Program No:1

Title: Design any database with atleast 3 entities and relationship between then. Apply DDL commands. Draw suitable ER Diagram suitable for the system

1. College Database
2. Library Database
3. Company Database
4. Or any other

---

1. Create table following tables with the attributes mentioned below
   STUDENT (USN,SNAME, ADDRESS ,PHONE)
   SEMSEC (SSID , SEM ,SEC) ,
   IAMARKS (USN ,SUBID,SSID ,TEST1,TEST2,TEST3 ,FINALIA)
   SUBJECT (SUBID,TITLE,SEM,CREDIT)
   CLASS (USN,SSID)
   Create database databasename;
   Use databasename;
   create table tablename (columnname1 datatype(size), columnname1 datatype(size), columnname1 datatype(size).. columnname_n datatype(size));
2. Add new column with name GENDER with datatype varchar and size 1 in table STUDENT
   ALTER TABLE table_name ADD column_name datatype;
3. Add two new columns FEES as integer and DOB as varchar in table STUDENT
   Alter table table_name ADD column columnname1 datatype, ADD column columnname2 datatype;
4. Change the data type of column DOB to Date type in student table
   Alter table table_name MODIFY COLUMN column_name datatype;
5. Change the name of column DOB to Dateofbirth in student table.
   ALTER TABLE tablename rename oldcolumnname TO newcolumnname;
6. Change the table name from STUDENT to NEWSTUDENT.
   Rename table old_tablename to new_tablename;
7. Delete the column PHONE from student table
   alter table table_name drop column_name;
8. Delete the table NEWSTUDENT
   drop table table_name;

Program No:2

Title: Design and implement a database and apply at least 10 different DML queries for the following task. For a given input string display only those records which match the given pattern or a phrase in the search string. Make use of wild characters and LIKE operator for the same. Make use of Boolean and arithmetic operators wherever necessary.

---

1. Create table emp with attributes EMP_ID integer, ENAME varchar (15), JOB varchar (15), HIREDATE DATE, SAL integer (10), COMM integer (10), DEPT_NO integer (15).

   CREATE TABLE emp_chk (EMP_ID integer, ENAME varchar (15), JOB varchar (15), HIREDATE DATE, SAL integer, COMM integer, DEPT_NO integer);

2. Insert the records into the EMP table and display all the records of the emp table.

   insert into table_name (Column_name1,Column_name1..column_name_n) values ('value1','value2',value3);
   *select * from table_name;*

3. Create another table emp1 with all the attributes of emp table and insert all the records of emp table into emp1 table at a time

   CREATE table new_tablename LIKE old_tablename;
   *Select * from new_tablename; (Schema is created for the table)*

   *INSERT INTO DB.new_tablename SELECT * FROM DB.old_tablename;*
   (Values of original table gets inserted to new created table)

   (Multiple values insertion to the DB)
   INSERT INTO sql_workbench.emp_chk (EMP_ID , ENAME, JOB, HIREDATE , SAL , COMM , DEPT_NO) values
   (0002, 'Aditya','Supervisor','2000-10-12',2000,6,5),
   (0003, 'Srihanth','Worker','2012-02-02',5000,8,2),
   (0004, 'Sathya','Emp1','2019-10-12',1000,1,8);

4. Display all the information of emp table whose salary is greater than 2000

   *SELECT * FROM DB.Table_name WHERE SAL>2000;*

5. Display distinct salary from emp table

   SELECT DISTINCT SAL FROM DB.Table_name;

6. Display the record of emp table in ascending order of hire date.

   *SELECT * FROM DB.tablename ORDER BY HIREDATE asc;*

7. Display the record of emp table in decreasing order of hire date.

   *SELECT * FROM DB.tablename ORDER BY HIREDATE desc;*

8. Update the salary of employee to 5000 whose employee id is 7839
   ALTER TABLE emp_chk
   ADD PRIMARY KEY (EMP_ID);
   //Above statements are required if we get error saying:
   [Code: 1175. You are using safe update mode and you tried to update a table without a WHERE that uses a KEY column. To disable safe mode, toggle the option in Preferences -> SQL Editor and reconnect.]

   update emp_chk set SAL = 5000 where EMP_ID = 5;

   Another solution for the above-mentioned problem is:
   SET SQL_SAFE_UPDATES = 0;
   update emp_chk set SAL = 3000 where EMP_ID = 5;
   SET SQL_SAFE_UPDATES = 1;

9. Delete the record of employee whose employee id is 7369
   delete from emp_chk where EMP_ID = 5;

10. Display the salary of employee whose salary is in the range of 2000 to 5000.
    *SELECT * FROM emp_chk WHERE SAL BETWEEN 2000 AND 5000;*

11. Display the records of employee whose jobs are *'Manager', 'Supervisor'*and *Worker*
    *SELECT * FROM emp_chk WHERE JOB IN ('Manager', 'Supervisor', Worker);*

12. Display name of employee whose name starts with A.
    *SELECT * FROM emp_chk WHERE ENAME LIKE 'A%';*

    Display name of employee whose name starts with A or S
    *SELECT * FROM emp_chk WHERE ENAME LIKE 'A%' or ENAME LIKE 'S%';*

13. Display the name of employees whose name does not start with A.
    *SELECT * FROM emp_chk WHERE ENAME NOT LIKE 'A%';*

14. Write a SQL statement to increase the salary of employees under the department 40, 90 and 110 according to the company rules that, salary will be increased by 25% for the department 40, 15% for department 90 and 10% for the department 110 and the rest of the departments will remain same.

```
UPDATE employees SET salary= CASE department_id
                             WHEN 40 THEN salary+(salary*.25)
                             WHEN 90 THEN salary+(salary*.15)
                             WHEN 110 THEN salary+(salary*.10)
                             ELSE salary
                           END
             WHERE department_id IN (40,50,50,60,70,80,90,110);
```

(Try below - DATA MANIPULATION -
SELECT
ORDER BY
SELECT TOP
SELECT DISTINCT
WHERE
NULL
AND
OR
IN
BETWEEN
LIKE
Column & Table Aliases
Joins
INNER JOIN
LEFT JOIN
RIGHT JOIN
FULL OUTER JOIN
Self Join
CROSS JOIN
GROUP BY
HAVING
GROUPING SETS
ROLLUP
Subquery
Correlated Subquery
EXISTS
ANY
ALL
UNION
INTERSECT

EXCEPT
INSERT
INSERT Multiple Rows
INSERT INTO SELECT
UPDATE
UPDATE JOIN
DELETE
MERGE)

Program No:3 – On Constraints

---

1. Consider the following schema for a Library Database:
BOOK (Book_id, Title, Publisher_Name, Pub_Year)
BOOK_AUTHORS (Book_id, Author_Name)
PUBLISHER (Name, Address, Phone)
BOOK_COPIES (Book_id, Branch_id, No-of_Copies)
BOOK_LENDING (Book_id, Branch_id, Card_No, Date_Out, Due_Date)
LIBRARY_BRANCH (Branch_id, Branch_Name, Address)

Write SQL queries to
1. Retrieve details of all books in the library – id, title, name of publisher, authors, number of copies in each branch, etc.
2. Get the particulars of borrowers who have borrowed more than 3 books, but from 1st Jan 2017 to 30th Sep 2017
3. Delete a book in BOOK table. Update the contents of other tables to reflect this data manipulation operation.
4. Partition the BOOK table based on year of publication. Demonstrate its working with a simple query.
5. Create a view of all books and its number of copies that are currently available in the library.
6. List the book id, title, publisher name, author name along with total number of copies of each textbook and sort the records in the descending order of book_id.
7. List the book id, title, publisher name, along with total number of copies of each textbook where number of copies is greater than 2 and sort the records in the ascending order of book_id.
8. Retrieve the book id along with date of publication details of books which are published in 2016.
9. Create / Alter table by ADD/ DROP CONSTRAINT – like NOT NULL, Unique, CHECK, Primary Key, INDEX, AUTO INCREMENT, Foreign Key, DEFAULT

Solutions –

Table Creation -

```
CREATE TABLE PUBLISHER
(PUBL_NAME VARCHAR (20) PRIMARY KEY,
PHONE BIGINT,
ADDRESS VARCHAR (20));

CREATE TABLE BOOK
```

```
(BOOK_ID INTEGER NOT NULL PRIMARY KEY,
TITLE VARCHAR (20),
PUB_YEAR VARCHAR (20),
PUBL_NAME varchar(20),
FOREIGN KEY (PUBL_NAME) REFERENCES PUBLISHER (PUBL_NAME) ON
DELETE CASCADE);

CREATE TABLE BOOK_AUTHORS
(AUTHOR_NAME VARCHAR (20),
BOOK_ID int,
FOREIGN KEY (BOOK_ID) REFERENCES BOOK (BOOK_ID) ON DELETE
CASCADE,
PRIMARY KEY (BOOK_ID, AUTHOR_NAME));
CREATE TABLE LIBRARY_BRANCH
(BRANCH_ID INTEGER PRIMARY KEY,
BRANCH_NAME VARCHAR (50),
ADDRESS VARCHAR (50));

CREATE TABLE BOOK_COPIES
(NO_OF_COPIES INTEGER,
Book_id int,
branch_id int,
FOREIGN KEY (BOOK_ID) REFERENCES BOOK (BOOK_ID) ON DELETE
CASCADE,
FOREIGN KEY (BRANCH_ID) REFERENCES LIBRARY_BRANCH
(BRANCH_ID) ON DELETE
CASCADE,
PRIMARY KEY (BOOK_ID, BRANCH_ID));

CREATE TABLE CARD
(CARD_NO INTEGER PRIMARY KEY);

CREATE TABLE BOOK_LENDING
(DATE_OUT DATE,
DUE_DATE DATE,
Book_id int,
Branch_id int,
card_no int,
FOREIGN KEY (BOOK_ID) REFERENCES BOOK (BOOK_ID) ON DELETE
CASCADE,
FOREIGN KEY (BRANCH_ID) REFERENCES LIBRARY_BRANCH
(BRANCH_ID) ON DELETE
CASCADE,
FOREIGN KEY (CARD_NO) REFERENCES CARD (CARD_NO) ON DELETE
CASCADE,
```

PRIMARY KEY (BOOK_ID, BRANCH_ID, CARD_NO));

Relational Schema -

Insertion of Values to Tables

```
INSERT INTO PUBLISHER VALUES ('MCGRAW-HILL', 9989076587, 'BANGALORE');
INSERT INTO PUBLISHER VALUES ('PEARSON', 9889076565, 'NEWDELHI');
INSERT INTO PUBLISHER VALUES ('Jaico', 7455679345, 'HYDRABAD');
INSERT INTO PUBLISHER VALUES ('LIVRE', 8970862340, 'CHENNAI');
INSERT INTO PUBLISHER VALUES ('PLANETA', 7756120238, 'BANGALORE');

INSERT INTO BOOK VALUES (1,'DBMS','2017-01-18', 'MCGRAW-HILL');
INSERT INTO BOOK VALUES (2,'ADBMS','2016-06-16', 'MCGRAW-HILL');
INSERT INTO BOOK VALUES (3,'CN','2016-09-26', 'PEARSON');
INSERT INTO BOOK VALUES (4,'CG','2015-05-18', 'PLANETA');
INSERT INTO BOOK VALUES (5,'OS','2016-05-09', 'PEARSON');

INSERT INTO BOOK_AUTHORS VALUES ('NAVATHE', 1);
INSERT INTO BOOK_AUTHORS VALUES ('NAVATHE', 2);
INSERT INTO BOOK_AUTHORS VALUES ('TANENBAUM', 3);
INSERT INTO BOOK_AUTHORS VALUES ('EDWARD ANGEL', 4);
INSERT INTO BOOK_AUTHORS VALUES ('GALVIN', 5);

INSERT INTO LIBRARY_BRANCH VALUES (10,'Branch1','BANGALORE');
INSERT INTO LIBRARY_BRANCH VALUES (11,'Branch2','BANGALORE');
INSERT INTO LIBRARY_BRANCH VALUES (12,'Branch3', 'BANGALORE');
INSERT INTO LIBRARY_BRANCH VALUES (13,'Branch4','MANGALORE');
INSERT INTO LIBRARY_BRANCH VALUES (14,'Branch5','Mysore');

INSERT INTO BOOK_COPIES VALUES (10, 1, 10);
INSERT INTO BOOK_COPIES VALUES (5, 1, 11);
INSERT INTO BOOK_COPIES VALUES (2, 2, 12);
INSERT INTO BOOK_COPIES VALUES (5, 2, 13);
INSERT INTO BOOK_COPIES VALUES (7, 3, 14);
INSERT INTO BOOK_COPIES VALUES (1, 5, 10);
INSERT INTO BOOK_COPIES VALUES (3, 4, 11);

INSERT INTO CARD VALUES (100);
INSERT INTO CARD VALUES (101);
INSERT INTO CARD VALUES (102);
INSERT INTO CARD VALUES (103);
```

```
INSERT INTO CARD VALUES (104);

INSERT INTO BOOK_LENDING VALUES ('2017-01-21','2017-06-21', 1, 10, 101);
INSERT INTO BOOK_LENDING VALUES ('2017-02-19','2017-07-19', 3, 14, 101);
INSERT INTO BOOK_LENDING VALUES ('2017-02-26','2017-07-26', 2, 13, 101);
INSERT INTO BOOK_LENDING VALUES ('2017-03-15','2017-09-15', 4, 11, 101);
INSERT INTO BOOK_LENDING VALUES ('2017-04-12','2017-10-12', 1, 11, 104);
```

Queries:

1. Retrieve details of all books in the library – id, title, name of publisher, authors, number of copies in each branch, etc.

Sol:

```
Use lab2;
SELECT B.BOOK_ID, B.TITLE, B.PUBL_NAME, A.AUTHOR_NAME,
C.NO_OF_COPIES, L.BRANCH_ID
FROM BOOK B, BOOK_AUTHORS A, BOOK_COPIES C, LIBRARY_BRANCH
L
WHERE B.BOOK_ID=A.BOOK_ID
AND B.BOOK_ID=C.BOOK_ID
AND L.BRANCH_ID=C.BRANCH_ID;
```

2. Get the particulars of borrowers who have borrowed more than 3 books, but from
Jan 2017 to Jun 2017.

Sol:

```
use lab2;

SELECT CARD_NO
FROM BOOK_LENDING
WHERE DATE_OUT BETWEEN '2017-01-21' AND '2017-09-15'
GROUP BY CARD_NO
HAVING COUNT(*)>3;
```

3. Delete a book in BOOK table. Update the contents of other tables to reflect this data
manipulation operation.

Sol:

```
DELETE FROM BOOK
WHERE BOOK_ID=3;
```

4. Partition the BOOK table based on year of publication. Demonstrate its working with a simple query.
Sol:

```
CREATE VIEW V_PUBLICATION AS
SELECT PUB_YEAR
FROM BOOK;
```

5. Create a view of all books and its number of copies that are currently available in the library.
Sol:

```
CREATE VIEW V_BOOKS AS
SELECT B.BOOK_ID, B.TITLE, C.NO_OF_COPIES
FROM BOOK B, BOOK_COPIES C, LIBRARY_BRANCH L
WHERE B.BOOK_ID=C.BOOK_ID
AND C.BRANCH_ID=L.BRANCH_ID;
```

6. List the book id, title ,publisher name, author name along with total number of copies of each textbook and sort the records in the descending order of book_id.

```
SELECT B.BOOK_ID, B.TITLE, B.PUBL_NAME,
A.AUTHOR_NAME,SUM(NO_OF_COPIES)
FROM BOOK B, BOOK_AUTHORS A, BOOK_COPIES C, LIBRARY_BRANCH
L
WHERE B.BOOK_ID=A.BOOK_ID
AND B.BOOK_ID=C.BOOK_ID
AND L.BRANCH_ID=C.BRANCH_ID
group by book_id
order by book_id DESC;
```

7. List the book id, title, publisher name, along with total number of copies of each textbook where number of copies is greater than 2 and sort the records in the ascending order of book_id.

```
SELECT B.BOOK_ID, B.TITLE, B.PUBL_NAME,
A.AUTHOR_NAME,SUM(NO_OF_COPIES)
FROM BOOK B, BOOK_AUTHORS A, BOOK_COPIES C, LIBRARY_BRANCH
L
WHERE B.BOOK_ID=A.BOOK_ID
AND B.BOOK_ID=C.BOOK_ID
AND L.BRANCH_ID=C.BRANCH_ID
group by book_id
having SUM(NO_OF_COPIES)>2
order by book_id;
```

8. Retrieve the book id along with date of publication details of books which are published in 2016.

```
SELECT BOOK_ID,PUB_YEAR FROM BOOK WHERE PUB_YEAR LIKE
'2016%';
```

Lab Programs – 4 AND 5 (On Aggregate Functions and Nested / Sub Queries)

Title: Implement nested sub queries (Single Row Query / Multiple Row Queries)
While Creating Table Declare constraints where ever required.

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
100	Steven	King	SKING	515.123.4567	1987-06-17	AD_VP	24000	0	0	90
101	Neena	Kochhar	NKOCHHAR	515.123.4568	1987-06-18	AD_VP	17000	0	100	90
107	Diana	Lorentz	DLORENTZ	590.423.5567	1987-06-24	IT_PROG	4200	0	103	60
108	Nancy	Greenberg	NGREENBE	515.124.4569	1987-06-25	FI_MGR	12000	0	101	100
114	Den	Raphaely	DRAPHEAL	515.127.4561	1987-07-01	CLK	11000	0	114	30

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
90	VP	0	1000
100	Finance	101	1400
30	Clerk	114	1200
60	IT	103	1100
70	Press	110	1300

LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	COUNTRY_ID
1000	Avenue	989	Roma		IT
1100	Rashti Road	10934	Venice		IT
1200	Shinju Rd	1689	Tokyo	Tokyo Prefecture	JP
1300	Kamiya Rd	6823	Hiroshima		JP
1400	Jabberwocky Rd	26192	Southlake	Texas	US

1. Create Table Employees, using constraints on required columns (PK, NOT NULL, CHECK, FK) –

```
create database Employees;

Use Employees;

/* LOCATION*/
```

```
CREATE TABLE IF NOT EXISTS LOCATION (
LOCATION_ID int NOT NULL PRIMARY KEY,
ADRESS varchar(20) DEFAULT NULL,
POSTAL_CODE int NOT NULL,
CITY VARCHAR(20) NOT NULL, STATE_PROVINCE varchar(20) DEFAULT NULL,
COUNTRY_ID varchar(20) NOT NULL);
Desc LOCATION;
/* Department*/
CREATE TABLE IF NOT EXISTS Department (
DEPARTMENT_ID int NOT NULL PRIMARY KEY,
DEPARTMENT_NAME varchar(20) DEFAULT NULL,
MANAGER_ID int UNIQUE NOT NULL ,
LOCATION_ID int DEFAULT NULL,
foreign key (LOCATION_ID) REFERENCES LOCATION (LOCATION_ID));
Desc Department;
/* employees*/
CREATE TABLE employees (
EMPLOYEE_ID decimal(6,0) NOT NULL PRIMARY KEY,
FIRST_NAME varchar(20) DEFAULT NULL,
LAST_NAME varchar(25) NOT NULL,
EMAIL varchar(25) NOT NULL,
PHONE_NUMBER varchar(20) DEFAULT NULL,
HIRE_DATE date NOT NULL,
JOB_ID varchar(10) NOT NULL,
SALARY decimal(8,2) DEFAULT NULL,
COMMISSION_PCT decimal(2,2) DEFAULT NULL,
MANAGER_ID int DEFAULT NULL ,
DEPARTMENT_ID int,
FOREIGN KEY(DEPARTMENT_ID) REFERENCES Department (DEPARTMENT_ID));
Desc employees;
```

2. Insert values as in Location, Department and Employees table respectively.

/*Location */

```
insert into LOCATION values (1000,'Avenue','989','Roma','Italy','IT');

insert into LOCATION values (1100,'Rashti ','10934','Venice','Italy','IT');

insert into LOCATION values (1200,'Shinju ','1689','Tokyo','Tokyo','JP');

insert into LOCATION values (1400,'Jabberwocky ','26192','Southlake','Michigan','US');

INSERT INTO LOCATION VALUES (1300, 'Shinju Rd', '6823', 'Tokoy', 'Texus', 'JP');
```

/*Department*/

```
INSERT INTO Department VALUES(90,'VP',0,1000);

INSERT INTO Department VALUES(100,'Finance',101,1400);

INSERT INTO Department VALUES(30,'Clerk ',114,1200);

INSERT INTO Department VALUES(60,'IT',103,1100);

INSERT INTO Department VALUES(70,'Press',110,1300);
```

/* Employees*/

```
INSERT INTO employees VALUES(100,'Steven','King','SKING','456789','1987-06-17','AD_VP',24000,0,'0',90);

INSERT INTO employees VALUES(101,'Neena','Kochhar','NKOCHHAR','23646','1987-06-18','AD_VP',17000,0,100,90);

INSERT INTO employees VALUES(107,'Diana','Lorentz','DLORENTZ','15863','1987-06-24','IT_PROG',4200,0,103,60);

INSERT INTO employees VALUES(108,'Nancy','Greenberg','NGREENBE','79546','1987-06-25','FI_MGR',12000,0,101,100);

INSERT INTO employees VALUES(114,'Den','Raphaely','DRAPHEAL','79556','1988-06-25','CLK',11000,0,114,30);
```

3. Write a SQL query to find those employees who receive a higher salary than the employee with ID 114. Return first name, last name
   Solution –

```
SELECT first_name, last_name
FROM employees
WHERE salary >
( SELECT salary
     FROM employees
       WHERE employee_id=114
);
```

4. Write a SQL query to find those employees whose salary matches the lowest salary of any of the departments. Return first name, last name and department ID.

Solution –

```
SELECT first_name, last_name, salary, department_id  FROM employees

WHERE salary IN

( SELECT MIN(salary)
FROM employees
GROUP BY department_id
);
```

5. write a SQL query to find those employees who earn more than the average salary. Return employee ID, first name, last name
   Solution –

```
SELECT employee_id, first_name,last_name
FROM employees
WHERE salary >
( SELECT AVG(salary)
FROM employees
      );
```

6. write a SQL query to find all those employees who work in the IT department. Return department ID, name (first), job ID and department name.
   Solution –

```
SELECT e.department_id, e.first_name, e.job_id , d.department_name
FROM employees e , department d
WHERE e.department_id = d.department_id
      AND  d.department_name = 'IT';
```

7. write a SQL query to find those employees whose salary falls within the range of the smallest salary and 10000. Return all the fields.

```sql
SELECT *
FROM employees
WHERE salary BETWEEN
(SELECT MIN(salary)
      FROM employees) AND 10000;
```

8. write a SQL query to find those employees who do not work in the departments where managers' IDs are between 100 and 110 (Begin and end values are included.). Return all the fields of the employees.

```sql
SELECT *
FROM employees
WHERE department_id NOT IN
(SELECT department_id
FROM department
WHERE manager_id BETWEEN 100 AND 110);
```

9. write a SQL query to find those employees who get second-highest salary. Return all the fields of the employees.

```sql
SELECT *
FROM employees
WHERE employee_id IN
(SELECT employee_id
FROM employees
WHERE salary =
(SELECT MAX(salary)
FROM employees
WHERE salary <
(SELECT MAX(salary)
FROM employees)));
```

10. write a SQL query to find those employees who earn more than the average salary and work in the same department as an employee whose first name contains the letter 'S'. Return employee ID, first name and salary.

```sql
SELECT employee_id, first_name , salary
FROM employees
WHERE salary >
(SELECT AVG (salary)
FROM employees )
AND  department_id IN
( SELECT department_id
FROM employees
WHERE first_name LIKE '%S%');
```

11. write a SQL query to calculate total salary of the departments where at least one employee works. Return department ID, total salary

```
SELECT department.department_id, result1.total_amt
FROM department,
( SELECT employees.department_id, SUM(employees.salary) total_amt
FROM employees
GROUP BY department_id) result1
      WHERE result1.department_id = department.department_id;
```

Lab Program 5 - 5 (On Aggregate Functions and Nested / Sub Queries)

12. Write a query to display the employee id, name (first name and last name) and the job id column with a modified title SALESMAN for those employees whose job title is ST_MAN and DEVELOPER for whose job title is IT_PROG

```
SELECT  employee_id,  first_name, last_name,
CASE job_id
WHEN ' FI_MGR' THEN ‘MANAGER’
WHEN 'IT_PROG' THEN 'DEVELOPER'
ELSE job_id
END AS designation,  salary
FROM employees;
```

13. write a SQL query to find those employees whose salaries exceed 50% of their department's total salary bill. Return first name, last name.

```
SELECT e1.first_name, e1.last_name
FROM employees e1
WHERE salary >
( SELECT (SUM(salary))*.5
FROM employees e2
WHERE e1.department_id=e2.department_id);
```

14. write a SQL query to find those employees who manage a department. Return all the fields of employees table.

```
SELECT *
FROM employees
WHERE employee_id=ANY
( SELECT manager_id FROM department );
```

15. write a SQL query to find those managers who supervise four or more employees. Return manager name, department ID.

```
SELECT first_name || ' ' || last_name AS Manager_name,department_id
     FROM employees
          WHERE employee_id IN
          (SELECT manager_id
          FROM employees
               GROUP BY manager_id
               HAVING COUNT(*)>=1);
```

16. write a SQL query to find the first name, last name, department, city, and state province for each employee.

```
SELECT E.first_name,E.last_name,
D.department_name, L.city, L.state_province
FROM employees E
JOIN department D
  ON E.department_id = D.department_id
    JOIN location L
  ON D.location_id = L.location_id;
```

17. write a SQL query to find the employees and their managers. These managers do not work under any manager. Return the first name of the employee and manager.

```
SELECT E.first_name AS "Employee Name",
   M.first_name AS "Manager"
    FROM employees E
      LEFT OUTER JOIN employees M
            ON E.manager_id = M.employee_id;
```

18. write a SQL query to calculate the average salary, the number of employees receiving commissions in that department. Return department name, average salary and number of employees

```
SELECT department_name, AVG(salary), COUNT(commission_pct)
     FROM department
          JOIN employees USING (department_id)
GROUP BY department_name;
```

19. write a SQL query to find the department name, department ID, and number of employees in each department.

```
SELECT d.department_name,
       e.*
FROM department d
JOIN
  (SELECT count(employee_id),
          department_id
   FROM employees
GROUP BY department_id) e USING (department_id);
```

Practice -

20. write a SQL query to find those employees who earn more than the average salary. Sort the result-set in descending order by salary. Return first name, last name, salary, and department ID.

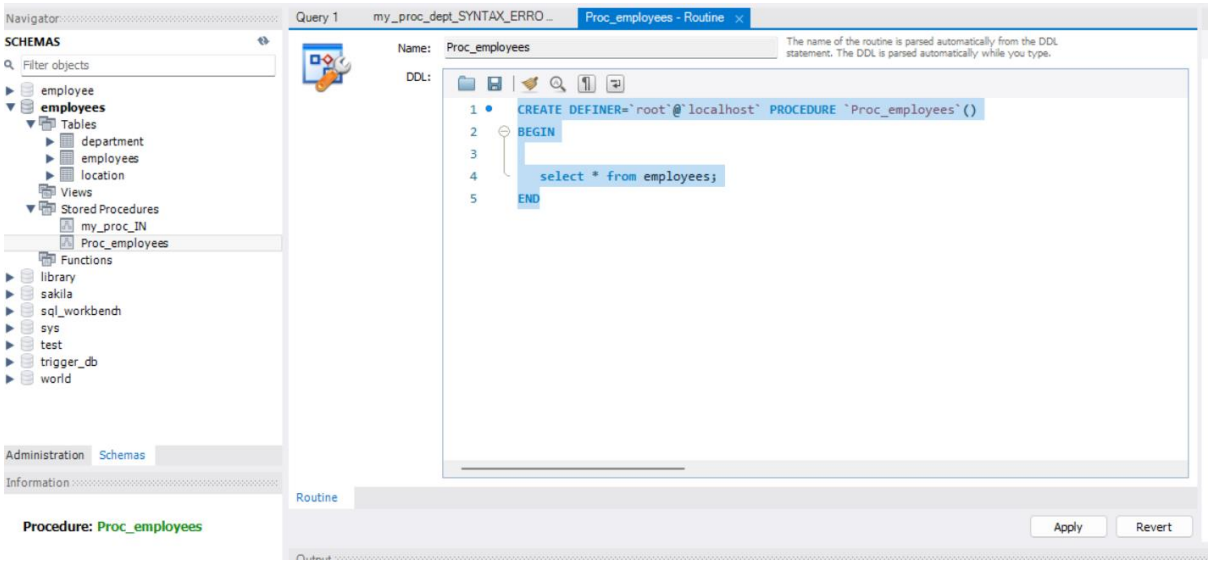
```
SELECT first_name, last_name , salary, department_id
  FROM employees
    WHERE salary > (
                SELECT AVG(salary)
                   FROM employees )
    ORDER BY salary DESC;
```

21. write a SQL query to find the first name, last name, department number, and department name for each employee

```
SELECT E.first_name , E.last_name ,
       E.department_id , D.department_name
        FROM employees E
         JOIN department D
          ON E.department_id = D.department_id;
```

```
mysql> DELIMITER //
mysql> create procedure proc_dept()
    -> begin
    -> select * from department;
    -> end//
Query OK, 0 rows affected (0.04 sec)

mysql> delimiter ;
mysql> call proc_dept();
+---------------+-----------------+------------+-------------+
| DEPARTMENT_ID | DEPARTMENT_NAME | MANAGER_ID | LOCATION_ID |
+---------------+-----------------+------------+-------------+
|            30 | Clerk           |        114 |        1200 |
|            60 | IT              |        103 |        1100 |
|            70 | Press           |        110 |        1300 |
|            90 | VP              |          0 |        1000 |
|           100 | Finance         |        101 |        1400 |
+---------------+-----------------+------------+-------------+
5 rows in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)
```

```
MySQL 8.0 Command Line Client - Unicode

mysql> use employees;
Database changed
mysql> call my_proc_Max_sal();
+-------------+
| MAX(salary) |
+-------------+
|    11000.00 |
|     4200.00 |
|    24000.00 |
|    12000.00 |
+-------------+
4 rows in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

mysql> DELIMITER //
mysql> create procedure avg_sal()
    -> begin
    -> select AVG(Salary) from employees;
    -> end//
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> call avg_sal();
+---------------+
| AVG(Salary)   |
+---------------+
| 13640.000000 |
+---------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

mysql>
```



Stored Procedure Programs with -

1. Simple SP creation and calling
2. SP using IN
3. SP using out
4. SP using INOUT
5. Drop Procedure
6. Display procedures used in DB
7. Definition of Stored Procedures
8. Nested Stored Procedures
9. Using Conditional Stored Procedures

Note use Previously created Employees table to perform the below Programs

1. Write a SQL query to find those employees whose salary is lowest / Highest / Avg salary by the departments

a) Lowest Salary

```
-→  Use DB;
```

```
-→ Delimiter //
Create Procedure my_proc_Min_Sal()
BEGIN

SELECT MIN(salary)FROM employees
GROUP BY department_id;
```

```
END//


->Delimiter ;

->Call my_proc_Min_Sal ();
```

b) Highest salary
C) Average salary

2. Write a SQL query to find those employees who earn more than the average salary. Sort the result-set in descending order by salary. Return first name, last name, salary, and department ID.

```
DELIMITER //

Create Procedure my_proc_Morethan_AVG_Sal()
BEGIN

SELECT first_name, last_name , salary, department_id
  FROM employees
    WHERE salary > (
                SELECT AVG(salary)
                   FROM employees )
    ORDER BY salary DESC;
      END//

      Delimiter;

      Call my_proc_Morethan_AVG_Sal();
```

3. Write a SQL query to find the first name, last name, department number, and department name for each employee

(`my_proc_emp_dept_details`)

To Create a parameterized stored procedure – Syntax

```
(IN | OUT | INOUT) (Parameter Name [datatype(length)])
```

4. Write a SQL query to find those employees who receive a higher salary than the employee with ID 114. Return first name, last name

```
/* IN Parameter*/

DELIMITER //

Create Procedure empl_info(IN empl_id integer)
BEGIN

SELECT employee_id,first_name,last_name
FROM employees
WHERE salary >
(SELECT salary
     FROM employees
       WHERE employee_id= empl_id);
END//

DELIMITER ;

CALL empl_info(114);
```

5. Write a SQL query to calculate the number of employees receiving salary more than 10000. Return number of employees.

```
/* OUT Parameter*/
```

/* To get the count of employee with Salary >10000, the `Emp` is declared as out parameter, and the data type as Integer. The count of the employees in query is then assigned to the **OUT** variable (`Emp`) using the **INTO** keyword.

```
DELIMITER //

Create Procedure empl_info_Out(OUT Emp Int)
BEGIN

SELECT COUNT(*) into Emp
      FROM department
           JOIN employees USING (department_id)
        where Salary >= 10000;
END//

DELIMITER ;
```

/* To store the value returned by the procedure, needs to be passed to a session variable named `@Emp_Count` */

```
CALL empl_info_Out(@Emp_Count);
Select @Emp_Count as Employee_Sal;
```

```
Expected Solution– Query should return, total number of employees
having salary >= 10000
```

6. Write a SQL query to find those employees who receive a higher salary than the employee with ID declared IN Parameter. Return Count of Employees whose salary is greater than Employee ID passed in the IN parameter

```
/* INOUT Parameter*/

DELIMITER //

Create Procedure empl_info_InOut(INOUT Emp_Sal interger ,IN empl_id
integer)

BEGIN

SELECT count(employee_id)into Emp_Sal,
FROM employees
WHERE salary >
(SELECT salary
     FROM employees
       WHERE employee_id= empl_id);
END//

DELIMITER ;

CALL empl_info_InOut(@Emp_Sal,114);
Select @Emp_Sal as No_of_Emp;

Expected Solution – Count of Employees whose salary is greater than
the salary of Parameter passed Employee Id is displayed.
```

7. Command to see the definition of Created Procedures

```
SHOW CREATE PROCEDURE empl_info_Out;
```

8. View the list of stored procedure in a database using a query

- Displays list of Procedures used in DB

```
select routine_name,
routine_type,definer,created,security_type,SQL_Data_Access
from information_schema.routines where
routine_type='PROCEDURE' and routine_schema='employees';
```

OR

```
SHOW PROCEDURE STATUS WHERE name LIKE '%Emp%';
```

9. Drop the existing Procedure from the database.

Syntax – DROP PROCEDURE IF EXISTS 'DB name'.'Procedure name';

```
DROP PROCEDURE IF EXISTS `employees`.`my_proc_Morethan_AVG_Sal`;
```

Note – To Alter Procedure

Updating procedure body, number, and types of parameters. **For example,** you want to add a new IN parameter, or change the data type of an existing parameter, etc, then there is no way to directly update these through a command.

The only option in such cases is to

- DROP the existing procedure.
- CREATE the new procedure with an updated body/parameter list etc.

Program 7 - Set 2 - Practice Stored Procedure Queries – https://www.softwaretestinghelp.com/mysql-stored-procedure/

```
1.    CREATE SCHEMA stored_proc

2.    CREATE TABLE studentMarks (stud_id SMALLINT(5) NOT NULL AUTO_INCREMENT
PRIMARY KEY, total_marks INT, grade VARCHAR(5));

Insert sample data ---
3.    INSERT INTO studentMarks(total_marks, grade) VALUES(450, 'A'), (480,
'A+'), (490, 'A++'), (440, 'B+'),(400,
'C+'),(380,'C'),(250,'D'),(200,'E'),(100,'F'),(150,'F'),(220, 'E');
```

4. Create Simple Stored Procedure to return all records from Student Marks table -

```
DELIMITER $$
CREATE PROCEDURE GetStudentData()
BEGIN
    SELECT * FROM studentMarks;
END$$
```

```
  DELIMITER ;

CALL GetStudentData();
```

5. Create a procedure to fetch the details of students with the student ID being passed as an Input parameter.

```
DELIMITER //

CREATE PROCEDURE stored_proc.spGetDetailsByStudentName(IN studentId
INT)
BEGIN
    SELECT * FROM studentMarks where stud_id = studentId;
END //
DELIMITER ;
```

CALL stored_proc.spGetDetailsByStudentName(1);

6. To calculate the average marks of all the students from the studentMarks table and return the average as an OUT field.

```
DELIMITER //

CREATE PROCEDURE stored_proc.spGetAverageMarks(OUT average
DECIMAL(5,2))
BEGIN
 SELECT AVG(total_marks) INTO average FROM studentMarks;
END //
DELIMITER ;
```

CALL stored_proc.spGetAverageMarks(@average_marks);

```
SELECT @average_marks;
```

7. Find the count of students who is having marks below the average marks of all the students using Local variables.

```
DELIMITER //
CREATE PROCEDURE stored_proc.spCountOfBelowAverage(OUT
countBelowAverage INT)
BEGIN
    DECLARE avgMarks DECIMAL(5,2) DEFAULT 0;
    SELECT AVG(total_marks) INTO avgMarks FROM studentMarks;
    SELECT COUNT(*) INTO countBelowAverage FROM studentMarks WHERE
total_marks < avgMarks;
END //
DELIMITER ;

CALL stored_proc.spCountOfBelowAverage(@countBelowAverage);
```

```
SELECT @countBelowAverage;
```

8. To fetch the highest marks from a student data table. We can have one like this with the highest marks stored in an OUT parameter.

```
DELIMITER //
CREATE PROCEDURE stored_proc.spGetMaxMarks(OUT highestMarks INT)
BEGIN
    SELECT MAX(total_marks) INTO highestMarks FROM studentMarks;
END //
DELIMITER ;

-- calling procedure
CALL stored_proc.spGetMaxMarks(@highestMarks);

-- obtaining value of the OUT parameter
SELECT @highestMarks;
```

9. Calling A Procedure From Another STORED PROCEDURE - (Nested Procedure)

Call a procedure from another procedure to return the overall result of a student. If student marks are above average – then the result would be PASS else – FAIL

Solution - create 2 procedures

a. **Procedure for calculating Average marks and creating Boolean values for the results**

```
DELIMITER $$

CREATE PROCEDURE stored_proc.spGetIsAboveAverage(IN studentId INT, OUT
isAboveAverage BOOLEAN)
BEGIN
    DECLARE avgMarks DECIMAL(5,2) DEFAULT 0;
    DECLARE studMarks INT DEFAULT 0;
    SELECT AVG(total_marks) INTO avgMarks FROM studentMarks;
    SELECT total_marks INTO studMarks FROM studentMarks WHERE stud_id =
studentId;
    IF studMarks > avgMarks THEN
        SET isAboveAverage = TRUE;
    ELSE
        SET isAboveAverage = FALSE;
    END IF;
END$$
DELIMITER ;
```

b. Declaring Students results with Boolean values based on Students marks

```
DELIMITER $$
CREATE PROCEDURE stored_proc.spGetStudentResult(IN studentId INT, OUT
result VARCHAR(20))
BEGIN
      -- nested stored procedure call
    CALL stored_proc.spGetIsAboveAverage(studentId,@isAboveAverage);
    IF @isAboveAverage = 0 THEN
        SET result = "FAIL";
    ELSE
        SET result = "PASS";
    END IF;
END$$
DELIMITER ;
```

Call the procedure to fetch results. (The average marks for all the entries in studentTable is 323.6)

For PASS – We will use studentID 2 – having total marks – 450
Since 450 > 323.6 – we will expect the result to be “PASS”

```
CALL stored_proc.spGetStudentResult(2,@result);
SELECT @result;
```

For FAIL result we will use studentId – 10 having total marks – 150
Since 150 < 323.6 – we will expect the result to be “PASS”

```
CALL stored_proc.spGetStudentResult(10,@result);
SELECT @result;
```

10. Write a procedure to take studentId and depending on the studentMarks we need to return the class according to the below criteria. (Using Conditional statements)

Marks >= 400 : Class – First Class
Marks >= 300 and Marks < 400 – Second Class
Marks < 300 – Failed

Solution –

```
DELIMITER $$
CREATE PROCEDURE stored_proc.spGetStudentClass(IN studentId INT, OUT class
VARCHAR(20))
BEGIN
```

```
    DECLARE marks INT DEFAULT 0;
    SELECT total_marks INTO marks FROM studentMarks WHERE stud_id = studentId;
        IF marks >= 400 THEN
        SET class = "First Class";
    ELSEIF marks >=300 AND marks < 400 THEN
        SET class = "Second Class";
    ELSE
        SET class = "Failed";
    END IF;
END$$
DELIMITER ;
```

a. For student ID – 1 – total_marks are 450 – hence the expected result is FIRST CLASS.

```
CALL stored_proc.spGetStudentClass(1,@class);
SELECT @class;
```

b. For student ID – 6 – total_marks is 380 – Hence the expected result is SECOND CLASS.

```
CALL stored_proc.spGetStudentClass(6,@class);
SELECT @class;
```

c. For student ID – 11 – total_marks are 220 – Hence expected result is FAILED.

```
CALL stored_proc.spGetStudentClass(11,@class);
SELECT @class;
```

Error Handling In STORED PROCEDURES –

Declaring a Handler for Error –

The syntax for declaring handler is
DECLARE {action} HANDLER FOR {condition} {statement}

{action} can have values

- **CONTINUE:** This would still continue executing the current procedure.
- **EXIT:** This procedure execution would halt and the flow would be terminated.

{condition}: It's the event that would cause the HANDLER to be invoked.

- Ex – and MySQL error code – ex. MySQL would throw error code 1062 for a duplicate PRIMARY KEY violation
- SQLWARNING / NOTFOUND – etc are used for more generic cases. **For example**, SQLWARNING condition is invoked whenever the MySQL engine issues any warning for the executed statement. **Example,** UPDATES are done without a WHERE clause.

{statement} – Can be one or multiple statements, which we want to execute when the handler is executing. It would be enclosed within BEGIN and END keywords similar to the actual PROCEDURE body.
Note: It's important to note that, you can declare multiple handlers in a given MySQL procedure body definition. This is analogous to having multiple catch blocks for different types of exceptions in a lot of programming languages like Java, C#, etc.

Example of HANDLER declaration for a DUPLICATE KEY INSERT (which is error code 1062) –

```
DECLARE EXIT HANDLER FOR 1062
BEGIN
  SELECT 'DUPLICATE KEY ERROR' AS errorMessage;
END;
```

- Created a handler for error 1062. As soon as this condition is invoked, the statements between the BEGIN and END block would be executed.
- In this case, the SELECT statement would return the errorMessage as 'DUPLICATE KEY ERROR'.
- Can add multiple statements as needed within this BEGIN..END block.

Using Handler Within STORED PROCEDURE Body –

11. Create a procedure that would insert a record in the studentMarks table and have IN parameters as studentId, total_marks, and grade. We are also adding an OUT parameter named rowCount which would return the total count of records in the studentMarks table.
Add the EXIT Error Handler for Duplicate Key record i.e. if someone invokes it for inserting a record with an existing studentID, then the Error handler would be invoked and will return an appropriate error.

Solution -

```
DELIMITER $$
CREATE PROCEDURE stored_proc.spInsertStudentData(IN studentId INT,
IN total_marks INT,
IN grade VARCHAR(20),
OUT rowCount INT)
BEGIN

-- error Handler declaration for duplicate key
DECLARE EXIT HANDLER FOR 1062
    BEGIN
    SELECT 'DUPLICATE KEY ERROR' AS errorMessage;
    END;

-- main procedure statements
INSERT INTO studentMarks(stud_id, total_marks, grade)
VALUES(studentId,total_marks,grade);
    SELECT COUNT(*) FROM studentMarks INTO rowCount;
END$$
```

```
DELIMITER ;

CALL stored_proc.spInsertStudentData(1,450,'A+',@rowCount);
```

The output would display an error message defined in the error handler.

Since it's an exit handler, the main procedure flow would not continue. Hence, in this case the rowCount OUT parameter would not be updated as this statement is after the INSERT statement that had generated the error condition.

If you,Retrieve the value of rowCount, you would get NULL

```
SELECT @rowCount;
```

12. Create a procedure that would insert a record in the studentMarks table and have IN parameters as studentId, total_marks, and grade. We are also adding an OUT parameter named rowCount which would return the total count of records in the studentMarks table.
Add the CONTINUE Error Handler for Duplicate Key record i.e. if someone invokes it for inserting a record with an existing studentID, then the Error handler would be invoked and will return an appropriate error but will continue the execution after handling the error code.

(Query same as 11 but with CONTINUE handler)

Solution –

DROP this procedure, and re-create with CONTINUE action instead of EXIT for the error handler.

```
DROP PROCEDURE stored_proc.spInsertStudentData

DELIMITER $$
CREATE PROCEDURE stored_proc.spInsertStudentData(IN studentId INT,
IN total_marks INT,
IN grade VARCHAR(20),
OUT rowCount INT)
BEGIN
    DECLARE CONTINUE HANDLER FOR 1062
    BEGIN
        SELECT 'DUPLICATE KEY ERROR' AS errorMessage;
    END;
    INSERT INTO studentMarks(stud_id, total_marks, grade)
VALUES(studentId,total_marks,grade);
    SELECT COUNT(*) FROM studentMarks INTO rowCount;
END$$
DELIMITER ;
```

Call this procedure with an existing student ID.

```
CALL stored_proc.spInsertStudentData(1,450,'A+',@rowCount);
```

The same error appears, but the rowCount would be updated this time, as we have used CONTINUE action instead of EXIT.

Fetch the value of the rowCount OUT parameter.

```
SELECT @rowCount;
```

Lab Program 8 - Triggers

Triggers -

Specify a trigger to get executed when an INSERT, UPDATE or DELETE Operation happens in a table.

SYNTAX -

```
CREATE TRIGGER

trigger_name

trigger_time

trigger_event

ON table_name FOR EACH ROW

[trigger_order]

body
```

trigger_name: A unique name for the trigger object within a table
trigger_time: Allowed values are BEFORE & AFTER – This field indicates whether trigger would be activated BEFORE or AFTER the event.
trigger_event: Is the actual event or action that would lead to invoking the trigger.
The allowed values for this field are:

- **INSERT:** For every row insert.
- **UPDATE**: For every row modification i.e. row updates using **UPDATE statements**.
- **DELETE**: When a row is deleted from the table.

table_name: The name of the MySQL table for which the trigger is being defined.
trigger_order: This is an optional field and is used to define the order in which the trigger would be executed. This is generally used when there are multiple triggers associated with the same events – In that case, we can define the execution order.

It uses the below syntax:
{FOLLOWS | PRECEDES} other_trigger
Note: If the trigger_body is going to contain multiple statements, we can use the syntax similar to how we use it for **creating a STORED PROCEDURE**.
The trigger body, in that case, would be enclosed between BEGIN and END commands and the entire Trigger definition would be between the DELIMITER commands to have multiple statements specified with BEGIN...END blocks.

```
DELIMITER $$

// Trigger Syntax

BEGIN

    -- Statements
```

```
END $$
DELIMITER;
```

Create table with below fields and data type -

Table name – item_inventory
Attributes – name – VARCHAR(30), price – DECIMAL(5,2), quantity INTEGER

```
CREATE TABLE item_inventory (id SMALLINT(5) NOT NULL AUTO_INCREMENT
PRIMARY KEY,
name VARCHAR(30), price DECIMAL(5,2), quantity INTEGER);
```

CREATE TRIGGERS –

i) INSERT TRIGGER –

We can create triggers for row INSERTS in an existing table. There can be 2 types of such triggers namely AFTER INSERT and BEFORE INSERT.

a) AFTER INSERT TRIGGER

We will create an AFTER INSERT trigger on the table mentioned in test data – item_inventory.

Create an inventory audit table first –

```
CREATE TABLE item_inventory_audit (id SMALLINT(5) NOT NULL
AUTO_INCREMENT PRIMARY KEY,
item_id VARCHAR(30), price DECIMAL(5,2), created_on DATETIME(6),
quantity INTEGER);
```

We will create a TRIGGER to add an entry in the item_inventory_audit table after every INSERT in the item_inventory table.

```
CREATE TRIGGER inventory.trigger_item_insert_audit

AFTER INSERT

ON inventory.item_inventory

FOR EACH ROW

INSERT INTO
inventory.item_inventory_audit(id,price,created_on,quantity)
VALUES(new.id, new.price, now(), new.quantity);
```

In the above TRIGGER definition

- We've created a trigger named – *trigger_item_insert_audit in database – inventory.*
- The trigger is created on the table – item_inventory FOR EACH INSERT.
- The trigger body – contains an INSERT statement which would mean that after INSERT of the row in item_inventory table, a new row would be inserted in item_inventory_audit table.

Now INSERT a record in the item_inventory table.

```
INSERT INTO inventory.item_inventory(name, price,
quantity) VALUES("Henko Washing Powder",150.50,100);
```

Now check if the corresponding record in the item_inventory_audit table has been created or not.

```
SELECT * FROM inventory.item_inventory_audit;
```

```
mysql> CREATE TRIGGER inventory.trigger_item_insert_audit
    -> AFTER INSERT
    -> ON inventory.item_inventory
    -> FOR EACH ROW
    -> INSERT INTO inventory.item_inventory_audit(id,price,created_on,quantity) VALUES(new.id, new.price, now(), new.qua
ntity);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO inventory.item_inventory(name, price, quantity) VALUES("Henko Washing Powder",150.50,100);
Query OK, 1 row affected (0.03 sec)

mysql> SELECT * FROM inventory.item_inventory_audit;
+----+---------+--------+----------------------------+----------+
| id | item_id | price  | created_on                 | quantity |
+----+---------+--------+----------------------------+----------+
|  1 | NULL    | 150.50 | 2022-09-13 22:40:08.000000 |      100 |
+----+---------+--------+----------------------------+----------+
1 row in set (0.00 sec)

mysql>
```

b) BEFORE INSERT TRIGGER

The Before INSERT triggers would be called BEFORE every row INSERT on the target table.
The syntax for Trigger creation remains the same, as we saw in the previous section.

BEFORE INSERT can be used when you want to update the values being inserted to some other value or default etc.

```
DELIMITER $$
CREATE TRIGGER inventory.trigger_item_before_insert
BEFORE INSERT
ON inventory.item_inventory
FOR EACH ROW
BEGIN
IF new.price = 100 THEN SET new.price = 1.02 * new.price;
```

```
END IF;
END $$
```

Suppose we want to have the price in the audit table to be 2% more than the actual price data being entered.

Here, we have created a BEFORE INSERT trigger, which would change the price of the new row being added to 1.02 times the original price.

```
INSERT INTO inventory.item_inventory(name, price, quantity) VALUES("Tomato Ketchup",120,20);
```

```
SELECT * FROM inventory.item_inventory_audit where id=2;
```

```
mysql> select * from inventory.item_inventory;
+----+----------------------+--------+----------+
| id | name                 | price  | quantity |
+----+----------------------+--------+----------+
|  1 | Henko Washing Powder | 150.50 |      100 |
|  2 | Tomato Ketchup       | 120.00 |       20 |
+----+----------------------+--------+----------+
2 rows in set (0.01 sec)

mysql> select * from inventory.item_inventory where id=2;
+----+---------------+--------+----------+
| id | name          | price  | quantity |
+----+---------------+--------+----------+
|  2 | Tomato Ketchup | 120.00 |       20 |
+----+---------------+--------+----------+
1 row in set (0.00 sec)

mysql> SELECT * FROM inventory.item_inventory_audit where id=2;
+----+---------+--------+----------------------------+----------+
| id | item_id | price  | created_on                 | quantity |
+----+---------+--------+----------------------------+----------+
|  2 | NULL    | 120.00 | 2022-09-13 23:00:04.000000 |       20 |
+----+---------+--------+----------------------------+----------+
1 row in set (0.00 sec)

mysql>
```

ii) UPDATE TRIGGER

We can create triggers for the row UPDATE of an existing table. Similar to INSERT, for UPDATE as well, we can have 2 types of triggers i.e. BEFORE UPDATE and AFTER UPDATE. Let's see examples for both such cases.
In the UPDATE trigger – we can retrieve both NEW and OLD values for the item being updated using ***new*** and ***old*** keywords respectively.
Let's create a BEFORE UPDATE Trigger which would set the quantity to be updated in the NEW row to be the difference between the OLD and NEW

value. **Example,** if someone is updating the quantity column in the existing table, then

- If new.quantity < old.quantity => set new.quantity = old.quantity – new.quantity
- Else new.quantity = 0

```
DELIMITER $$
CREATE TRIGGER inventory.trigger_item_before_update
BEFORE UPDATE
ON inventory.item_inventory
FOR EACH ROW
BEGIN
    IF old.quantity > new.quantity THEN SET new.quantity = old.quantity - new.quantity;
    ELSE SET new.quantity = 0;
    END IF;
END $$
```

Let's also create an AFTER UPDATE Trigger which would INSERT a new row in the audit table with the new quantity column value calculated in the BEFORE UPDATE TRIGGER as shown above.

```
DELIMITER $$
CREATE TRIGGER inventory.trigger_item_after_update
AFTER UPDATE
ON inventory.item_inventory
FOR EACH ROW
BEGIN
    INSERT INTO
inventory.item_inventory_audit(item_id,price,created_on,quantity)
VALUES(old.id, old.price, now(), new.quantity);
END $$
```

Suppose we did an initial INSERT in the item_inventory table using the below query:

```
INSERT INTO inventory.item_inventory(name, price, quantity)
VALUES("Cookies",90,60);
```

This would have triggered the BEFORE INSERT and AFTER INSERT Triggers respectively.

Now let's run an UPDATE for this Item (which bears an item ID – 3)

```
UPDATE inventory.item_inventory SET quantity=20 WHERE id=3;
```

Now the sequence of trigger execution for UPDATE would be

=> UPDATE STATEMENT => BEFORE UPDATE TRIGGER => AFTER UPDATE TRIGGER

The Logic in BEFORE UPDATE TRIGGER – would update the quantity of new row to the difference of the new and original values

Since we are updating the quantity to 20 and the original value was 60, the actual value that would get inserted into the table would be 40.

Let's do a SELECT on the inventory_audit table to verify the results.

```
SELECT * FROM inventory.item_inventory_audit WHERE id=3;
```

```
mysql> use inventory
Database changed
mysql> INSERT INTO inventory.item_inventory(name, price, quantity) VALUES("Cookies",90,60);
Query OK, 1 row affected (0.02 sec)

mysql> UPDATE inventory.item_inventory SET quantity=20 WHERE id=2;
Query OK, 1 row affected (0.02 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from item_inventory;
+----+----------------------+--------+----------+
| id | name                 | price  | quantity |
+----+----------------------+--------+----------+
|  1 | Henko Washing Powder | 150.50 |      100 |
|  2 | Tomato Ketchup       | 120.00 |        0 |
|  3 | Cookies              |  90.00 |       60 |
+----+----------------------+--------+----------+
3 rows in set (0.00 sec)

mysql> UPDATE inventory.item_inventory SET quantity=20 WHERE id=3;
Query OK, 1 row affected (0.02 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from item_inventory;
+----+----------------------+--------+----------+
| id | name                 | price  | quantity |
+----+----------------------+--------+----------+
|  1 | Henko Washing Powder | 150.50 |      100 |
|  2 | Tomato Ketchup       | 120.00 |        0 |
|  3 | Cookies              |  90.00 |       40 |
+----+----------------------+--------+----------+
3 rows in set (0.00 sec)

mysql> select item_inventory_audit where id=3;
ERROR 1054 (42S22): Unknown column 'item_inventory_audit' in 'field list'
mysql> select * from item_inventory_audit where id=3;
+----+---------+-------+----------------------------+----------+
| id | item_id | price | created_on                 | quantity |
+----+---------+-------+----------------------------+----------+
|  3 | NULL    | 90.00 | 2022-09-13 23:23:40.000000 |       60 |
+----+---------+-------+----------------------------+----------+
1 row in set (0.00 sec)

mysql>
```

The first row you see above is added by the INSERT Trigger.

The second row is added as part of UPDATE

1. UPDATED quantity = 20
2. BEFORE UPDATE TRIGGER => changes new.quantity = 60 – 20 => 40
3. AFTER UPDATE TRIGGER => Inserts a row in audit_table with updated quantity – i.e 40

iii) DELETE TRIGGER

To delete or drop MySQL TRIGGER, we can use the DROP TRIGGER command.

Syntax –

```
DROP TRIGGER [if exists] schema.trigger_name;
```

If we want to remove the BEFORE INSERT trigger we created in previously, the trigger was named as *inventory.trigger_item_before_insert*
we use the DROP command to delete this:

```
DROP TRIGGER inventory.trigger_item_before_insert;
```

Once the trigger is deleted, any actions associated with the trigger would be void and no such events would be raised.

iv) List triggers in a Database -

List all the triggers in the inventory schema, then we can use the below command.

```
SHOW TRIGGERS FROM inventory;
```

Possible Viva Questions on Triggers –

Q #1) What are Triggers in MySQL?
Answer: Simply stated, A trigger in MySQL is a kind of action in response to any defined event.
For example, Suppose we want to perform some action (in the form of executing some statements/insert data in another table) on INSERTING or UPDATING a row in some other table, we can use TRIGGERS for that purpose.

Q #2) How are Triggers implemented in MySQL?
Answers: Triggers are named objects within MySQL DATABASE i.e. Trigger is associated with a table in MySQL database.
The triggers can be created using CREATE TRIGGER command and can be one of 2 types:

- **Row-level:** These are called for each row. **For example,** on every INSERT of the row in the table.
- **Statement level:** These are called once during the statement execution. The statement might impact one or more rows but the associated trigger would be called just once.

Q #3) Can there be Multiple Triggers associated with the same event?
Answer: Yes, it's perfectly fine to have multiple triggers associated with the same event. **For example,** we can define multiple triggers to get executed when a row is INSERTED into a table.

The order of the trigger execution is by default in the order in which Triggers were created. This sequence can also be changed, using the **trigger_order** field while creating the trigger.

Q #4) Is it possible to update an existing trigger?

Answer: In MySQL, there's no statement to update or alter an existing trigger. If after trigger creation, you need to change the trigger. You can DROP the existing trigger using the DROP TRIGGER command and create a new one with the same name.

Q #5) Applications of Using Triggers –

There are numerous real-life applications of triggers -

- **Checking for data Integrity / Error handling:** This is done especially by using BEFORE triggers – suppose you want to check whether the inserted column value follows a proper syntax or it's within a certain range etc.
- **Creating change log entries**: The triggers are widely used to create audit trails or changelogs of activity in a database or table. **For example,** if you want to track all updates to a table, then we can create an UPDATE trigger on that table which would create another entry in an audit table.
- **Derive / Update additional data field**: At times it might be required to update one or more columns in response to the original UPDATE or INSERT. Triggers can be useful in those situations.
- **Replicate data across different tables**: At times, we might want to insert / update data to other tables as a result of a data insertion or update on another table. In such cases, we can define triggers to achieve the desired table data updates.