**MODULE: 5**

**(Database)**

**1) What do you understand By Database?**

* Database is a way to organized collection of data.
* In the Database data is stored in such a manner, so it can easily be access, manage and update.
* In Database data stores systematically into tabular form.
* In this data can easily be access through SQL.
* Data can be insert, delete or update.
* It is easy to secure data into database.
* It takes less space in the memory to store data.
* Data can easily be import & export from Database table.
* It provides facility to backup or retrieve the data.

**There are some database types:**

* Relational Database
* Distributed Database
* Object Oriented Database
* No SQL Database
* Graph Database

**2) What is Normalization?**

Normalization is a process in which database table is designed to minimize the data redundancy for duplicates data and also minimize the inconsistency for the data that has duplicates entries with same or different details.

* Its main goal is to organize the data efficiency into database.
* In the normalization process storage space used as minimum.
* In this data manipulation (update and changes) and retrieval (search & queries fast) is easy.

**3) What is Difference between DBMS and RDBMS?**

|  |  |
| --- | --- |
| DBMS | RDBMS |
| DBMS refers to Database Management System | **RDBMS** refers to **Relational Database Management System** |
| It is a software which is used to define, create & maintain a database & provides controlled access to the data. | It is advance version or extension of DBMS. |
| It stores data as file. | RDBMS stores data into tabular form. |
| Data is generally stored in hierarchical form. | It uses a table structure, where row and column are used to contain corresponding values. |
|  | |  |  |  | | --- | --- | --- | | **ID** | **Name** | **Course** | | 01 | Jerry | Python | | 02 | Jack | Java | | 03 | Tom | PHP | |
| DBMS doesn’t have any normalization process. | RDBMS uses normalization process. |
| There is no relationship between data. | Data stores into tabular form, so it is related to each other. |
| It doesn’t provide any security for data manipulation. | RDBMS uses multilevel of data security. |
| Data redundancy is common in this model. | In this key & indexes doesn’t allow data redundancy. |
| It is used for small organization to deals with small size of data. | It is used to handle large amount of data. |
| It supports single user. | It supports multiple user. |
| Data fetches slowly for large amount of data | Data fetch faster because of relation data approach. |
| Ex. XML, Microsoft access, etc. | **Ex.** MySQL, SQL server, oracle, etc. |

**4) What is MF Cod Rule of RDBMS Systems?**

**MF Cod Rule** is defined by Edges f. Codd in 1970. Also known as **Codd’s 12rules**. It is basic conditions and rules of RDBMS. The aim of cod rule is to define a database system which completely follow the principles of **RDBMS.** This rule ensure that any database system works in a seriously relational manner or not.

**The 12 rules are here:**

* **Data Representation:** All the information of data representation should be relational or into tabular form. (because row and columns always connected with each other into tabular form)
* **Guaranteed Access:** Any of data can easily be find out by their table name, column or primary key.
* **Systematic Null Handling:** Null value supports to handle any of unknown or missing data.
* **Dynamic Online Catalog:** There should be data relation or table structure in the system so that it is easily accessible.
* **Comprehensive Data Sub Language:** Should be proper language support like SQL, which supports into definition, manipulation and access the data.
* **View Updating Rule:** If table is created logically, can be modify.
* **High level Insert, update, delete:** Apart from single row, multiple rows can also be insert, delete or update at a time.
* **Physical Data Independence:** After changing the physical method of data storage, queries should remain the same.
* **Logical Data Independence:** Existing queries should not be affected into changing data structure.
* **Integrity Independence:** Primary foreign key should be defined logically into database not at application level.
* **Distribution Independence:** Query results should not be affected wherever the data is physically stored.
* **Non-Subversion Rule:** Database key or rules should not be broken whenever any low-level operation is running into system. It is mandatory to follow rules into each steps/stage.

**5) What do you understand By Data Redundancy?**

Data redundancy refers to storing data in more than one place. In data redundancy it is possible that data may be redundant by known or unknown mistake.

**Types of Data Redundancy**

* **Intention Redundancy:** It means data may have been redundant by known mistake**.** Sometimes data get lost due to server failure and it is difficult to back up again. So that data can be stored into multiple places. Storing the data at multiple places helps into data redundancy through backup whenever data get lost.
* **Unintentional Redundancy:** It means data may have been redundant by unknown mistake. It means sometimes same types of data stored with multiple version into more than one place of system server. It may be the reason of data inconsistency (same types of data stored into multiple version).

**Benefits of Data Redundancy**

* If data is lost, another copy of that data can be used.
* To easily access the data according to requirements, copy of data can be stored anywhere.
* In case of system crash, data can be backed up through data redundancy.

**Disadvantages of Data Redundancy**

* On the availability of more than one data copies, it should be updated everywhere. If not updated, can be occurs mismatched version of data.
* Due to data redundancy, more storage space will be required.
* It consumes time or may be difficult to manage multiple copies of data.

**6) What is DDL Interpreter?**

**DDL** refers to **Data Definition Language.**

* DDL interpreter is a software, which processes DDL commands.
* It is a type of SQL subset, which is used to define & manage database structure.
* It’s primary works to create, modify and delete the objects of database. These objects are such as table, schemas, index, view etc.

**DDL commands works for**

* **CREATE:** To create new database table

Syntax:

**CREATE TABLE** table\_name

**(**

Column\_name DATATYPE (LENGTH),

Column\_name DATATYPE ………,

Column\_name DATATYPE ………

**);**

Ex:

**CREATE TABLE** employees

**(**

ID int,

NAME varchar (50),

AGE int

**);**

* **ALTER:** To modify existing database table.

**(**used to add new column and delete or modify existing column**)**

Syntax: (add new column)

**ALTER TABLE** table\_name **ADD** column\_name **DATATYPE (**values**);**

Ex:

**ALTER TABLE** employees **ADD** salary **DECIMAL (**10, 2**);**

Syntax: (change into existing column datatype)

**ALTER TABLE** table\_name **MODIFY** column\_name **DATATYPE (**values**);**

Ex:

**ALTER TABLE** employees **MODIFY** Name **VARCHAR (**100**);**

* **DROP:** To delete database objects such as table, view, index.

Syntax:

**DROP TABLE** table\_name**;**

Ex:

**DROP TABLE** employees;

* **TRUNCATE:** To delete all the records from the existing table but it preserves the table structure.

Syntax:

**TRUNCATE TABLE** table\_name**;**

Ex:

**TRUNCATE TABLE** employees;

**Note:** DDL commands use to define or manage database structure. DDL interpreter reads this command and execute it and also apply required changes into database.

**7) What is DML Compiler in SQL?**

**DML** refers to **DATA MANIPULATION LANGUAGE**

DML compiler is a software component which process to DML command and convert it into executable form of DBMS. DML commands mainly used to manipulate (insert, delete, update) data into database.

**DML commands works for**

* **INSERT:** To add new records as row

Syntax:

**INSERT INTO** table\_name **(fields)**

**VALUES (-----, -----, -----),**

**(-----, -----, -----),**

**(-----, -----, -----),**

**(-----, -----, -----);**

Ex:

**INSERT INTO** employees **(**ID, NAME, AGE**)**

**VALUES (**1, ‘AB’, 30**),**

**(**2, ‘CD’, 35**),**

**(**3, ‘EF’, 40**);**

* **UPDATE:** To modify existing records of database table.

**(**used to add new column and delete or modify existing column**)**

Syntax: (add new column)

**UPDATE** table\_name **SET** field = value **WHERE ID =** row\_number;

Ex:

**UPDATE** employees **SET** Age = 31 **WHERE ID = 1;**

* **DELETE:** To delete the records from the existing table

Syntax:

**DELETE FROM** table\_name **WHERE ID =** row\_number**;**

Ex:

**DELETE FROM** employees **WHERE ID =** 1;

* **SELECT:** To retrieve/show data from the existing table.

1. To show complete table

Syntax: **SELECT \* FROM** table\_name;

Ex: **SELECT \* FROM** employees**;**

1. To show data of any particular row

Syntax: **SELECT \* FROM** table\_name **WHERE ID =** row\_number;

Ex: **SELECT \* FROM** employees **WHERE ID = 2;**

1. To show data of any particular field

Syntax: **SELECT** fields\_name **FROM** table\_name;

Ex: **SELECT** Name, Age **FROM** employees**;**

1. To show data for one and more fields of that row, without viewing the data for an entire row selected in the table.

Syntax: **SELECT** fields\_name **FROM** table\_name **WHERE ID =** row\_number;

Ex: **SELECT** Name, Age **FROM** employees **WHERE ID = 3;**

**Role of DML Compiler:**

* **Parsing:** It means to analyze any query or statement. Parsing process reads DML commands line by line and check the syntax. It ensures that query follows SQL rules or not, if it gets any error, generate it(error) during parsing process and this error can become the reason of not execution.

Parsing is the first process when DML compiler receive any query.

* **Optimization:** DML compiler optimize query, which help to improve performance and fast execution of query.
* **Execution Plan:** Compiler creates an execution plan that defines how data can be fetched or modified.
* **Execution:** DML commands are executed to select, insert, update, or delete data in a database.

**Note: 1)** DDL commands use to define or manage database structure.

**2)** DML commands handle the data manipulation (insert, delete, update, select). The role of the DML compiler is to process SQL DML commands and make them executable for the DBMS, enabling data manipulation within the database.

**8) What is SQL Key Constraints writing an Example of SQL Key Constraints.**

SQL Key Constraints are rules or restrictions applied to the columns of a table. They ensure data integrity or uniqueness.

* These Constraints define that how data is stored and how relationships between tables are established.
* The main function of constraints is to uniquely identify the records in a table.

**Some important key constraints are:**

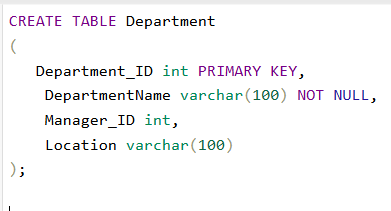
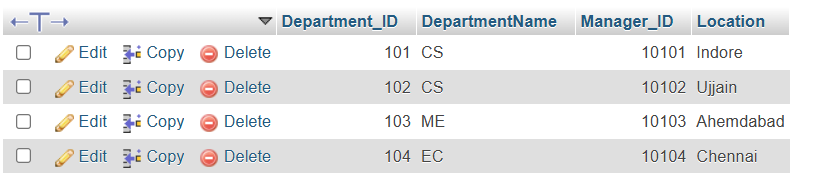
1. **Primary Key:**

* This constraint uniquely identifies each record in a table.
* The primary key of any two different rows cannot be the same.
* A Primary key does not allow null values in the column.

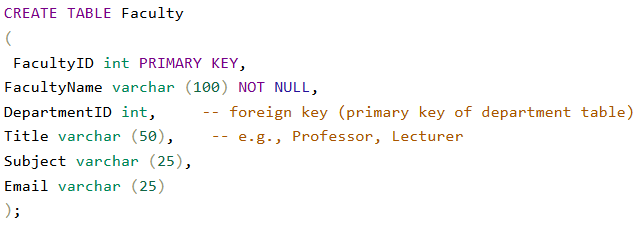
1. **Foreign Key:**

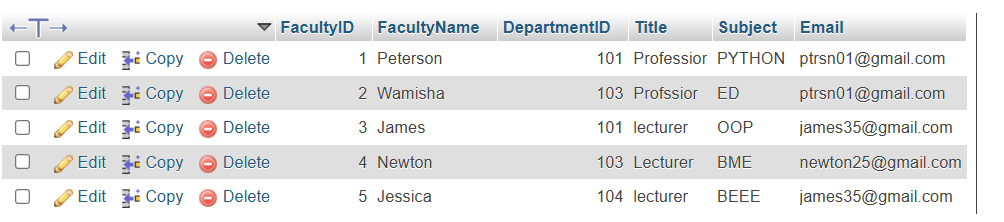
* A Foreign key constraints links one table to another, establishing relationship between both tables.
* foreign key is a reference to the primary key of another table.

**Step 1: created table “department” and Insert data**



**Step 2: created table ‘’faculty” and insert data**



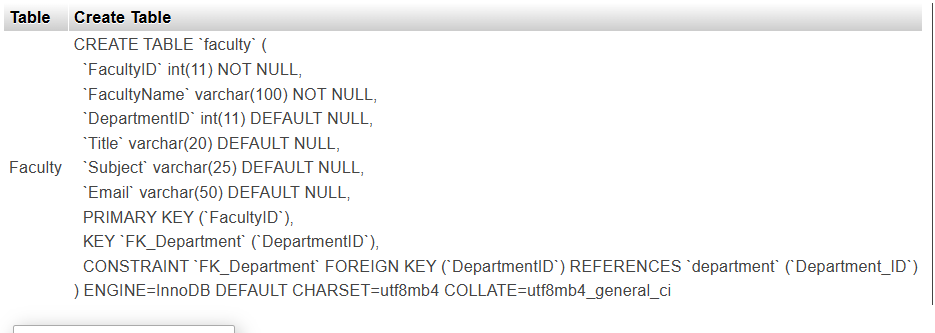
****

**Step 3: created table ‘’faculty” and insert data**

(Alter table to modify existing table to make department ID from department table to the foreign key of faculty table)

**ALTER TABLE** Faculty **--**. To modify existing database table

**ADD CONSTRAINT** FK\_Department

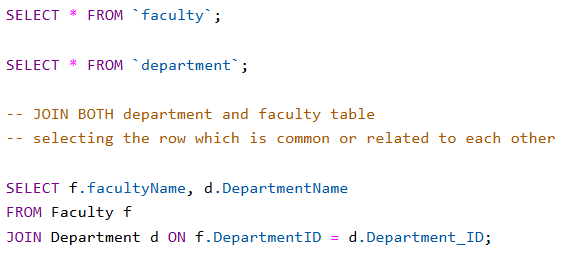
**FOREIGN KEY** (Department\_ID) **REFERENCES** Department (Departm ent\_ID);

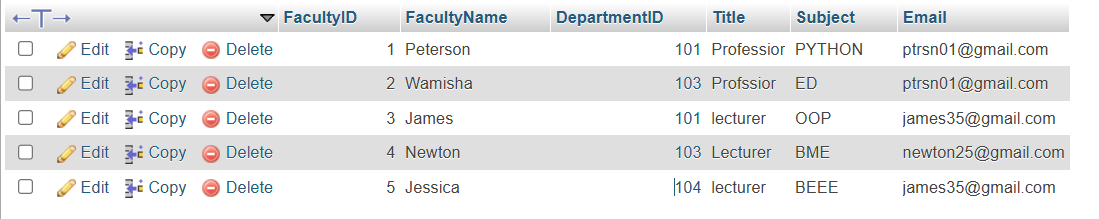
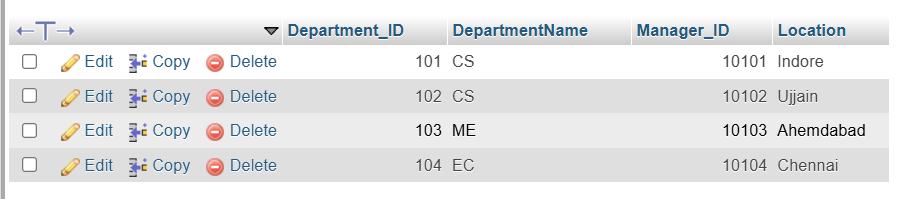
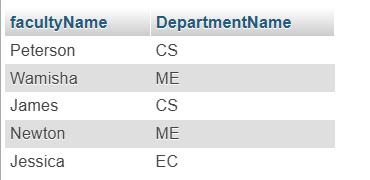
**Step 4: to see foreign key**

(To check that both tables are connected with each other through foreign key or not)

**SHOW CREATE TABLE** Faculty;

**Step 5: Data validation**

****(To verify that foreign key relation is proper working or not)

****

**9) What is save Point? How to create a save Point write a Query?**

**Save point** is a **Transaction Control Command**. It marks a specific point within a transaction, allowing you to roll back to that point if needed. This means you can undo changes made to the database and return it to a previous state.

* The use of a save point is to roll back a transaction to a specific point. Instead of rolling back the entire transaction.
* you can return the transaction only to the specific save point.
* A save point acts as a temporary marker within the transaction.

**Works of save Point**

* **Transaction Control:** In a transaction with multiple steps, you can create save points to ensure that if something goes wrong, you can roll back to a specific point.
* **Rollback Control:** Without cancelling the entire transaction, you can use a savepoint to selectively roll back any part of the transaction.

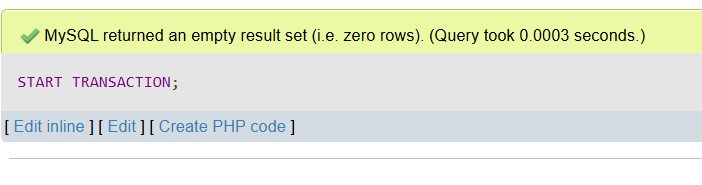
**Syntax:**

**SAVEPOINT savepoint\_name;**

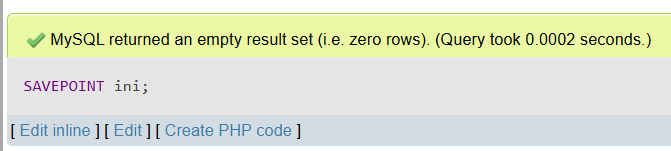
**Keypoints about Save point :-**

* **Partial Rollback:** Can be selectively roll back any part of a transaction using a savepoint.
* **Multiple Save points:** Can be creates multiple save points within a single transaction and roll back to each savepoint as needed.
* **Used with rollback:** To see the full power of save points, can be use "ROLLBACK TO SAVEPOINT." This allows you to return the transaction to that specific savepoint, resetting all subsequent changes.
* Savepoint provide flexibility and control within transaction.

**Ex.**

1. **Start Transaction;**

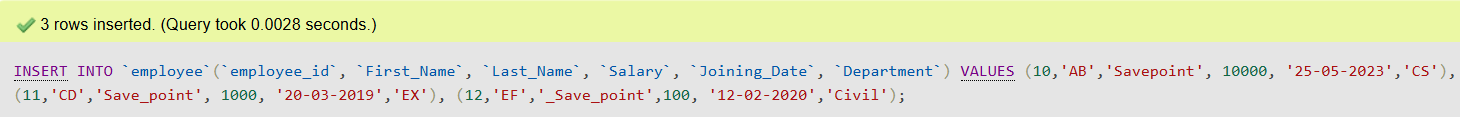
**2) SAVEPOINT ini;**

 (transaction named ini)

**3)**  **INSERT INTO `employee`(`employee\_id`, `First\_Name`, `Last\_Name`, `Salary`, `Joining\_Date`, `Department`)**

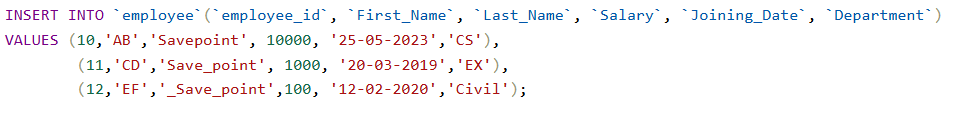
**VALUES (10,'AB','Savepoint', 10000, '25-05-2023','CS'),**

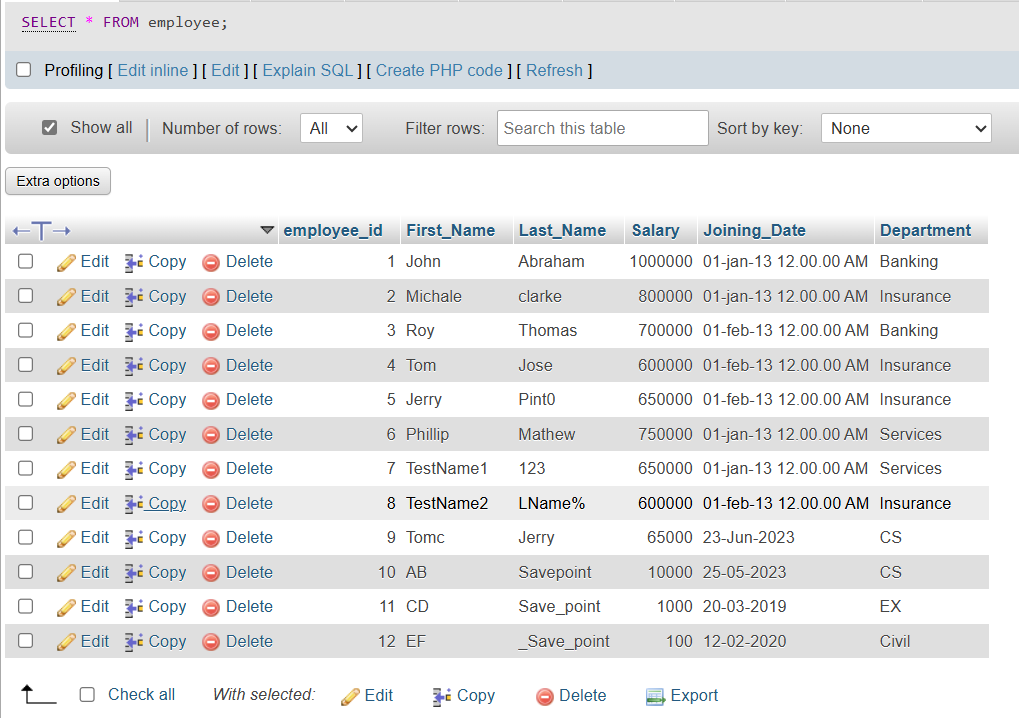
**(11,'CD','Save\_point', 1000, '20-03-2019','EX'),**

** (12,'EF','\_Save\_point',100, '12-02-2020','Civil');**

****

**4) Insert data**

****

****

**EX.**

START TRANSACTION;

**-- 1st INSERT data**

INSERT INTO employee (employee\_id, First\_Name, Last\_Name, Salary, Joining\_Date, Department)

values(11, 'AB', 'CD', 10000, '23-04-2022', 'CS'),

(12, 'PQ', 'RS', 20000, '15-03-2020', 'DS'),

(13, 'IJ', 'KL', 30000, '10-02-2019', 'CIVIL'),

(14, 'PQ', 'OP', 40000, '05-01-2018', 'AIML');

SELECT \* FROM employee,

SAVEPOINT INS;

**-- 2nd UPDATE record**

UPDATE employee SET First\_Name='NM' where employee\_id = 13;

SELECT \* FROM employee,

SAVEPOINT upd;

**-- 3rd DELETE record**

DELETE FROM employee where employee\_id = 11; -- (employee id 11 delete hogi)

SELECT \* FROM employee,

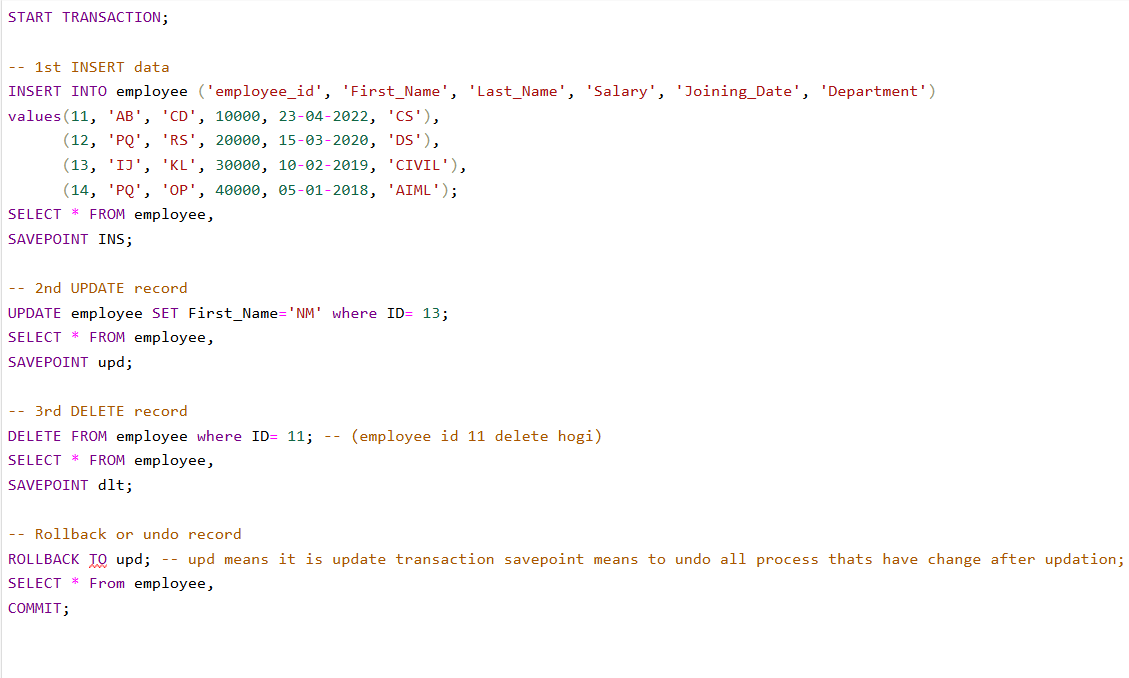
SAVEPOINT dlt;

**-- Rollback or undo record**

-- upd means it is update transaction savepoint means to undo all process thats have change after updation;

ROLLBACK TO upd;

SELECT \* From employee;

****

**10) What is trigger and how to create a Trigger in SQL?**

Stored program which is automatically executed in response to specific events(insert, update, delete) known as trigger.

* This means that when data is inserted, deleted or updated in any table, the trigger runs automatically.
* Triggers are mainly used for data integrity, automatic logging and auditing.
* Whenever data updated in any table, it’s automatically logging into another table(a copy of the table).

**There are two types of trigger:**

1. **Before Trigger:** The process to run actions before any specific event as insert, delete or update actions.
2. **After Trigger:** The process to run actions after successfully complete any specific events as insert, update or delete.

**These two triggers work as**

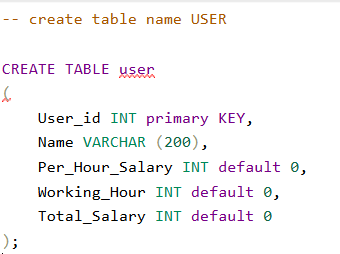
1. Before Insert **b)** After Insert
2. Before Update **d)** After Update
3. Before Delete  **f)** After Delete

|  |
| --- |
| **Before Insert**   **After Insert**  **Insert** |
| **Before Update**   **After Update**  **Update** |
| **Before Delete**   **After Delete**  **Delete** |

**Use Case: -**

**Let’s create a table named user and manage salary by using trigger**

1. **Create a table named “users”**

-- create table name USER

**CREATE TABLE** user

(

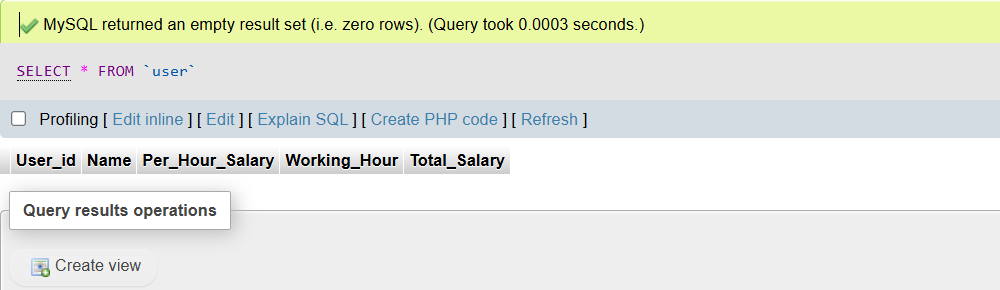
User\_id INT primary KEY,

Name VARCHAR (200),

Per\_Hour\_Salary INT default 0,

Working\_Hour INT default 0,

Total\_Salary INT default 0

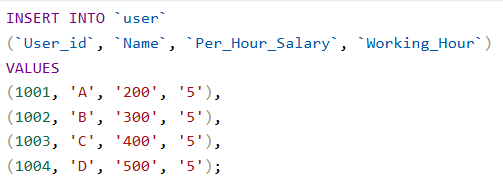
);

1. **Insert data and manage salary**

(Total salary will automatically be calculated through trigger)

INSERT INTO `user`

(`User\_id`, `Name`, `Per\_Hour\_Salary`, `Working\_Hour`)

VALUES

(1001, 'A', '200', '5'),

(1002, 'B', '300', '5'),

(1003, 'C', '400', '5'),

(1004, 'D', '500', '5');

(All details as user’s Total salary will automatically calculate through TRIGGER)

1. **Create trigger (BEFORE INSERT)**

(To create triggers will be take new data with delimiter)

**DELIMITER//**

**CREATE trigger before\_insert\_users -- before\_insert\_user (trigger name)**

**BEFORE INSERT -- event like (before/after insert, delete, update)**

**ON user FOR EACH ROW -- means trigger will automatically run before insert for each row**

**-- trigger body will be written between BEGIN, END**

**BEGIN**

**-- create trigger or write syntax for that value we want it should be calculated automatically**

**SET NEW.Total\_Salary= NEW.Per\_Hour\_Salary \* NEW.Working\_Hour;**

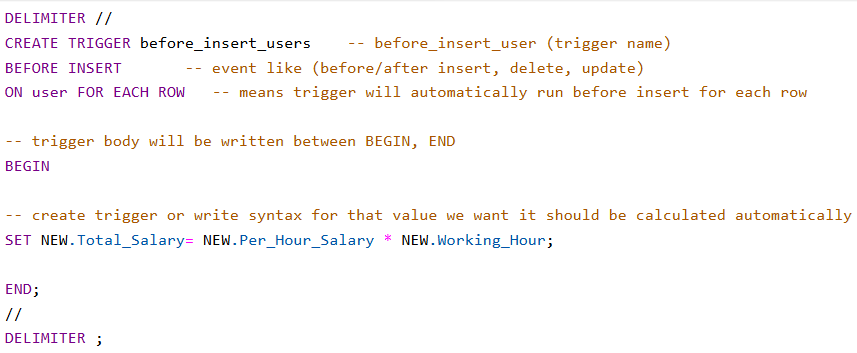
**END;**

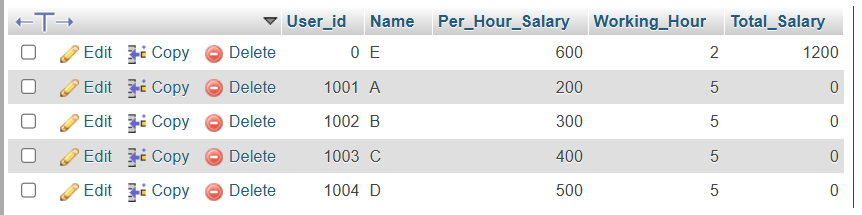
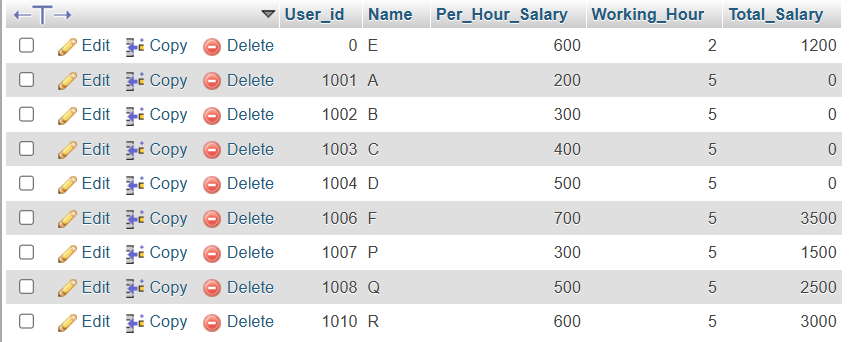
**//**

**DELIMITER ;**

1. **Insert Data** after using **BEFORE INSERT**

(To check that trigger is executing or not will be insert the data into existing table user)

INSERT INTO `user`(`User\_id`, `Name`, `Per\_Hour\_Salary`, `Working\_Hour`)

VALUES ('[1005','E','600','2');

1. **Create trigger (After Update)**

**DELIMITER //**

**CREATE TRIGGER before\_update\_users -- before\_update\_user (trigger name)**

**BEFORE UPDATE -- event like (before/after insert, delete, update)**

**ON user FOR EACH ROW -- means trigger will automatically run before update for each row**

**-- trigger body will be written between BEGIN, END**

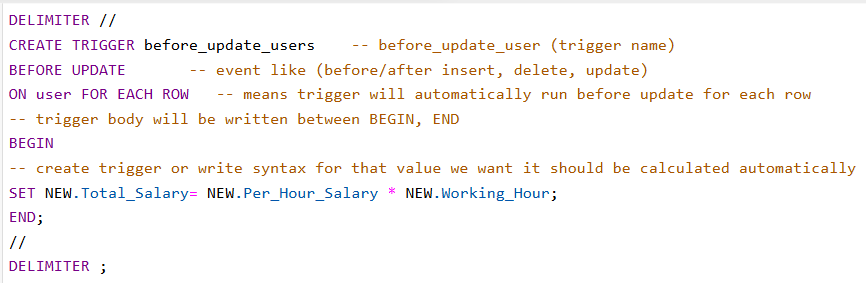
**BEGIN**

**-- create trigger or write syntax for that value we want it should be calculated automatically**

**SET NEW.Total\_Salary= NEW.Per\_Hour\_Salary \* NEW.Working\_Hour;**

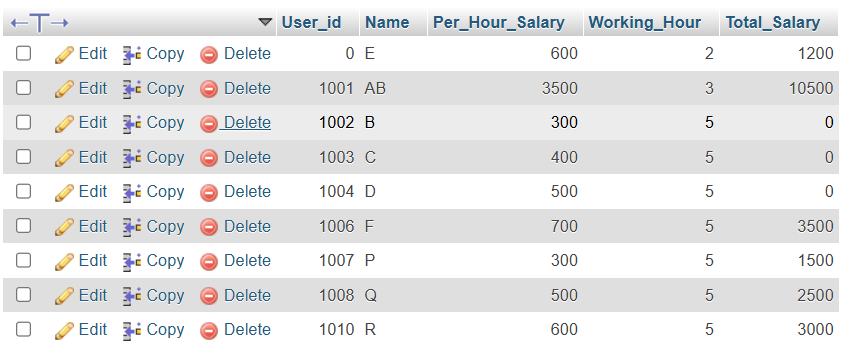
**END;**

**//**

**DELIMITER ;**

1. **Update Data**

(To check that trigger is executing or not will be update the data into existing table user)

UPDATE `user` SET `User\_id`='1001',`Name`='AB',`Per\_Hour\_Salary`='3500',`Working\_Hour`='3' WHERE `User\_id` = 1001;

1. **Create trigger ( BEFORE DELETE)**

**DELIMITER //**

**CREATE TRIGGER before\_delete\_users -- before\_delete\_users (trigger name)**

**BEFORE DELETE -- event like (before/after insert, delete, update)**

**ON user FOR EACH ROW -- means trigger will automatically run before delete for each row**

**-- trigger body will be written between BEGIN, END**

**BEGIN**

**-- create trigger or write syntax for that value we want it should be calculated automatically**

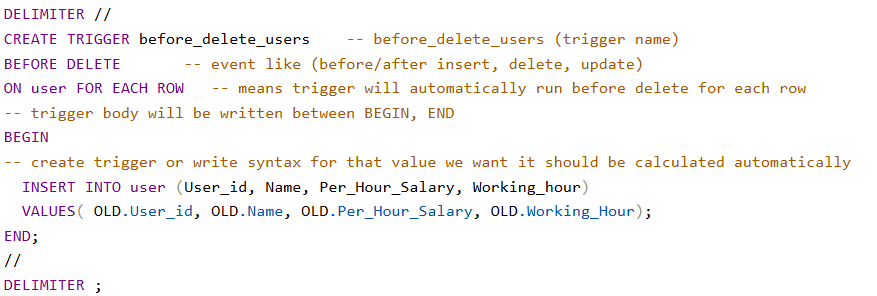
**INSERT INTO user (User\_id, Name, Per\_Hour\_Salary, Working\_hour)**

**VALUES( OLD.User\_id, OLD.Name, OLD.Per\_Hour\_Salary, OLD.Working\_Hour);**

**END;**

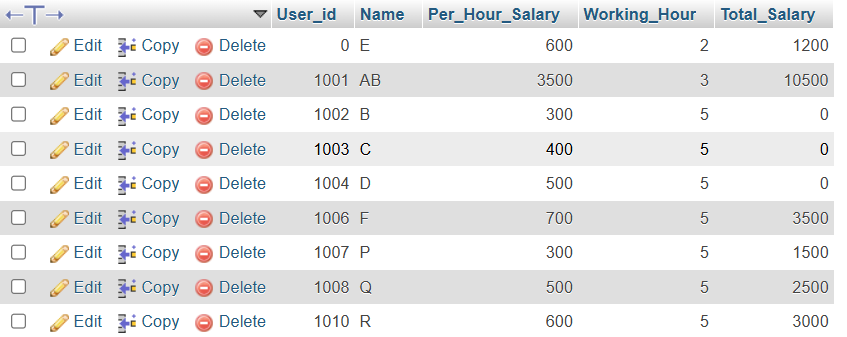
**//**

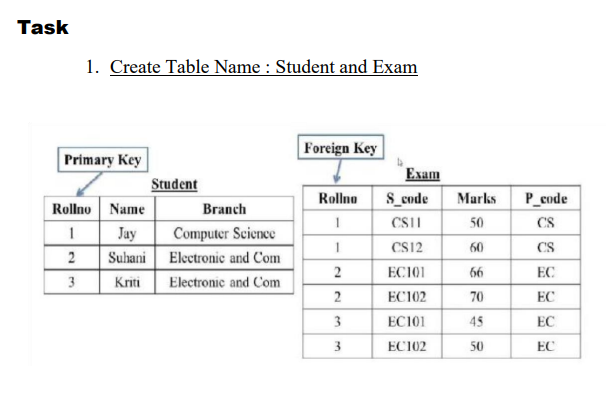
**DELIMITER ;**

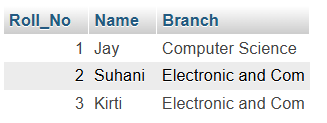
****

1. **DELETE Data**

(To check that trigger is executing or not will be delete the data into existing table user)

DELETE FROM `user` WHERE `User\_id`= 1009;c

****In**SERT INTO `student`(`Roll\_No`, `Name`, `Branch`)**

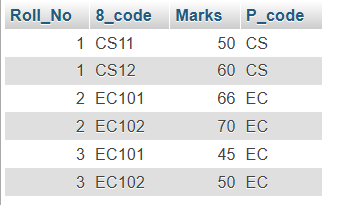
**VALUES**

**(1,'Jay', 'Computer Science'),**

**(2, 'Suhani', 'Electronic and Com'),**

**(3, 'Kirti', 'Electronic and Com');**

**INSERT INTO `exam`(`Roll\_No`, `8\_code`, `Marks`, `P\_code`)**

**VALUES ('1','CS11','50','CS'),**

**('1','CS12','60','CS'),**

**('2','EC101','66','EC'),**

**('2','EC102','70','EC'),**

**('3','EC101','45','EC'),**

**('3','EC102','50','EC');**

1. **Added primary key into student table**

**ALTER TABLE student**

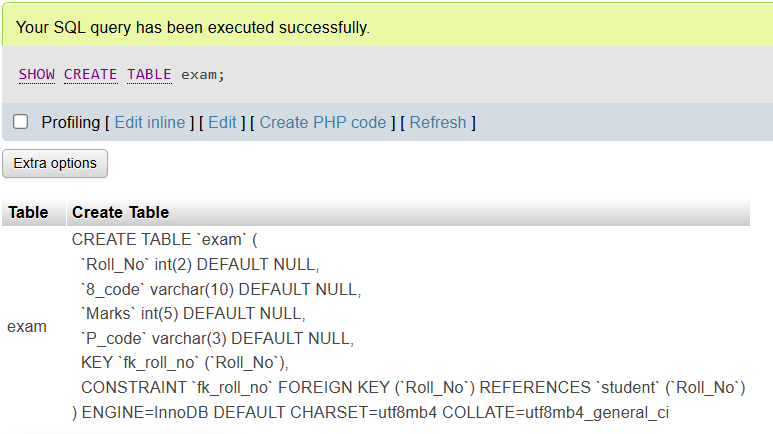
**ADD PRIMARY KEY (Roll\_No);**

1. **connect primary key of student table with foreign key of exam table**

**ALTER TABLE exam**

**ADD CONSTRAINT fk\_roll\_no**

**FOREIGN KEY (roll\_no) REFERENCES student (roll\_no);**

****

**-- after add foreign key both table joined**

**-- show the result from table**

**select \* from exam;**

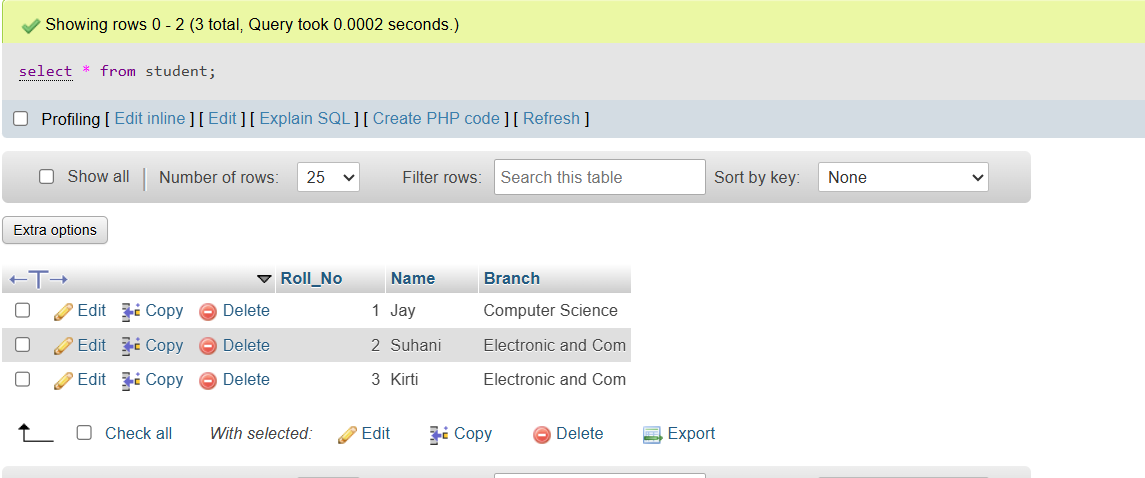
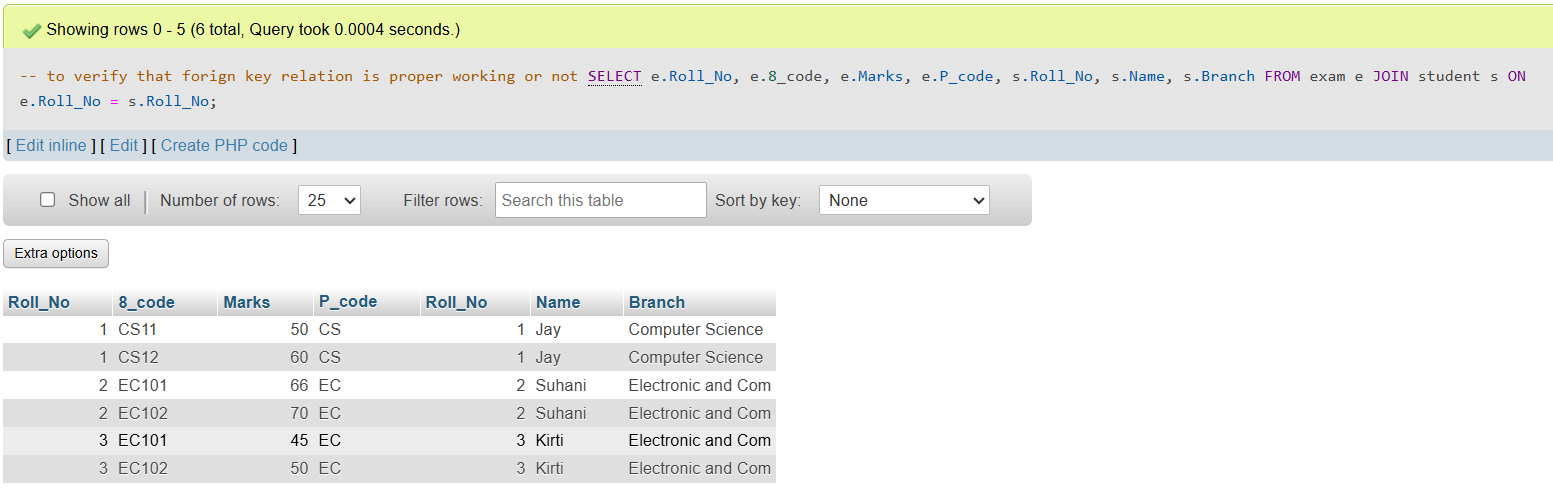
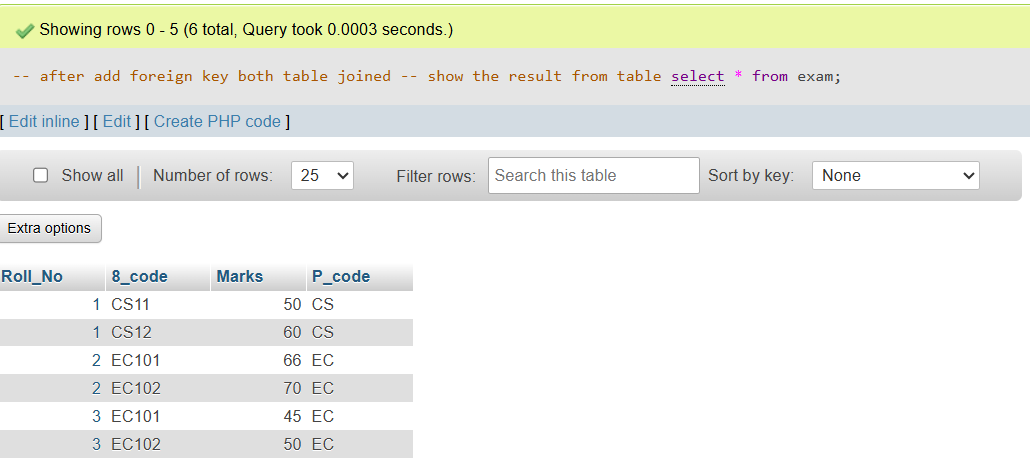
**select \* from student;**

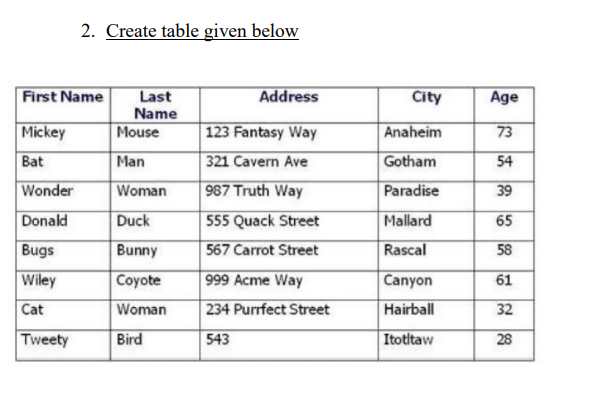
**-- to verify that forign key relation is proper working or not**

**SELECT e.Roll\_No, e.8\_code, e.Marks, e.P\_code, s.Roll\_No, s.Name, s.Branch**

**FROM exam e**

**JOIN student s ON e.Roll\_No = s.Roll\_No;**

** **

****

**INSERT INTO `person\_details`**

**(`First\_Name`, `Last\_Name`, `Address`, `City`, `Age`) VALUES**

**('Mickey','Mouse','123 Fantasy way','Anaheim','73'),**

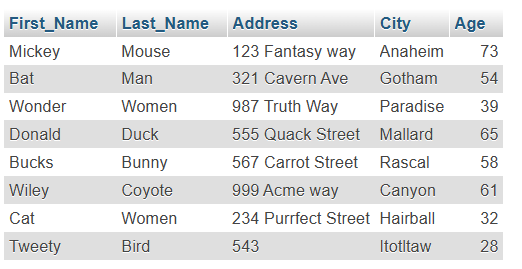
**('Bat','Man','321 Cavern Ave','Gotham','54'),**

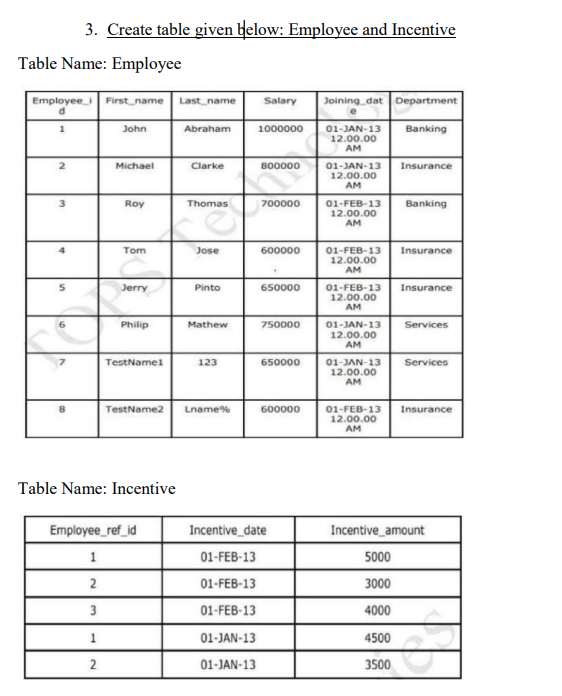
**('Wonder','Women','987 Truth Way','Paradise','39'),**

**('Donald','Duck','555 Quack Street','Mallard','65'),**

**('Bucks','Bunny','567 Carrot Street','Rascal','58'),**

**('Wiley','Coyote','999 Acme way','Canyon','61'),**

**('Cat','Women','234 Purrfect Street','Hairball','32'),('Tweety','Bird','543','Itotltaw','28');**

****

**CREATE TABLE employee(employee\_id int PRIMARY KEY AUTO\_INCREMENT ,**

**First\_Name varchar(40),**

**Last\_Name varchar(40),**

**Salary int,**

**Joining\_Date varchar(40),**

**Department varchar(40);**

**INSERT INTO employee(employee\_id, First\_Name, Last\_Name, Salary, Joining\_Date, Department)**

**VALUES**

**(1,"John","Abraham",1000000,"01-jan-13 12.00.00 AM","Banking"),**

**(2,"Michale","clarke",800000,"01-jan-13 12.00.00 AM”, “Insurance"),**

**(3,"Roy","Thomas",700000,"01-feb-13 12.00.00 AM","Banking"),**

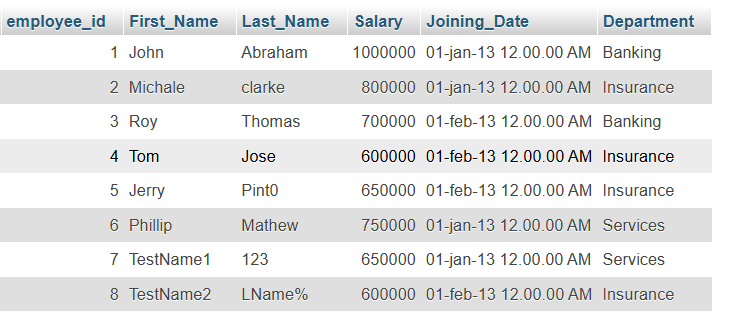
**(4,"Tom","Jose",600000,"01-feb-13 12.00.00 AM”, “Insurance"),**

**(5,"Jerry","Pint0",650000,"01-feb-13 12.00.00 AM”, “Insurance"),**

**(6,"Phillip","Mathew",750000,"01-jan-13 12.00.00 AM”, “Services"),**

**(7,"TestName1","123",650000,"01-jan-13 12.00.00 AM”, “Services"),**

**(8,"TestName2","LName%",600000,"01-feb-13 12.00.00 AM”, “Insurance");**

****

**INSERT INTO `incentive`(`Employee\_ref\_id`, `Incentive\_date`, `Incentive\_amount`)**

**VALUES**

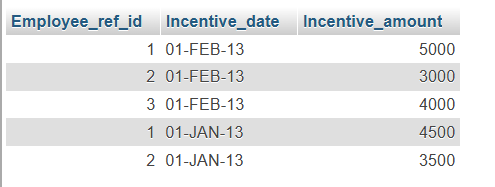
**('1','01-FEB-13','5000'),**

**('2','01-FEB-13','3000'),**

**('3','01-FEB-13','4000'),**

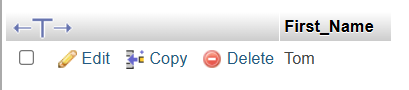
**('1','01-JAN-13','4500'),**

**('2','01-JAN-13','3500');**

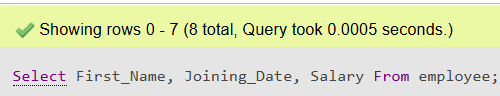
****

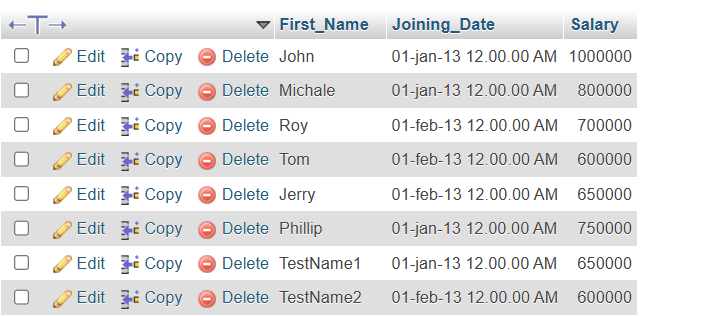
1. **Get First\_Name from employee table using Tom name “Employee Name”.**

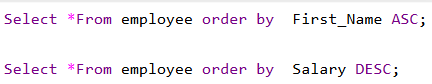
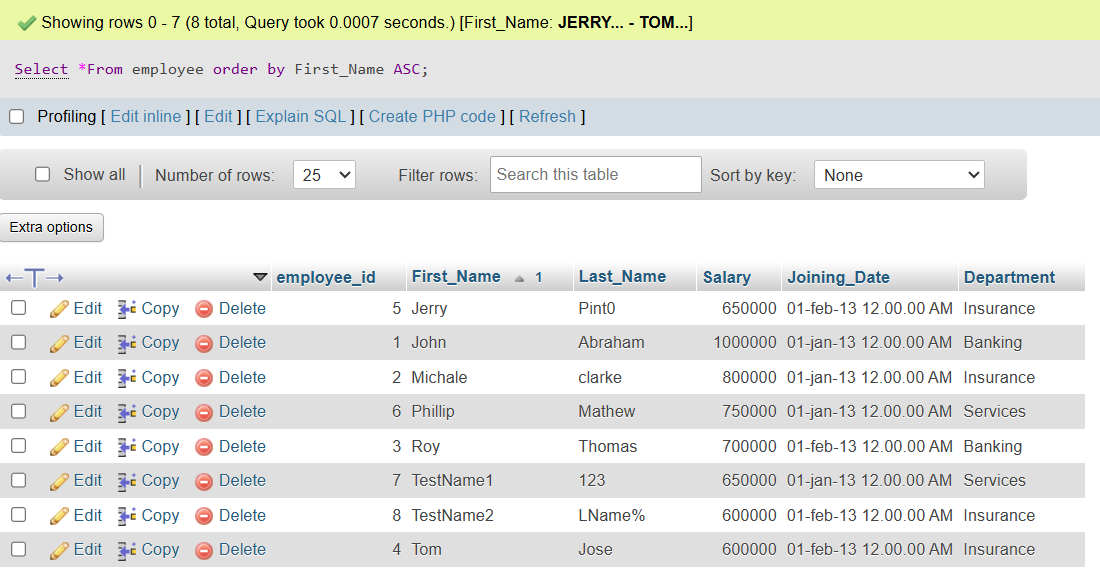
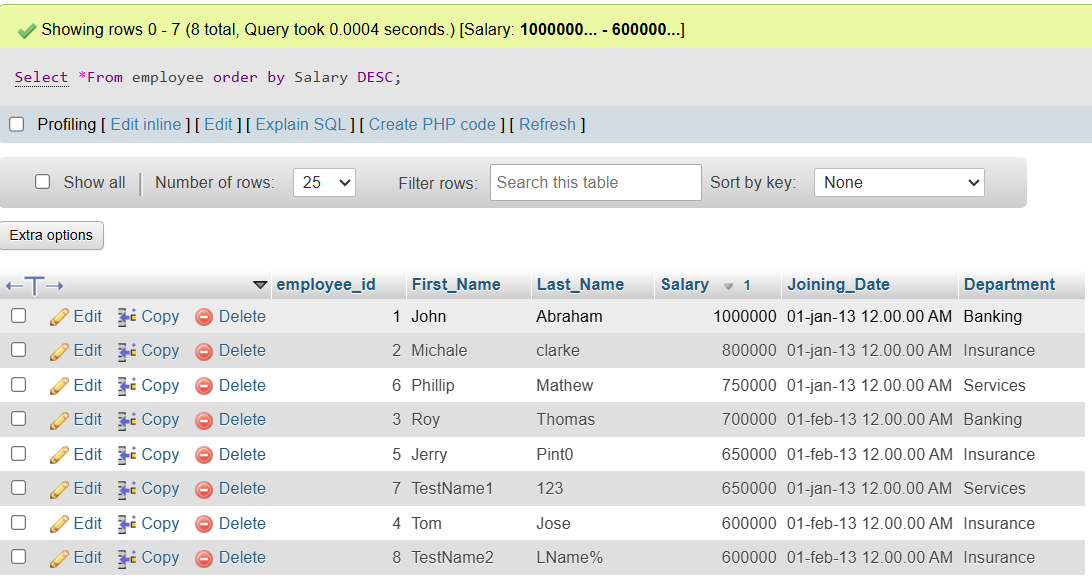
****

****

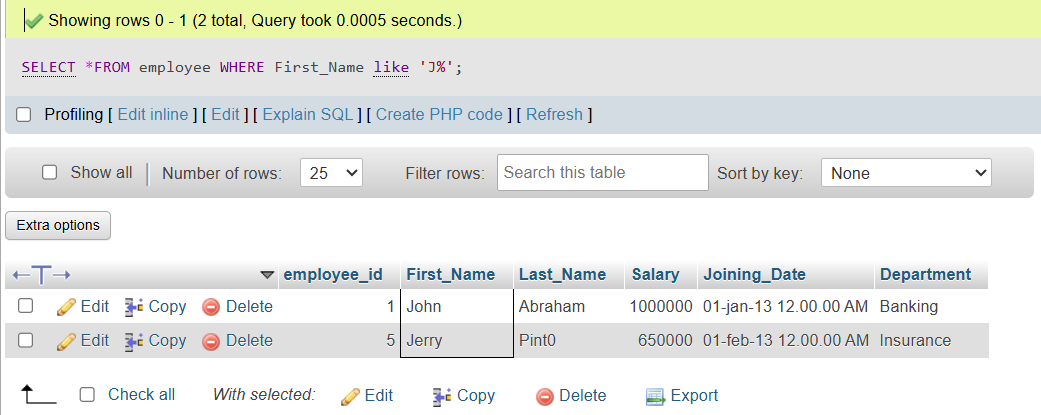
1. **Get FIRST\_NAME, Joining Date, and Salary from employee table.**

****

****

1. **Get all employee details from the employee table order by First\_Name Ascending and Salary descending?**
2. **First\_Name ascending order**
3.  **Salary Descending order**
4. **Get employee details from employee table whose first name contains ‘J’.**

**SELECT \*FROM employee WHERE First\_Name Like ‘J%’;**

****

1. **Get department wise maximum salary from employee table order by salary ascending**

**SELECT employee.employee\_id,**

**employee.Department,**

**employee.First\_Name,**

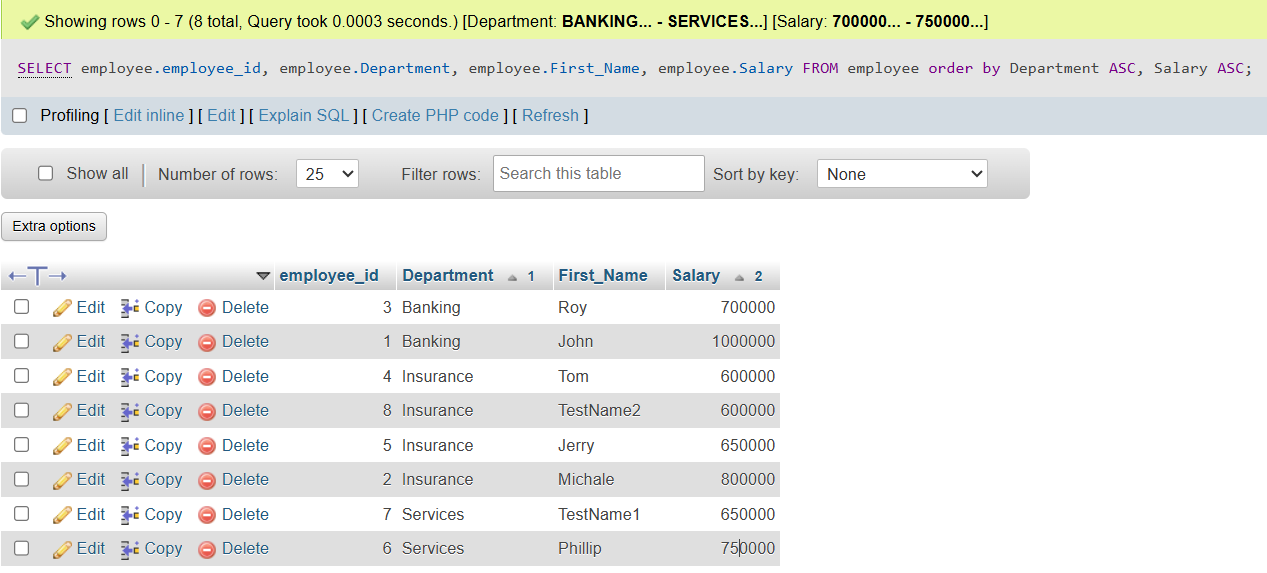
**employee.Salary**

**FROM employee**

**order by**

**Department ASC,**

**Salary ASC;**

****

**f) Select first\_name, incentive amount from employee and incentives table for those employees who have incentives and incentive amount greater than 3000**

-- employee id and first\_name (from employee table) left join into incentive table

SELECT incentive.Employee\_ref\_id,

incentive.Incentive\_date,

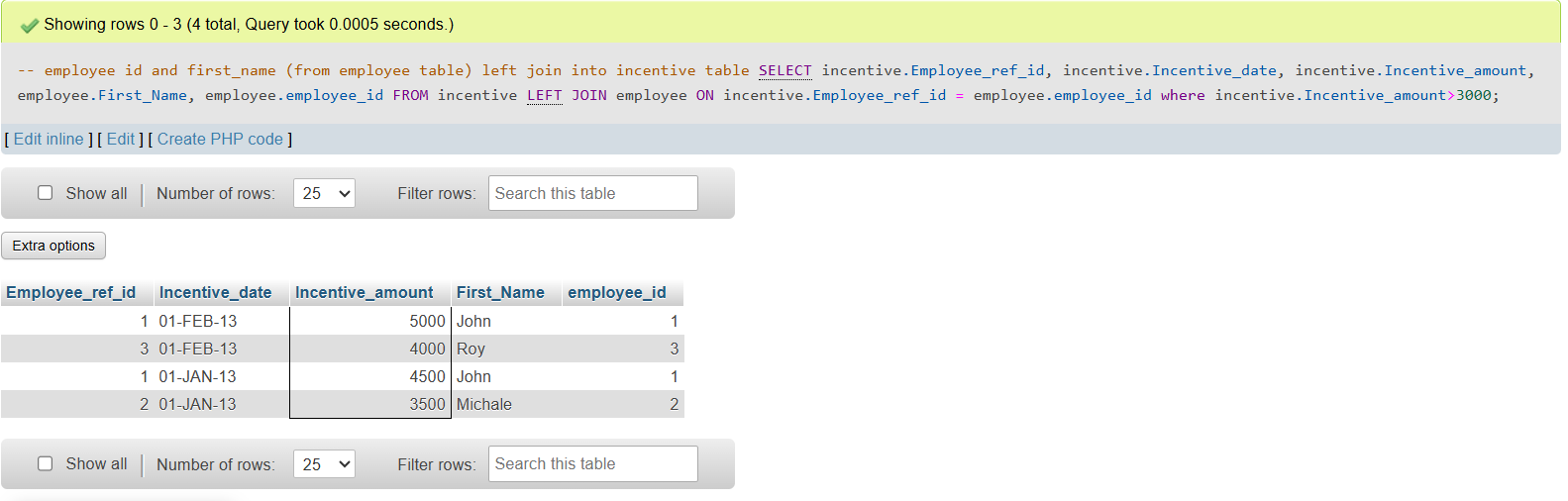
incentive.Incentive\_amount,

employee.First\_Name,

employee.employee\_id

FROM incentive

LEFT JOIN employee ON incentive.Employee\_ref\_id = employee.employee\_id where incentive.Incentive\_amount>3000;



1. **Create After Insert trigger on Employee table which insert records in view table.**

**CREATE TABLE employee\_viewtable(**

**employee\_id int,**

**First\_Name varchar(40),**

**Last\_Name varchar(40),**

**Salary int,**

**Joining\_Date varchar(40),**

**Department varchar(40)**

**);**

**DELIMITER //**

**CREATE TRIGGER after\_employee\_inserts**

**AFTER INSERT ON employee\_viewtable**

**FOR EACH ROW**

**BEGIN**

**INSERT INTO employee\_01trigger (**

**t\_employee\_id, t\_First\_Name, t\_Last\_Name, t\_Salary, t\_Joining\_Date, t\_Department**

**)**

**VALUES (**

**NEW.employee\_id, NEW.First\_Name, NEW.Last\_Name, NEW.Salary, NEW.Joining\_Date, NEW.Department**

**);**

**END //**

**DELIMITER ;**

**Retrieve the below data from above table**

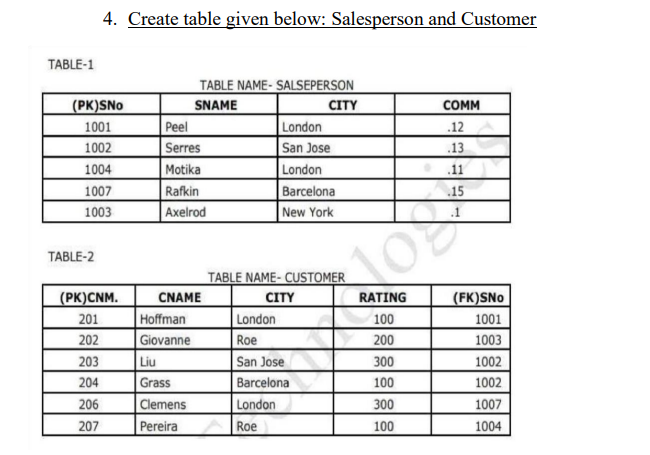
**a) All orders for more than $1000.**

**b) Names and cities of all salespeople in London with commission above 0.12**

**c) All salespeople either in Barcelona or in London**

**d) All salespeople with commission between 0.10 and 0.12. (Boundary values should be excluded).**

**e) All customers excluding those with rating <= 100 unless they are located in Rome**

**** **CREATE TABLE salse\_person**

**(**

**S\_NO int PRIMARY KEY,**

**S\_NAME varchar(25),**

**CITY varchar (25),**

**COMM double**

**);**

**CREATE TABLE customer**

**(**

**CNM int PRIMARY KEY,**

**C\_NAME varchar (25),**

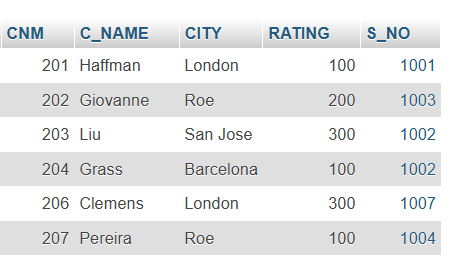
**CITY varchar (25),**

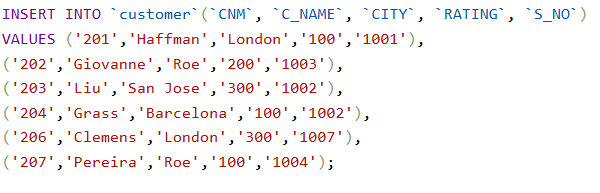
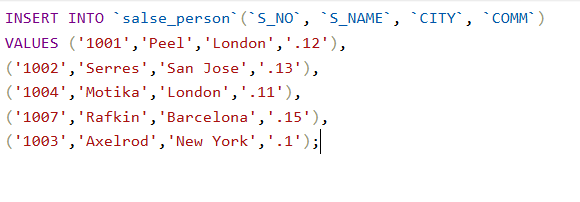
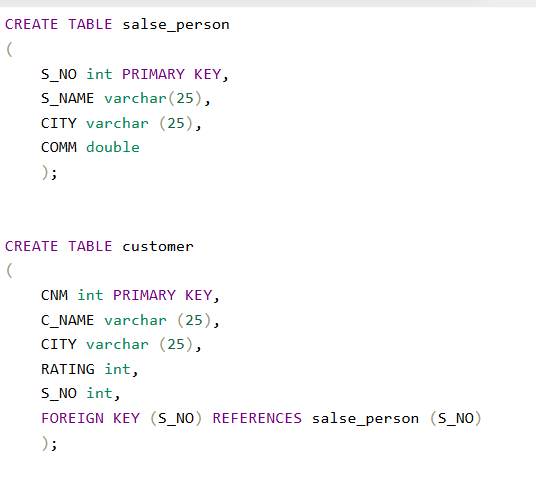
**RATING int,**

**S\_NO int,**

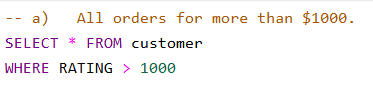
**FOREIGN KEY (S\_NO) REFERENCES salse\_person (S\_NO)**

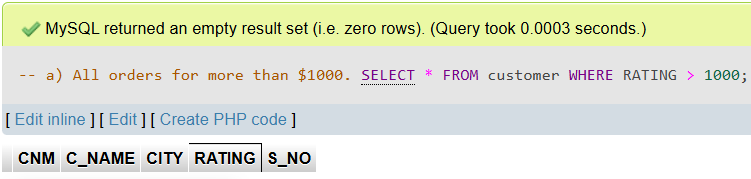
**);**

** salse\_person customer**

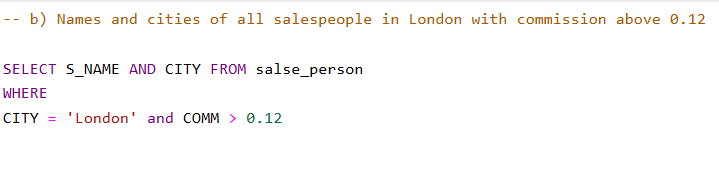


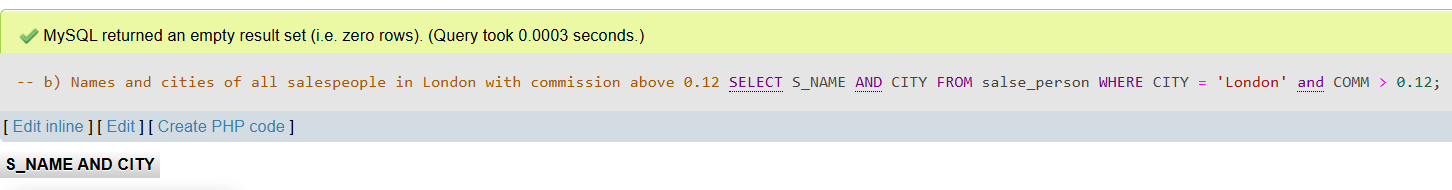
1. **All orders for more than $1000.**

****

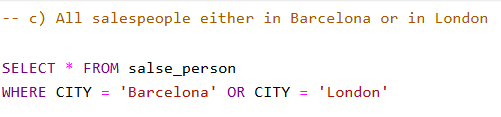
****

1. **Names and cities of all salespeople in London with commission above 0.12**

****

****

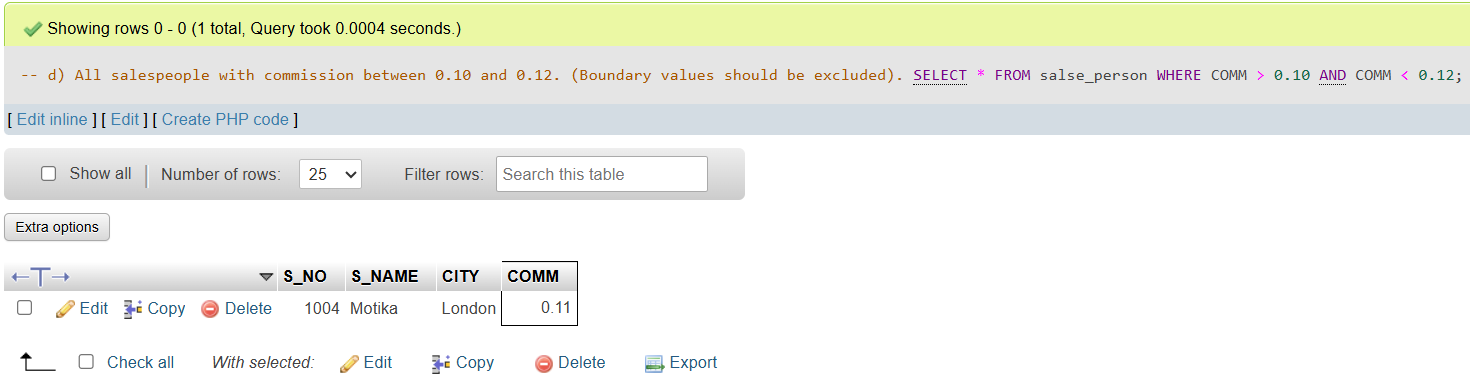
1. **All salespeople either in Barcelona or in London**

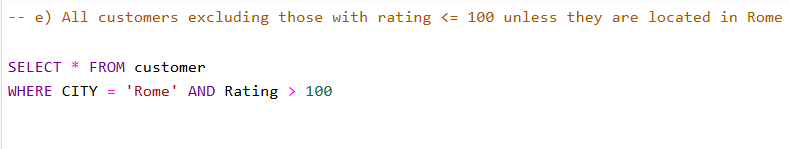
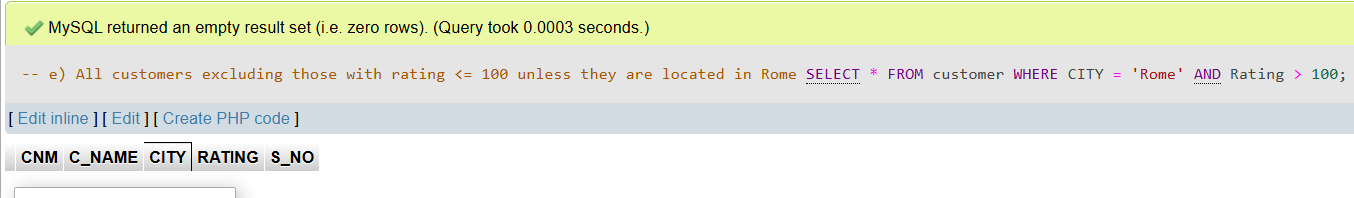
****

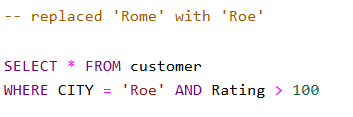
****

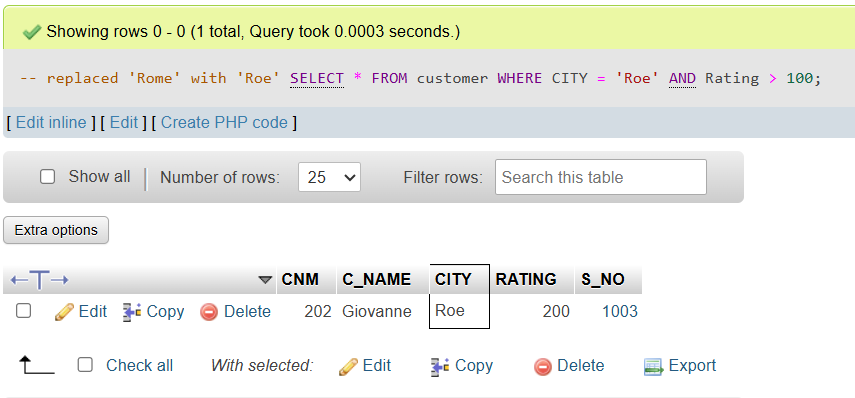
1. **All salespeople with commission between 0.10 and 0.12. (Boundary values should be excluded).**

****

****

**e) All customers excluding those with rating <= 100 unless they are located in Rome**

****

****