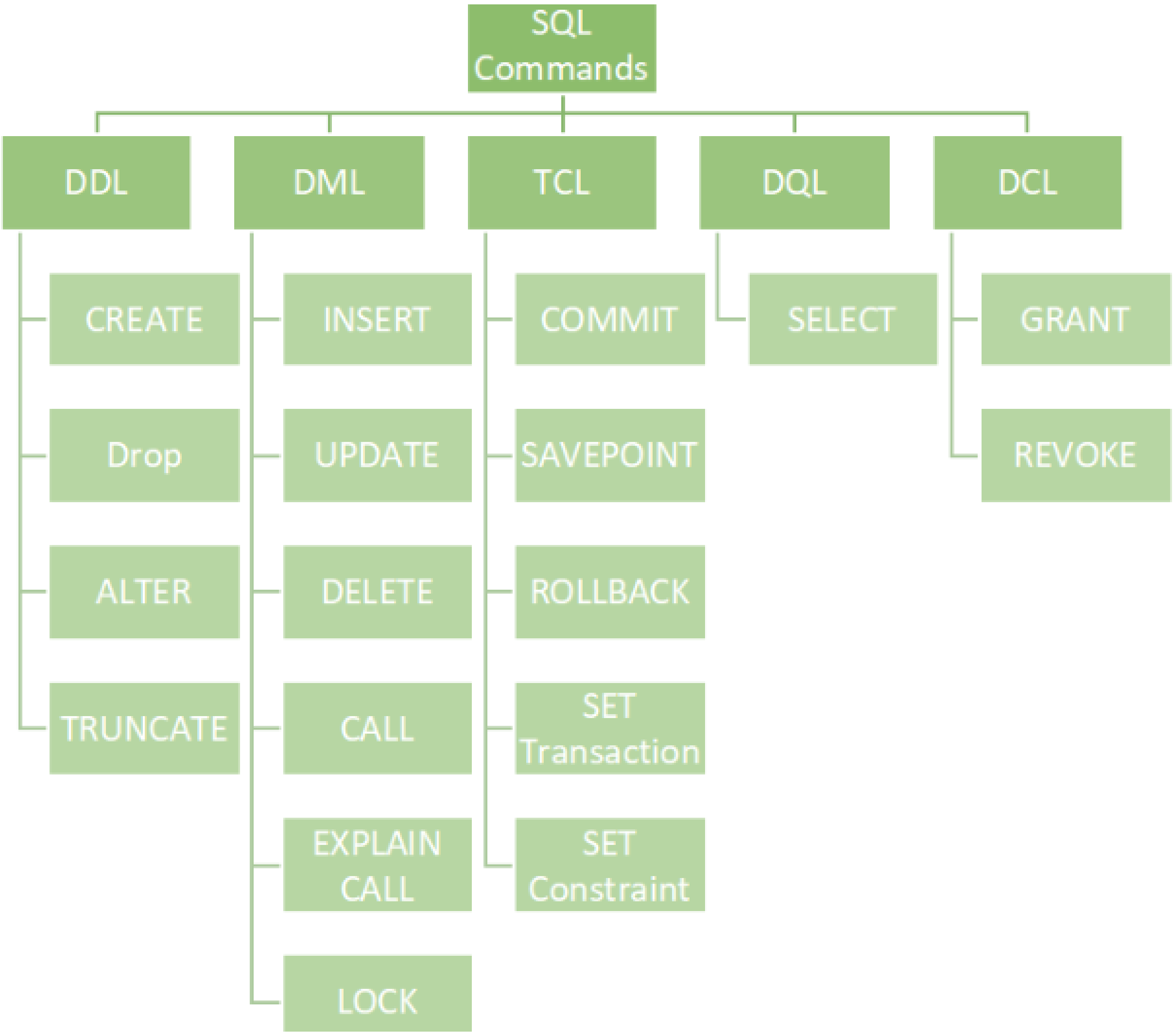
```
    mysql  -u root -p

    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**

  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**

  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**

  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**

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

- Delete all the records from the table – **DELETE**

  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)**

i. Create table Employee (Emp_Id,Emp_Name,Dept_Id,Salary) And Also create another table Department (Dept_Id,Dept_Name,Dept_Head)
ii. Insert minimum of 4 rows.
iii. Set Primary Key and Foreign Key constraints.
iv. Display the records.
v. Update a record.

vi. Delete a record.

Ans:

i)

```
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);
```

ii)

```
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');
```

iii)

```
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);
```

iv)

```
SQL> select * from employee;

SQL> select * from department;
```

v)

```
SQL> update department set Dept_Head='Fasil' where Dept_id=4;
```

vi)

```
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.

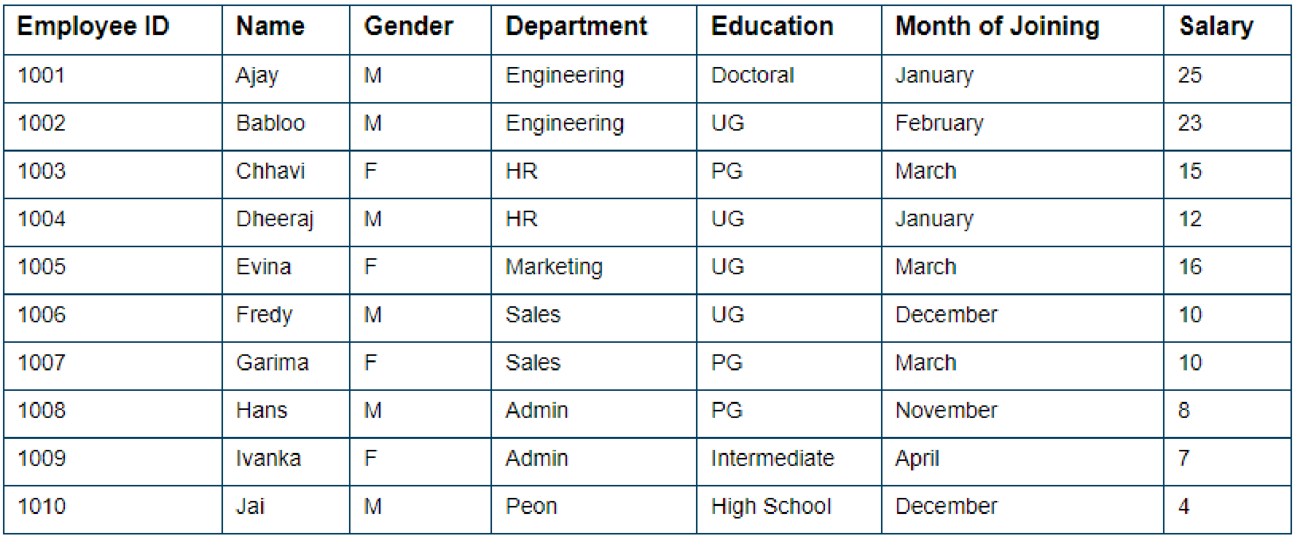
It can only be used with the SQL SELECT clause

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.**



Employee ID	Name	Gender	Department	Education	Month of Joining	Salary
1001	Ajay	M	Engineering	Doctoral	January	25
1002	Babloo	M	Engineering	UG	February	23
1003	Chhavi	F	HR	PG	March	15
1004	Dheeraj	M	HR	UG	January	12
1005	Evina	F	Marketing	UG	March	16
1006	Fredy	M	Sales	UG	December	10
1007	Garima	F	Sales	PG	March	10
1008	Hans	M	Admin	PG	November	8
1009	Ivanka	F	Admin	Intermediate	April	7
1010	Jai	M	Peon	High School	December	4

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.

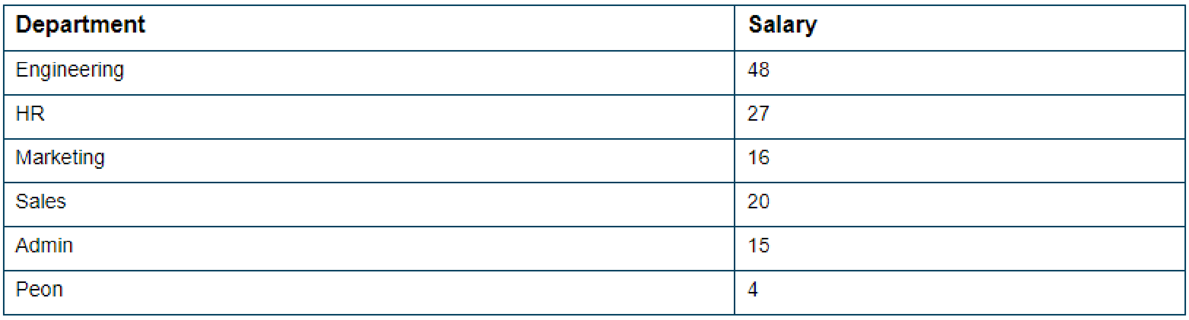
(vii) Rename the columns "Name" and "Education" to "First_Name" and "Qualification", respectively.

(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.

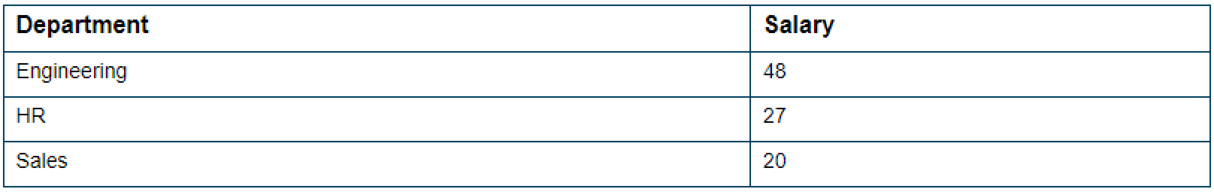
3) Find the departments in which the SUM of the salaries is greater than or equal to 15 lacs and arrange the Salary in descending order.

(i) SELECT Department, SUM(Salary)
FROM Employee
GROUP BY Department;



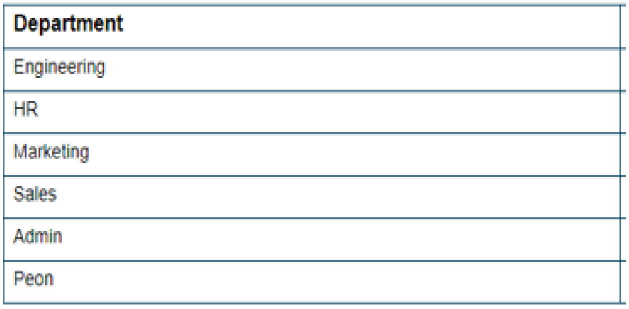
Department	Salary
Engineering	48
HR	27
Marketing	16
Sales	20
Admin	15
Peon	4

(ii) SELECT Department, SUM(Salary)
FROM Employee
GROUP BY Department
HAVING SUM(Salary) >= 20;



Department	Salary
Engineering	48
HR	27
Sales	20

(iii) SELECT DISTINCT Department FROM Employee;



Department
Engineering
HR
Marketing
Sales
Admin
Peon

(iv) SELECT COUNT(Salary) FROM Employee;

10

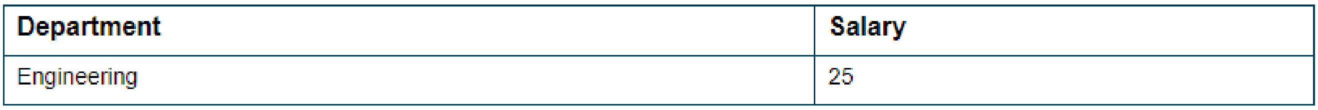
(v) SELECT * FROM Employee where Department='Engineering' and Education='Doctorial';

(vi) SELECT * FROM Employee where Department='Engineering' or Education='Doctorial';

(vii)
```
SELECT Name as First_Name,Education as Qualification from Employee;
```

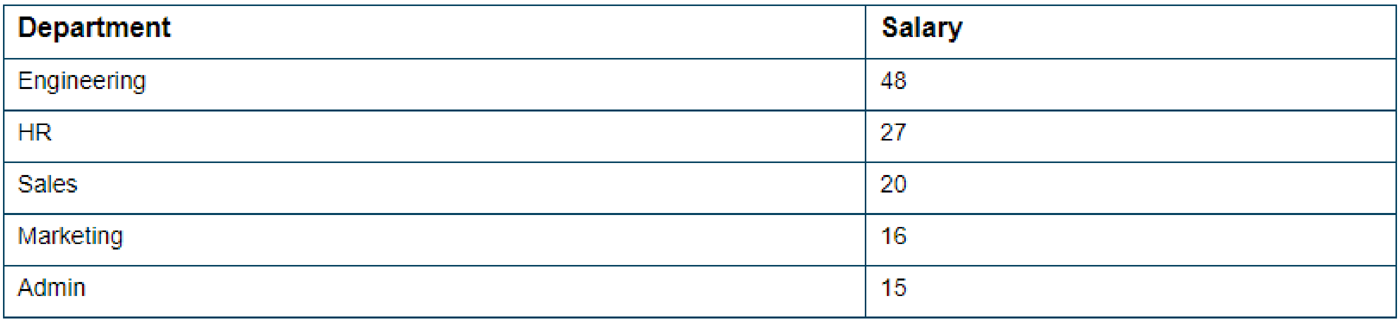
(viii)
```
SELECT * from Employee where Name like 'A%';
```

2)
```
SELECT Department, SUM(Salary)
FROM Employee
WHERE Education <> ‘UG’
GROUP BY Department
HAVING SUM(Salary) >= 20;
```



Department	Salary
Engineering	25

3)
```
SELECT Department, SUM(Salary)

FROM Employee

GROUP BY Department

HAVING SUM(Salary) >= 15

ORDER BY SUM(Salary) DESC;
```



Department	Salary
Engineering	48
HR	27
Sales	20
Marketing	16
Admin	15

Creating Views

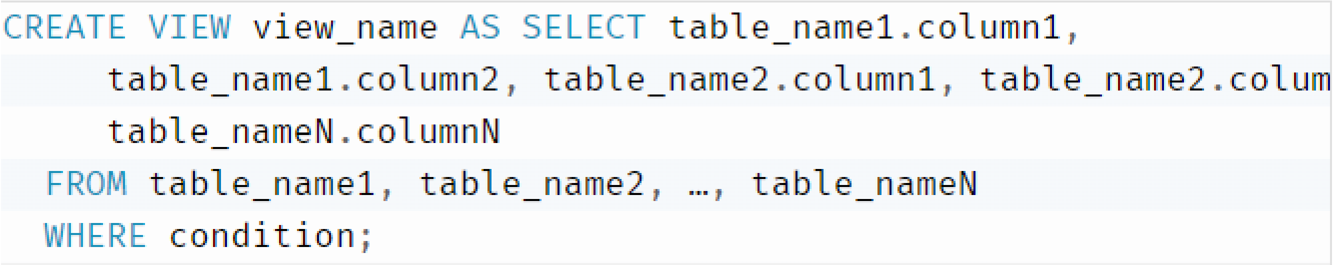
- Views in SQL are considered as a virtual table. A view also contains rows and columns.
- To create the view, we can select the fields from one or more tables present in the database.

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];
```

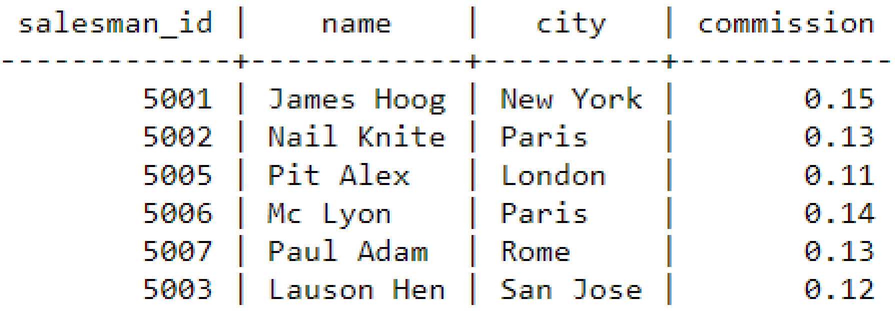
Creating View from multiple tables:



```
CREATE VIEW view_name AS SELECT table_name1.column1,
    table_name1.column2, table_name2.column1, table_name2.colum
    table_nameN.columnN
 FROM table_name1, table_name2, …, table_nameN
 WHERE condition;
```

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

Table: salesman



```
 salesman_id |    name    |   city   | commission
-------------+------------+----------+------------
        5001 | James Hoog | New York |       0.15
        5002 | Nail Knite | Paris    |       0.13
        5005 | Pit Alex   | London   |       0.11
        5006 | Mc Lyon    | Paris    |       0.14
        5007 | Paul Adam  | Rome     |       0.13
        5003 | Lauson Hen | San Jose |       0.12
```

(ii) 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
```

(iii) 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

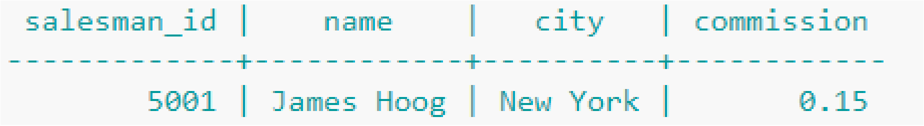
```
 ord_no     purch_amt   ord_date    customer_id salesman_id
----------  ----------  ----------  ----------- -----------
70001       150.5       2012-10-05   3005         5002
70009       270.65      2012-09-10   3001         5005
70002       65.26       2012-10-05   3002         5001
70004       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
```

(iv) **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.**

(v) **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).

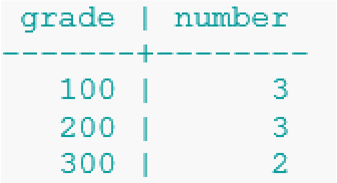
```
CREATE VIEW newyorkstaff AS SELECT * FROM salesman
WHERE city = 'New York';

SELECT * FROM newyorkstaff;
```



```
 salesman_id |    name    |   city   | commission
-------------+------------+----------+------------
        5001 | James Hoog | New York |       0.15
```

(ii)

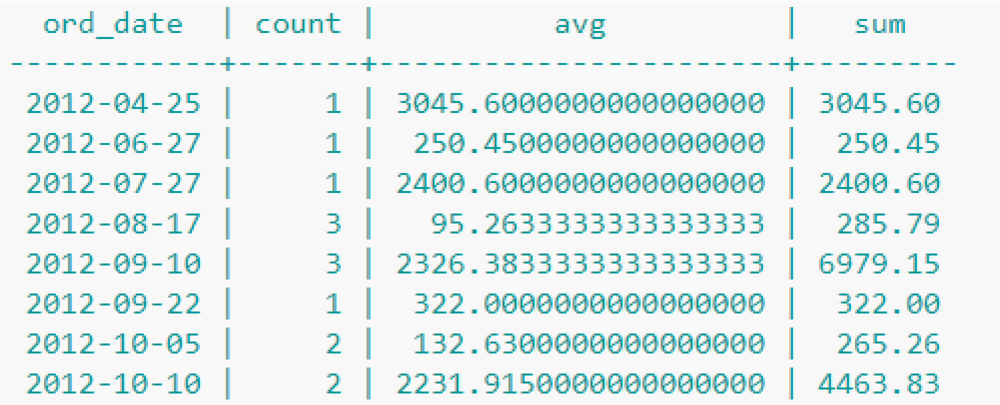
```
CREATE VIEW gradecount (grade, number) AS SELECT grade, COUNT(*)
FROM customer GROUP BY grade;
SELECT * FROM gradecount;
```



```
 grade | number
-------+--------
   100 |      3
   200 |      3
   300 |      2
```

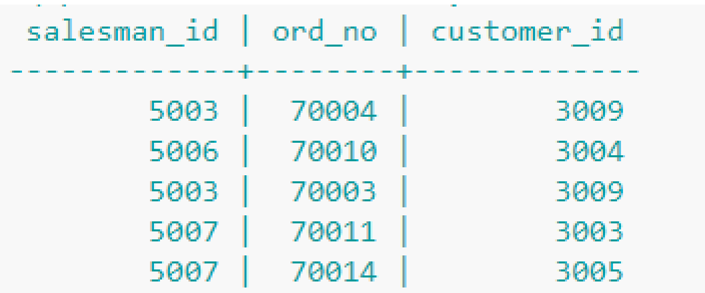
(iii)

```
CREATE VIEW totalforday AS SELECT ord_date, COUNT (DISTINCT
customer_id), AVG(purch_amt), SUM(purch_amt) FROM orders GROUP BY
ord_date;
```



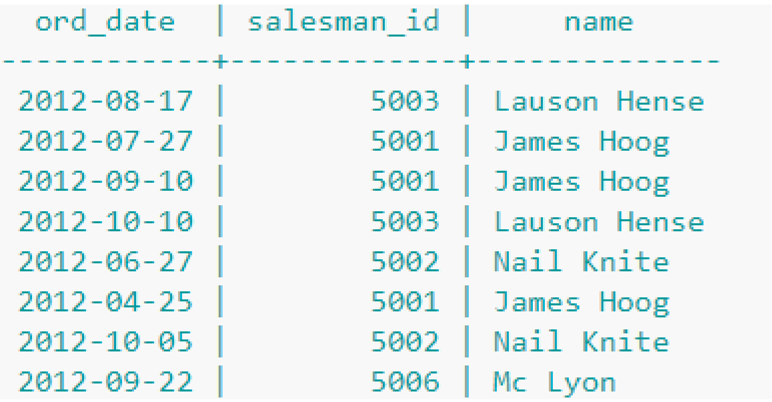
```
  ord_date   | count |          avg           |   sum
------------+-------+-----------------------+----------
 2012-04-25 |     1 | 3045.6000000000000000 | 3045.60
 2012-06-27 |     1 |  250.4500000000000000 |  250.45
 2012-07-27 |     1 | 2400.6000000000000000 | 2400.60
 2012-08-17 |     3 |   95.2633333333333333 |  285.79
 2012-09-10 |     3 | 2326.3833333333333333 | 6979.15
 2012-09-22 |     1 |  322.0000000000000000 |  322.00
 2012-10-05 |     2 |  132.6300000000000000 |  265.26
 2012-10-10 |     2 | 2231.9150000000000000 | 4463.83
```

(iv) CREATE VIEW sorder AS SELECT salesman_id, ord_no, customer_id FROM orders WHERE ord_date IN ('2012-08-17', '2012-10-10');



```
 salesman_id | ord_no | customer_id
-------------+--------+-------------
        5003 |  70004 |        3009
        5006 |  70010 |        3004
        5003 |  70003 |        3009
        5007 |  70011 |        3003
        5007 |  70014 |        3005
```

(v) 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);



```
  ord_date  | salesman_id |     name
------------+-------------+--------------
 2012-08-17 |        5003 | Lauson Hense
 2012-07-27 |        5001 | James Hoog
 2012-09-10 |        5001 | James Hoog
 2012-10-10 |        5003 | Lauson Hense
 2012-06-27 |        5002 | Nail Knite
 2012-04-25 |        5001 | James Hoog
 2012-10-05 |        5002 | Nail Knite
 2012-09-22 |        5006 | Mc Lyon
```

Subquery

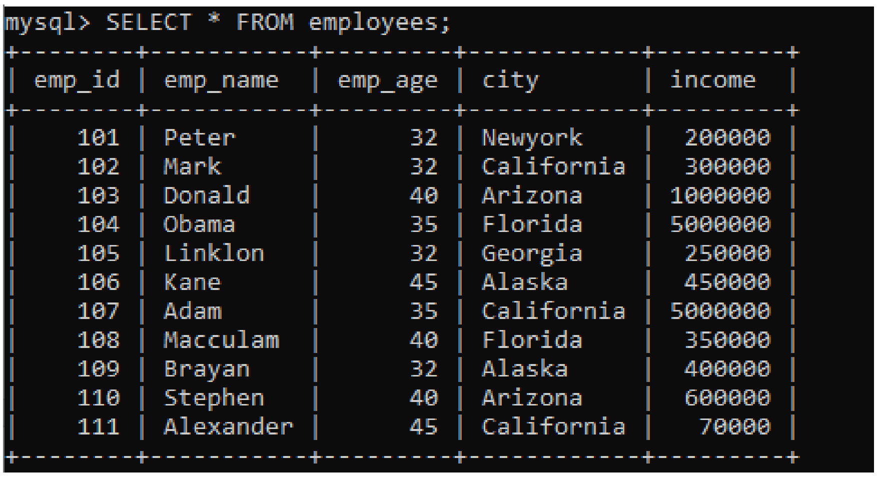
A subquery in MySQL is a query, which is nested into another SQL query and embedded with SELECT, INSERT, UPDATE or DELETE statement along with the various operators. We can also nest the subquery with another subquery. A subquery is known as the **inner query**, and the query that contains subquery is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

SELECT column_list (s) FROM table_name

WHERE column_name OPERATOR

(SELECT column_list (s) FROM table_name [WHERE])

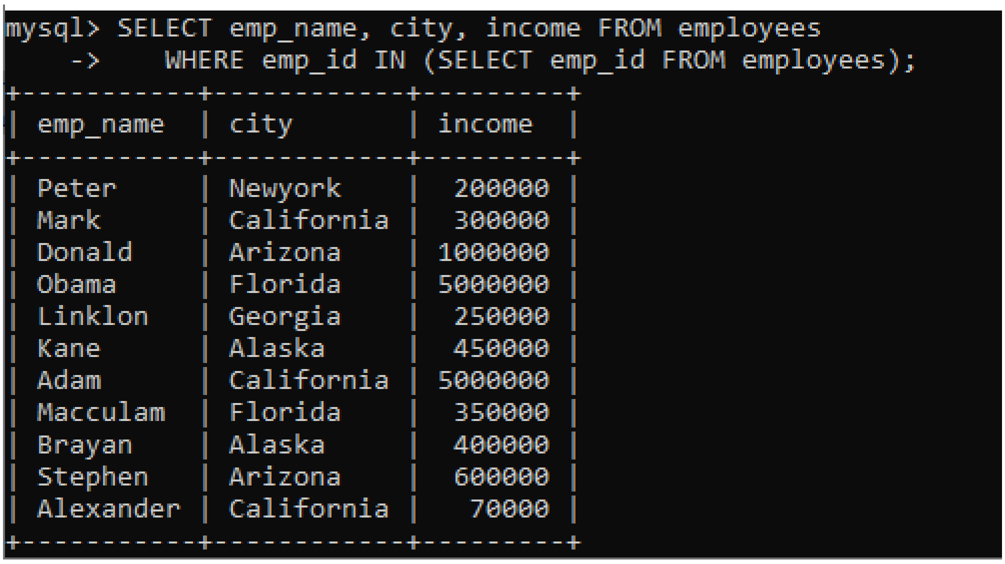
7) Table: employees



```
mysql> SELECT * FROM employees;
+--------+-----------+---------+------------+---------+
| emp_id | emp_name  | emp_age | city       | income  |
+--------+-----------+---------+------------+---------+
|    101 | Peter     |      32 | Newyork    |  200000 |
|    102 | Mark      |      32 | California |  300000 |
|    103 | Donald    |      40 | Arizona    | 1000000 |
|    104 | Obama     |      35 | Florida    | 5000000 |
|    105 | Linklon   |      32 | Georgia    |  250000 |
|    106 | Kane      |      45 | Alaska     |  450000 |
|    107 | Adam      |      35 | California | 5000000 |
|    108 | Macculam  |      40 | Florida    |  350000 |
|    109 | Brayan    |      32 | Alaska     |  400000 |
|    110 | Stephen   |      40 | Arizona    |  600000 |
|    111 | Alexander |      45 | California |   70000 |
+--------+-----------+---------+------------+---------+
```

(i) **Find employee detail whose id matches in a subquery**:

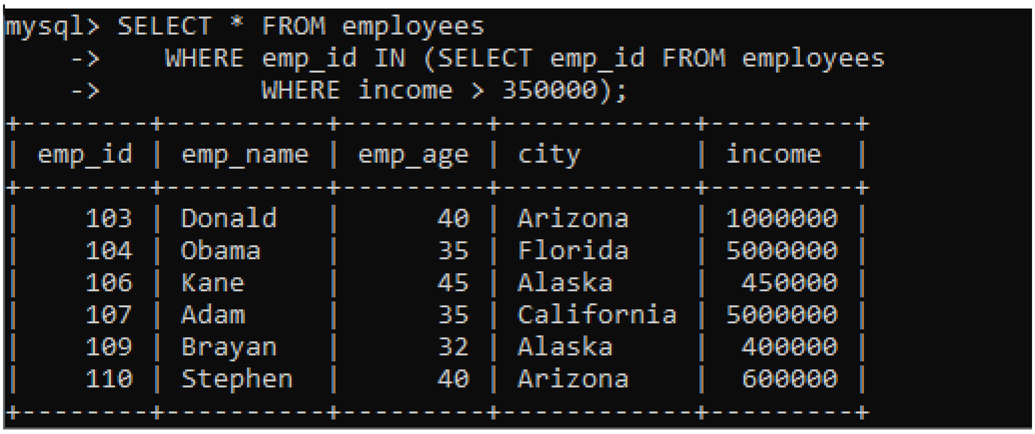
```
SELECT emp_name, city, income FROM employees
  WHERE emp_id IN (SELECT emp_id FROM employees);
```



```
mysql> SELECT emp_name, city, income FROM employees
    ->     WHERE emp_id IN (SELECT emp_id FROM employees);
+-----------+------------+---------+
| emp_name  | city       | income  |
+-----------+------------+---------+
| Peter     | Newyork    |  200000 |
| Mark      | California |  300000 |
| Donald    | Arizona    | 1000000 |
| Obama     | Florida    | 5000000 |
| Linklon   | Georgia    |  250000 |
| Kane      | Alaska     |  450000 |
| Adam      | California | 5000000 |
| Macculam  | Florida    |  350000 |
| Brayan    | Alaska     |  400000 |
| Stephen   | Arizona    |  600000 |
| Alexander | California |   70000 |
+-----------+------------+---------+
```

(ii) **Find employee detail whose income is more than 350000 with the help of subquery:**

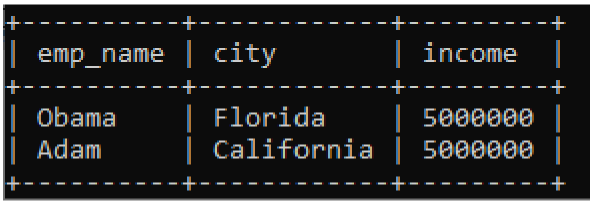
```
SELECT * FROM employees
  WHERE emp_id IN (SELECT emp_id FROM employees
      WHERE income > 350000);
```



```
mysql> SELECT * FROM employees
    ->     WHERE emp_id IN (SELECT emp_id FROM employees
    ->           WHERE income > 350000);
+--------+----------+---------+------------+---------+
| emp_id | emp_name | emp_age | city       | income  |
+--------+----------+---------+------------+---------+
|    103 | Donald   |      40 | Arizona    | 1000000 |
|    104 | Obama    |      35 | Florida    | 5000000 |
|    106 | Kane     |      45 | Alaska     |  450000 |
|    107 | Adam     |      35 | California | 5000000 |
|    109 | Brayan   |      32 | Alaska     |  400000 |
|    110 | Stephen  |      40 | Arizona    |  600000 |
+--------+----------+---------+------------+---------+
```

(iii) Find employee details with maximum income using a subquery.

```
SELECT emp_name, city, income FROM employees
 WHERE income = (SELECT MAX(income) FROM employees);
```

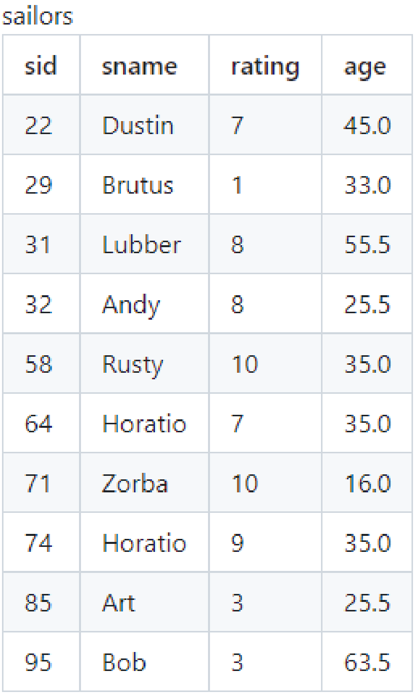


```
+----------+------------+---------+
| emp_name | city       | income  |
+----------+------------+---------+
| Obama    | Florida    | 5000000 |
| Adam     | California | 5000000 |
+----------+------------+---------+
```

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

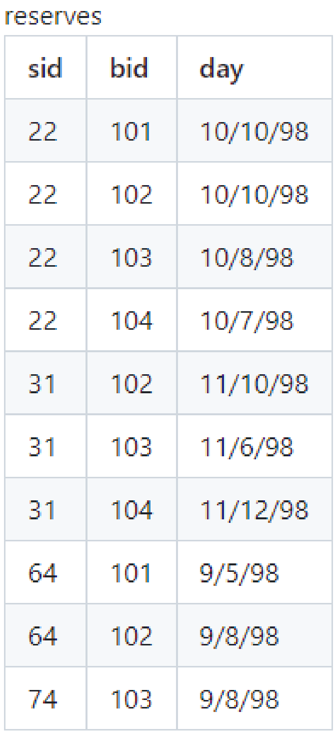
Boats(bid: integer, bname: string, color: string);

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



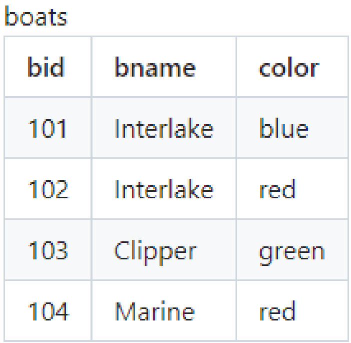
sailors

sid	sname	rating	age
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5



reserves

sid	bid	day
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98



boats

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

a) **Count the number of distinct boat colors:**

```
SELECT COUNT(DISTINCT color) FROM boats;
```

b) **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);
```

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

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

d) **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;
```

e) 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;
```

f) 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;
```

g) Find the ids of sailors who have reserved a red boat or a green boat.

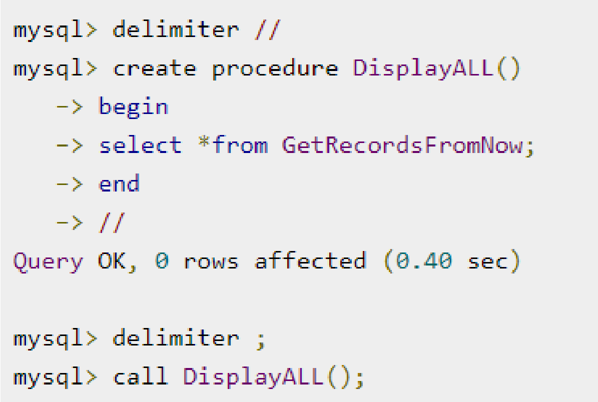
```
SELECT R.sid
FROM boats B, reserves R
WHERE R.bid = B.bid AND B.color = ‘red’
UNION
SELECT R2.sid
FROM boats B2, reserves R2
WHERE R2.bid = B2.bid AND B2.color = ‘green’;
```

h) 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



```
mysql> delimiter //
mysql> create procedure DisplayALL()
   -> begin
   -> select *from GetRecordsFromNow;
   -> end
   -> //
Query OK, 0 rows affected (0.40 sec)

mysql> delimiter ;
mysql> call DisplayALL();
```

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.

2. 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)
```

4 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)
```

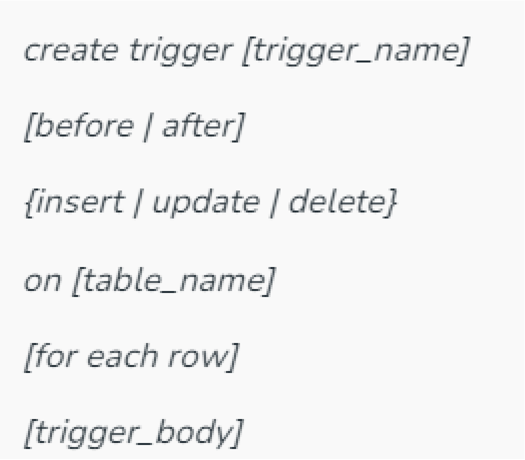
5 Write a PL/SQL procedure to find maximum of three values.

6 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:



```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
[for each row]
[trigger_body]
```

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

1. 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.

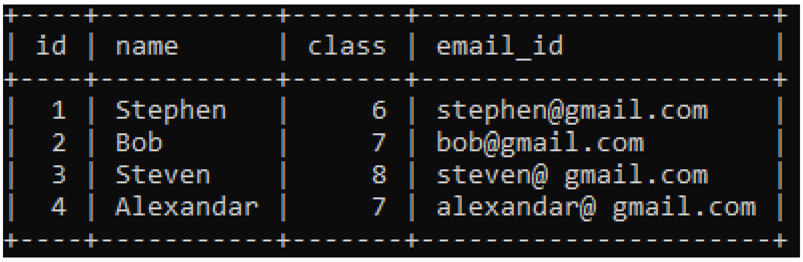
2. **MySQL Trigger : Example BEFORE UPDATE**

```
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      |
+-----------+----------+------+------+------+------+------+-------+-----------+-----------
```

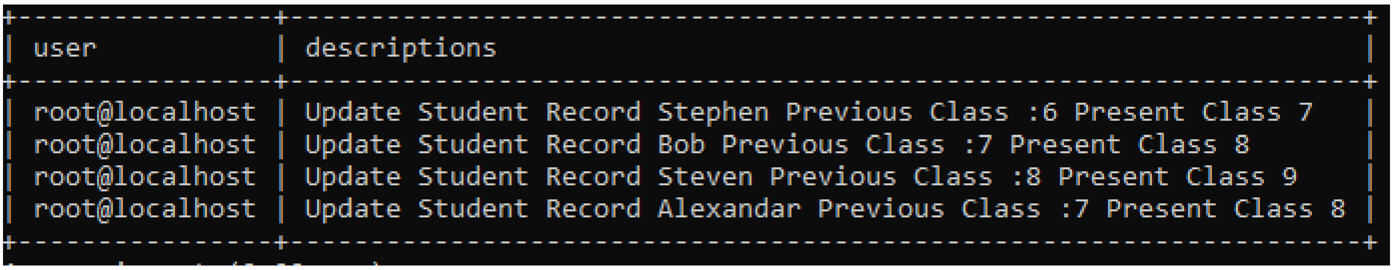
3 AFTER UPDATE Trigger Example

```
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;
```



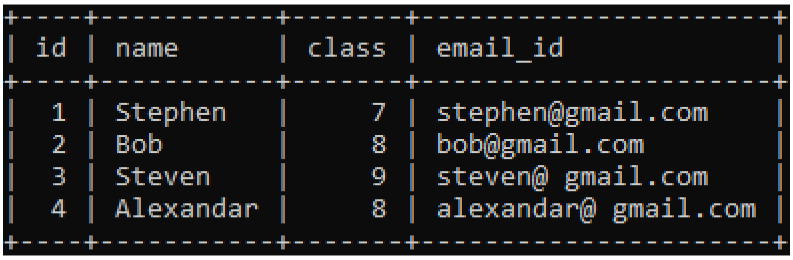
```
+----+-----------+-------+----------------------+
| id | name      | class | email_id             |
+----+-----------+-------+----------------------+
|  1 | Stephen   |     6 | stephen@gmail.com    |
|  2 | Bob       |     7 | bob@gmail.com        |
|  3 | Steven    |     8 | steven@ gmail.com    |
|  4 | Alexandar |     7 | alexandar@ gmail.com |
+----+-----------+-------+----------------------+
```

```
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;
```



```
+----------------+------------------------------------------------------------------+
| user           | descriptions                                                     |
+----------------+------------------------------------------------------------------+
| root@localhost | Update Student Record Stephen Previous Class :6 Present Class 7  |
| root@localhost | Update Student Record Bob Previous Class :7 Present Class 8      |
| root@localhost | Update Student Record Steven Previous Class :8 Present Class 9   |
| root@localhost | Update Student Record Alexandar Previous Class :7 Present Class 8 |
+----------------+------------------------------------------------------------------+
```

```
mysql> select * from students;
```



```
+----+-----------+-------+----------------------+
| id | name      | class | email_id             |
+----+-----------+-------+----------------------+
|  1 | Stephen   |     7 | stephen@gmail.com    |
|  2 | Bob       |     8 | bob@gmail.com        |
|  3 | Steven    |     9 | steven@ gmail.com    |
|  4 | Alexandar |     8 | alexandar@ gmail.com |
+----+-----------+-------+----------------------+
```

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.

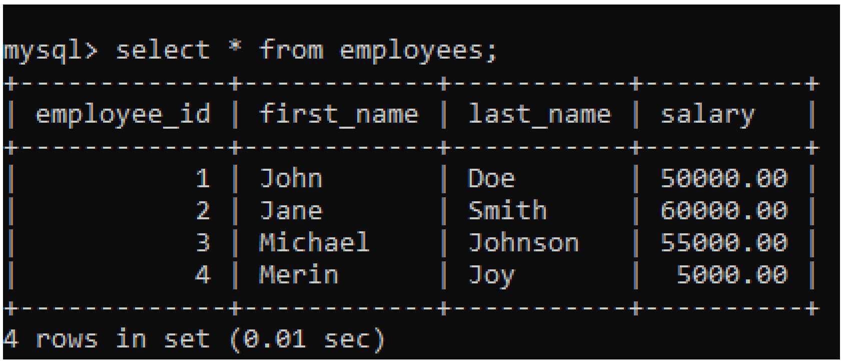
Cursor Actions

- **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

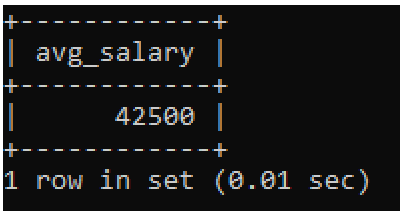
```
DELIMITER //
CREATE PROCEDURE calculate_av_salary()
BEGIN
  DECLARE cur_salary INT;
  DECLARE total_salary INTEGER DEFAULT 0;
  DECLARE num_rows INTEGER DEFAULT 0;
  DECLARE avg_salary INTEGER DEFAULT 0;
  DECLARE done BOOLEAN DEFAULT FALSE;
  DECLARE salary_cursor CURSOR FOR SELECT salary FROM employees;
  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> 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 | Merin      | Joy       |  5000.00 |
+-------------+------------+-----------+----------+
4 rows in set (0.01 sec)
```

```
mysql> call calculate_av_salary();
```



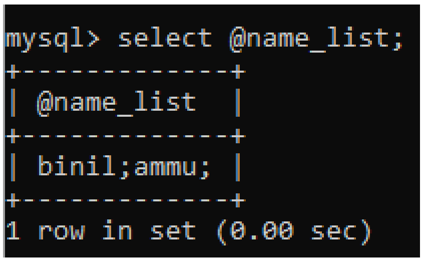
```
+------------+
| avg_salary |
+------------+
|      42500 |
+------------+
1 row in set (0.01 sec)
```

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;
```



```
mysql> select @name_list;
+-------------+
| @name_list  |
+-------------+
| binil;ammu; |
+-------------+
1 row in set (0.00 sec)
```

What is MongoDB?

MongoDB, the most popular NoSQL database, is an open-source document-oriented database. The term ‘NoSQL’ means ‘non-relational’. It means that MongoDB isn’t based on the table-like relational database structure but provides an altogether different mechanism for storage and retrieval of data. Unlike standard relational databases, MongoDB stores data in a JSON document structure form. This makes it easy to operate with dynamic and unstructured data and MongoDB is an open-source and cross-platform database System.

Create a Database

You can change or create a new database by typing use then the name of the database.

Eg: use test

Create Collection same ae table in SQL)

You can create a collection using the createCollection() database method.

Eg: db.createCollection("student")

Insert Documents (Documents are equivalent to records or rows in a relational database table)

```
db.student.insert({rollno:101,name:"Alex",Branch:"MCA"})

db.student.insert({rollno:102,name:"Maya",Branch:"MBA"})

db.student.insert({rollno:101,name:"Niya",Branch:"BCA"})
```

To select data from a collection in MongoDB, we can use the find() method.

```
db.student.find().pretty()    // to display table in neat format
```

Output:

```
switched to db test
{ "ok" : 1 }
WriteResult({ "nInserted" : 1 })
WriteResult({ "nInserted" : 1 })
WriteResult({ "nInserted" : 1 })
{
  "_id" : ObjectId("65a7ea9d5bf43ea549d25fb7"),
  "rollno" : 101,
  "name" : "Alex",
  "Branch" : "MCA"
}
{
  "_id" : ObjectId("65a7ea9d5bf43ea549d25fb8"),
  "rollno" : 102,
  "name" : "Maya",
  "Branch" : "MBA"
}
{
  "_id" : ObjectId("65a7ea9d5bf43ea549d25fb9"),
  "rollno" : 101,
  "name" : "Niya",
  "Branch" : "BCA"
}
```

1. **Designing NoSQL Database - Employee Management**
    - Create a NoSQL database named "Employee".
    - Create a collection named "EMPL" with fields: "Empno", "Name", "Salary", and "Role".
    - Insert 10 records into the "EMPL" collection.
    - Display the data from the "EMPL" collection in a proper format.
    - Retrieve employees from the "EMPL" collection based on their roles.
    - Update the salary of an employee in the "EMPL" collection.

```
use Employee
db.createCollection("EMPL")
db.EMPL.insertMany([
  { "Empno": 1, "Name": "John Doe", "Salary": 60000, "Role": "Manager" },
  { "Empno": 2, "Name": "Alice Smith", "Salary": 50000, "Role": "Developer" },
  // Insert more records here...
])
db.EMPL.find().pretty()
db.EMPL.find({ "Role": "Manager" }).pretty()
db.EMPL.updateOne(
 { "Name": "Alice Smith" },
{ $set: { "Salary": 55000 } })
```

2. **Performing CRUD Operations - Product Catalog**
    - Create a NoSQL database named "ProductCatalog".
    - Create a collection named "Products" with fields: "ProductID", "ProductName", "Price", and "Quantity".
    - Insert several records into the "Products" collection.
    - Display the data from the "Products" collection.
    - Update the details of a specific product. (For example, increase the quantity of laptops by 10)
    - Delete a product from the collection. (For example, remove the smartphone from the catalog.)

```
use ProductCatalog
db.createCollection("Products")
db.Products.insertMany([
  { "ProductID": 1, "ProductName": "Laptop", "Price": 1000, "Quantity": 20 },
  { "ProductID": 2, "ProductName": "Smartphone", "Price": 500, "Quantity": 30 },
  // Insert more records here...
])
db.Products.find().pretty()
db.Products.updateOne(
  { "ProductName": "Laptop" },
  { $inc: { "Quantity": 10 } }
)
db.Products.deleteOne({ "ProductName": "Smartphone" })
```