## Indian Institute of Information Technology Surat

****

# Lab Report on

# Advanced Database Management (CS 604) Practical

**Submitted by**

### [RAHUL KUMAR SINGH] (UI21CS44)

**Course Faculty**

### Mr. Rishi Sharma

## Department of Computer Science and Engineering

## Indian Institute of Information Technology Surat

## Gujarat-394190, India

**Jan-2024**

## Lab No: 1

**Aim: Perform basic SQL Query on three tables (Employee, Title, Bonus)**

**Description:** Create a Database for an Organization and create the following tables in the Organization Database:

*Employee(EMP\_ID(PK), FIRST\_NAME, LAST\_NAME, SALARY, JOINING\_DATE, DEPARTMENT)*

*Bonus (EMP\_REF\_ID(FK EMP\_ID), BONUS\_AMOUNT, BONUS\_DATE)*

*Title (EMP\_REF\_ID(FKEMP\_ID), EMP\_TITLE, AFFECTED\_FROM)*

Insert a minimum of 50 records in each table. Retrieve the following information from the Organization database:

1. SQL query to print all Employee details from the Employee table order by FIRST\_NAME Ascending and DEPARTMENT Descending.

2. SQL query to fetch the count of employees working in the department ‘Admin’.

3. SQL query to fetch Employee names with salaries >= 50000 and <= 100000.

4. SQL query to print details of the Workers who are also Managers.

5. SQL query to fetch duplicate records having matching data in some fields of a table.

6. SQL query to show only even rows from a table.

7. SQL query to show records from one table that another table does not have. Find employees in employee table that do not exist in bonus table (i.e. who did not get bonus)

8. SQL query to show the to pn(say10) records of a table.

9. Find people who have the same salary

10. SQL query to fetch the first 50% records from a table.

11. Find the highest 2 salaries without LIMIT or TOP.

12. Create a trigger to ensure that no employee of age less than 18 can be inserted in the database.

13. Create a trigger which will work before deletion in the employee table and create a duplicate copy of the record in another table employee\_backup.

14. Create a trigger to count the number of new tuples inserted using each insert statement.

## Source Code:

**Database Creation:**

CREATE DATABASE IF NOT EXISTS Organization;

USE Organization;

**Create the Employee table**

CREATE TABLE IF NOT EXISTS Employee (

EMP\_ID INT PRIMARY KEY,

FIRST\_NAME VARCHAR(50),

LAST\_NAME VARCHAR(50),

SALARY DECIMAL(10, 2),

JOINING\_DATE DATE,

DEPARTMENT VARCHAR(50)

);

**Create the Bonus table**

CREATE TABLE IF NOT EXISTS Bonus (

EMP\_REF\_ID INT,

BONUS\_AMOUNT DECIMAL(10, 2),

BONUS\_DATE DATE,

FOREIGN KEY (EMP\_REF\_ID) REFERENCES Employee(EMP\_ID)

);

**Create the Title table**

CREATE TABLE IF NOT EXISTS Title (

EMP\_REF\_ID INT,

EMP\_TITLE VARCHAR(50),

AFFECTED\_FROM DATE,

FOREIGN KEY (EMP\_REF\_ID) REFERENCES Employee(EMP\_ID)

);

**Task 1:**

SELECT \* FROM Employee ORDER BY FIRST\_NAME ASC, DEPARTMENT DESC;

**Task 2:**

SELECT COUNT(\*) FROM Employee WHERE DEPARTMENT = 'Admin';

**Task 3:**

SELECT FIRST\_NAME, LAST\_NAME FROM Employee WHERE SALARY BETWEEN 50000 AND 100000;

**Task 4:**

SELECT Employee.\* FROM Employee INNER JOIN Title ON Employee.EMP\_ID = Title.EMP\_REF\_ID WHERE Title.EMP\_TITLE = 'Manager';

**Task 5:**

SELECT SALARY, DEPARTMENT, COUNT(\*) FROM Employee GROUP BY SALARY, DEPARTMENT HAVING COUNT(\*) > 1;

**Task 6:**

WITH RankedRows AS (SELECT \*, ROW\_NUMBER() OVER (ORDER BY (SELECT NULL)) AS RowNum FROM Employee) SELECT \* FROM RankedRows WHERE RowNum % 2 = 0;

**Task 7:**

SELECT Employee.\* FROM Employee LEFT JOIN Bonus ON Employee.EMP\_ID = Bonus.EMP\_REF\_ID WHERE Bonus.EMP\_REF\_ID IS NULL;

**Task 8:**

SELECT \* FROM Employee LIMIT 10;

**Task 9:**

SELECT SALARY, COUNT(\*) FROM Employee GROUP BY SALARY HAVING COUNT(\*) > 1;

**Task 10:**

WITH RankedRows AS (SELECT \*, ROW\_NUMBER() OVER (ORDER BY (SELECT NULL)) AS RowNum FROM Employee) SELECT \* FROM RankedRows WHERE RowNum <= (SELECT COUNT(\*)/2 FROM Employee);

**Task 11:**

WITH RankedRows AS (SELECT \*, ROW\_NUMBER() OVER (ORDER BY Salary DESC) AS RowNum FROM Employee) SELECT \* FROM RankedRows WHERE RowNum <= 2;

**Task 12:**

DELIMITER //

CREATE TRIGGER age\_insert\_employee

BEFORE INSERT ON Employee

FOR EACH ROW

BEGIN

DECLARE emp\_age INT;

SET emp\_age = YEAR(CURDATE()) - YEAR(NEW.JOINING\_DATE) - (DATE\_FORMAT(CURDATE(), '%m%d') < DATE\_FORMAT(NEW.JOINING\_DATE, '%m%d'));

IF emp\_age < 18 THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Cannot insert employee with age less than 18.';

END IF;

END;

//

DELIMITER ;

**Task 13:**

CREATE TABLE IF NOT EXISTS Employee\_backup (

EMP\_ID INT PRIMARY KEY,

FIRST\_NAME VARCHAR(50),

LAST\_NAME VARCHAR(50),

SALARY DECIMAL(10, 2),

JOINING\_DATE DATE,

DEPARTMENT VARCHAR(50)

);

DELIMITER //

CREATE TRIGGER before\_delete\_employee BEFORE DELETE ON Employee FOR EACH ROW

BEGIN

INSERT INTO employee\_backup (EMP\_ID, FIRST\_NAME, LAST\_NAME, SALARY, JOINING\_DATE, DEPARTMENT) VALUES (OLD.EMP\_ID, OLD.FIRST\_NAME, OLD.LAST\_NAME, OLD.SALARY, OLD.JOINING\_DATE, OLD.DEPARTMENT);

END;

//

DELIMITER ;

**Task 14:**

CREATE TABLE insert\_count (

table\_name VARCHAR(255) PRIMARY KEY,

insert\_count INT DEFAULT 0

);

DELIMITER //

CREATE TRIGGER after\_insert\_count\_employee

AFTER INSERT ON Employee

FOR EACH ROW

BEGIN

INSERT INTO insert\_count (table\_name, insert\_count) VALUES ('Employee', 1) ON DUPLICATE KEY UPDATE insert\_count = insert\_count + 1;

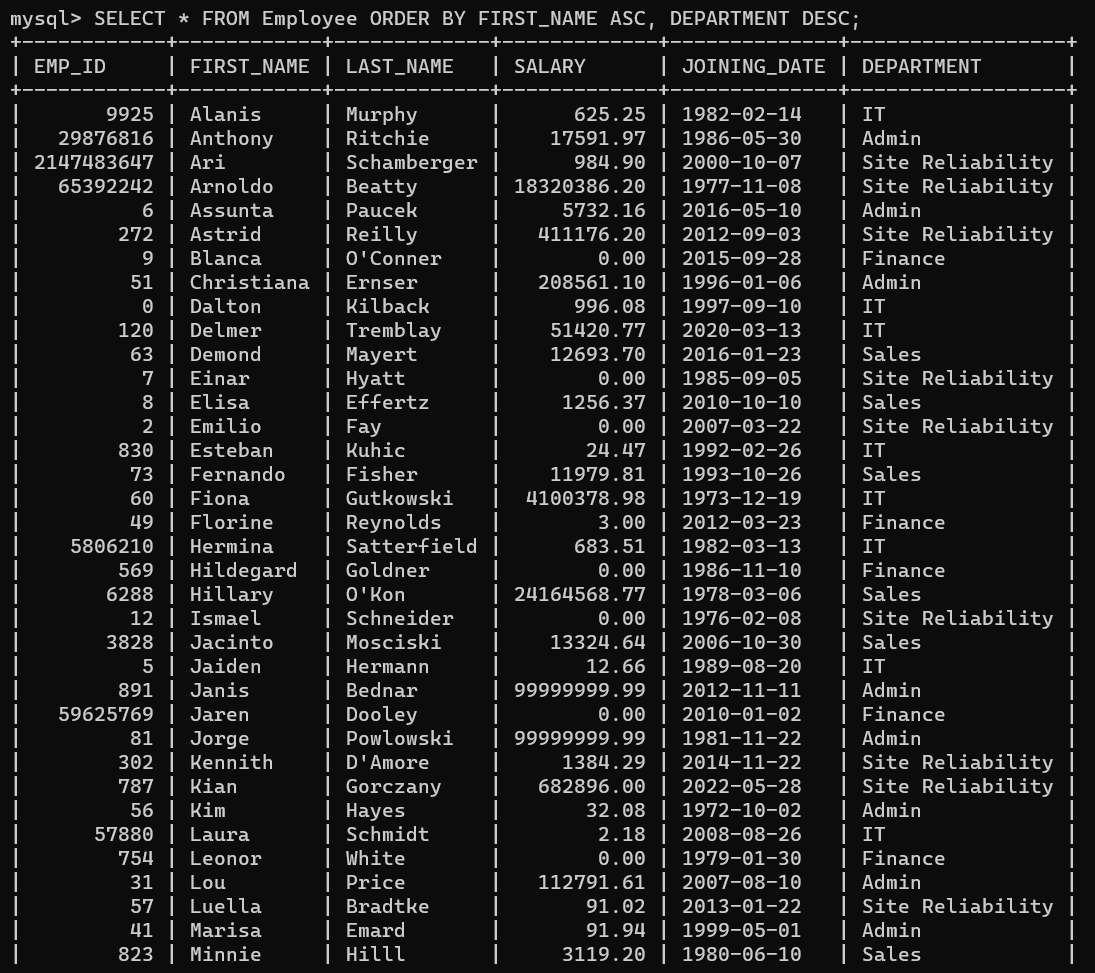
END;

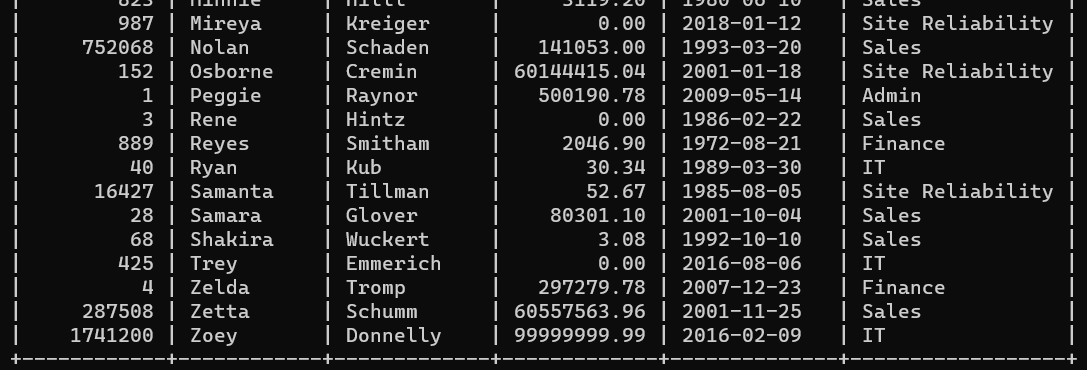
//

DELIMITER ;

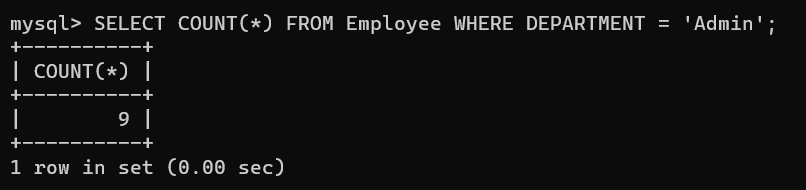
## Output:

**Task 1:**

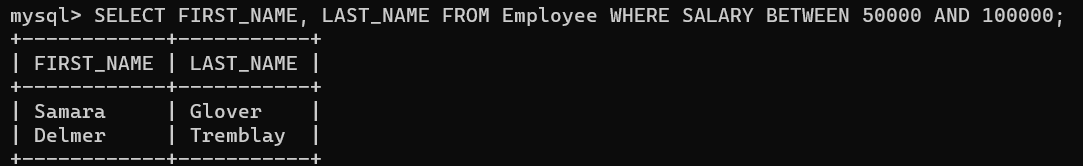
****

****

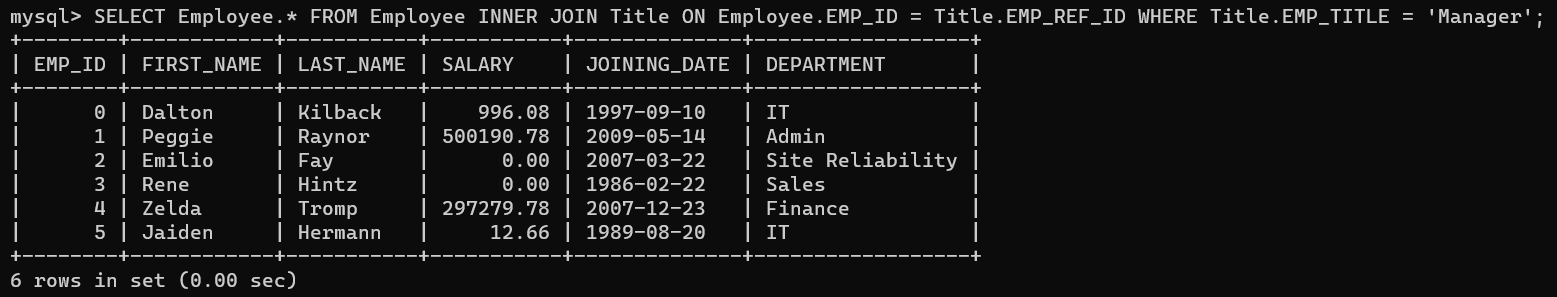
**Task 2:**

****

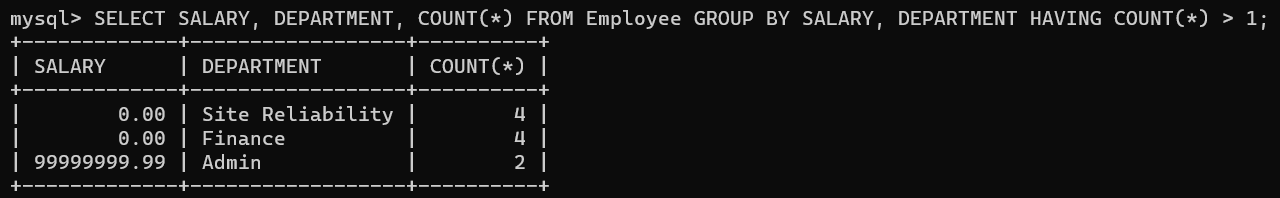
**Task 3:**

****

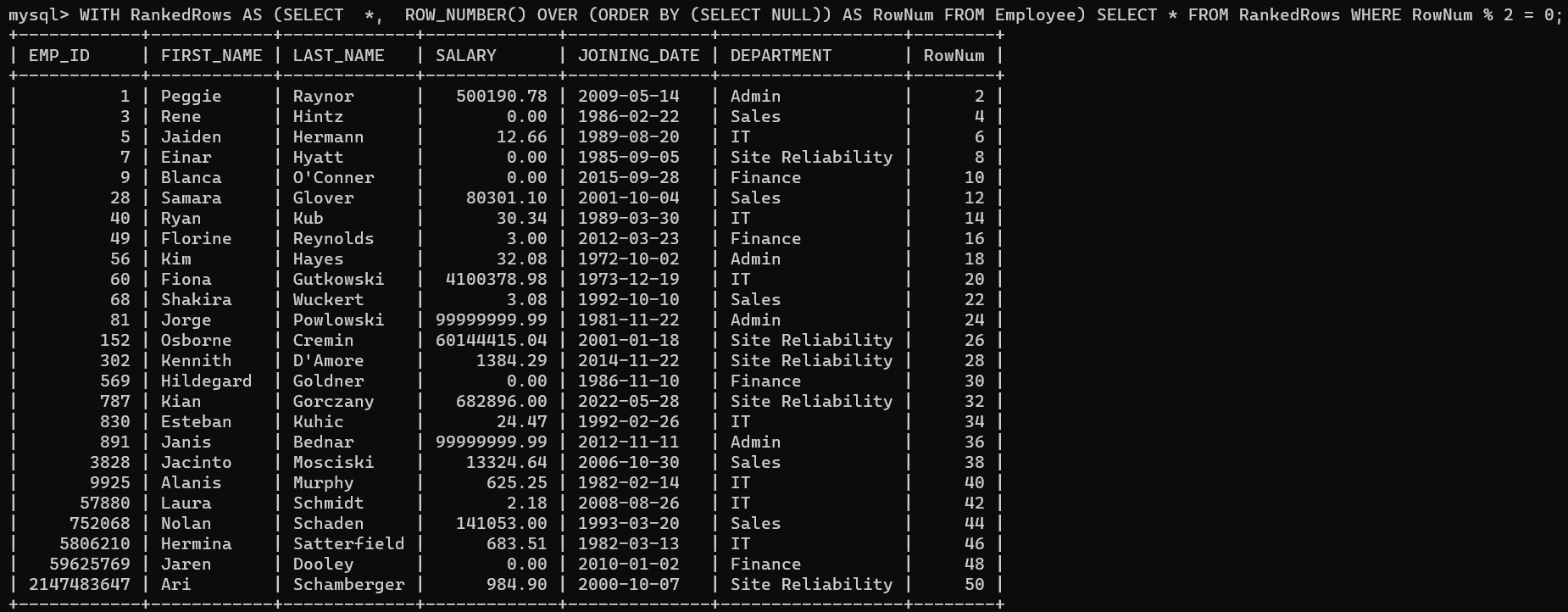
**Task 4:**

****

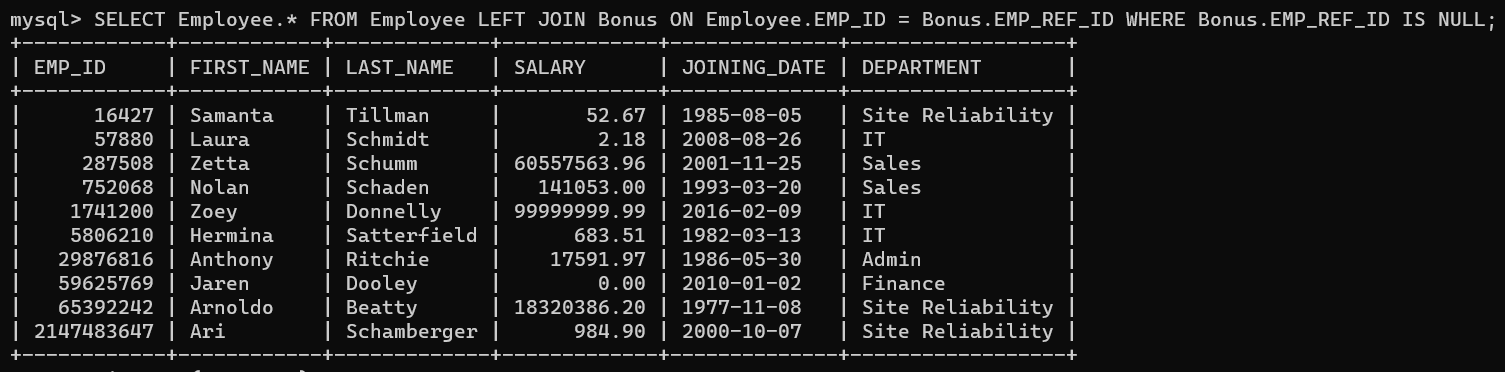
**Task 5:**

****

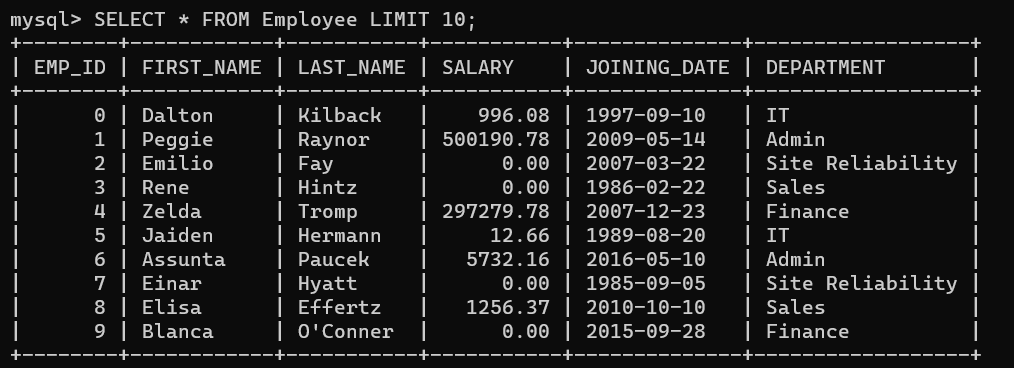
**Task 6:**

****

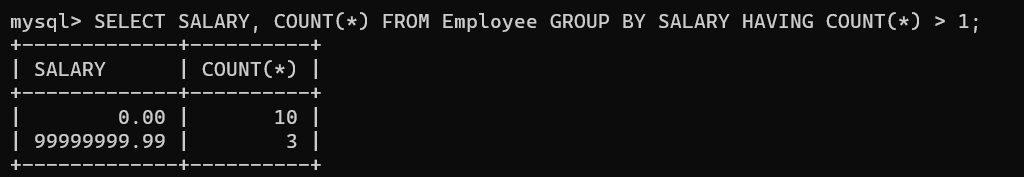
**Task 7:**

****

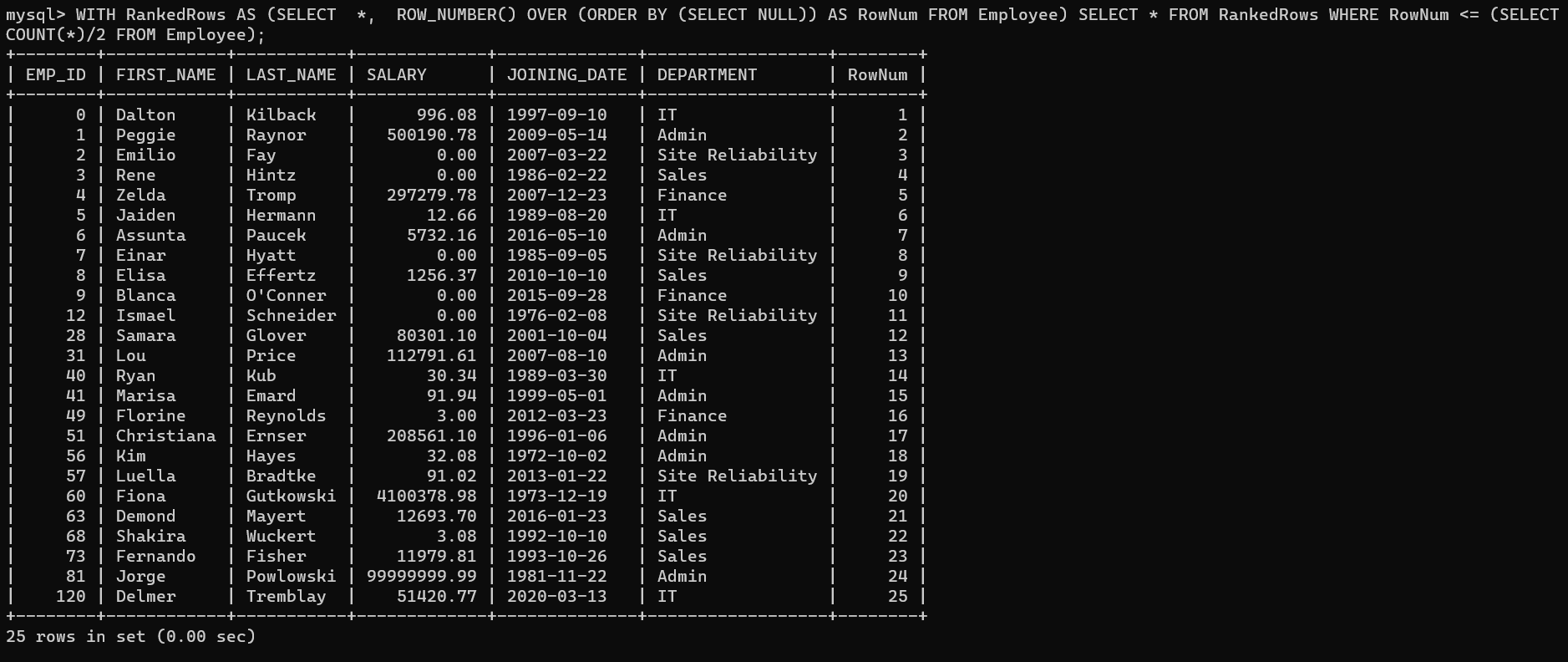
**Task 8:**

****

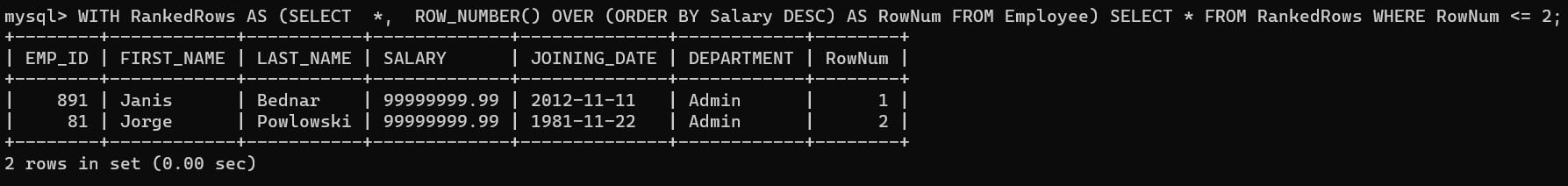
**Task 9:**

****

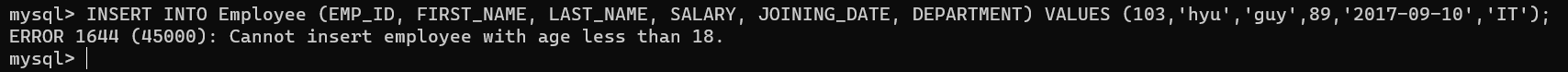
**Task 10:**

****

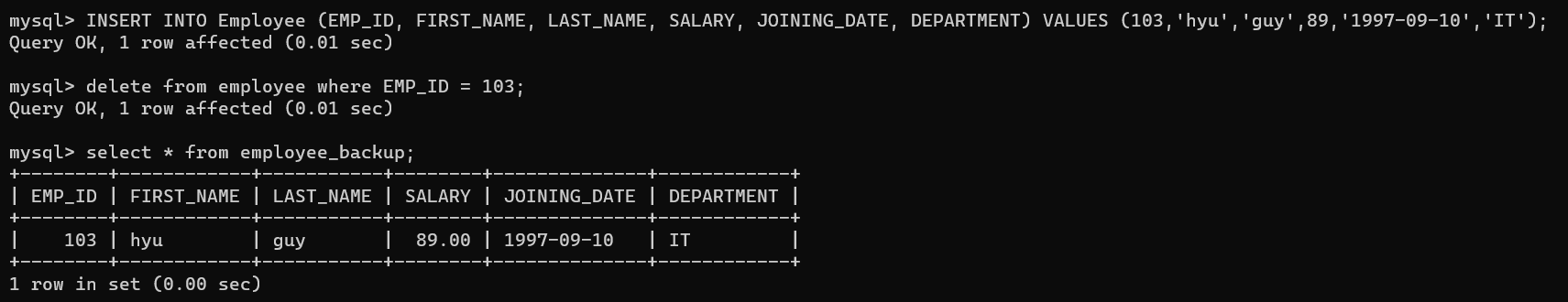
**Task 11:**

****

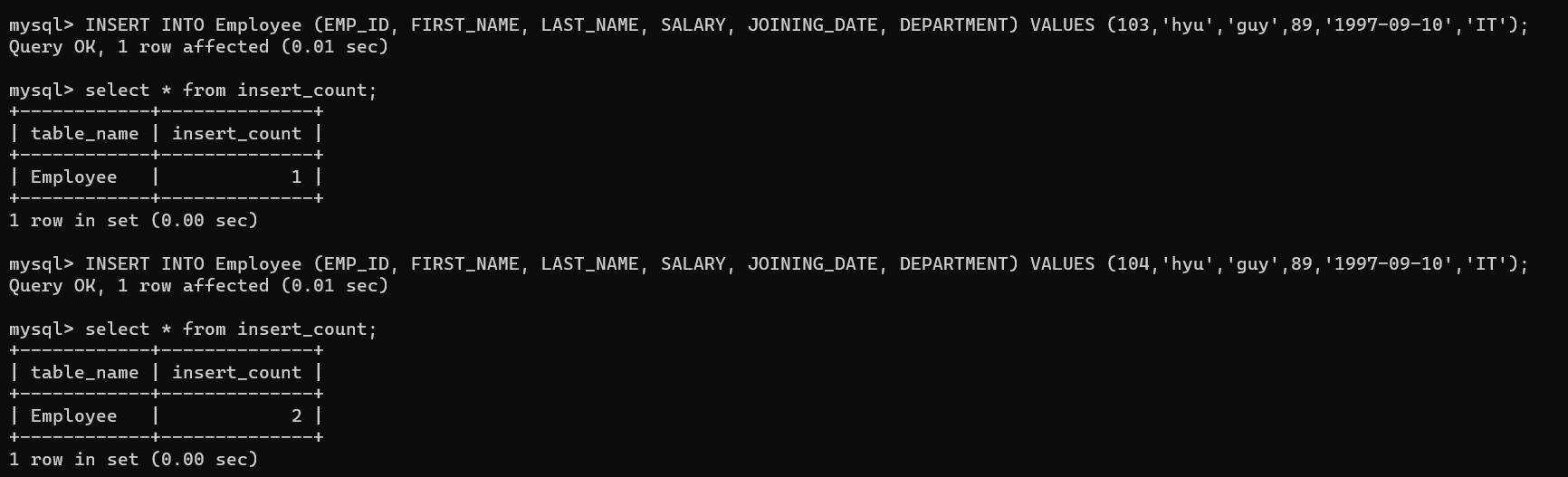
**Task 12:**

****

**Task 13:**

****

**Task 14:**

****

## Conclusion:

* Triggers are powerful mechanisms in SQL that can be used to enforce data integrity, automate tasks, and maintain historical records.
* Triggers enhance the reliability and security of the database by enforcing rules and executing actions automatically in response to specific events.
* Using basic sql statements to solve complex queries.

## Lab No: 2

**Aim: Write a PL/SQL code block to find total and average of 6 subjects and display the grade.**

**Description:** The provided PL/SQL code block calculates the total and average scores of a student in six subjects and determines the corresponding grade based on a simple grading system.

* The code assumes the existence of a table named student\_scores with columns for student ID and scores in each of the six subjects.
* It fetches the scores for a specified student ID, calculates the total and average, and assigns a grade (A, B, C, D, or F) based on the average score.
* The results, including the student ID, total, average, and grade, are then displayed using the DBMS\_OUTPUT.PUT\_LINE function.

## Source Code:

**With Table:**

-- Table Creation

CREATE TABLE student\_scores (

student\_id NUMBER,

sub1 NUMBER,

sub2 NUMBER,

sub3 NUMBER,

sub4 NUMBER,

sub5 NUMBER,

sub6 NUMBER

);

-- PL/SQL block to calculate total, average, and display grade

DECLARE

v\_student\_id student\_scores.student\_id%TYPE;

v\_sub1 student\_scores.sub1%TYPE;

v\_sub2 student\_scores.sub2%TYPE;

v\_sub3 student\_scores.sub3%TYPE;

v\_sub4 student\_scores.sub4%TYPE;

v\_sub5 student\_scores.sub5%TYPE;

v\_sub6 student\_scores.sub6%TYPE;

v\_total NUMBER;

v\_average NUMBER;

v\_grade VARCHAR2(2);

BEGIN

SELECT student\_id, sub1, sub2, sub3, sub4, sub5, sub6

INTO v\_student\_id, v\_sub1, v\_sub2, v\_sub3, v\_sub4, v\_sub5, v\_sub6

FROM student\_scores

WHERE student\_id = 1;

-- Calculate total and average

v\_total := v\_sub1 + v\_sub2 + v\_sub3 + v\_sub4 + v\_sub5 + v\_sub6;

v\_average := v\_total / 6;

-- Grade Determination

IF v\_average >= 90 THEN

v\_grade := 'A';

ELSIF v\_average >= 80 THEN

v\_grade := 'B';

ELSIF v\_average >= 70 THEN

v\_grade := 'C';

ELSIF v\_average >= 60 THEN

v\_grade := 'D';

ELSE

v\_grade := 'F';

END IF;

-- Results

DBMS\_OUTPUT.PUT\_LINE('Student ID: ' || v\_student\_id);

DBMS\_OUTPUT.PUT\_LINE('Total: ' || v\_total);

DBMS\_OUTPUT.PUT\_LINE('Average: ' || v\_average);

DBMS\_OUTPUT.PUT\_LINE('Grade: ' || v\_grade);

END;

/

**Without\_Table:**

SET SERVEROUTPUT ON;

DECLARE

subject1 NUMBER := 85;

subject2 NUMBER := 92;

subject3 NUMBER := 78;

subject4 NUMBER := 90;

subject5 NUMBER := 88;

subject6 NUMBER := 95;

total NUMBER;

average NUMBER;

grade VARCHAR2(2);

BEGIN

total := subject1 + subject2 + subject3 + subject4 + subject5 + subject6;

average := total / 6;

IF average >= 90 THEN

grade := 'A';

ELSIF average >= 80 THEN

grade := 'B';

ELSIF average >= 70 THEN

grade := 'C';

ELSIF average >= 60 THEN

grade := 'D';

ELSE

grade := 'F';

END IF;

DBMS\_OUTPUT.PUT\_LINE('Total: ' || total);

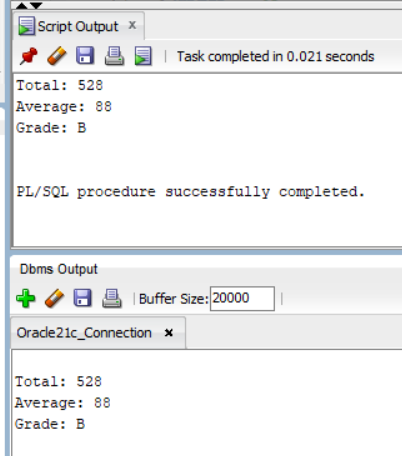
DBMS\_OUTPUT.PUT\_LINE('Average: ' || average);

DBMS\_OUTPUT.PUT\_LINE('Grade: ' || grade);

END;

/

## Output:



## Conclusion:

* The code is structured in a modular manner using a PL/SQL block for better understanding.
* The code is designed for execution in interactive environments.
* Utilized DECLARE and BEGIN sections to define variables and execute procedural logic.
* Applied the DBMS\_OUTPUT.PUT\_LINE function for displaying total marks, average marks, and the corresponding grade.

## Lab No: 3

**Aim: Write a PL/SQL (MySQL Procedure) code block to perform specific tasks on tables Teacher, Class and Pay\_scale.**

**Description:** Consider the following table to write PL/SQL code as specified under

* Teacher (t\_no, f\_name, l\_name, salary, supervisor, joining\_date, birth\_date, title)
* Class (class\_no, t\_no, room\_no)
* Pay\_scale (Min\_limit, Max\_limit, grade)

1. Accept a range of salary and print the details of teachers from the teacher table.

2. By using cursor - Calculate the bonus amount to be given to a teacher depending on the following conditions:

a) if salary < 10000 then bonus is 10% of the salary.

b) if the salary is between 10000 and 20000 then bonus is 20% of the salary.

c) if the salary is between 20000 and 25000 then bonus is 25% of the salary.

d) if the salary exceeds 25000 then the bonus is 30% of the salary.

3. Using a simple LOOP structure, list the first 10 records of the ‘teachers’ table.

4. Accept the room number and display the teacher details like t\_no, f\_name, l\_name, birth\_date, title from table Teacher.

## Source Code:

**Teacher Table:**

CREATE TABLE Teacher (

t\_no INT PRIMARY KEY,

f\_name VARCHAR(50) NOT NULL,

l\_name VARCHAR(50) NOT NULL,

salary DECIMAL(10, 2) NOT NULL,

supervisor INT,

joining\_date DATE NOT NULL,

birth\_date DATE NOT NULL,

title VARCHAR(50) NOT NULL

);

**Class Table:**

CREATE TABLE Class (

class\_no INT PRIMARY KEY,

t\_no INT,

room\_no INT,

FOREIGN KEY (t\_no) REFERENCES Teacher(t\_no),

UNIQUE KEY unique\_teacher\_class (t\_no, room\_no)

);

**Pay\_scale Table:**

CREATE TABLE Pay\_scale (

Min\_limit DECIMAL(10, 2) NOT NULL,

Max\_limit DECIMAL(10, 2) NOT NULL,

grade VARCHAR(10) PRIMARY KEY

);

**Task 1:**

DELIMITER //

CREATE PROCEDURE GetTeachersBySalaryRange(

IN minSalary DECIMAL(10, 2),

IN maxSalary DECIMAL(10, 2)

)

BEGIN

SELECT \*

FROM Teacher

WHERE salary BETWEEN minSalary AND maxSalary;

END //

DELIMITER ;

**Task 2:**

DELIMITER //

CREATE PROCEDURE CalculateTeacherBonus()

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE t\_no\_var INT;

DECLARE salary\_var DECIMAL(10, 2);

DECLARE bonus\_var DECIMAL(10, 2);

DECLARE teacher\_cursor CURSOR FOR

SELECT t\_no, salary

FROM Teacher;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN teacher\_cursor;

teacher\_loop: LOOP

FETCH teacher\_cursor INTO t\_no\_var, salary\_var;

IF done THEN

LEAVE teacher\_loop;

END IF;

IF salary\_var < 10000 THEN

SET bonus\_var = salary\_var \* 0.10;

ELSEIF salary\_var BETWEEN 10000 AND 20000 THEN

SET bonus\_var = salary\_var \* 0.20;

ELSEIF salary\_var BETWEEN 20000 AND 25000 THEN

SET bonus\_var = salary\_var \* 0.25;

ELSE

SET bonus\_var = salary\_var \* 0.30;

END IF;

SELECT t\_no\_var AS Teacher\_ID, salary\_var AS Salary, bonus\_var AS Bonus;

END LOOP;

CLOSE teacher\_cursor;

END //

DELIMITER ;

**Task 3:**

DELIMITER //

CREATE PROCEDURE ListFirst10Teachers()

BEGIN

DECLARE teacher\_cursor CURSOR FOR

SELECT t\_no, f\_name, l\_name, salary, supervisor, joining\_date, birth\_date, title

FROM Teacher;

OPEN teacher\_cursor;

teacher\_loop: LOOP

FETCH teacher\_cursor INTO t\_no\_var, f\_name\_var, l\_name\_var, salary\_var, supervisor\_var, joining\_date\_var, birth\_date\_var, title\_var;

IF counter >= 10 OR t\_no\_var IS NULL THEN

LEAVE teacher\_loop;

END IF;

SET counter = counter + 1;

SELECT t\_no\_var AS Teacher\_ID, f\_name\_var AS First\_Name, l\_name\_var AS Last\_Name, salary\_var AS Salary,

supervisor\_var AS Supervisor, joining\_date\_var AS Joining\_Date, birth\_date\_var AS Birth\_Date, title\_var AS Title;

END LOOP;

CLOSE teacher\_cursor;

END //

DELIMITER ;

**Task 4:**

DELIMITER //

CREATE PROCEDURE GetTeachersByRoomNumber(IN roomNumber INT)

BEGIN

SELECT t.t\_no, t.f\_name, t.l\_name, t.birth\_date, t.title

FROM Teacher t

JOIN Class c ON t.t\_no = c.t\_no

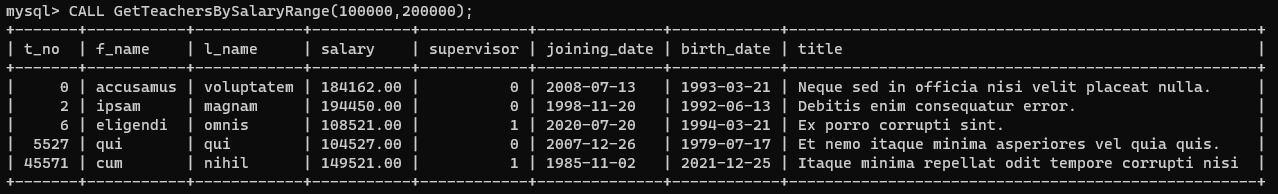
WHERE c.room\_no = roomNumber;

END //

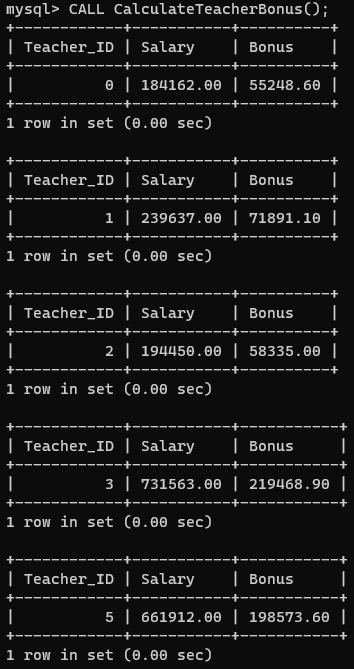
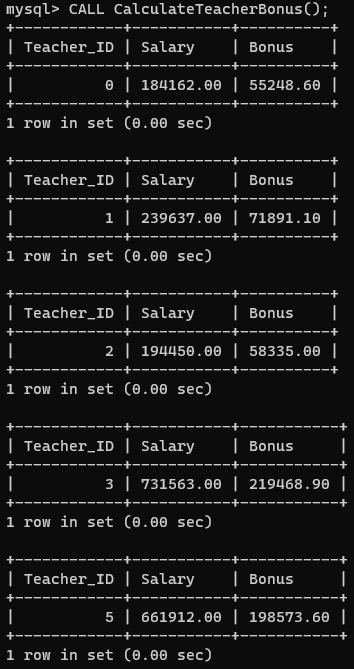
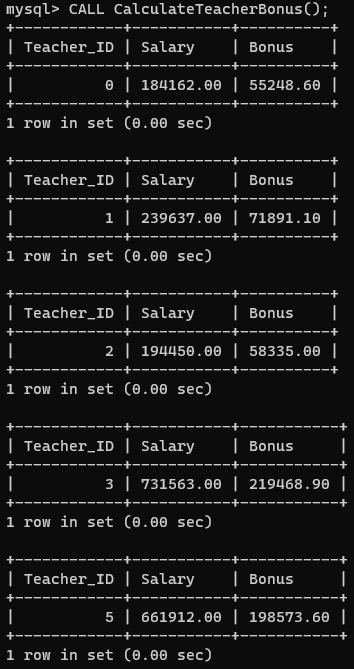
DELIMITER ;

## Output:

**Task 1:**

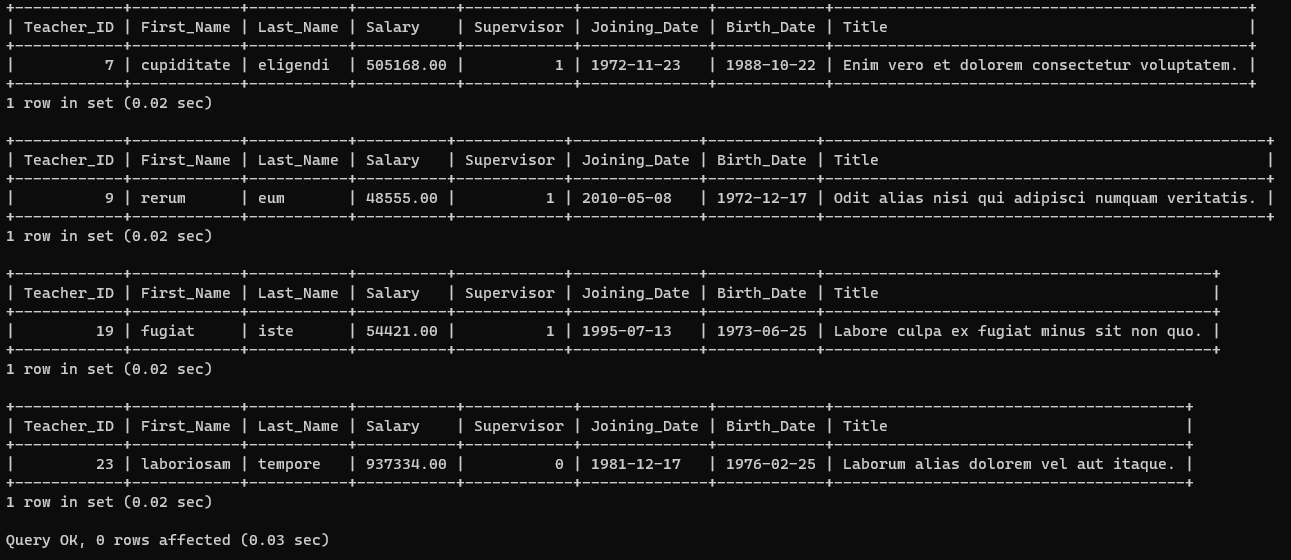
****

**Task 2:**

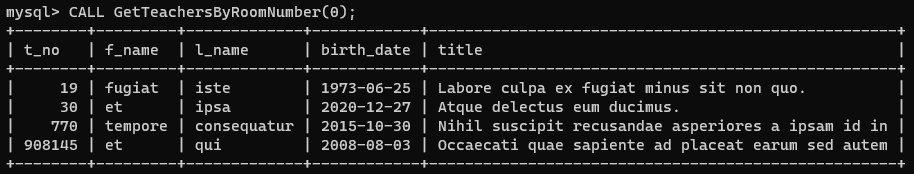
****

**Task 3:**

****

****

**Task 4:**

****

## Conclusion:

* GetTeachersBySalaryRange: Accepts a salary range and retrieves details of teachers from the Teacher table within that range.
* CalculateTeacherBonus: Uses a cursor to calculate the bonus amount for teachers based on salary conditions, considering different bonus percentages for specific salary ranges.
* ListFirst10Teachers: Utilizes a simple LOOP structure to list the first 10 records from the Teacher table.
* GetTeachersByRoomNumber: Accepts a room number and displays specific details (t\_no, f\_name, l\_name, birth\_date, title) of teachers associated with that room from the Teacher table.
* The code is structured in a modular manner using a MySQL Procedure block for better understanding.
* The code is designed for execution in interactive environments.
* Utilized DECLARE and BEGIN sections to define variables and execute procedural logic.
* Applied the DBMS\_OUTPUT.PUT\_LINE function for displaying output for all the procedures.

.

## Lab No: 4

**Aim: Design and develop a suitable Student Database application. One of the attributes to me maintained is the attendance of a student in each subject for which he/she has enrolled.**

**Description:** Using TRIGGERS, we write active rules to do the following:

a) Whenever attendance is updated, check if the attendance is less than 85%; if so notify the Head of Department concerned.

b) Whenever the marks in the Internal Assessment Test are entered, check if the marks are less than 40%; if so, notify the Head of the Department concerned.

## Source Code:

**Student\_Attendance Table:**

CREATE TABLE Student\_Attendance (

t\_no INT PRIMARY KEY,

Day1 TINYINT, Day2 TINYINT, Day3 TINYINT, Day4 TINYINT, Day5 TINYINT,

Day6 TINYINT, Day7 TINYINT, Day8 TINYINT, Day9 TINYINT, Day10 TINYINT

);

**Student\_Marks Table:**

CREATE TABLE Student\_Marks (

t\_no INT PRIMARY KEY,

Sub1 DECIMAL(5,2), Sub2 DECIMAL(5,2), Sub3 DECIMAL(5,2),

Sub4 DECIMAL(5,2), Sub5 DECIMAL(5,2), Sub6 DECIMAL(5,2)

);

**Task 1:**

DELIMITER //

CREATE TRIGGER after\_insert\_attendance\_student

AFTER INSERT ON Student\_Attendance

FOR EACH ROW

BEGIN

IF ((NEW.Day1+NEW.Day2+NEW.Day3+NEW.Day4+NEW.Day5+NEW.Day6+NEW.Day7+NEW.Day8+NEW.Day9+NEW.Day10)\*(100/10) < 85) THEN

-- CALL NotifyHeadOfDepartment(NEW.t\_no, 'Low internal assessment marks');

SIGNAL SQLSTATE '02000' SET MESSAGE\_TEXT = "Notice: Attendance for the entered student is less than 85%";

END IF;

END;

//

DELIMITER ;

**Task 2:**

DELIMITER //

CREATE TRIGGER after\_insert\_marks\_student

AFTER INSERT ON Student\_Marks

FOR EACH ROW

BEGIN

IF ((NEW.Sub1+NEW.Sub2+NEW.Sub3+NEW.Sub4+NEW.Sub5+NEW.Sub6)/6 < 40) THEN

-- CALL NotifyHeadOfDepartment(NEW.t\_no, 'Low internal assessment marks');

SIGNAL SQLSTATE '02000' SET MESSAGE\_TEXT = "Notice: Marks for Internal Assessment are less than 40%";

END IF;

END;

//

DELIMITER ;

**Test:**

CREATE TRIGGER AttendanceTrigger

AFTER UPDATE ON Enrollments

FOR EACH ROW

BEGIN

IF NEW.Attendance < 85 THEN

CALL NotifyHeadOfDepartment(NEW.StudentID, 'Low attendance');

END IF;

END;

CREATE TRIGGER AssessmentTrigger

AFTER UPDATE ON Enrollments

FOR EACH ROW

BEGIN

IF NEW.InternalAssessmentMarks < 40 THEN

CALL NotifyHeadOfDepartment(NEW.StudentID, 'Low internal assessment marks');

END IF;

END;

DELIMITER //

CREATE PROCEDURE NotifyHeadOfDepartment(IN studentID INT, IN message VARCHAR(255))

BEGIN

SELECT studentID, ": ", message as Output;

-- DECLARE departmentHeadEmail VARCHAR(255);

-- SELECT Email INTO departmentHeadEmail

-- FROM DepartmentHeads

-- WHERE DepartmentID = (SELECT DepartmentID FROM Students WHERE StudentID = studentID);

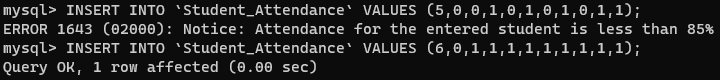
-- CALL SendEmail(departmentHeadEmail, message);

END //

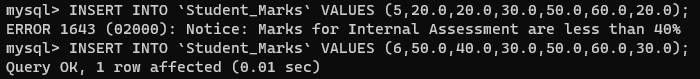
DELIMITER ;

## Output:

**Task 1:**

****

**Task 2:**

****

## Conclusion:

* The code is structured in a modular manner using a MySQL Procedure block for better understanding.
* The code is designed for execution in interactive environments.
* Utilized BEGIN sections to define variables and execute trigger logic.
* Created “NotifyHeadOfDepartment” and “SendEmail” Procedure to notify and send mail to the respected authority.
* Applied the DBMS\_OUTPUT.PUT\_LINE function for displaying output for all the procedures.

## Lab No: 5

**Aim: To implement Deadlock Detection Algorithm for Distributed Database using Wait-for Graph to check for Deadlock.**

**Description:**

* DetectDeadlock Procedure:
* Creates a temporary table for wait-for graph.
* Populates wait-for graph with data from deadlock\_info.
* DepthFirstSearch Procedure:
* Simulates stack for DFS using a temporary table.
* Detects cycles by recursively traversing wait-for graph.
* DetectDeadlock Procedure Modification:
* Inserts multiple rows into dfs\_stack from wait\_for\_graph where requesting\_node=start\_node.

## Source Code:

**Table:**

CREATE TABLE Deadlock\_Info (

transaction\_id INT PRIMARY KEY AUTO\_INCREMENT,

requesting\_node INT,

holding\_node INT

);

**Detect Deadlock Procedure:**

DELIMITER //

CREATE PROCEDURE DetectDeadlock()

BEGIN

DECLARE result INT DEFAULT 0;

DECLARE temp INT DEFAULT 0;

DECLARE start\_node INT;

DECLARE current\_node INT;

DECLARE done INT DEFAULT 0;

CREATE TEMPORARY TABLE IF NOT EXISTS wait\_for\_graph (

requesting INT,

holding INT

);

INSERT INTO wait\_for\_graph SELECT requesting\_node, holding\_node FROM deadlock\_info;

SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

CREATE TEMPORARY TABLE IF NOT EXISTS distinct\_nodes (

node INT

);

INSERT INTO distinct\_nodes

SELECT DISTINCT requesting

FROM wait\_for\_graph;

WHILE (SELECT COUNT(\*) FROM distinct\_nodes) > 0 DO

SELECT node INTO start\_node FROM distinct\_nodes ORDER BY node LIMIT 1;

DELETE FROM distinct\_nodes WHERE node = start\_node;

SELECT CONCAT("Start Node: ",start\_node) as message;

SELECT result;

CALL DepthFirstSearch(start\_node, start\_node, temp);

IF temp = 1 THEN

SET result = 1;

END IF;

DELETE FROM wait\_for\_graph;

INSERT INTO wait\_for\_graph SELECT requesting\_node, holding\_node FROM deadlock\_info;

END WHILE;

SELECT result;

IF result = 0 THEN

SELECT "No Deadlock Detected!" as message;

ELSE

SELECT "Deadlock Detected!" as message;

END IF;

DROP TEMPORARY TABLE IF EXISTS wait\_for\_graph;

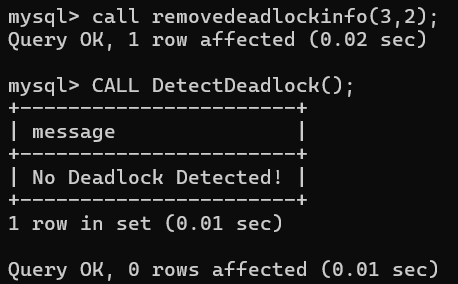
DROP TEMPORARY TABLE IF EXISTS distinct\_nodes;

END //

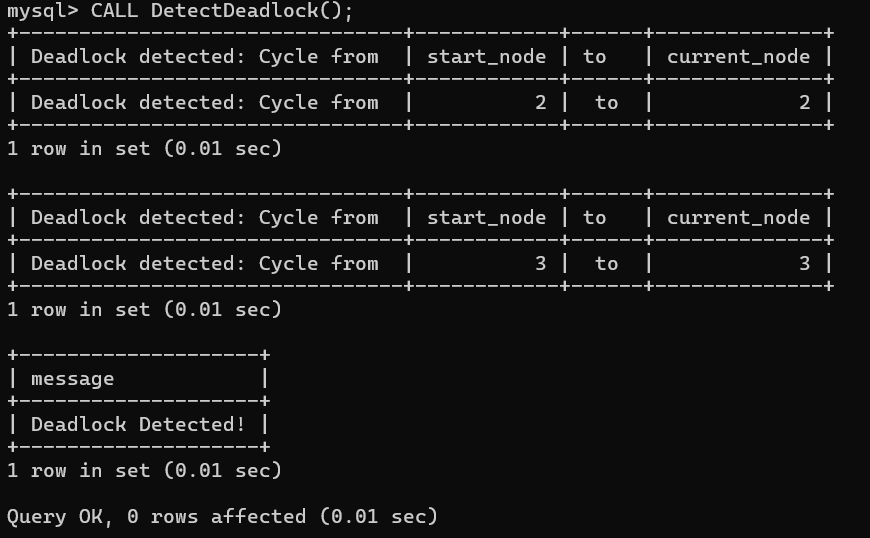
DELIMITER ;

## Output:

**In case of No Deadlock:**

****

**In case of Deadlock:**

****

## Conclusion:

* Detects deadlocks in a distributed database using a wait-for graph.
* Implemented depth-first search within MySQL stored procedures.
* Procedures manage deadlock information, simulate DFS, and execute deadlock detection.
* MySQL limitations for complex algorithms; use external languages for efficiency.
* Deadlock detection results in cycles which means careful consideration using it.

## Lab No: 6

**Aim: To Implement Object Oriented Approach for writing PL/SQL codes (MySQL)**

**Description:**

1. Write a PL/SQL code to create a class for a "Person" with attributes such as name, age, and address.
2. Write a PL/SQL code to Implement methods in the "Person" class to display the details and update the age.
3. Write a PL/SQL code to implement a method to calculate the annual bonus based on the salary in the "Employee" class.
4. Write a PL/SQL code to create a "Manager" subclass inheriting from the "Employee" class, and add an attribute to store the number of employees managed.

## Source Code:

**A) Implementing a "Person" class:**

Drop TABLE Person;

CREATE TABLE Person (

objectId VARCHAR(100) PRIMARY KEY,

name VARCHAR(100),

age INT,

address VARCHAR(200)

);

-- Dropping Procedures

DROP PROCEDURE DisplayDetails;

DROP PROCEDURE UpdateAge;

DROP PROCEDURE AppendPerson;

**B) Implementing methods to create object, display details and update age:**

DELIMITER //

CREATE PROCEDURE DisplayDetails(IN object\_id VARCHAR(100))

BEGIN

SELECT name, age, address FROM Person WHERE objectId = object\_id;

END //

CREATE PROCEDURE UpdateAge(IN object\_id VARCHAR(100), IN new\_age INT)

BEGIN

UPDATE Person SET age = new\_age WHERE objectId = object\_id;

END //

CREATE PROCEDURE AppendPerson(IN object\_id VARCHAR(100), IN person\_name VARCHAR(100), IN new\_age INT, IN person\_address VARCHAR(100))

BEGIN

INSERT INTO Person values (object\_id, person\_name, new\_age, person\_address);

END //

DELIMITER ;

-- Calling Procedures:

CALL AppendPerson("person1", "Rahul Kumar Singh", 20, "Raigarh");

CALL DisplayDetails("person1");

CALL UpdateAge("person1", 21);

CALL DisplayDetails("person1");

**C) Implementing methods to create object, display details and calculate the annual bonus based on salary:**

-- Dropping Procedures

DROP PROCEDURE DisplayEmpDetails;

DROP PROCEDURE AppendEmployee;

DROP PROCEDURE CalculateAnnualBonus;

DELIMITER //

CREATE PROCEDURE DisplayEmpDetails(IN object\_id VARCHAR(100))

BEGIN

SELECT \* FROM Employee WHERE objectId = object\_id;

END //

CREATE PROCEDURE AppendEmployee(IN object\_id VARCHAR(100), IN person\_name VARCHAR(100), IN new\_age INT, IN person\_address VARCHAR(100), IN salary DECIMAL(10,2))

BEGIN

INSERT INTO Employee values (object\_id, person\_name, new\_age, person\_address, salary);

END //

CREATE PROCEDURE CalculateAnnualBonus(IN object\_id VARCHAR(100))

BEGIN

SELECT salary \* 0.1 FROM Employee WHERE objectId = object\_id; -- 10% bonus rate.

END //

DELIMITER ;

-- Calling Procedures

CALL CalculateAnnualBonus(4000);

**D) Implementing an "Employee" and "Manager" subclass:**

Drop TABLE Employee;

Drop TABLE Manager;

CREATE TABLE Employee (

objectId VARCHAR(100) PRIMARY KEY,

name VARCHAR(100),

age INT,

address VARCHAR(200),

salary DECIMAL(10,2)

);

CREATE TABLE Manager AS

SELECT \* FROM Employee;

ALTER TABLE Manager

ADD num\_employees\_managed INT;

-- Dropping Procedures

DROP PROCEDURE DisplayManDetails;

DROP PROCEDURE CalculateManagerBonus;

DROP PROCEDURE AppendManager;

DELIMITER //

CREATE PROCEDURE DisplayManDetails(IN object\_id VARCHAR(100))

BEGIN

SELECT \* FROM Manager WHERE objectId = object\_id;

END //

CREATE PROCEDURE AppendManager(IN object\_id VARCHAR(100), IN person\_name VARCHAR(100), IN new\_age INT, IN person\_address VARCHAR(100), IN salary DECIMAL(10,2), IN num\_emp INT)

BEGIN

INSERT INTO Manager values (object\_id, person\_name, new\_age, person\_address, salary, num\_emp);

END //

CREATE PROCEDURE CalculateManagerBonus(IN object\_id VARCHAR(100))

BEGIN

SELECT salary \* 0.15 + num\_employees\_managed \* 1000 FROM Manager WHERE objectId = object\_id; -- Bonus

END //

DELIMITER ;

-- Calling Procedures

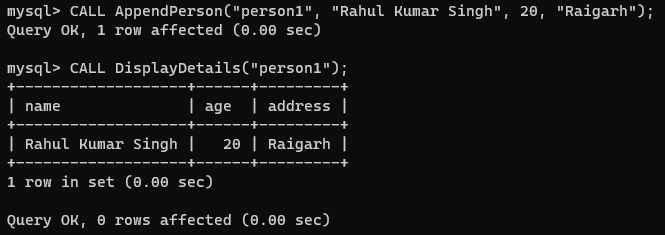
CALL AppendManager("manager1", "Rahul Kumar Singh", 20, "Raigarh", 10000.00, 10);

CALL DisplayManDetails("manager1");

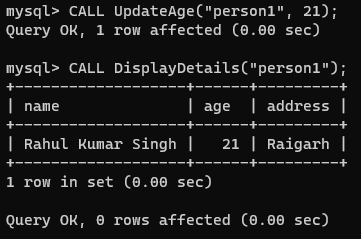
CALL CalculateManagerBonus("manager1");

## Output:

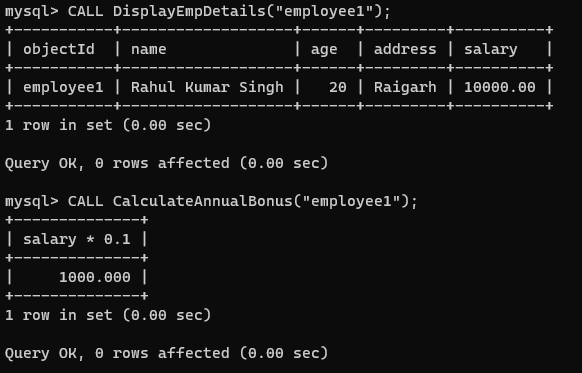
**A) Implementing a "Person" class:**

****

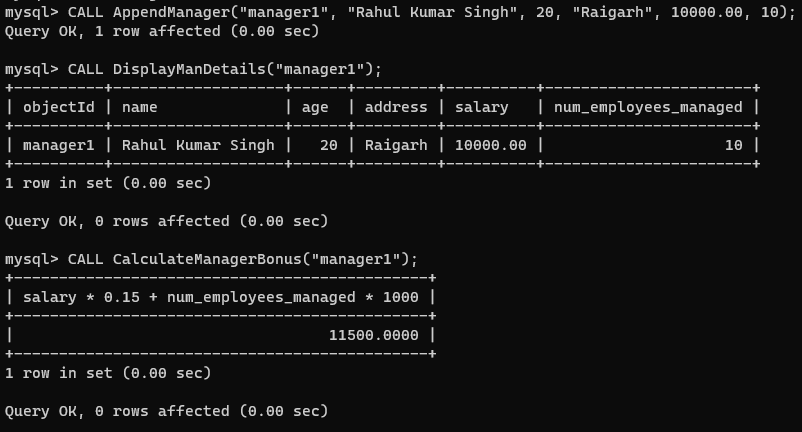
**B) Implementing methods to create object, display details and update age:**

****

**C) Implementing methods to create object, display details and calculate the annual bonus based on salary:**

****

**D) Implementing an "Employee" and "Manager" subclass:**

****

## Conclusion:

* Help us to understand the importance of object-oriented approach.
* Provide various features of object-oriented approach like Polymorphism, Inheritance and Encapsulation.
* To be able to implement the MySQL code into an Object-oriented programming model.

## Lab No: 7

**Aim: To Implement basic relational methods and statements using PostgreSQL.**

**Description:**

1. Write a PSQL statement to create a simple table countries including columns country\_id,country\_name and region\_id.

2. Write a PSQL statement to create a simple table countries including columns country\_id,country\_name and region\_id which already exist.

3. Write a PSQL statement to create the structure of a table dup\_countries similar to countries.

4. Write a PSQL statement to create a duplicate copy of countries table including structure and data by name dup\_countries.

5. Write a PSQL statement to create a table countries set a constraint NULL.

6. Write a PSQL statement to create a table named jobs including columns job\_id, job\_title, min\_salary, max\_salary and check whether the max\_salary amount exceeding the upper limit 25000.

7. Write a PSQL statement to create a table named countries including columns country\_id, country\_name and region\_id and make sure that no countries except Italy, India and China will be entered in the table.

8. Write a PSQL statement to create a table named countries including columns country\_id,country\_name and region\_id and make sure that no duplicate data against column country\_id will be allowed at the time of insertion.

9. Write a PSQL statement to create a table named jobs including columns job\_id, job\_title, min\_salary and max\_salary, and make sure that, the default value for job\_title is blank and min\_salary is 8000 and max\_salary is NULL will be entered automatically at the time of insertion if no value assigned for the specified columns.

10. Write a PSQL statement to create a table named countries including columns country\_id, country\_name and region\_id and make sure that the country\_id column will be a key field which will not contain any duplicate data at the time of insertion.

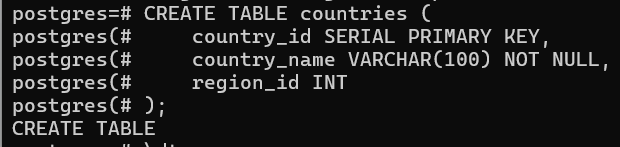
11. Write a PSQL statement to create a table countries including columns country\_id, country\_name and region\_id and make sure that the column country\_id will be unique and store an auto-incremented value.

Click me to see the solution

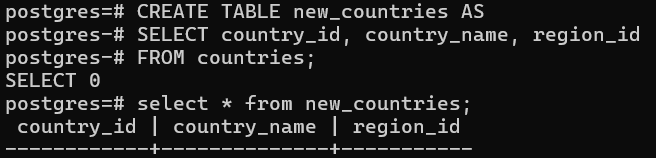
12. Write a PSQL statement to create a table countries including columns country\_id, country\_name and region\_id and make sure that the combination of columns country\_id and region\_id will be unique.

## Source Code & Output:

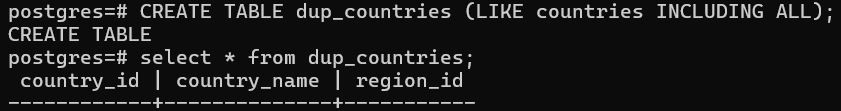
**Task 1:**

****

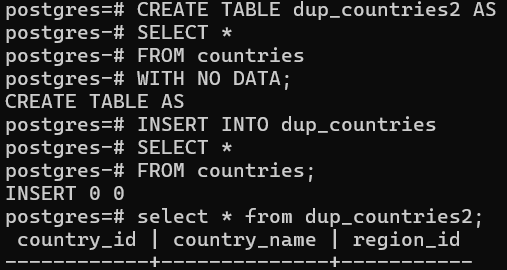
**Task 2:**

****

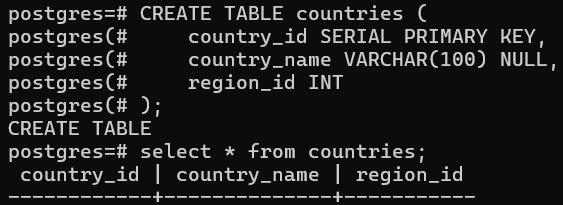
**Task 3:**

****

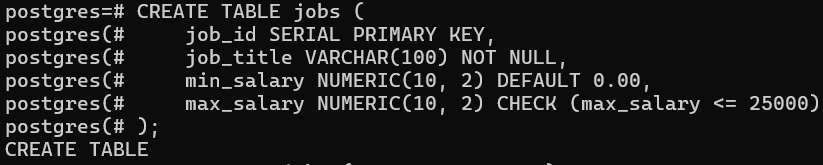
**Task 4:**

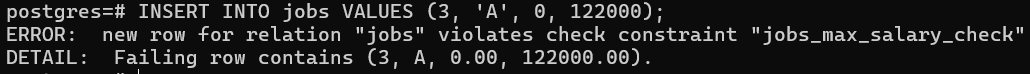
****

**Task 5:**

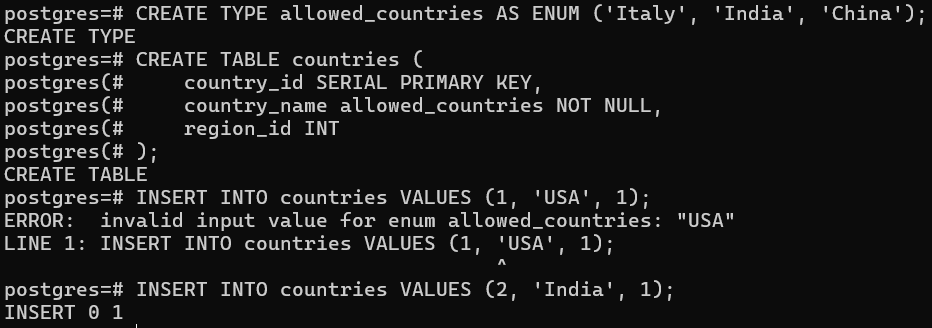
****

**Task 6:**

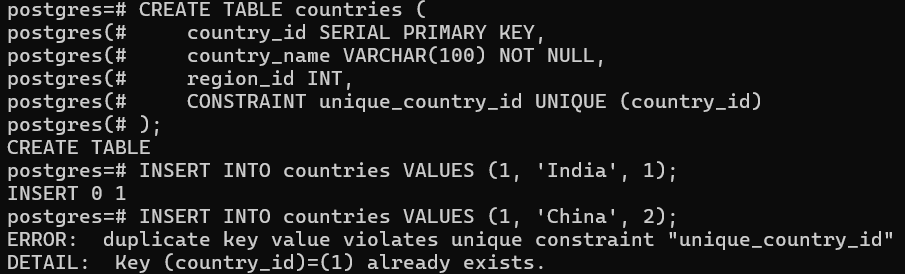
****

****

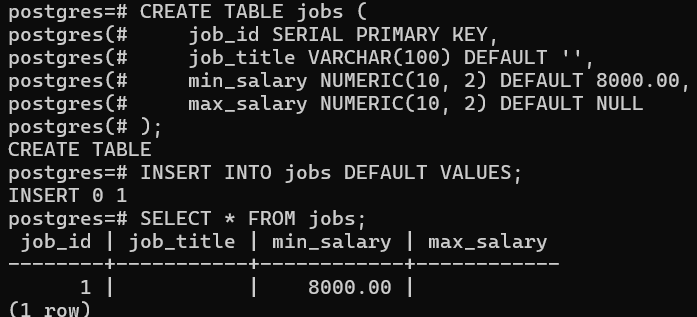
**Task 7:**

****

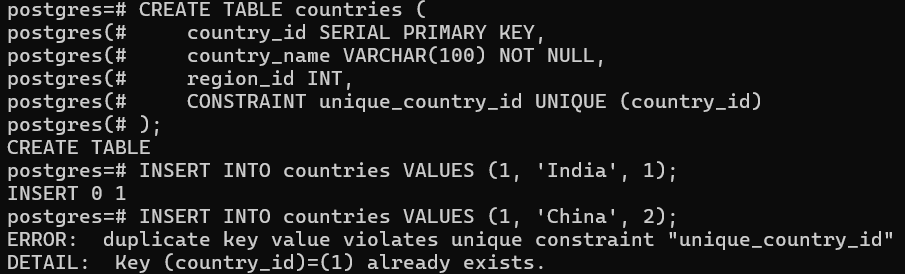
**Task 8:**

****

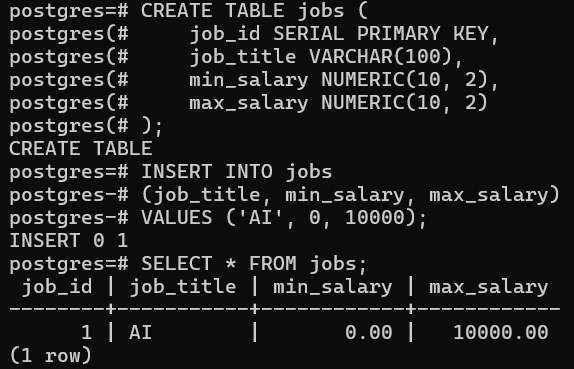
**Task 9:**

****

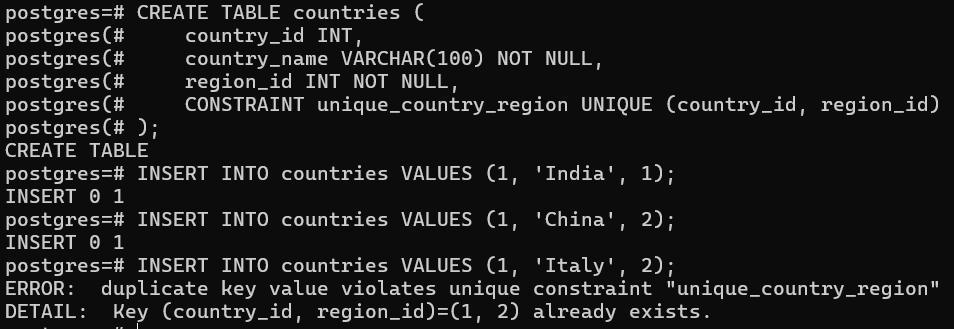
**Task 10:**

****

**Task 11:**

****

**Task 12:**

****

## Conclusion:

* PSQL is PostgreSQL command-line interface for database management.
* Similarity to MySQL language in terms of both statements and structures.
* SERIAL in place of AUTO\_INCREMENT
* Doesn't have an explicit statement for displaying a list of table (but could be done by using command

**\dt**)