedures with INOUT Parameter

In this procedure, we have used the INOUT parameter as **'var1'** of integer type. Its body part first fetches the marks from the table with the specified **id** and then stores it into the same variable var1. The var1 first acts as the IN parameter and then OUT parameter.

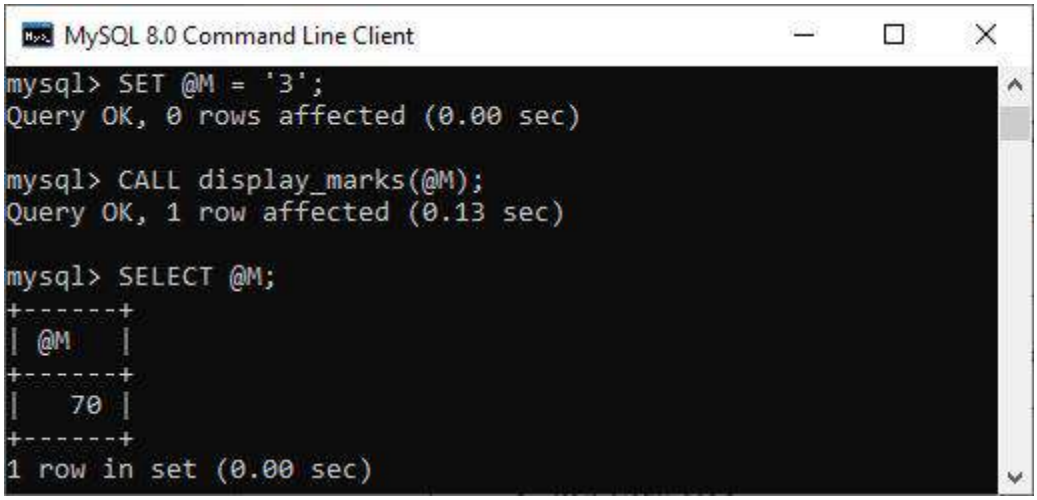
Therefore, we can call it the INOUT parameter mode. See the procedure code:

```
DELIMITER &&
CREATE PROCEDURE display_marks (INOUT var1 INT)
BEGIN
   SELECT marks INTO var1 FROM student_info WHERE stud_id = var1;
END &&
DELIMITER ;
```

After successful execution, we can call the procedure as follows:

```
mysql> SET @M = '3';
mysql> CALL display_marks(@M);
mysql> SELECT @M;
```

We will get the below output:




## Show or list stored procedures in MySQL

When we have several procedures in the MySQL server, it is very important to list all procedures. It is because sometimes the procedure names are the same in many databases.

In that case, this query is very useful. We can list all procedure stored on the current MySQL server as follows:

```
PROCEDURE STATUS [LIKE 'pattern' | WHERE search_condition]
```

**This statement displays all stored procedure names, including their characteristics**.

If we want to display procedures in a particular database, we need to use the **WHERE** clause.

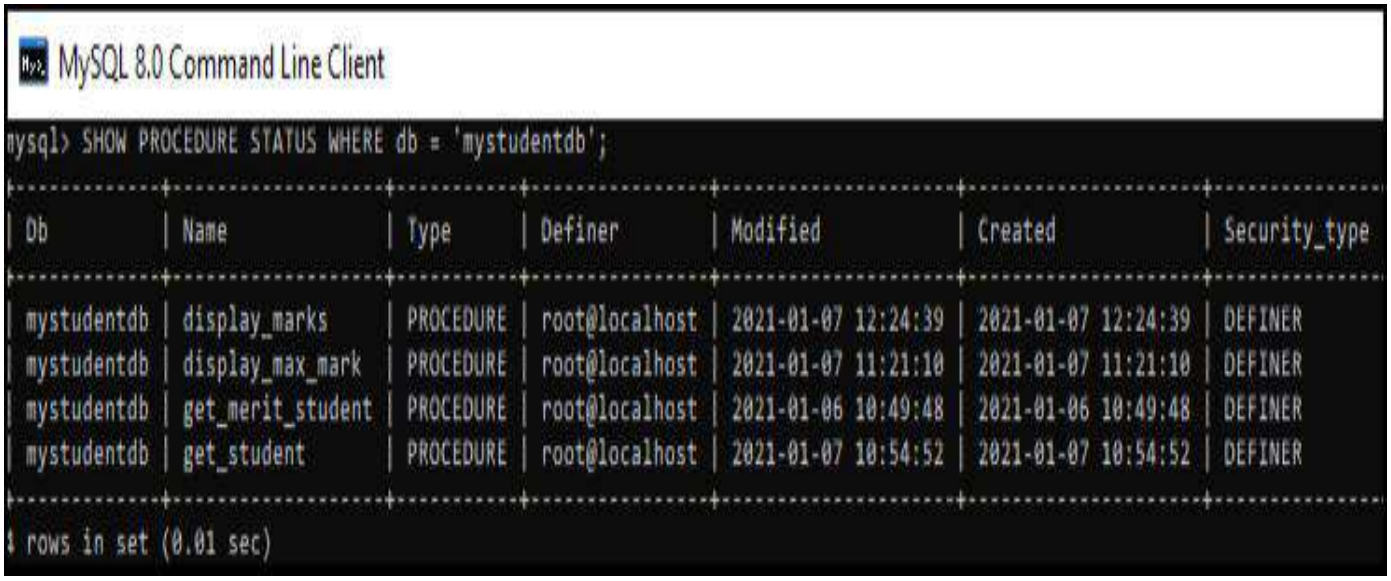
In case we want to list stored procedures with a specific word, we need to use the **LIKE** clause.

We can list all stored procedure in the MySQL **mystudentsb** database using the below statement:

```
mysql> SHOW PROCEDURE STATUS WHERE db = 'mystudentdb';
```

It will give the below output where we can see that the mystudentdb database contains

**four stored procedures**:




## Delete/drop stored procedures in MySQL

MySQL also allows a command to drop the procedure. When the procedure is dropped,

it is removed from the database server also. The following statement is used to drop a

stored procedure in MySQL:

```
DROP PROCEDURE [ IF EXISTS ] procedure_name;
```

Suppose we want to remove the procedure named **display_marks** from the mystudentdb database.

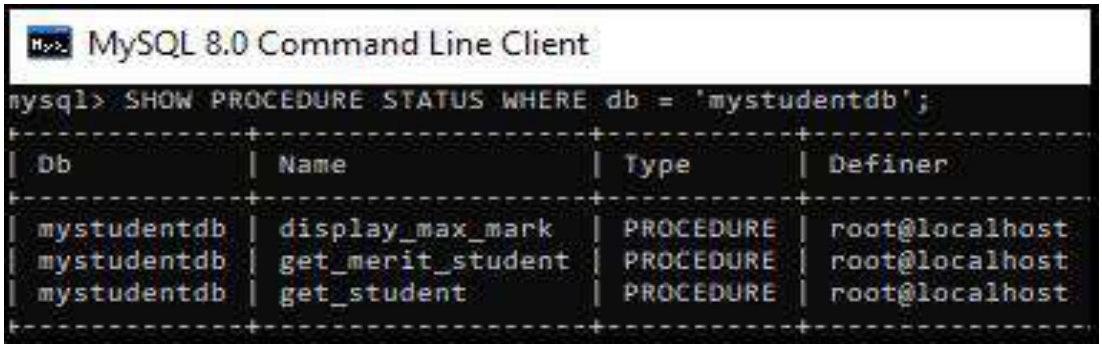
We can do this by first selecting the database and then use the syntax as follows to remove

the procedure:

mysql> **DROP PROCEDURE** display_marks;

We can verify it by listing the procedure in the specified database using the

**SHOW PROCEDURE STATUS** command. See the below output:




## Alter the procedure in MySQL

MySQL does not allow any command to alter the procedure in MySQL.

However,it provides a command that is used to change the characteristics of a stored procedure.

This command may alter more than one change in the procedure but does not

modify the stored procedure's parameters or body. If we want to make such changes,

we must **drop and re-create the procedure** using the DROP PROCEDURE and

CREATE PROCEDURE statement.

**The below statement is used to change the characteristics of a procedure but not the**

**actual procedure**:

**ALTER PROCEDURE** procedure_name [characteristics ...]

```
characteristics: {
    COMMENT 'string'
  | LANGUAGE SQL
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}
```

Suppose we want to **add a comment** to the existing procedure. In such a case, we can use

the **ALTER** statement as follows to accomplish this task:

```
mysql> ALTER PROCEDURE get_merit_student
COMMENT 'It displays all records';
```

After executing this statement, we can verify it by using the below statement:

```
mysql> SHOW CREATE PROCEDURE get_merit_student \G;
```

It will display the below output where we can see that the **comment** is added successfully.

