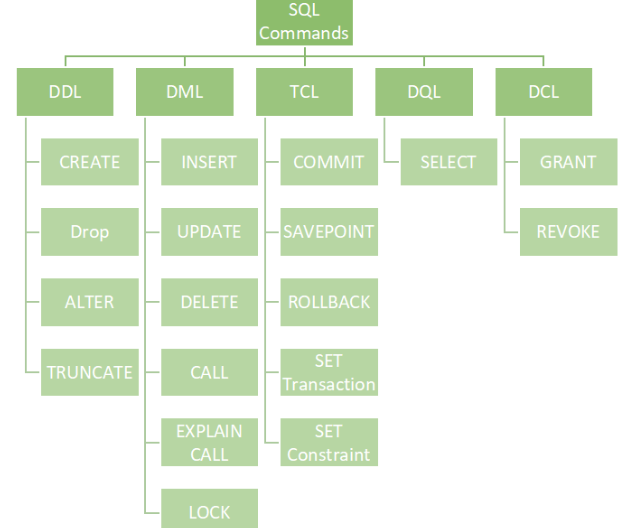
|  |
| --- |
| mysql -u root -p  Enter password: root  create database MCA;  use MCA;  CREATE TABLE student ( RollNo INT PRIMARY KEY, Name VARCHAR(255), Course VARCHAR(50), Year INT );  desc student; |

****

**Data Definition Language (DDL)**

DDL is used for specifying the database schema. It is used for creating tables, schema, indexes, constraints etc. in database. Let’s see the operations that we can perform on database using DDL:

* To create the database instance – [**CREATE**](https://beginnersbook.com/2014/05/sql-create-database-statement/)

CREATE DATABASE databaseName;

You can verify the successful creation of database using show databases statement.

SHOW DATABASES;

* To alter the structure of database – **ALTER**

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

To add a column in a table, use the following syntax:

ALTER TABLE *table\_name* ADD *column\_name datatype*;

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

ALTER TABLE *table\_name* DROP COLUMN *column\_name*;

To rename a column in a table, use the following syntax:

ALTER TABLE *table\_name* RENAME COLUMN *old\_name* to *new\_name*;

OR

Alter Table table\_name change old\_name new\_name data\_type;

* To drop database instances – [**DROP**](https://beginnersbook.com/2014/05/sql-drop-database-statement/)

The DROP DATABASE statement is used to drop an existing SQL database.

DROP DATABASE databasename;

The DROP TABLE statement is used to drop an existing table in a database.

DROP TABLE table\_name;

* To delete tables in a database instance – **TRUNCATE**

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

TRUNCATE TABLE table\_name;

* To rename database instances – **RENAME**

The **SQL RANME DATABASE...TO** statement is used to rename an existing user-created database.

RENAME DATABASE OldDatabaseName TO NewDatabaseName;

* To Comment – **Comment**

Single line comments start with –

--Select all:  
SELECT \* FROM Customers;

All of these commands either defines or update the database schema that’s why they come under Data Definition language.

**Data Manipulation Language (DML)**

DML is used for accessing and manipulating data in a database. The following operations on database comes under DML:

* To read records from table(s) – [**SELECT**](https://beginnersbook.com/2014/05/sql-select-query/)

SELECT column1, column2, ...FROM table\_name WHERE condition;

* To insert record(s) into the table(s) – **INSERT**

INSERT INTO table\_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);

* Update the data in table(s) – [**UPDATE**](https://beginnersbook.com/2014/05/update-query-in-sql/)

UPDATE table\_name SET column1 = value1, column2 = value2, ...WHERE condition;

* Delete all the records from the table – [**DELETE**](https://beginnersbook.com/2014/05/delete-query-in-sql/)

DELETE FROM table\_name WHERE condition;

**Data Control language (DCL**) 🡪 **(grant,revoke)**

DCL is used for granting and revoking user access on a database –

* To grant access to user – GRANT

First, we need to create a new user. The syntax is:

CREATE USER name IDENTIFIED BY 'password';

Eg: create user ‘abin’ identified by ‘ncs007’;

Next, execute the SHOW GRANT statement to check the privileges assigned to “abin” using the following query:

SHOW GRANTS FOR abin;

GRANT privilege\_name(s) ON object TO user\_name;

Eg: grant all on employee to abin;

* To revoke access from user – REVOKE

REVOKE privilege\_name ON object\_name FROM user\_name;

Eg:revoke all on employee from abin;

**Transaction Control Language (TCL)🡪** **(commit,rollback,savepoint).**

The changes in the database that we made using DML commands are either performed or rollbacked using TCL.

* COMMIT

The COMMIT command is used to save changes made during a transaction to the database permanently:

Eg: -- SQL statements

COMMIT;

* ROLLBACK

The ROLLBACK command is used to undo changes made during a transaction:

Eg: -- SQL statements

ROLLBACK;

* SAVEPOINT

The SAVEPOINT command allows you to set a point within a transaction to which you can later roll back:

Eg: -- SQL statements

SAVEPOINT my\_savepoint;

-- More SQL statements

ROLLBACK TO my\_savepoint;

**Practical Questions**

**1. Execute DDL statements**

* + Create a table Student with fields (RollNo,Name,Age,Course,Year).
  + Alter table.
  + Drop table.
  + Truncate table.

Write necessary query statements.

Ans:

**Create Table:**

CREATE TABLE Student ( RollNo INT PRIMARY KEY, Name VARCHAR(255), Age INT,Course VARCHAR(50), Year INT );

This creates a table named "Student" with columns RollNo, Name, Course, and Year.

**Alter Table:** For example, let's say you want to add a new column "Marks" to the existing table:

ALTER TABLE Student ADD Marks INT;

This adds a new column "Marks" of type INT to the "Student" table.

**Drop Table:** To delete (drop) the entire table:

DROP TABLE Student;

This removes the entire "Student" table and all its data.

**Truncate Table:** To remove all rows from the table but keep the table structure:

TRUNCATE TABLE Student;

This deletes all rows from the "Student" table, leaving an empty table with the same structure.

2. **Execute DML statements(Select,Insert,Update,Delete)**

1. Create table Employee (Emp\_Id,Emp\_Name,Dept\_Id,Salary) And Also create another table Department (Dept\_Id,Dept\_Name,Dept\_Head)
2. Insert minimum of 4 rows.
3. Set Primary Key and Foreign Key constraints.
4. Display the records.
5. Update a record.
6. Delete a record.

Ans:

1. SQL> create table employee(Emp\_id int,Emp\_name varchar(20),Dept\_Id int,Salary int);

SQL> insert into employee values(101,'Ben',1,1000);

SQL> insert into employee values(102,'Biby',2,1500);

SQL> insert into employee values(103,'Benoi',3,2500);

SQL> insert into employee values(104,'Joel',4,3500);

1. SQL> create table department(Dept\_id int,Dept\_name varchar(20),Dept\_Head varchar(20));

SQL> insert into department values (1,'Accounting','Alan');

SQL> insert into department values (2,'Production','Arun');

SQL> insert into department values (3,'HR','Nandu');

SQL> insert into department values (4,'Research','Atul');

1. SQL> alter table employee add primary key(Emp\_id);

SQL> alter table department add primary key(Dept\_id);

SQL> alter table employee add foreign key(Dept\_Id) references department(Dept\_id);

1. SQL> select \* from employee;

SQL> select \* from department;

1. SQL> update department set Dept\_Head='Fasil' where Dept\_id=4;
2. SQL> delete from employee where Emp\_id=105;

**3. Create a table and execute DCL statements (grant,revoke).**

Ans:

SQL> CREATE TABLE EMPLOYEE (

Emp\_Id INTEGER PRIMARY KEY,

Emp\_Name TEXT NOT NULL,

Emp\_Add TEXT NOT NULL,

Emp\_Phone TEXT NOT NULL,

Dept\_Id TEXT NOT NULL,

Dept\_Name TEXT NOT NULL,

Salary TEXT NOT NULL);

INSERT INTO EMPLOYEE VALUES (1, 'Ramesh', 'GNoida','9855498465', '3445', 'Sales','25000');

INSERT INTO EMPLOYEE VALUES (2, 'Suresh', 'GNoida','98565498465', '0072', 'Sales','75000');

INSERT INTO EMPLOYEE VALUES (3, 'Rajesh', 'GNoida','9855497865', '2324', 'Sales','28000');

INSERT INTO EMPLOYEE VALUES (4, 'Shyamu', 'BSB','9853698465', '8883', 'Sales','35000');

INSERT INTO EMPLOYEE VALUES (5, 'Ramu', 'BSB','9855498235', '74568', 'Sales','96000');

SQL> select \* from EMPLOYEE;

SQL> create user ‘xyz’ identified by ‘a11’;

SQL>show grants for xyz;

SQL> grant all on employee to xyz;

SQL> select \* from EMPLOYEE;

SQL>revoke all on employee from xyz;

SQL> select \* from EMPLOYEE;

**4. Create a table and execute TCL statements (commit,rollback,savepoint).**

Ans:

SQL> create table student(rollno int,name varchar(20),course varchar(20));

SQL> insert into student values(1,'Ammu','MCA');

SQL> insert into student values(2,'Amal','MCA');

SQL> start transaction;

SQL> savepoint a;

SQL> select \* from student;

SQL> insert into student values(3,'Anju','MCA');

SQL> insert into student values(4,'Anet','MCA');

SQL> select \* from student;

SQL> rollback to a;

SQL> select \* from student;

SQL> commit;

SQL> insert into student values(5,'Arun','MCA');

SQL> select \* from student;

SQL> rollback;

SQL> select \* from student;

**5. Accessing database (SELECT, Filtering using WHERE, HAVING, GROUP BY, ORDER BY Clauses)**

The HAVING clause in SQL is similar to the WHERE clause that is used to filter the data but in a different way.

HAVING clause is used to filter the result obtained by the GROUP BY clause

It is used with the aggregation function

It can include one or more conditions

The order of execution of the HAVING clause is after the GROUP BY clause and before the ORDER BY clause.

It can only be used with the SQL SELECT clause

**Syntax**

**SELECT column\_names**

**FROM table\_name**

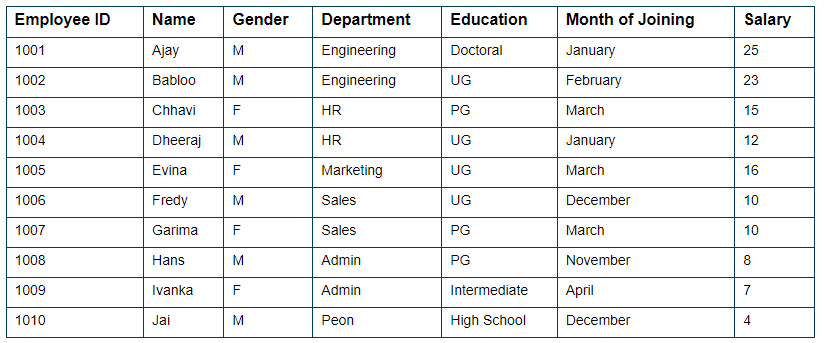
**WHERE conditions**

**GROUP BY column\_name**

**HAVING conditions**

**ORDER BY column\_name;**

* **Consider the database (Employee) that contains the record Employee ID, Name, Department, Education, and their salary in Lacs.**

****

**1) (i): Calculate the sum of salaries of each department.**

**(ii): Find the departments in which the SUM of the salaries is greater than or equal to 20 lacs**

**(iii)** **Display distinct department of Employee.**

**(iv) Display total number of Salary of Employees.**

**(v) Display details of Employees from Employee tables in which Department of the employee is Engineering and Education is Doctorial.**

**(vi) Display details of Employees from Employee tables in which Department of the employee is Engineering or Education is Doctorial.**

**(vii) Rename the columns "Name" and "Education" to "First\_Name" and "Qualification", respectively.**

**(viii)List records of Employees whose names start with “A”.**

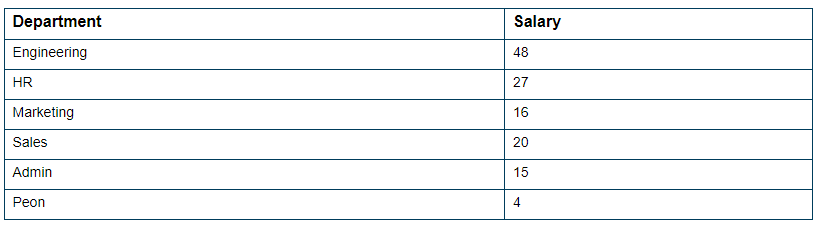
**2) Find the department in which SUM salary is greater than or equal to 20 lacs, but the education of employees is not UG.**

**3) Find the departments in which the SUM of the salaries is greater than or equal to 15 lacs and arrange the Salary in descending order.**

1. SELECT Department, SUM(Salary)

FROM Employee

GROUP BY Department;



1. SELECT Department, SUM(Salary)

FROM Employee

GROUP BY Department

HAVING SUM(Salary) >= 20;



1. SELECT DISTINCT Department FROM Employee;



1. SELECT COUNT(Salary) FROM Employee;

10

1. SELECT \* FROM Employee where Department='Engineering' and Education='Doctorial';
2. SELECT \* FROM Employee where Department='Engineering' or Education='Doctorial';
3. SELECT Name as First\_Name,Education as Qualification from Employee;
4. SELECT \* from Employee where Name like 'A%';

2) SELECT Department, SUM(Salary)

FROM Employee

WHERE Education <> ‘UG’

GROUP BY Department

HAVING SUM(Salary) >= 20;



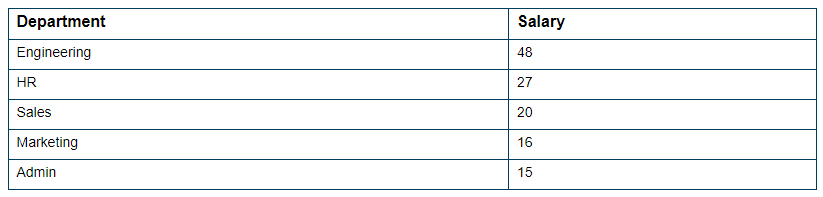
3) SELECT Department, SUM(Salary)

FROM Employee

GROUP BY Department

HAVING SUM(Salary) >= 15

ORDER BY SUM(Salary) DESC;



**Creating Views**

* Views in SQL are considered as a virtual table. A view also contains rows and columns.
* To create the view, we can select the fields from one or more tables present in the database.

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

The basic **CREATE VIEW** syntax is as follows −

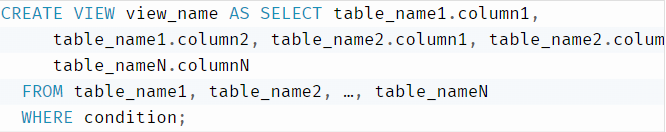
CREATE VIEW view\_name AS

SELECT column1, column2.....

FROM table\_name

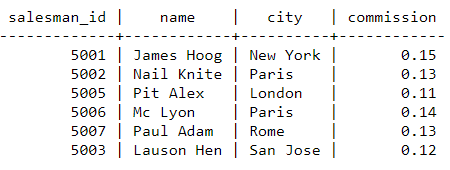
WHERE [condition];

**Creating View from multiple tables:**



**6.** **(i)** **From the following table, create a view for those salespeople who belong to the city of New York.**

**Table: salesman**

****

1. **From the following table, create a view that counts the number of customers in each grade**.

**Table: customer**

customer\_id | cust\_name | city | grade | salesman\_id

-------------+----------------+------------+-------+-------------

3002 | Nick Rimando | New York | 100 | 5001

3007 | Brad Davis | New York | 200 | 5001

3005 | Graham Zusi | California | 200 | 5002

3008 | Julian Green | London | 300 | 5002

3004 | Fabian Johnson | Paris | 300 | 5006

3009 | Geoff Cameron | Berlin | 100 | 5003

3003 | Jozy Altidor | Moscow | 200 | 5007

3001 | Brad Guzan | London | 100 | 5005

1. **From the following table, create a view to count the number of unique customers, compute the average and the total purchase amount of customer orders by each date.**

**Table: orders**

ord\_no purch\_amt ord\_date customer\_id salesman\_id

---------- ---------- ---------- ----------- -----------

70001 150.5 2012-10-05 3005 5002

70009 270.65 2012-09-10 3001 5005

70002 65.26 2012-10-05 3002 5001

70004 110.5 2012-08-17 3009 5003

70007 948.5 2012-09-10 3005 5002

70005 2400.6 2012-07-27 3007 5001

70008 5760 2012-09-10 3002 5001

70010 1983.43 2012-10-10 3004 5006

70003 2480.4 2012-10-10 3009 5003

70012 250.45 2012-06-27 3008 5002

70011 75.29 2012-08-17 3003 5007

70013 3045.6 2012-04-25 3002 5001

1. **From the order table, create a view to find the salespersons who issued orders on either August 17th, 2012 or October 10th, 2012. Return salesperson ID, order number and customer ID.**
2. **From the salesman and order tables, create a view to find the salesperson who handles a customer who makes the highest order of the day. Return order date, salesperson ID, name.**

**6 (i)**. CREATE VIEW newyorkstaff AS SELECT \* FROM salesman

WHERE city = 'New York';

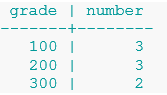
SELECT \* FROM newyorkstaff;

****

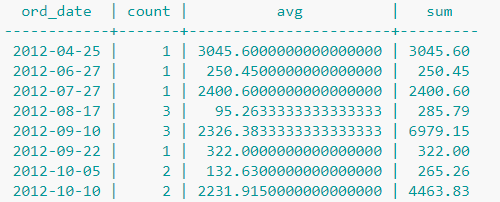
**(ii)** CREATE VIEW gradecount (grade, number) AS SELECT grade, COUNT(\*)

FROM customer GROUP BY grade;

SELECT \* FROM gradecount;

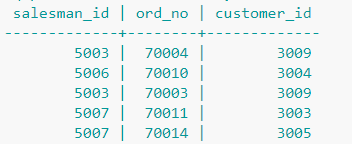


1. CREATE VIEW totalforday AS SELECT ord\_date, COUNT (DISTINCT customer\_id), AVG(purch\_amt), SUM(purch\_amt) FROM orders GROUP BY ord\_date;



1. CREATE VIEW sorder AS SELECT salesman\_id, ord\_no, customer\_id

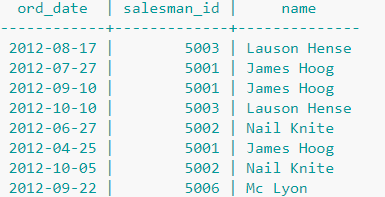
FROM orders WHERE ord\_date IN ('2012-08-17', '2012-10-10');



1. CREATE VIEW elitsalesman AS SELECT b.ord\_date, a.salesman\_id, a.name

FROM salesman a, orders b WHERE a.salesman\_id = b.salesman\_id

AND b.purch\_amt=(SELECT MAX (purch\_amt) FROM orders c WHERE c.ord\_date = b.ord\_date);



# **Subquery**

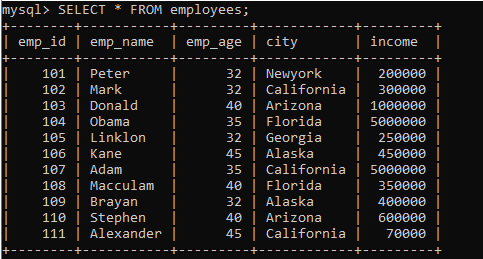
A subquery in MySQL is a query, which is nested into another SQL query and embedded with SELECT, INSERT, UPDATE or DELETE statement along with the various operators. We can also nest the subquery with another subquery. A subquery is known as the **inner query**, and the query that contains subquery is known as the **outer query**.

SELECT column\_list (s) FROM table\_name

WHERE column\_name OPERATOR

(SELECT column\_list (s) FROM table\_name [WHERE])

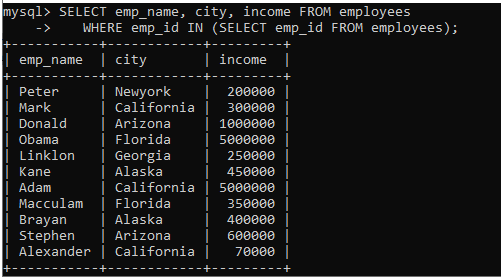
**7) Table: employees**

****

1. **Find employee detail whose id matches in a subquery**:

**SELECT** emp\_name, city, income **FROM** employees

**WHERE** emp\_id IN (**SELECT** emp\_id **FROM** employees);

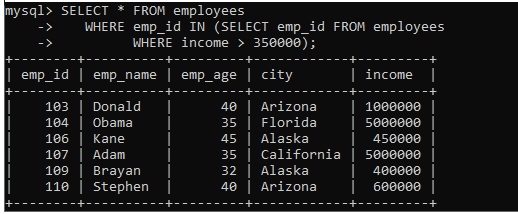


1. **Find employee detail whose income is more than 350000** **with the help of subquery:**

**SELECT** \* **FROM** employees

**WHERE** emp\_id IN (**SELECT** emp\_id **FROM** employees

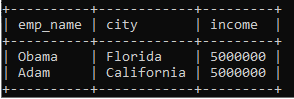
**WHERE** income > 350000);



1. **Find employee details with maximum income using a subquery.**

**SELECT** emp\_name, city, income **FROM** employees

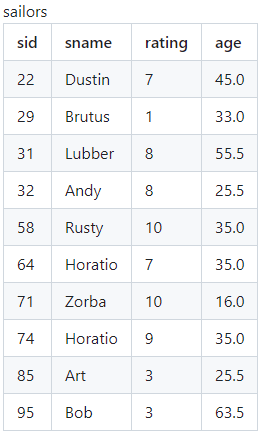
**WHERE** income = (**SELECT** **MAX**(income) **FROM** employees);

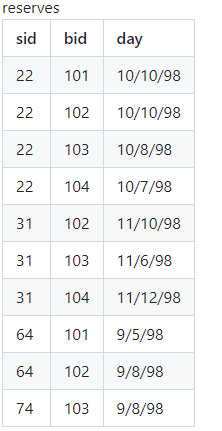
****

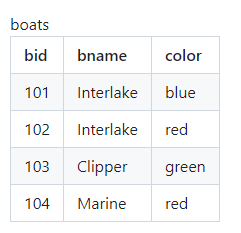
**8)** Sailors(sid: integer, sname: string, rating: integer, age: real);

Boats(bid: integer, bname: string, color: string);

Reserves(sid: integer, bid: integer, day: date).

****

****

****

1. **Count the number of distinct boat colors:**

SELECT COUNT(DISTINCT color) FROM boats;

1. **Find all information of sailors who have reserved boat number 101.**

SELECT sailors.\* FROM Sailors, Reserves WHERE Sailors.sid = Reserves.sid AND Reserves.bid = 101;

**OR**

select \* from sailors where sid in (select sid from reserves where bid=101);

1. **Find names of sailors who have reserved at least one boat.**

SELECT sname FROM sailors S, Reserves R WHERE S.sid = R.sid;

1. **Find names of sailors who have reserved a red boat and list in the order of their age.**

select sname,age from sailors where sid in (select sid from reserves,boats where reserves.bid=boats.bid and color='red') order by age;

1. **Display boat names and the names of sailors who have sailed on them:**

SELECT b.bname, s.sname

FROM boats b

INNER JOIN reserves r ON b.bid = r.bid INNER JOIN sailors s ON s.sid=r.sid;

1. **Find the ids and names of sailors who have reserved two different boats on the same day.**

SELECT DISTINCT S.sid, S.sname

FROM sailors S, reserves R1, reserves R2

WHERE S.sid = R1.sid AND R1.day = R2.day

AND R1.bid <> R2.bid;

SELECT!DISTINCT!S.sid,!S.name!

FROM!Sailors!S,!Reserves!R1,!Reserves!R2!

WHERE!S.sid!=!R1.sid!AND!R1.sid!=!R2.sid!

AND!R1.day!=!R2.day!AND!R1.bid!<>!R2.bid

SELECT!DISTINCT!S.sid,!S.name!

FROM!Sailors!S,!Reserves!R1,!Reserves!R2!

WHERE!S.sid!=!R1.sid!AND!R1.sid!=!R2.sid!

AND!R1.day!=!R2.day!AND!R1.bid!<>!R2.bid

SELECT!DISTINCT!S.sid,!S.name!

FROM!Sailors!S,!Reserves!R1,!Reserves!R2!

WHERE!S.sid!=!R1.sid!AND!R1.sid!=!R2.sid!

AND!R1.day!=!R2.day!AND!R1.bid!<>!R2.bid

1. **Find the ids of sailors who have reserved a red boat or a green boat.**

SELECT R.sid

FROM boats B, reserves R

WHERE R.bid = B.bid AND B.color = ‘red’

UNION

SELECT R2.sid

FROM boats B2, reserves R2

WHERE R2.bid = B2.bid AND B2.color = ‘green’;

1. **Find the names of sailors who have reserved all boats.**

SELECT S.sname

FROM Sailors S

WHERE NOT EXISTS (SELECT B.bid

FROM Boats B

WHERE NOT EXISTS(SELECT R.bid

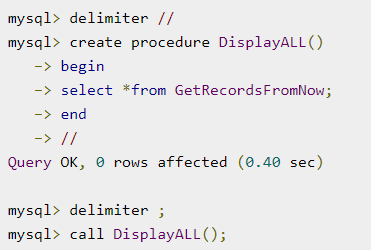
FROM Reserves R

WHERE R.bid = B.bid

AND R.sid = S.sid));

**PL/SQL Programs**

**Procedure**

****

1. Write a PL/SQL procedure to display all fields from a table.

mysql> delimiter //

mysql> create procedure display()

-> begin

-> select \* from student;

-> end

-> //

Query OK, 0 rows affected (0.02 sec)

mysql> delimiter ;

mysql> call display();

+--------+-------+--------+------+

| RollNo | Name | Course | Year |

+--------+-------+--------+------+

| 1 | ammu | bams | 2023 |

| 2 | binil | bca | 2020 |

+--------+-------+--------+------+

2 rows in set (0.00 sec)

Query OK, 0 rows affected (0.02 sec)

|  |
| --- |
| (`) backticks are used to denote identifiers such as table names, column names, stored procedure names, etc. This is particularly useful when the identifier contains special characters or spaces, allowing MySQL to interpret the identifier correctly. |

1. Write a PL/SQL procedure to add two numbers.

mysql> DELIMITER //

mysql> CREATE PROCEDURE `sum`(IN a INT, IN b INT)

-> BEGIN

-> DECLARE c INT;

-> SET c = a + b;

-> SELECT CONCAT('Sum of two numbers = ', c) AS Result;

-> END //

Query OK, 0 rows affected (0.02 sec)

mysql> delimiter ;

mysql> CALL `sum`(5, 10);

+-------------------------+

| Result |

+-------------------------+

| Sum of two numbers = 15 |

+-------------------------+

1 row in set (0.01 sec)

Query OK, 0 rows affected (0.02 sec)

3. Write a PL/SQL procedure to check whether a number is odd or even.

mysql> DELIMITER //

mysql> CREATE PROCEDURE CheckOddOrEven(IN input\_number INT)

-> BEGIN

-> IF MOD(input\_number, 2) = 0 THEN

-> SELECT 'Even' AS result;

-> ELSE

-> SELECT 'Odd' AS result;

-> END IF;

-> END //

Query OK, 0 rows affected (0.02 sec)

mysql> delimiter ;

mysql> call CheckOddOrEven(2);

+--------+

| result |

+--------+

| Even |

+--------+

1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)

1. Write a PL/SQL procedure to find Factorial of a number

mysql> Delimiter //

mysql> CREATE PROCEDURE Factorial( in a int)

-> begin

-> declare f int default 1;

-> while a > 0 do

-> set f = f \* a;

-> set a = a - 1;

-> end while;

-> SELECT CONCAT('Factorial = ', f) AS Result;

-> end //

Query OK, 0 rows affected (0.01 sec)

mysql> delimiter ;

mysql> call Factorial(6);

+-----------------+

| Result |

+-----------------+

| Factorial = 720 |

+-----------------+

1 row in set (0.01 sec)

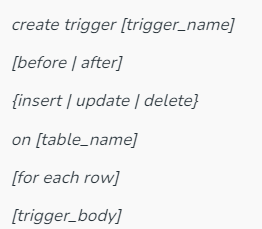
Query OK, 0 rows affected (0.01 sec)

1. Write a PL/SQL procedure to find maximum of three values.
2. Write a PL/SQL procedure to find the sum of digits

**SQL Trigger**

A trigger is a stored procedure in a database that automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when specific table columns are updated in simple words a trigger is a collection of SQL statements with particular names that are stored in system memory.

**Syntax:**



1. Create trigger [trigger\_name]: Creates or replaces an existing trigger with the trigger\_name.
2. [before | after]: This specifies when the trigger will be executed.
3. {insert | update | delete}: This specifies the DML operation.
4. On [table\_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each affected row.
6. [trigger\_body]: This provides the operation to be performed as the trigger is fired
7. Execution of trigger

The database consists of three tables: "students", "courses", and "enrollment\_log". The "students" table stores information about students, including their student ID, name, and email address. The "courses" table contains details about courses offered by the university, such as course ID, title, and instructor. The "enrollment\_log" table records all enrollment transactions, tracking the enrollment of students into courses. Create a trigger named "after\_enrollment" that automatically inserts a log entry into the "enrollment\_log" table whenever a student enrolls in a course. The log entry should include details about the enrollment action (e.g., "Student enrolled in course"), along with the timestamp of the enrollment.

mysql> CREATE TABLE employees (

-> employee\_id INT PRIMARY KEY,

-> first\_name VARCHAR(50),

-> last\_name VARCHAR(50),

-> salary DECIMAL(10, 2)

-> );

Query OK, 0 rows affected (0.11 sec)

mysql> CREATE TABLE audit\_log (

-> log\_id INT PRIMARY KEY AUTO\_INCREMENT,

-> action VARCHAR(255),

-> timestamp TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

-> );

Query OK, 0 rows affected (0.03 sec)

mysql> INSERT INTO employees (employee\_id, first\_name, last\_name, salary)

-> VALUES

-> (1, 'John', 'Doe', 50000.00),

-> (2, 'Jane', 'Smith', 60000.00),

-> (3, 'Michael', 'Johnson', 55000.00);

Query OK, 3 rows affected (0.02 sec)

mysql> INSERT INTO audit\_log (action, timestamp)

-> VALUES

-> ('New employees added', CURRENT\_TIMESTAMP);

Query OK, 1 row affected (0.01 sec)

mysql> select \* from employees;

+---------------+--------------+------------+----------+

| employee\_id | first\_name | last\_name | salary |

+----------------+-------------+------------+----------+

| 1 | John | Doe | 50000.00 |

| 2 | Jane | Smith | 60000.00 |

| 3 | Michael | Johnson | 55000.00 |

+-------------+------------+-----------+----------+

3 rows in set (0.00 sec)

mysql> select \* from audit\_log;;

+--------+----------------------------+---------------------+

| log\_id | action | timestamp |

+--------+----------------------------+---------------------+

| 1 | New employees added | 2024-03-19 15:29:12 |

+--------+----------------------------+------------------------+

1 row in set (0.00 sec)

mysql> DELIMITER //

mysql>

mysql> CREATE TRIGGER after\_employee\_insert

-> AFTER INSERT ON employees

-> FOR EACH ROW

-> BEGIN

-> INSERT INTO audit\_log (action, timestamp)

-> VALUES ('New employee inserted', NOW());

-> END //

Query OK, 0 rows affected (0.02 sec)

mysql> DELIMITER ;

mysql> INSERT INTO employees (employee\_id, first\_name, last\_name, salary)

-> VALUES (4, 'Minnu', 'Joseph', 65000.00);

Query OK, 1 row affected (0.01 sec)

mysql> select \* from employees;

+-------------+------------+-----------+----------+

| employee\_id | first\_name | last\_name | salary |

+-------------+------------+-----------+----------+

| 1 | John | Doe | 50000.00 |

| 2 | Jane | Smith | 60000.00 |

| 3 | Michael | Johnson | 55000.00 |

| 4 | Minnu | Joseph | 65000.00 |

+-------------+------------+-----------+----------+

4 rows in set (0.00 sec)

mysql> select \* from audit\_log;

+--------+-----------------------------+-------------------------+

| log\_id | action | timestamp |

+--------+-----------------------------+-------------------------+

| 1 | New employees added | 2024-03-19 15:29:12 |

| 2 | New employee inserted | 2024-03-19 20:15:58 |

+--------+-----------------------------+--------------------------+

2 rows in set (0.01 sec)

|  |
| --- |
| CREATE TRIGGER after\_employee\_insert  AFTER INSERT ON employees  FOR EACH ROW  BEGIN  INSERT INTO audit\_log (action, timestamp)  VALUES ('New employee inserted', NOW());  END;  In this trigger:   * **after\_employee\_insert** is the name of the trigger. * **AFTER INSERT ON employees** specifies that the trigger should fire after an insertion into the **employees** table. * **FOR EACH ROW** indicates that the trigger should execute once for each row affected by the insert operation. * **BEGIN ... END** encloses the trigger's action. * **INSERT INTO audit\_log (action, timestamp) VALUES ('New employee inserted', NOW());** is the action performed by the trigger, which inserts a new record into the **audit\_log** table with the action description and the current timestamp.   Now, whenever a new employee record is inserted into the **employees** table, a corresponding record will automatically be inserted into the **audit\_log** table with the action description "New employee inserted" and the current timestamp. |

1. **MySQL Trigger : Example BEFORE UPDATE**

mysql> CREATE TABLE Std1 (

-> studentID INT PRIMARY KEY,

-> Name VARCHAR(20),

-> sub1 INT,

-> sub2 INT,

-> sub3 INT,

-> sub4 INT,

-> sub5 INT,

-> total INT,

-> per\_marks INT,

-> grade VARCHAR(20)

-> );

Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO Std1 (studentID, Name, sub1, sub2, sub3, sub4, sub5)

-> VALUES

-> (1, 'Student1', 0, 0, 0, 0, 0),

-> (2, 'Student2', 0, 0, 0, 0, 0);

Query OK, 2 rows affected (0.01 sec)

Records: 2 Duplicates: 0 Warnings: 0

mysql> select \* from std1;

+-----------+----------+------+------+------+------+------+-------+-----------+-----------+

| studentID | Name | sub1 | sub2 | sub3 | sub4 | sub5 | total | per\_marks | grade |

+-----------+----------+------+------+------+------+------+-------+-----------+-----------+

| 1 | Student1 | 0 | 0 | 0 | 0 | 0 | NULL | NULL| NULL |

| 2 | Student2 | 0 | 0 | 0 | 0 | 0 | NULL | NULL | NULL |

+-----------+----------+------+------+------+------+------+-------+-----------+-----------

mysql> DELIMITER //

mysql> CREATE TRIGGER std\_before\_update

-> BEFORE UPDATE ON Std1

-> FOR EACH ROW

-> BEGIN

-> DECLARE total\_marks INT;

-> DECLARE per\_marks INT;

-> DECLARE grade VARCHAR(20);

-> SET total\_marks = NEW.sub1 + NEW.sub2 + NEW.sub3 + NEW.sub4 + NEW.sub5;

-> SET per\_marks = total\_marks / 5;

-> IF per\_marks >= 90 THEN

-> SET grade = 'EXCELLENT';

-> ELSEIF per\_marks >= 75 AND per\_marks < 90 THEN

-> SET grade = 'VERY GOOD';

-> ELSEIF per\_marks >= 60 AND per\_marks < 75 THEN

-> SET grade = 'GOOD';

-> ELSEIF per\_marks >= 40 AND per\_marks < 60 THEN

-> SET grade = 'AVERAGE';

-> ELSE

-> SET grade = 'NOT PROMOTED';

-> END IF;

-> SET NEW.total = total\_marks;

-> SET NEW.per\_marks = per\_marks;

-> SET NEW.grade = grade;

-> END;

-> //

Query OK, 0 rows affected (0.01 sec)

mysql>

mysql> DELIMITER ;

mysql> UPDATE Std1 SET sub1 = 90, sub2 = 75, sub3 = 90, sub4 = 95, sub5 = 85 WHERE studentID = 1;

Query OK, 1 row affected (0.01 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> SELECT \* FROM Std1;

+-----------+----------+------+------+------+------+------+-------+-----------+-----------+

| studentID | Name | sub1 | sub2 | sub3 | sub4 | sub5 | total | per\_marks | grade |

+-----------+----------+------+------+------+------+------+-------+-----------+-----------+

| 1 | Student1 | 90 | 75 | 90 | 95 | 85 | 435 | 87 | VERY GOOD |

| 2 | Student2 | 0 | 0 | 0 | 0 | 0 | NULL | NULL | NULL |

+-----------+----------+------+------+------+------+------+-------+-----------+-----------

### **3 AFTER UPDATE Trigger Example**

mysql> CREATE TABLE students(

-> id int NOT NULL AUTO\_INCREMENT,

-> name varchar(45) NOT NULL,

-> class int NOT NULL,

-> email\_id varchar(65) NOT NULL,

-> PRIMARY KEY (id)

-> );

Query OK, 0 rows affected (0.05 sec)

mysql> INSERT INTO students(name, class, email\_id)

-> VALUES ('Stephen', 6, 'stephen@gmail.com'),

-> ('Bob', 7, 'bob@gmail.com'),

-> ('Steven', 8, 'steven@ gmail.com'),

-> ('Alexandar', 7, 'alexandar@ gmail.com');

Query OK, 4 rows affected (0.01 sec)

Records: 4 Duplicates: 0 Warnings: 0

mysql> CREATE TABLE student\_log (

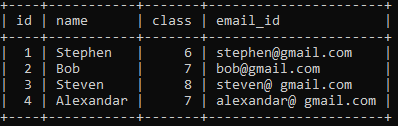
-> user VARCHAR(45) NOT NULL,

-> descriptions VARCHAR(65) NOT NULL

-> );

Query OK, 0 rows affected (0.04 sec)

mysql> select \* from students;



mysql> DELIMITER //

mysql> CREATE TRIGGER after\_update\_stdnts

-> AFTER UPDATE

-> ON students

-> FOR EACH ROW

-> BEGIN

-> INSERT INTO student\_log VALUES (user(),

-> CONCAT('Update Student Record ', OLD.name, ' Previous Class :',

-> OLD.class, ' Present Class ', NEW.class));

-> END //

Query OK, 0 rows affected (0.02 sec)

mysql> delimiter ;

mysql> select \* from student\_log;

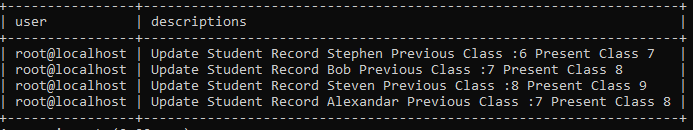
Empty set (0.01 sec)

mysql> update students set class=class+1;

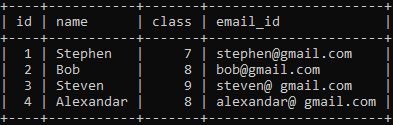
Query OK, 4 rows affected (0.02 sec)

Rows matched: 4 Changed: 4 Warnings: 0

mysql> select \* from student\_log;



mysql> select \* from students;



**Execution of cursor**

The major function of a cursor is to retrieve data, one row at a time, from a result set, unlike the SQL commands which operate on all the rows in the result set at one time. Cursors are used when the user needs to update records in a singleton fashion or in a row by row manner, in a database table.

**Cursor Actions**

* **Declare Cursor:** A cursor is declared by defining the SQL statement that returns a result set.
* **Open:** A Cursor is opened and populated by executing the SQL statement defined by the cursor.
* **Fetch:** When the cursor is opened, rows can be fetched from the cursor one by one or in a block to perform data manipulation.
* **Close:** After data manipulation, close the cursor explicitly.

**4.Write a program in PL/SQL to find average salary using cursor**

DELIMITER //

CREATE PROCEDURE calculate\_av\_salary()

BEGIN

DECLARE cur\_salary INT;

DECLARE total\_salary INTEGER DEFAULT 0;

DECLARE num\_rows INTEGER DEFAULT 0;

DECLARE avg\_salary INTEGER DEFAULT 0;

DECLARE done BOOLEAN DEFAULT FALSE;

DECLARE salary\_cursor CURSOR FOR SELECT salary FROM employees;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN salary\_cursor;

fetch\_loop: LOOP

FETCH salary\_cursor INTO cur\_salary;

IF done THEN

LEAVE fetch\_loop;

END IF;

IF cur\_salary IS NOT NULL THEN

SET total\_salary = total\_salary + cur\_salary;

SET num\_rows = num\_rows + 1;

END IF;

END LOOP fetch\_loop;

CLOSE salary\_cursor;

IF num\_rows > 0 THEN

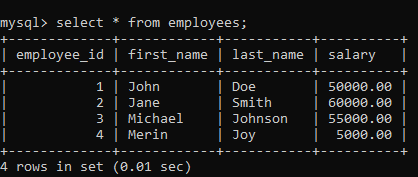
SET avg\_salary = total\_salary / num\_rows;

END IF;

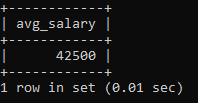
SELECT avg\_salary;

END //

DELIMITER ;



mysql> call calculate\_av\_salary();



**5** **Write a program in PL/SQL to list the name of students using cursor**

delimiter $$

create procedure list\_names(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 student;

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 $$

delimiter ;

**CALLING**

mysql> set @name\_list="";

Query OK, 0 rows affected (0.00 sec)

mysql> call list\_name(@name\_list);

Query OK, 0 rows affected (0.00 sec)

mysql> select @name\_list;

