Introduction to DBMS

# Introduction to SQL

**Lab 1: Create a new database named school\_db and a table called students with the following columns: student\_id, student\_name, age, class, and address.**

**Answer:**

**Query 1: Create Database**

CREATE DATABASE school\_db;

**Query 2: Use Database**

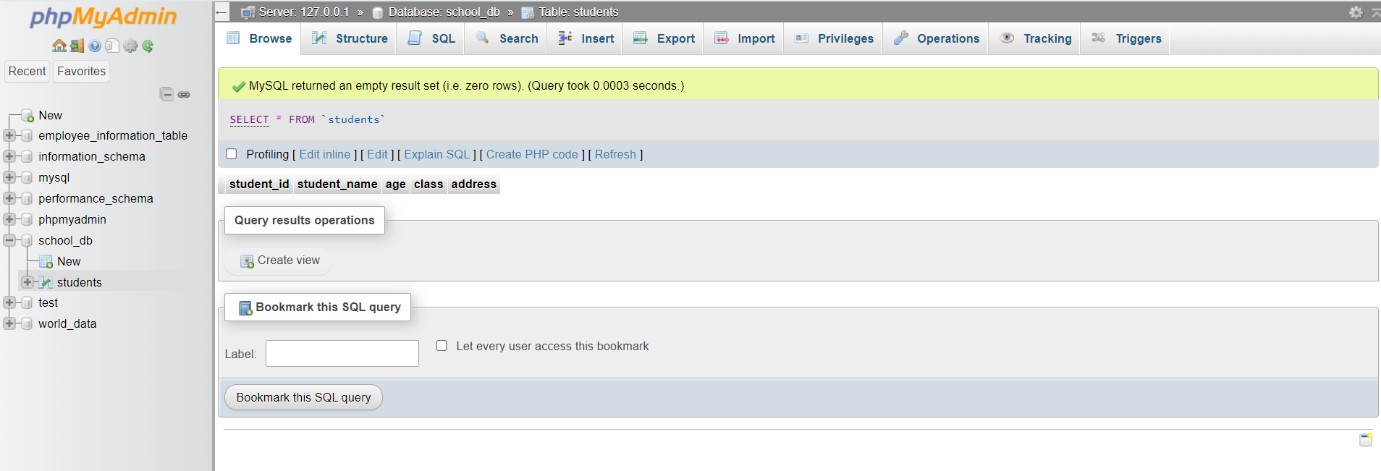
USE school\_db;

**Query 3: Create Table students** CREATE TABLE students ( student\_id INT PRIMARY KEY, student\_name VARCHAR(50),

age INT,

class VARCHAR(20), address VARCHAR(100)

);



**Lab 2: Insert five records into the students table and retrieve all records using the SELECT statement.**

**Answer:**

**Query 1: Insert 5 Records**

INSERT INTO students (student\_id, student\_name, age, class, address) VALUES

(1, 'Raj Patel', 22, '8A', 'Rajkot'),

(2, 'Pratham Patel', 13, '6B', 'Ahmedabad'),

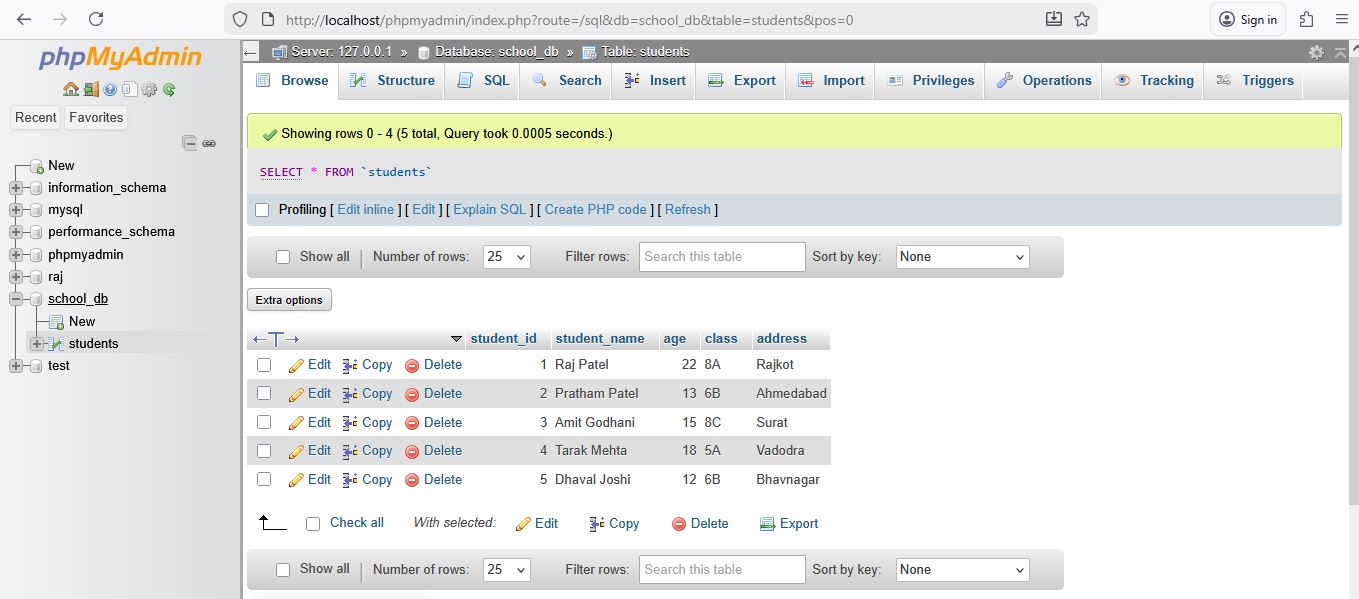
(3, 'Amit Godhani', 15, '8C', 'Surat'),

(4, 'Tarak Mehta', 18, '5A', 'Vadodra'),

(5, 'Dhaval Joshi', 12, '6B', 'Bhavnagar');

**Query 2: Retrieve All Records**

SELECT \* FROM students;



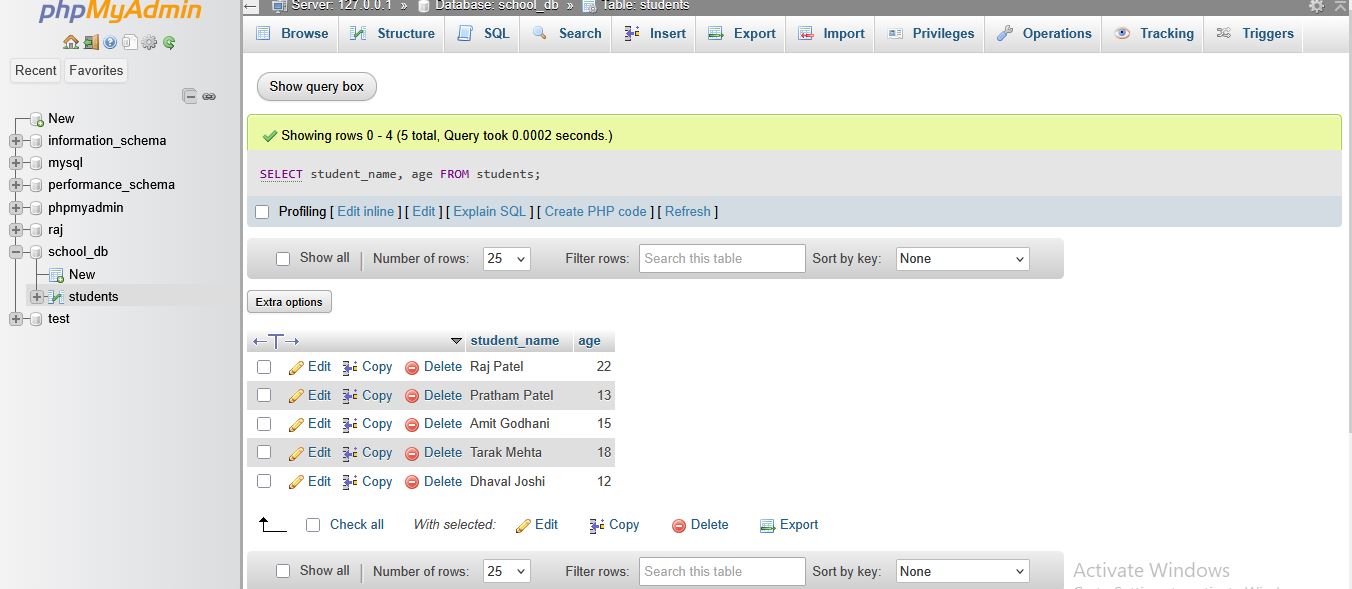
# SQL Syntax

**Lab 1: Write SQL queries to retrieve specific columns (student\_name and age) from the students table.**

**Answer:**

**Query: Retrieve Specific Columns (student\_name and age)**

SELECT student\_name, age FROM students;

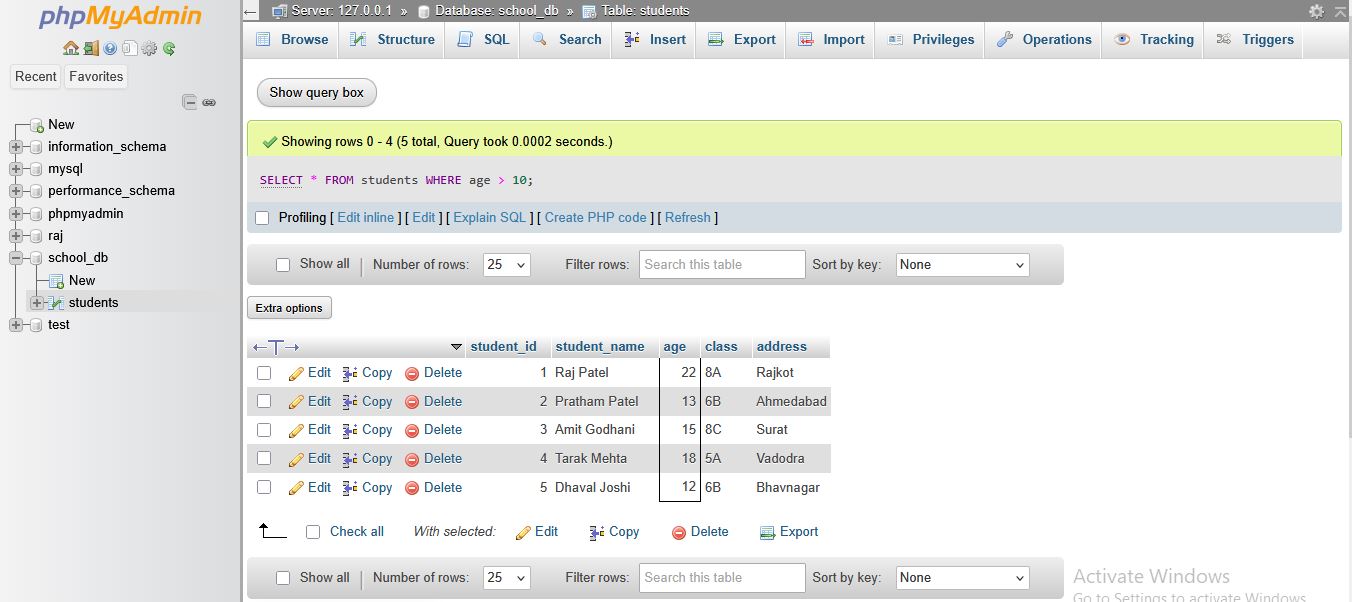


**Lab 2: Write SQL queries to retrieve all students whose age is greater than 10.**

**Answer:**

**Query: Retrieve All Students Whose Age > 10**

SELECT \* FROM students WHERE age > 10;



# SQL Constraints

**Lab 1: Create a table teachers with the following columns: teacher\_id (Primary Key), teacher\_name (NOT NULL), subject (NOT NULL), and email (UNIQUE).**

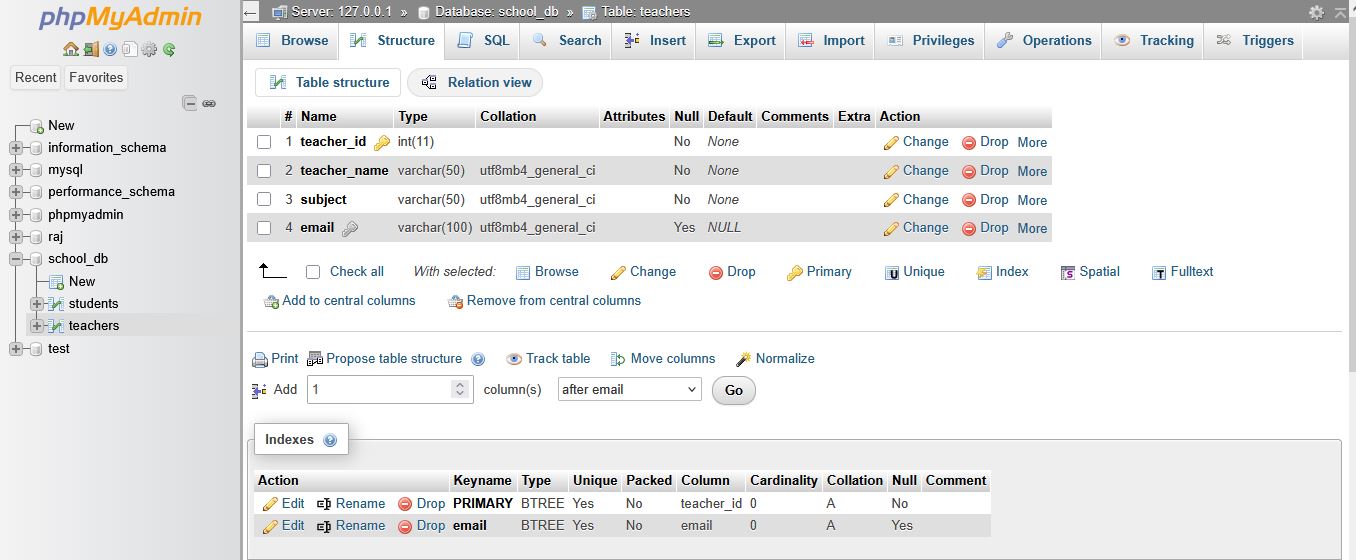
**Answer:**

**Query: Create teachers Table**

CREATE TABLE teachers

( teacher\_id INT PRIMARY KEY, teacher\_name VARCHAR(50) NOT NULL, subject VARCHAR(50) NOT NULL,

email VARCHAR(100) UNIQUE );

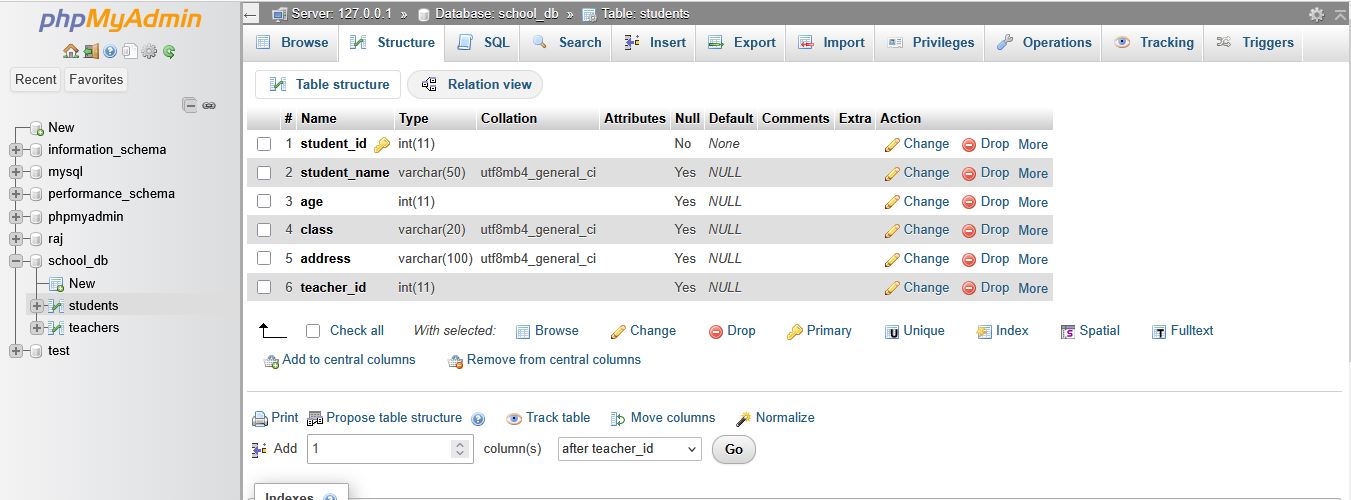


**Lab 2: Implement a FOREIGN KEY constraint to relate the teacher\_id from the teachers table with the students table.**

**Answer:**

**Query 1: Add teacher\_id column to students table**

ALTER TABLE students ADD teacher\_id INT;



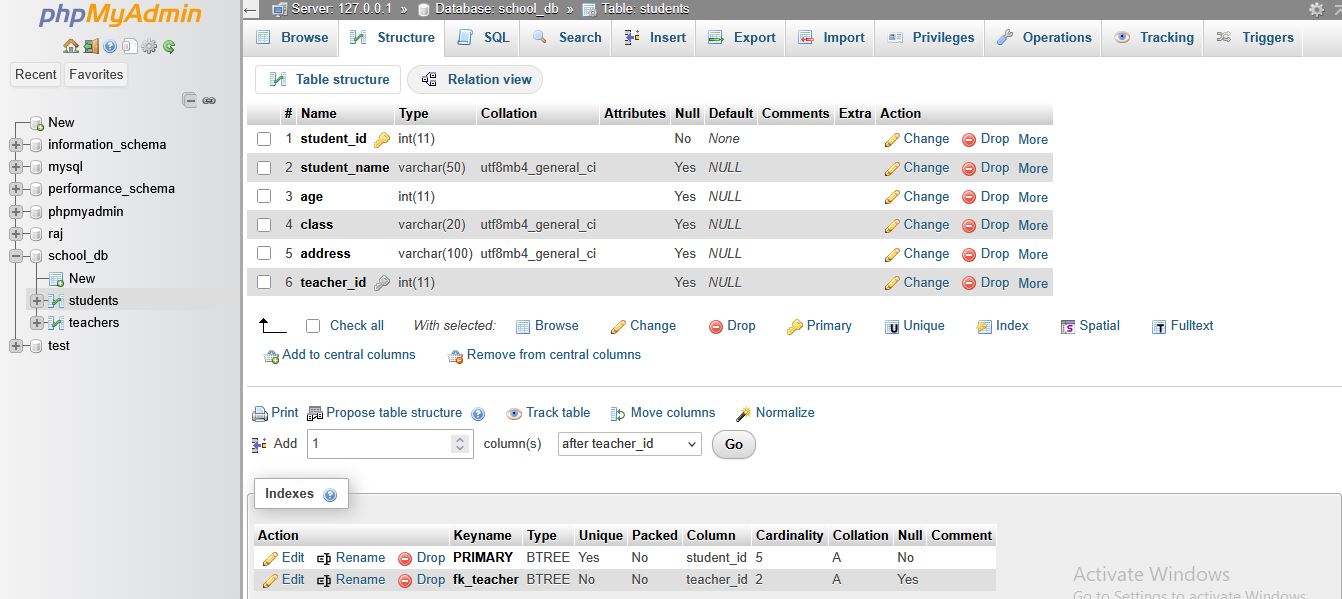
**Query 2: Add FOREIGN KEY Constraint**

ALTER TABLE students

ADD CONSTRAINT fk\_teacher

FOREIGN KEY (teacher\_id)

REFERENCES teachers(teacher\_id);



# Main SQL Commands and Sub-commands (DDL)

**Lab 1: Create a table courses with columns: course\_id, course\_name, and course\_credits. Set the course\_id as the primary key.**

**Answer:**

**Query: Create a table courses** CREATE TABLE courses

( course\_id INT PRIMARY KEY, course\_name VARCHAR(100),

course\_credits INT );

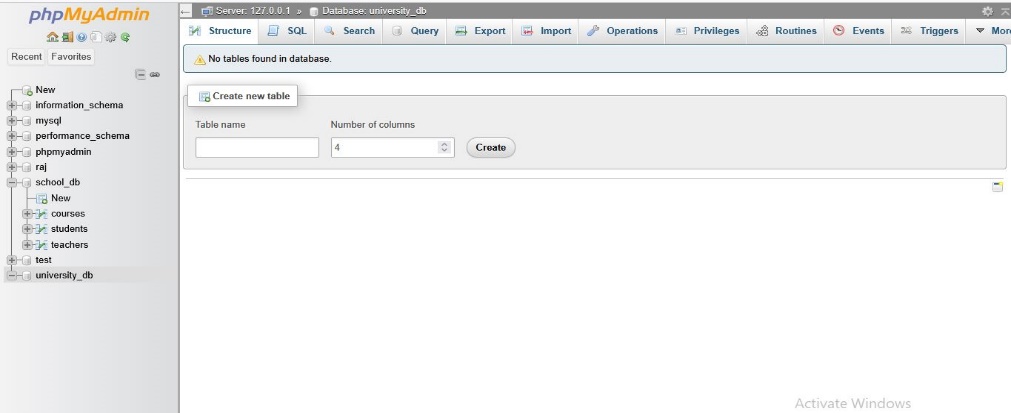


**Lab 2: Use the CREATE command to create a database university\_db.**

**Answer:**

**Query: Create a Database university\_db**

CREATE DATABASE university\_db;



# ALTER Command

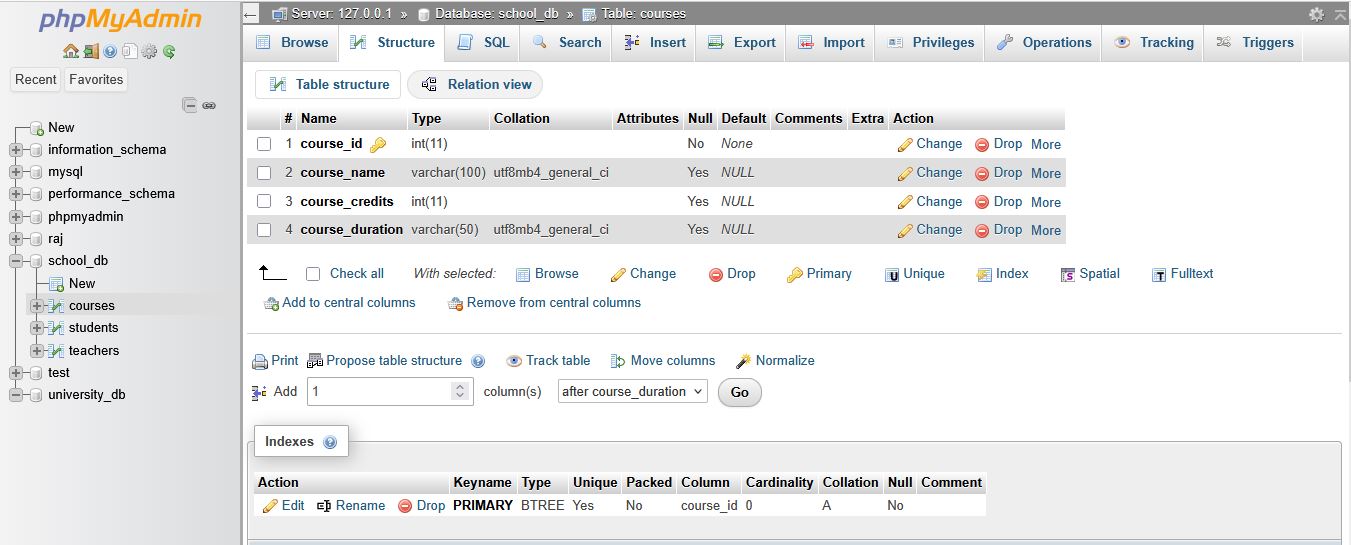
**Lab 1: Modify the courses table by adding a column course\_duration using the ALTER**

**command.**

**Answer:**

**Query: Add a New Column (course\_duration) to courses Table**

ALTER TABLE courses ADD course\_duration VARCHAR(50);

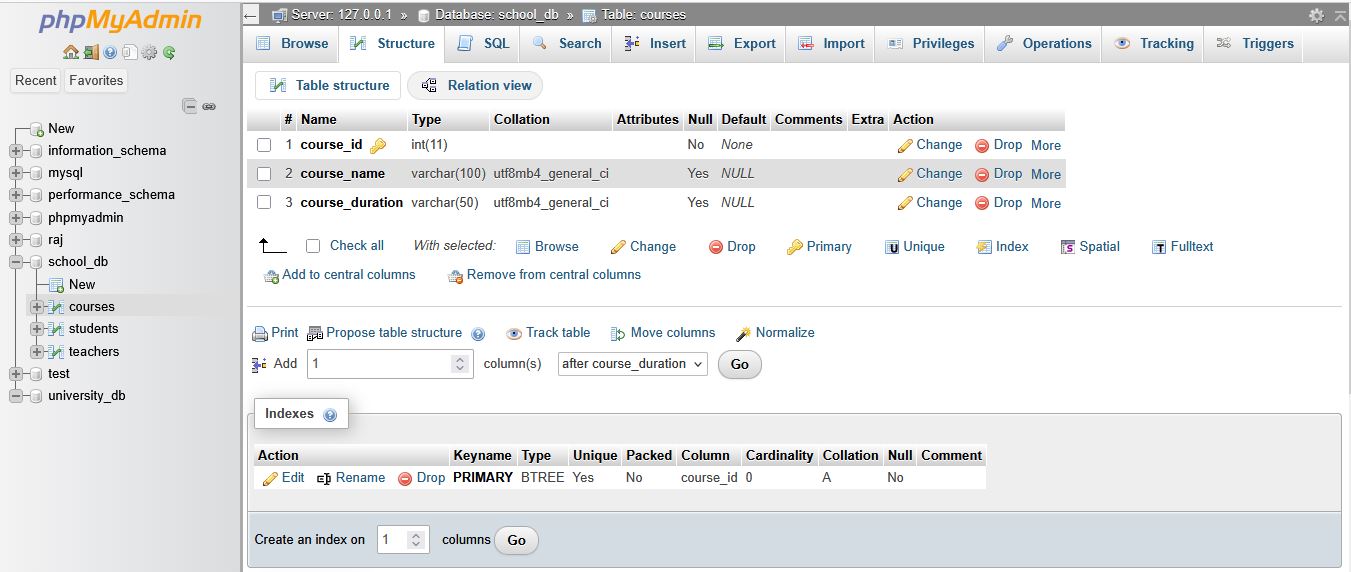


**Lab 2: Drop the course\_credits column from the courses table.**

**Answer:**

**Query: Drop the Column (course\_credits) from courses Table**

ALTER TABLE courses DROP COLUMN course\_credits;



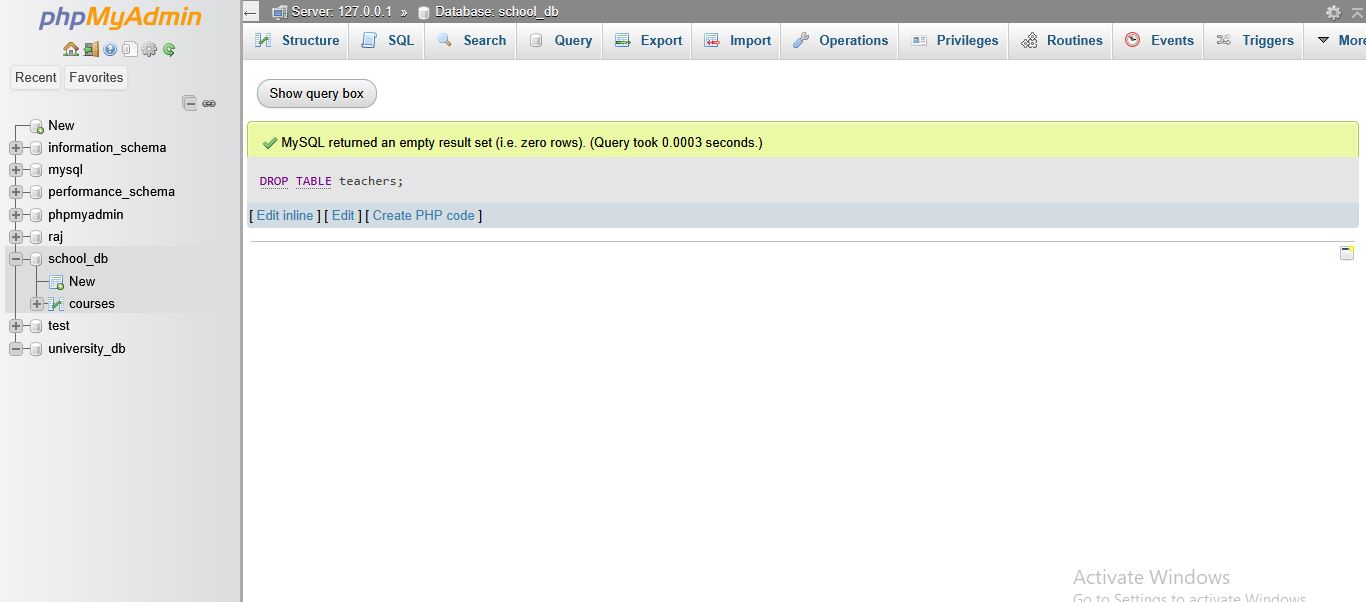
# DROP Command

**Lab 1: Drop the teachers table from the school\_db database.**

**Answer:**

**Query: Drop the teachers table**

DROP TABLE teachers;

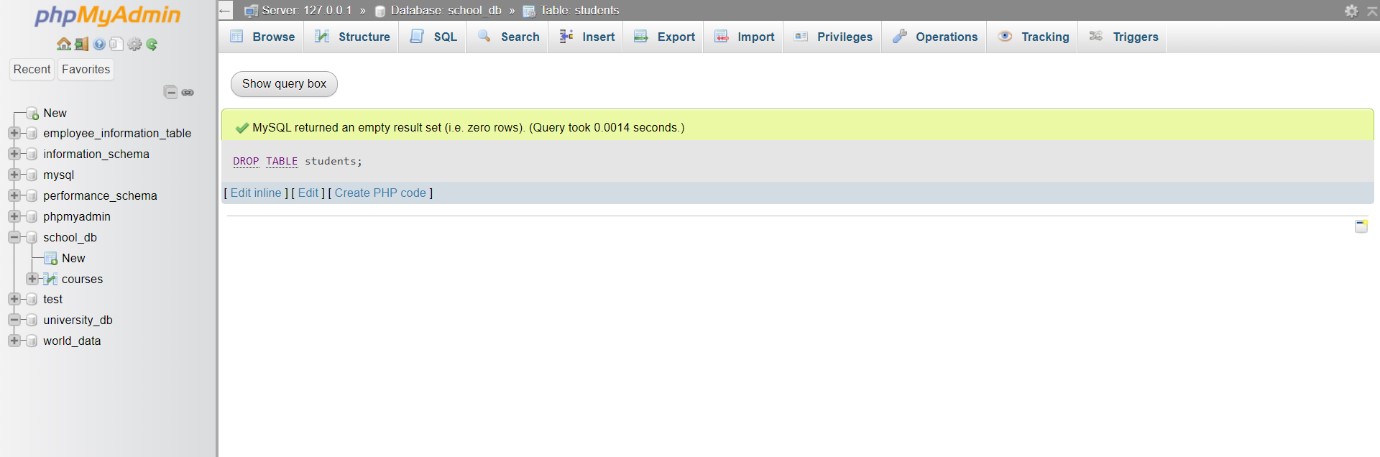


**Lab 2: Drop the students table from the school\_db database and verify that the table has been removed.**

**Answer:**

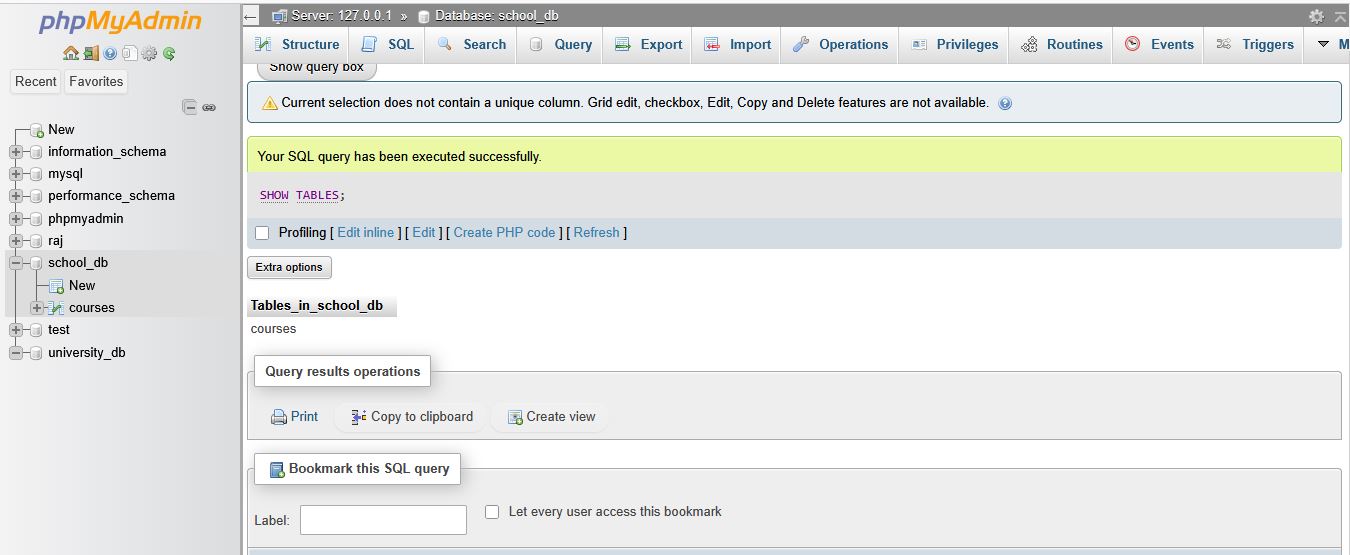
**Query 1: Drop the students table and verify removal**

DROP TABLE students;



**Query 2: Verify that the table is removed**

SHOW TABLES;



# Data Manipulation Language (DML)

**Lab 1: Insert three records into the courses table using the INSERT command.**

**Answer:**

**Query: Insert 3 Records into courses Table**

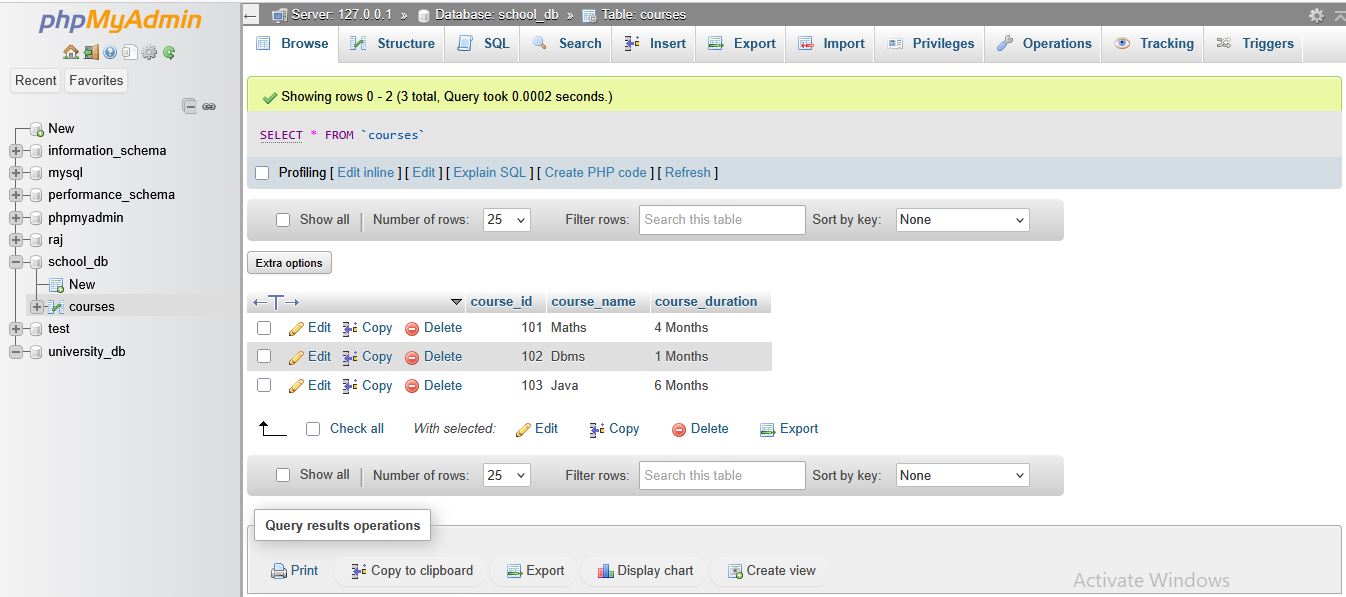
INSERT INTO courses (course\_id, course\_name, course\_duration)

VALUES

(101, 'Maths', '4 Months'),

(102, 'Dbms', '1 Months'),

(103, 'Java', '6 Months');



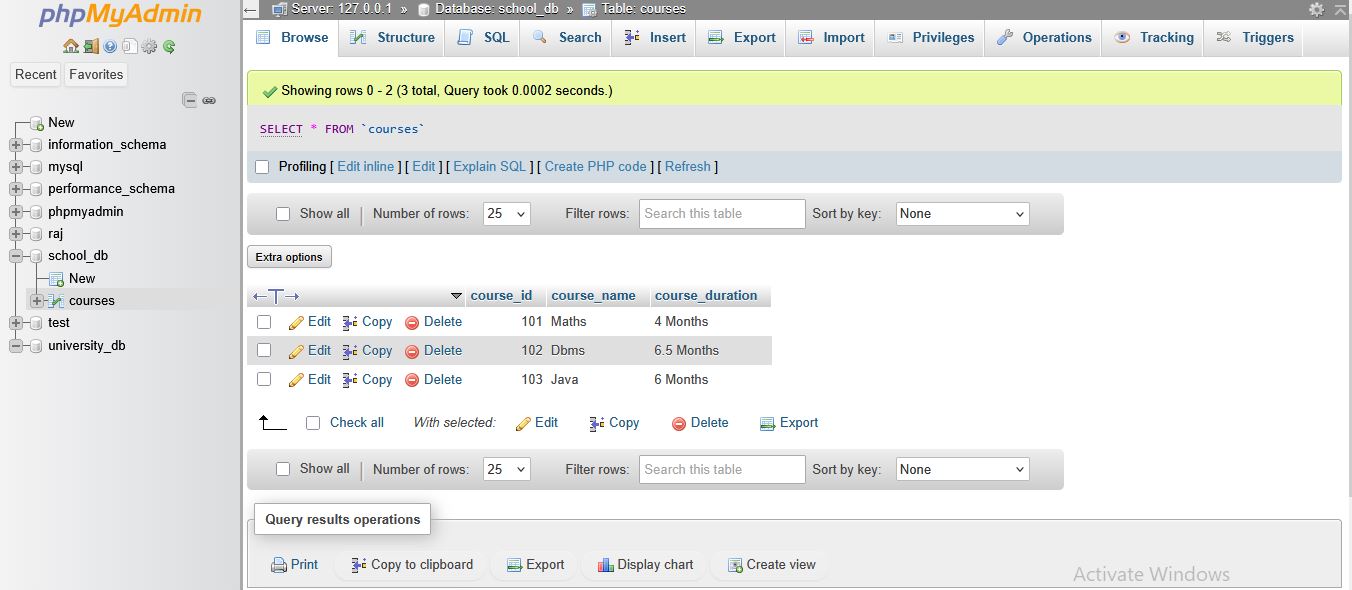
**Lab 2: Update the course duration of a specific course using the UPDATE command.**

**Answer:**

**Query: Update Course Duration of a Specific Course**

**Example:** Update duration of course\_id = 103

UPDATE courses SET course\_duration = '6.5 Months' WHERE course\_id = 102;



**Lab 3: Delete a course with a specific course\_id from the courses table using the DELETE**

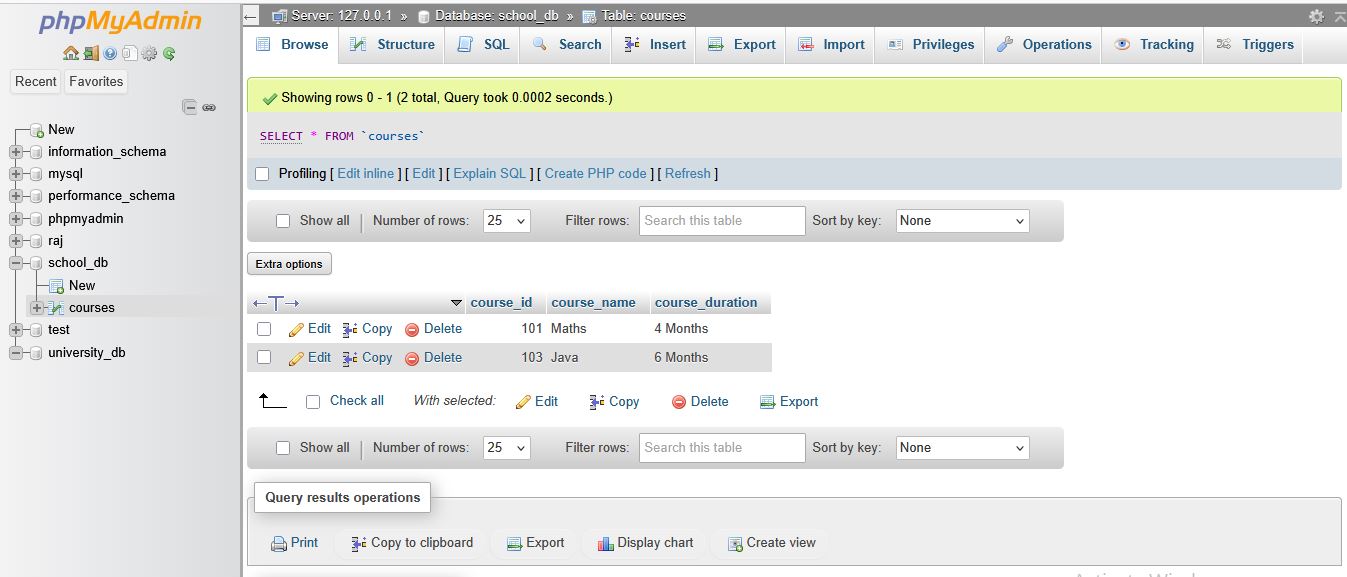
**command.**

**Answer:**

**Query: Delete a course using DELETE Command**

**Example:** Delete course with course\_id = 102

DELETE FROM courses WHERE course\_id = 102;



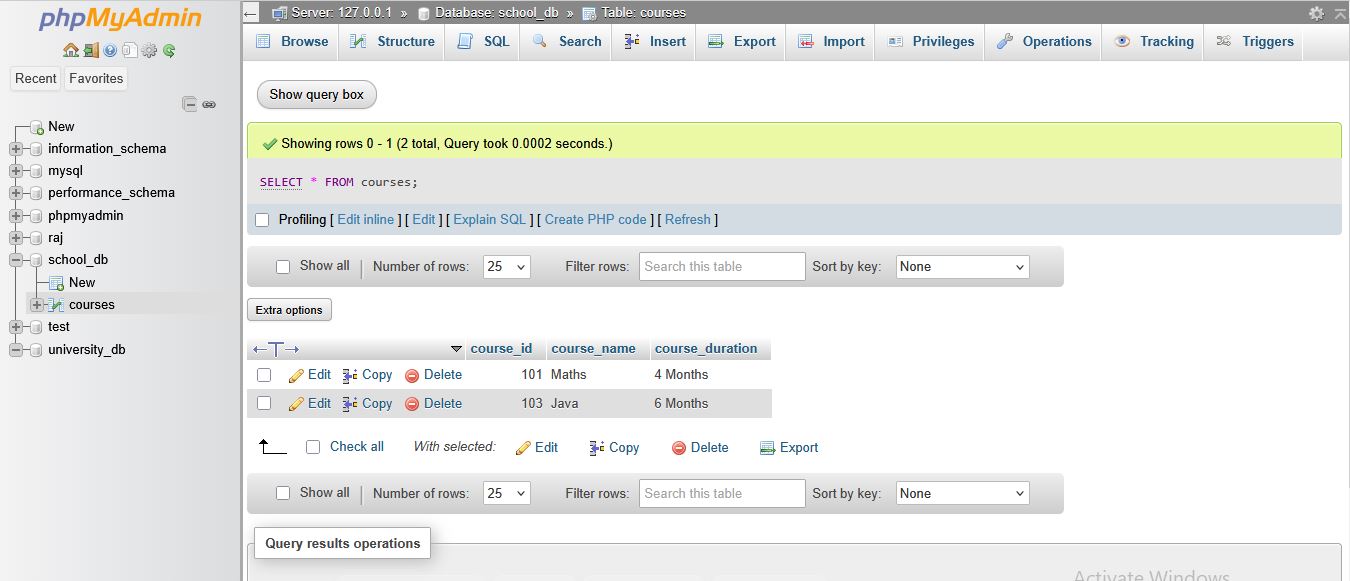
# Data Query Language (DQL)

**Lab 1: Retrieve all courses from the courses table using the SELECT statement.**

**Answer:**

**Query: Retrieve all courses**

SELECT \* FROM courses;

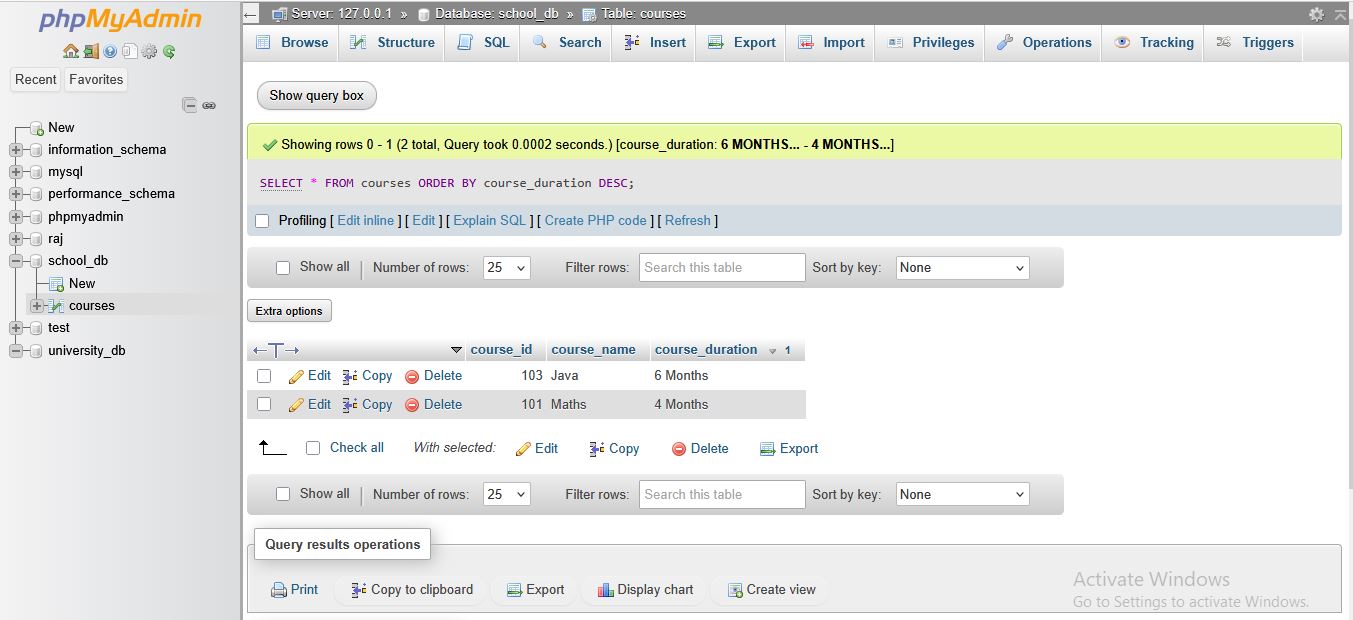


**Lab 2: Sort the courses based on course\_duration in descending order using ORDER BY.**

**Answer:**

**Query: Sort Courses by course\_duration (Descending Order)**

SELECT \* FROM courses ORDER BY course\_duration DESC;

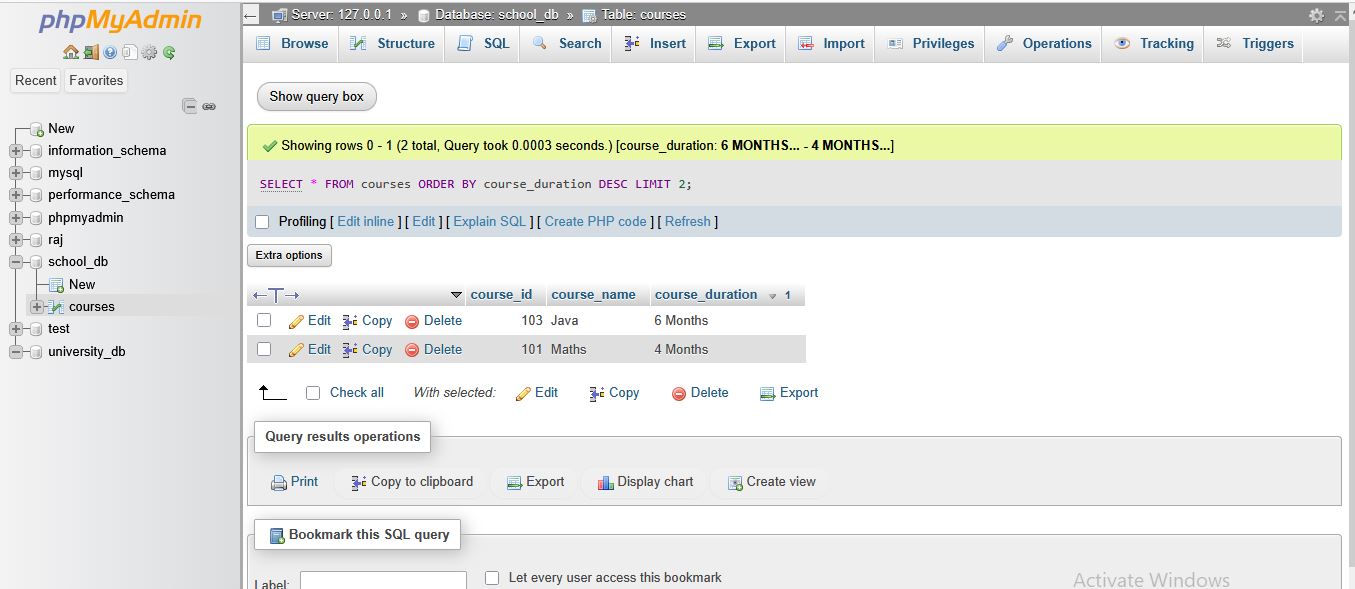


**Lab 3: Limit the results of the SELECT query to show only the top two courses using LIMIT.**

**Answer:**

**Query: Show only top 2 courses using LIMIT**

SELECT \* FROM courses ORDER BY course\_duration DESC LIMIT 2;



# Data Control Language (DCL)

**Lab 1: Create two new users user1 and user2 and grant user1 permission to SELECT from the courses table.**

**Answer:**

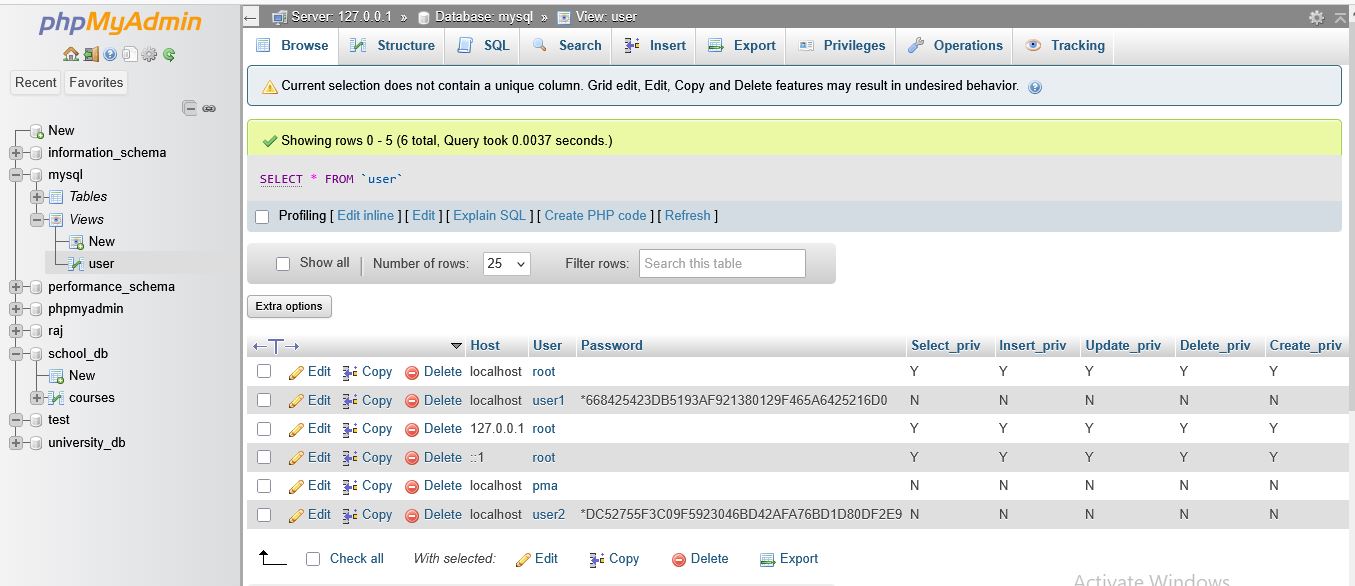
**Query 1: Create two new users user1 and user2**

**Step 1: Create user1**

CREATE USER 'user1'@'localhost' IDENTIFIED BY 'password1';

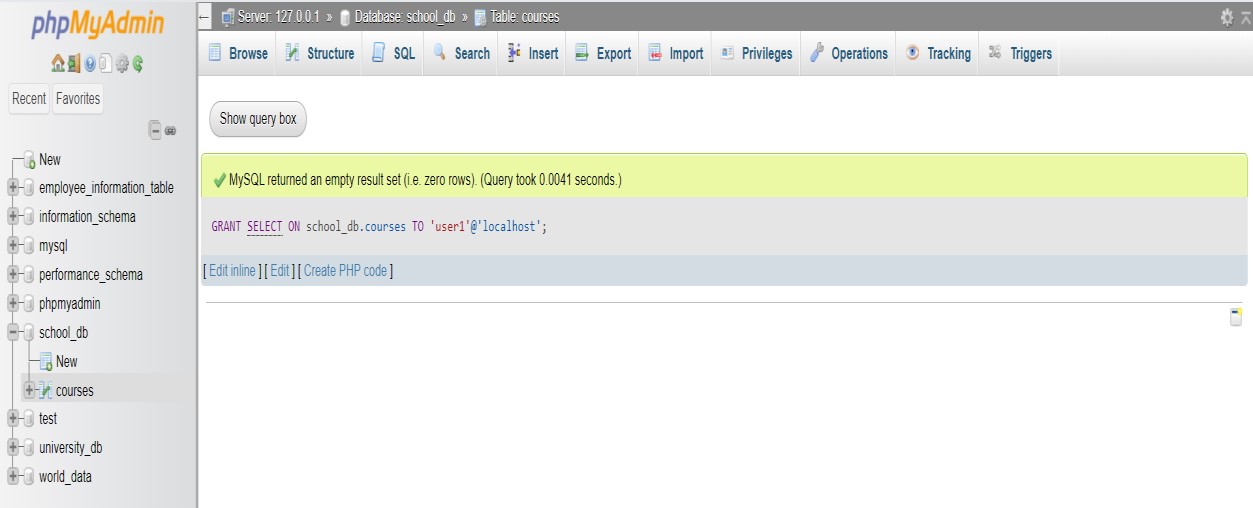
**Step 2: Create user2**

CREATE USER 'user2'@'localhost' IDENTIFIED BY 'password2';



**Query 2: Grant SELECT permission on courses table to user1**

GRANT SELECT ON school\_db.courses TO 'user1'@'localhost';



**Lab 2: Revoke the INSERT permission from user1 and give it to user2.**

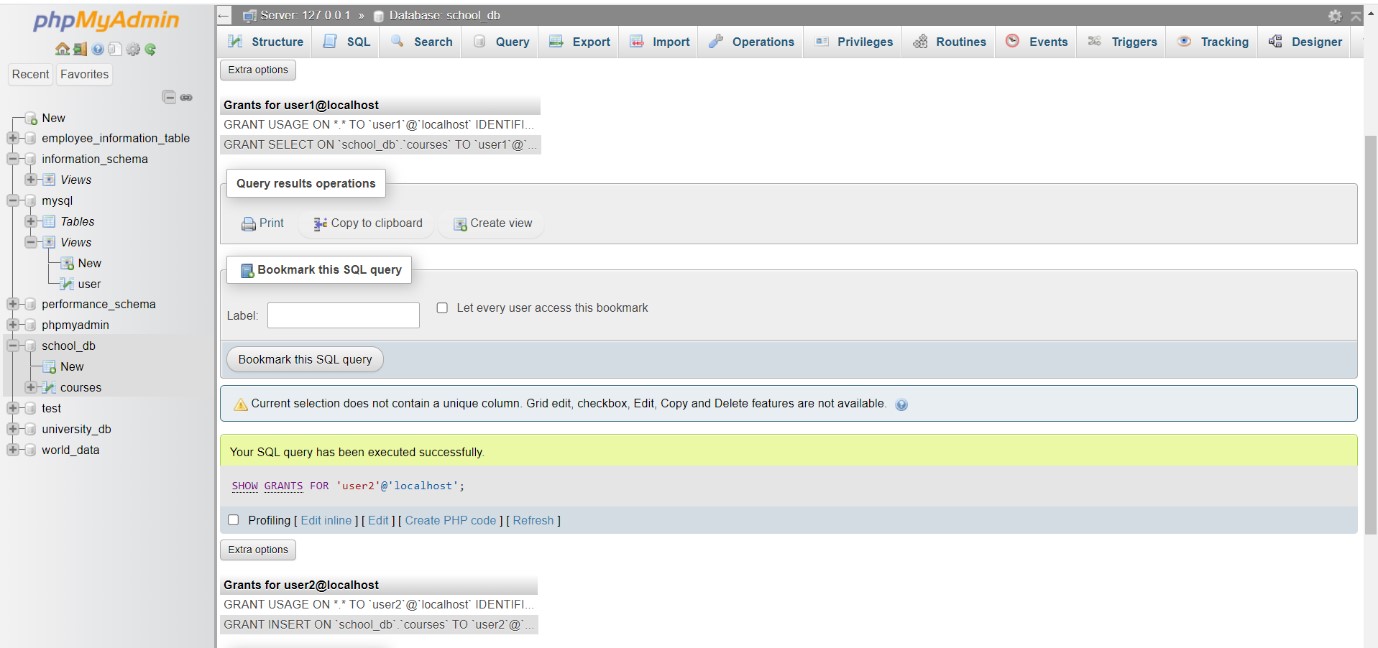
**Answer:**

**Query 1: Revoke INSERT permission from user1**

REVOKE INSERT ON school\_db.courses FROM 'user1'@'localhost';

**Query 2: Grant INSERT permission to user2**

GRANT INSERT ON school\_db.courses TO 'user2'@'localhost';



# Transaction Control Language (TCL)

**Lab 1: Insert a few rows into the courses table and use COMMIT to save the changes.**

**Answer:**

**Query: Insert a few rows + COMMIT**

START TRANSACTION;

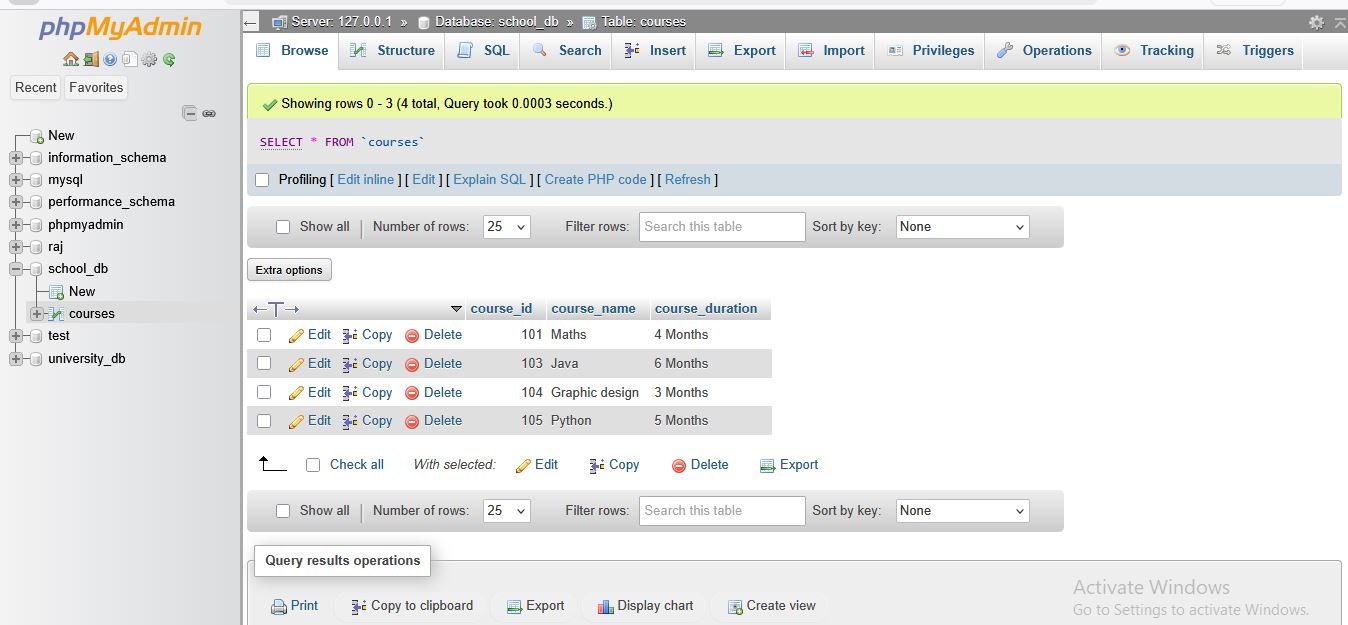
INSERT INTO courses (course\_id, course\_name, course\_duration)

VALUES

(104, ' Graphic design, '3 Months'),

(105, 'Python', '5 Months');

COMMIT;



* Rows **104 and 105** will appear permanently.

**Lab 2: Insert additional rows, then use ROLLBACK to undo the last insert operation.**

**Answer:**

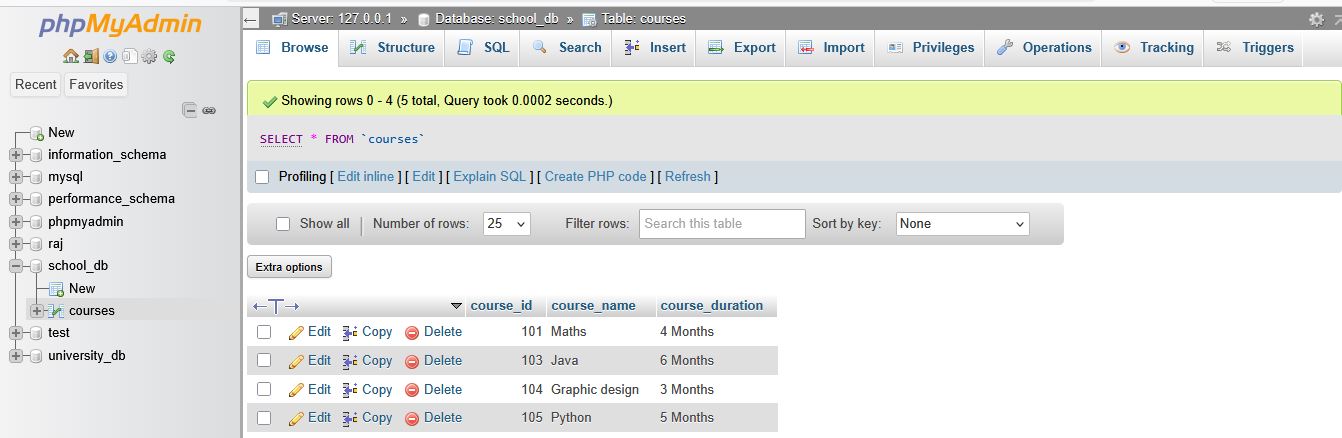
**Query: Insert additional rows + ROLLBACK**

START TRANSACTION;

INSERT INTO courses (course\_id, course\_name, course\_duration)

VALUES (106, 'Cloud Computing', '5 Months');

ROLLBACK;



* **Course\_id 106** will **NOT appear** because rollback cancelled the insert.

**Lab 3: Create a SAVEPOINT before updating the courses table, and use it to roll back specific changes.**

**Answer:**

**Query: SAVEPOINT + ROLLBACK TO SAVEPOINT**

START TRANSACTION;

-- Create savepoint

SAVEPOINT before\_update;

-- Update a course

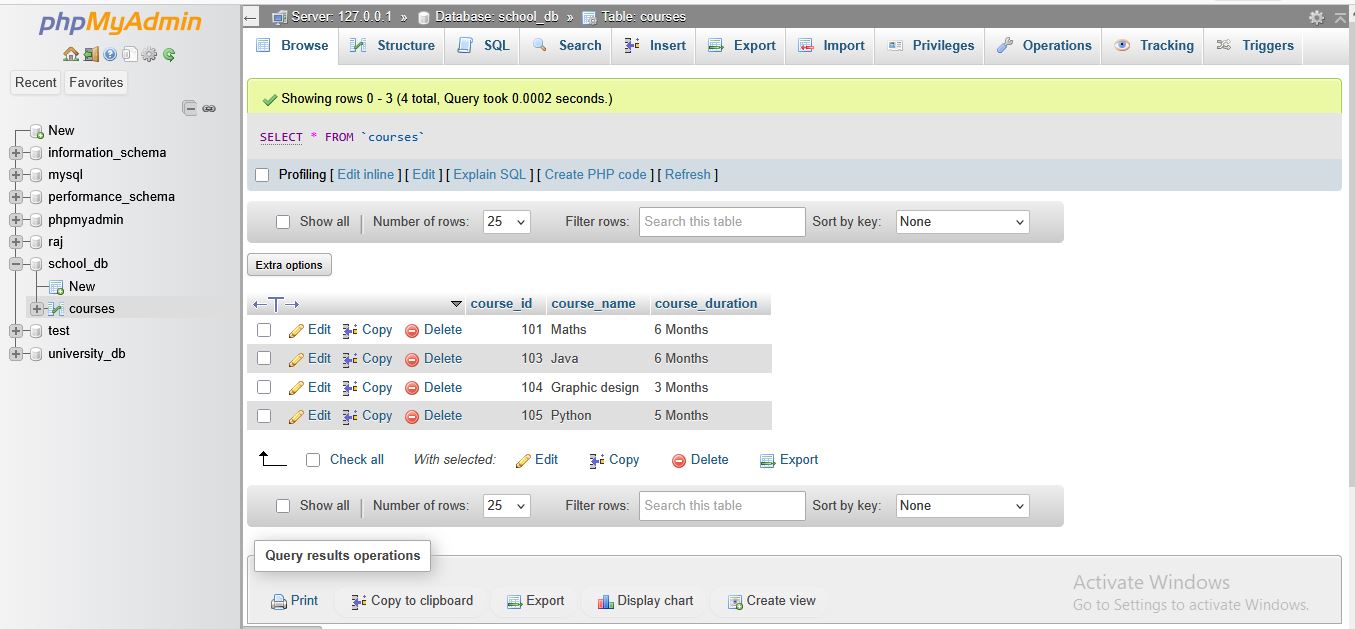
UPDATE courses SET course\_duration = '6 Months' WHERE course\_id = 101;

-- Rollback only this update

ROLLBACK TO before\_update;

-- Commit remaining changes

COMMIT;



• Course\_id 101 duration will remain ‘3 Months’ (rollback undo the update)

# SQL Joins

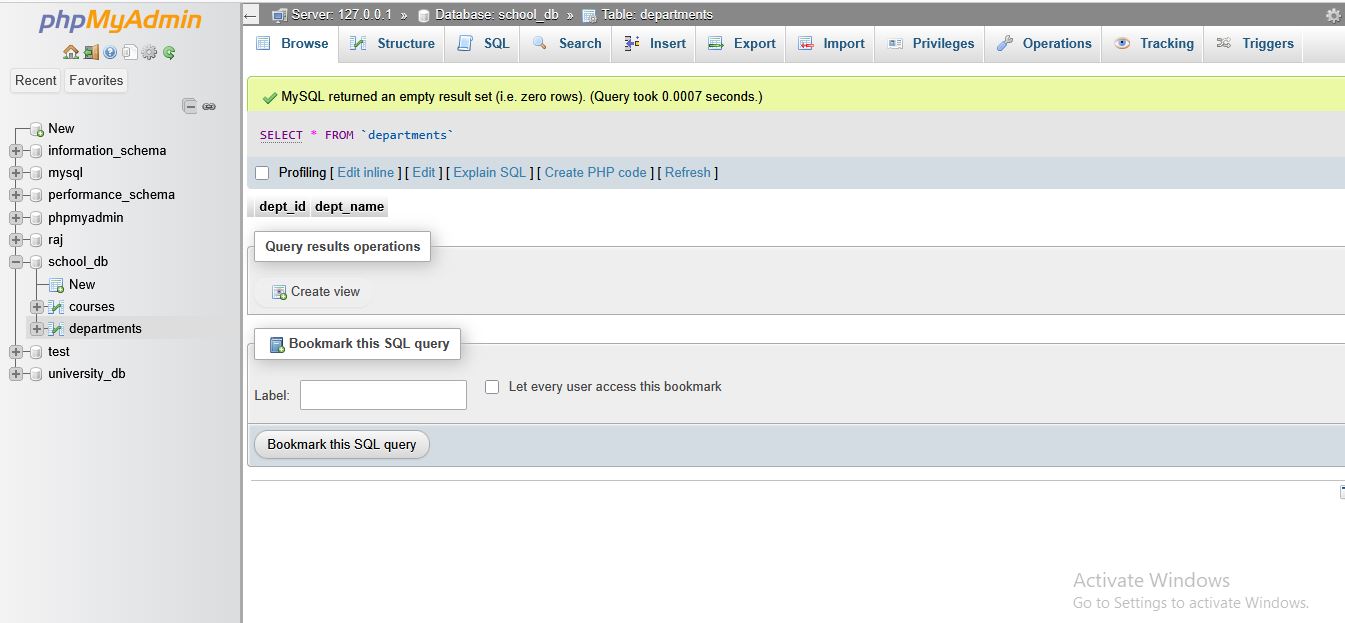
**Lab 1: Create two tables: departments and employees. Perform an INNER JOIN to display employees along with their respective departments.**

**Answer:**

**Query 1: Create two tables**

1. CREATE TABLE departments ( dept\_id INT PRIMARY KEY, dept\_name VARCHAR(50)

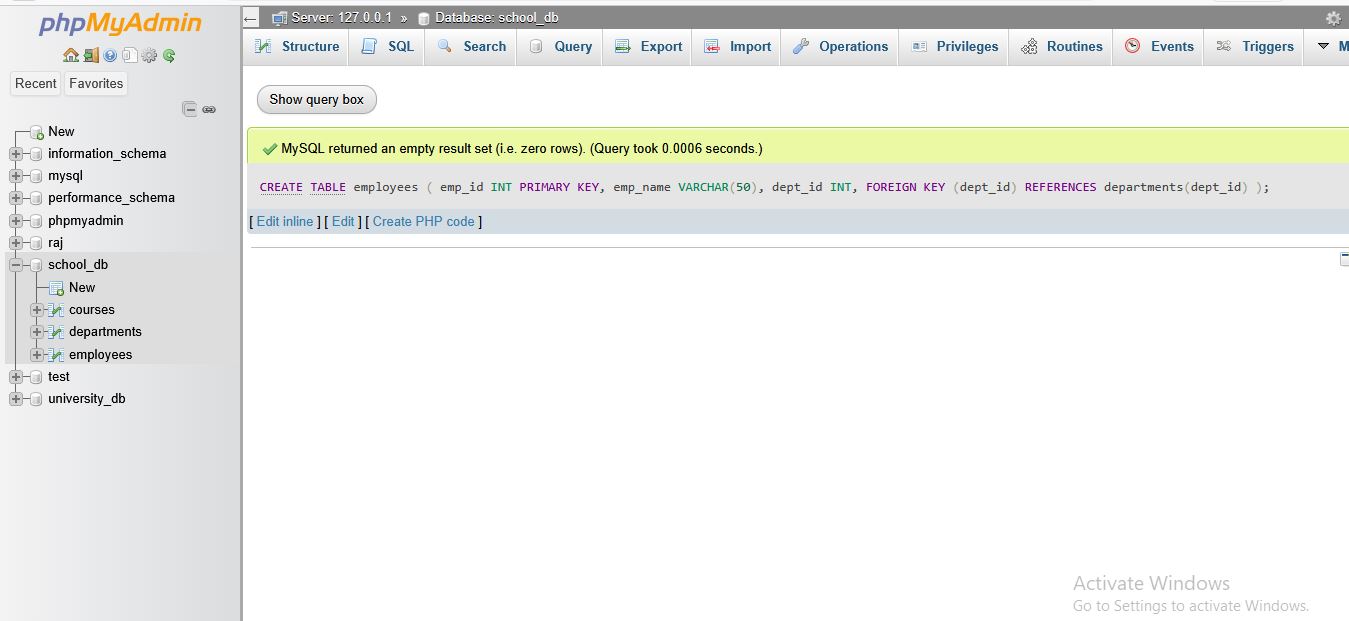
);



1. CREATE TABLE employees
2. ( emp\_id INT PRIMARY KEY, emp\_name VARCHAR(50),

dept\_id INT,

FOREIGN KEY (dept\_id) REFERENCES departments(dept\_id) );



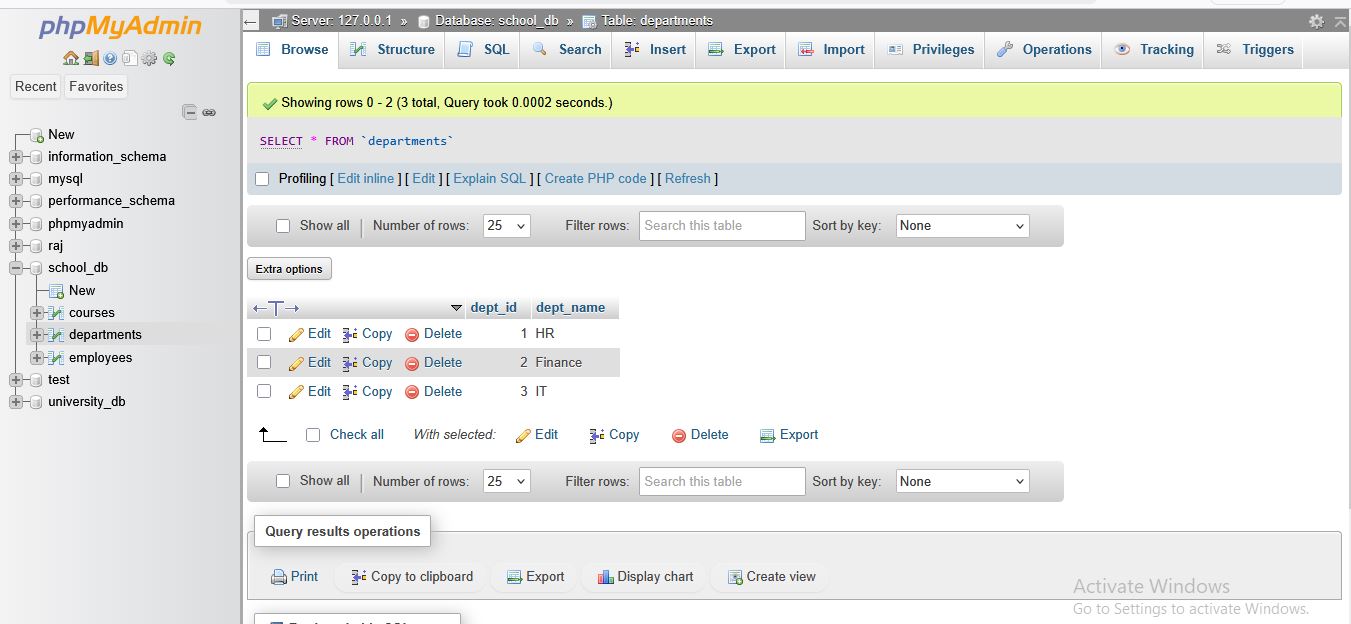
**Query 2: Insert sample records**

1. INSERT INTO departments VALUES

(1, 'HR'),

(2, 'Finance'),

(3, 'IT');

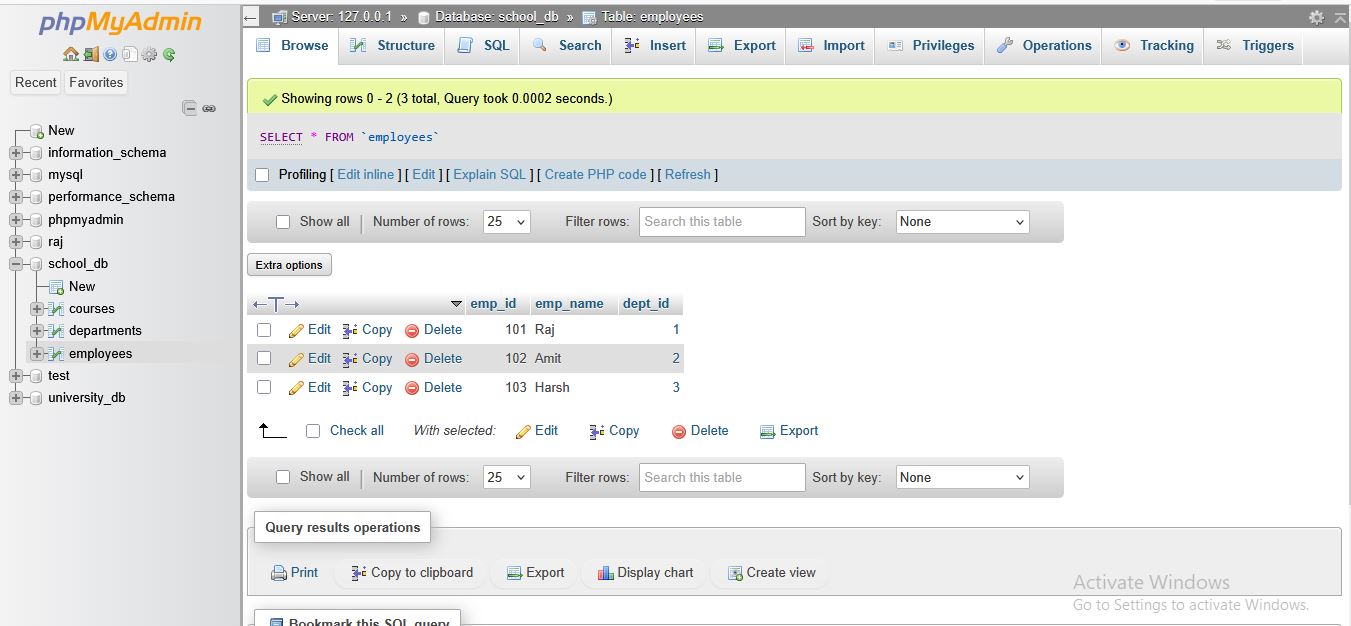


1. INSERT INTO employees VALUES

(101, 'Raj', 1),

(102, 'Amit', 2),

(103, 'Harsh', 3);

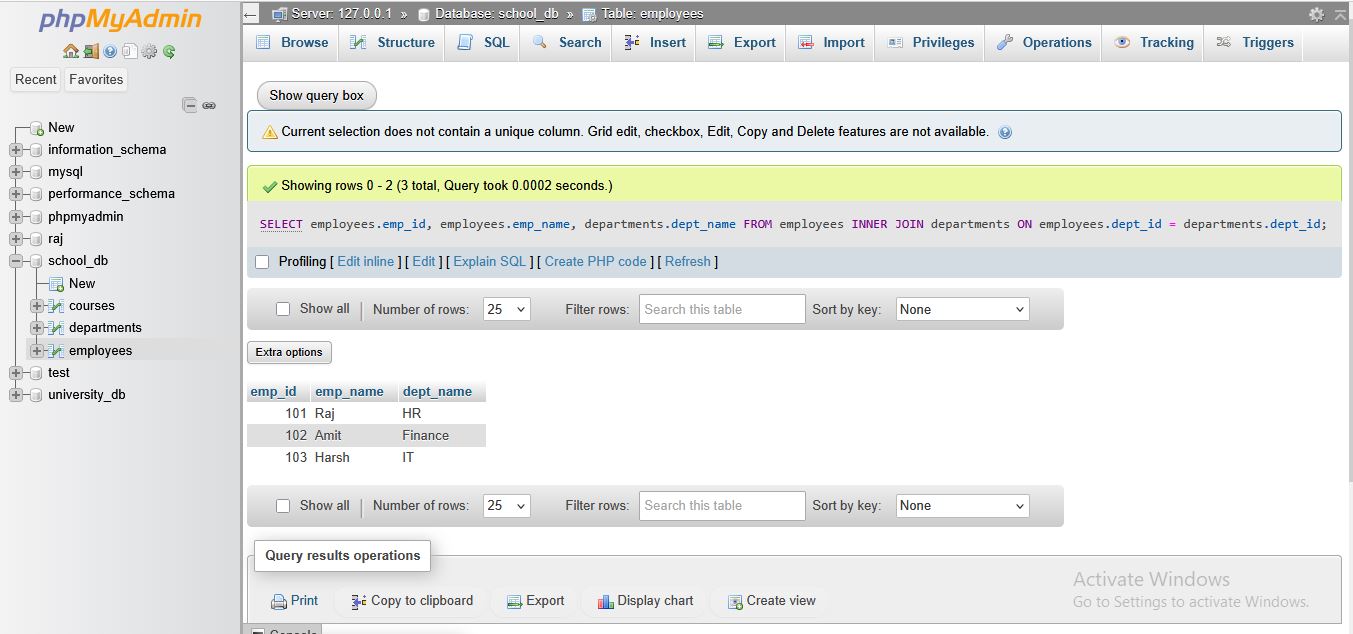


**Query 3: Perform INNER JOIN**

SELECT employees.emp\_id, employees.emp\_name, departments.dept\_name

FROM employees

INNER JOIN departments ON employees.dept\_id = departments.dept\_id;



• INNER JOIN shows **only matching** employee – department records.

**Lab 2: Use a LEFT JOIN to show all departments, even those without employees.**

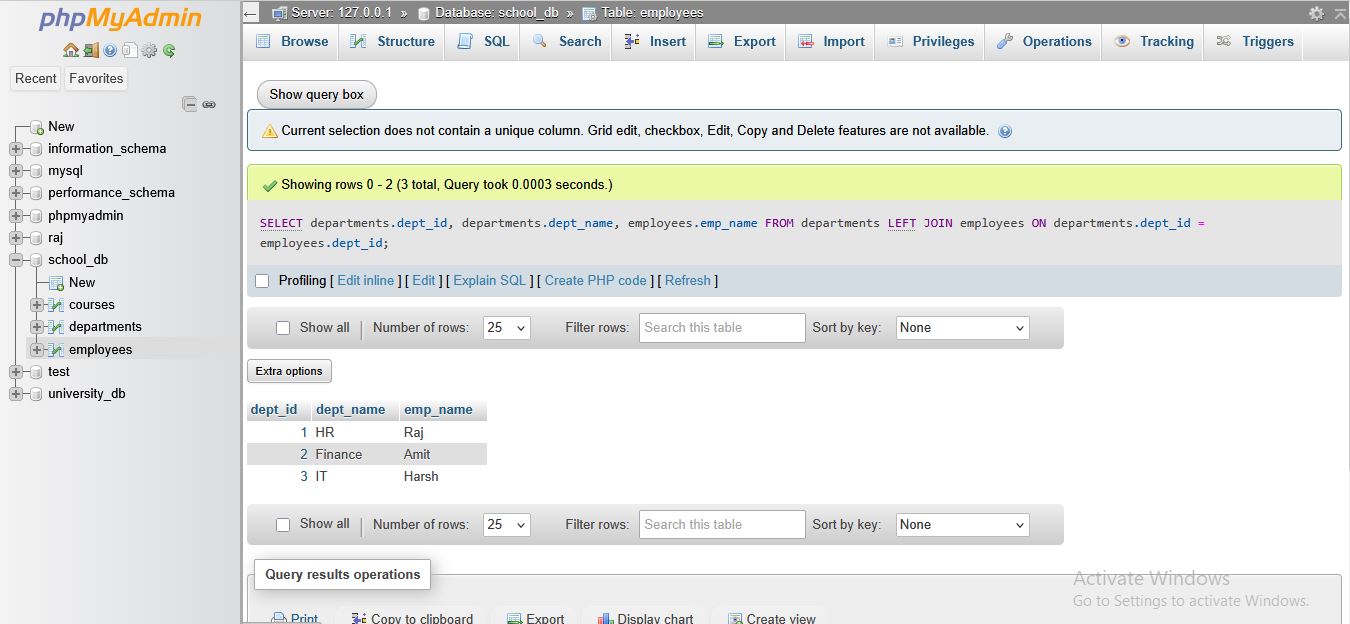
**Answer:**

**Query: LEFT JOIN (Show all departments even without employees)**

SELECT departments.dept\_id, departments.dept\_name, employees.emp\_name

FROM departments

LEFT JOIN employees ON departments.dept\_id = employees.dept\_id;



# SQL Group By

**Lab 1: Group employees by department and count the number of employees in each department using GROUP BY.**

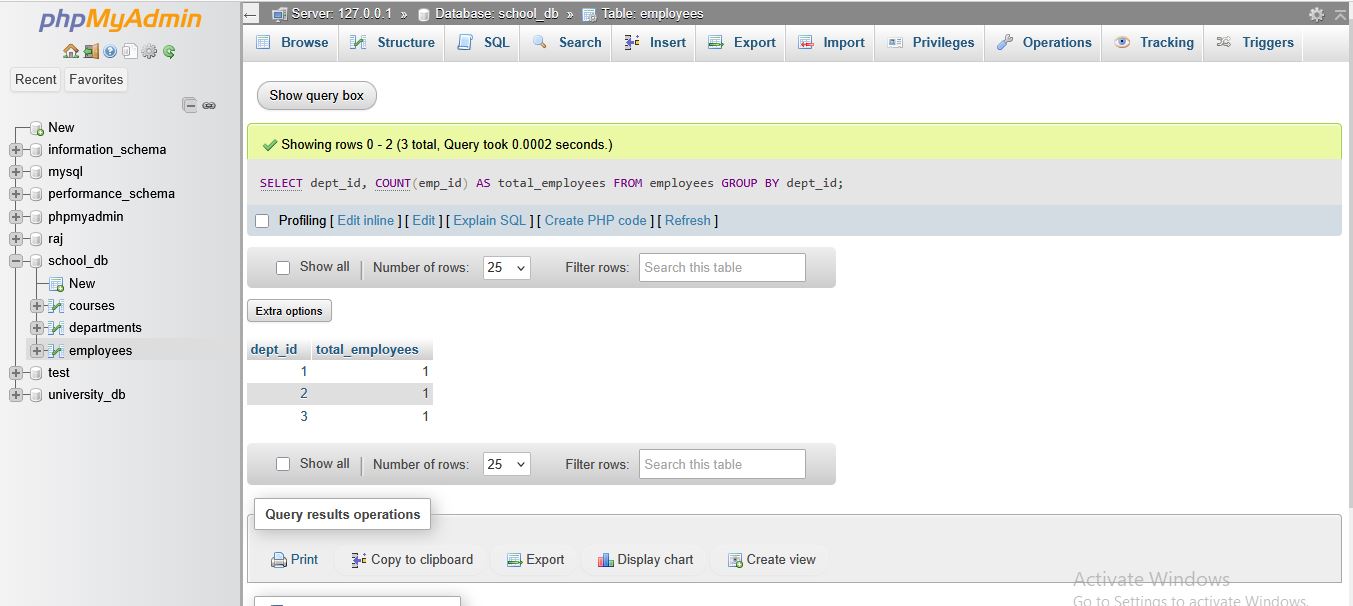
**Answer:**

**Query: GROUP BY – Count Employees in Each Department**

SELECT dept\_id, COUNT(emp\_id) AS total\_employees

FROM employees

GROUP BY dept\_id;



**Lab 2: Use the AVG aggregate function to find the average salary of employees in each department.**

**Answer:**

**Important note:**

* **To use Lab 2, your employees table must have a salary column, e.g.:**

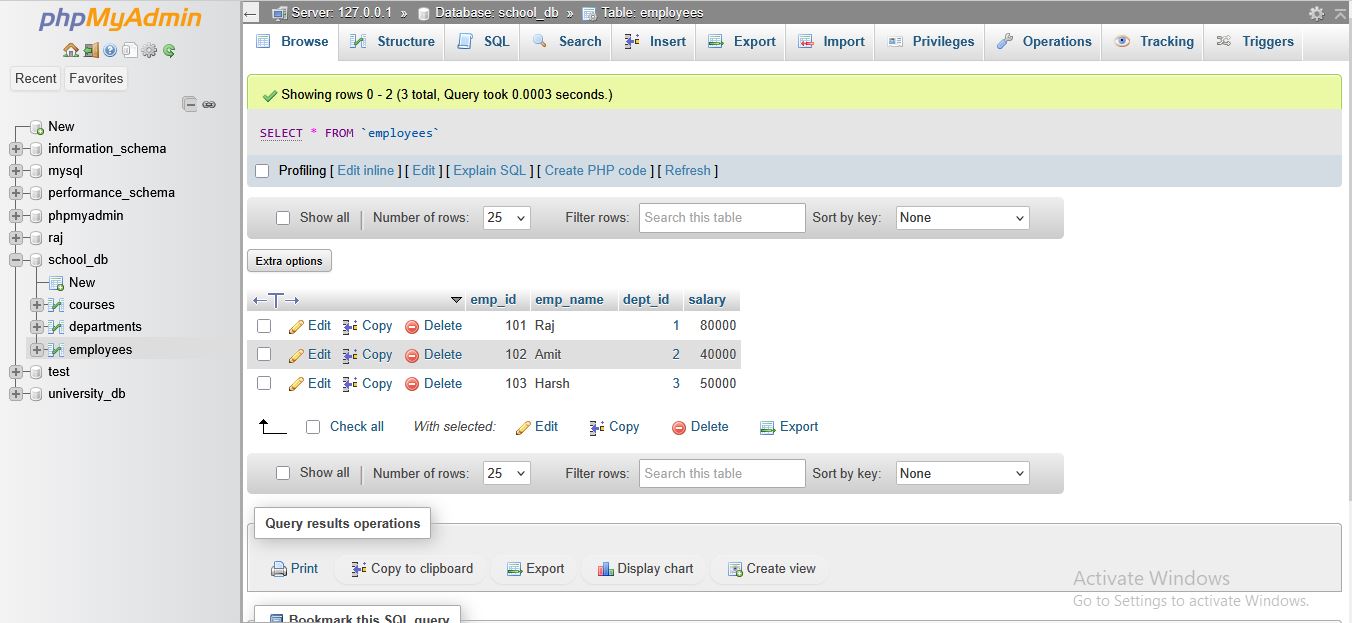
ALTER TABLE employees ADD salary INT;

* **And insert/update salaries:**

UPDATE employees SET salary = 80000 WHERE emp\_id = 101;

UPDATE employees SET salary = 40000 WHERE emp\_id = 102;

UPDATE employees SET salary = 50000 WHERE emp\_id = 103;

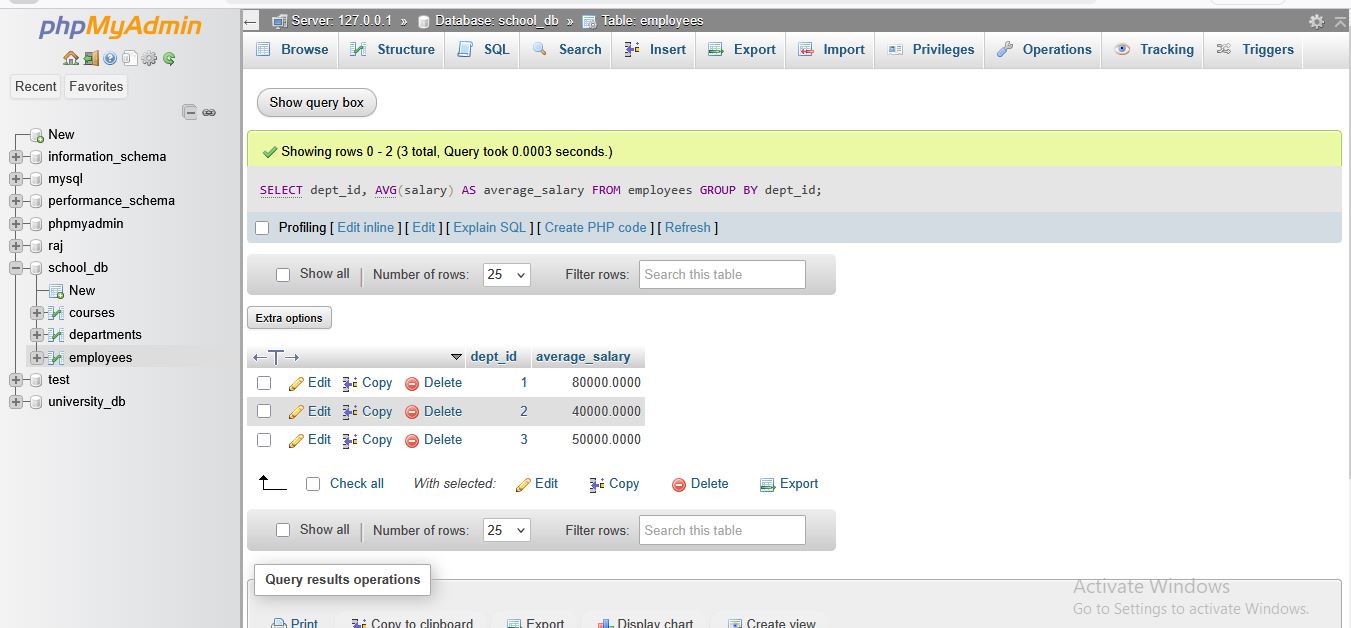


**Query: Find Average Salary per Department (Using AVG)**

SELECT dept\_id, AVG(salary) AS average\_salary

FROM employees

GROUP BY dept\_id;



# SQL Stored Procedure

**Lab 1: Write a stored procedure to retrieve all employees from the employees table based on department.**

**Answer:**

**Query: Stored Procedure to Retrieve Employees by Department**

* **Stored Procedure**

DELIMITER $$

CREATE PROCEDURE GetEmployeesByDepartment(IN dept INT)

BEGIN

SELECT emp\_id, emp\_name, dept\_id

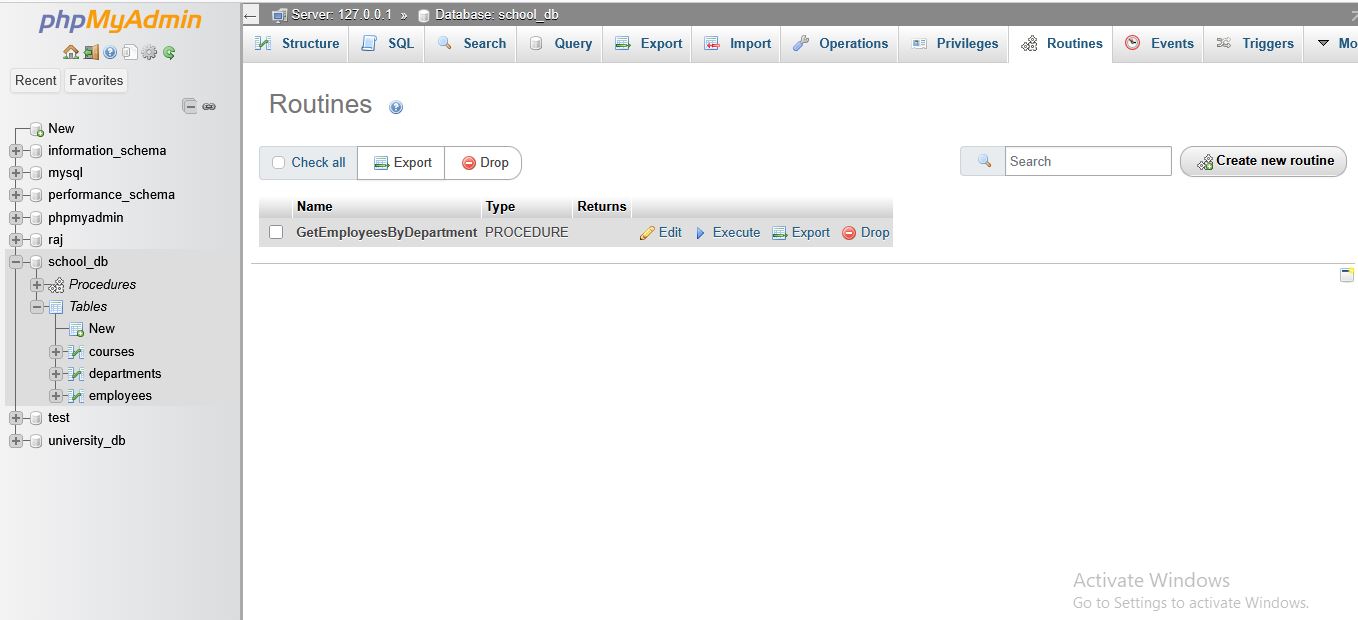
FROM employees

WHERE dept\_id = dept;

END $$

DELIMITER ;

* **How to Execute the Procedure** CALL GetEmployeesByDepartment(2);



* This procedure returns employees who work in a specific department.

**Lab 2: Write a stored procedure that accepts course\_id as input and returns the course**

**details.**

**Answer:**

**Query: Stored Procedure to Retrieve Course Details by course\_id**

* **Stored Procedure**

DELIMITER $$

CREATE PROCEDURE GetCourseDetails(IN c\_id INT)

BEGIN

SELECT course\_id, course\_name, course\_duration

FROM courses

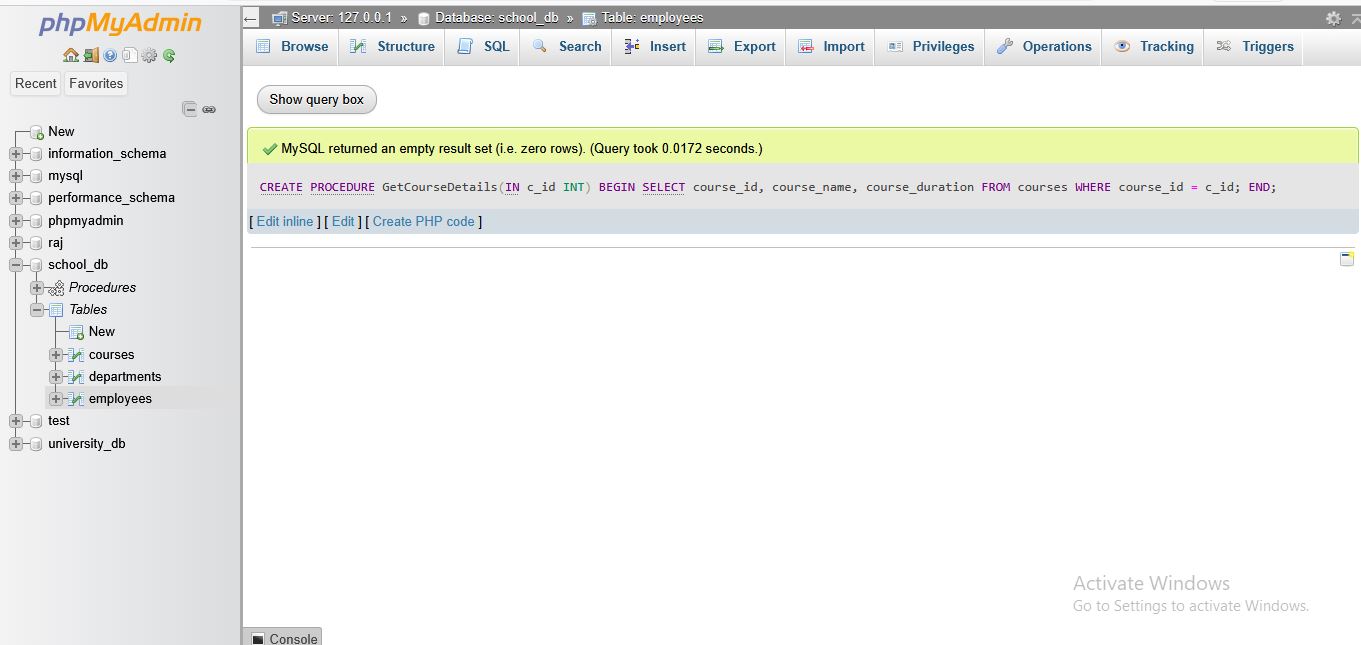
WHERE course\_id = c\_id;

END $$

DELIMITER ;

* **How to Execute the Procedure**

CALL GetCourseDetails(101);



* This procedure returns only the details of the course whose ID you pass.

# SQL View

**Lab 1: Create a view to show all employees along with their department names. Answer:**

**Query: Create a View to Show Employees with Department Names**

* **Step 1: Create a VIEW using JOIN**

CREATE VIEW employee\_department\_view AS

SELECT e.emp\_id, e.emp\_name, e.salary, d.dept\_name

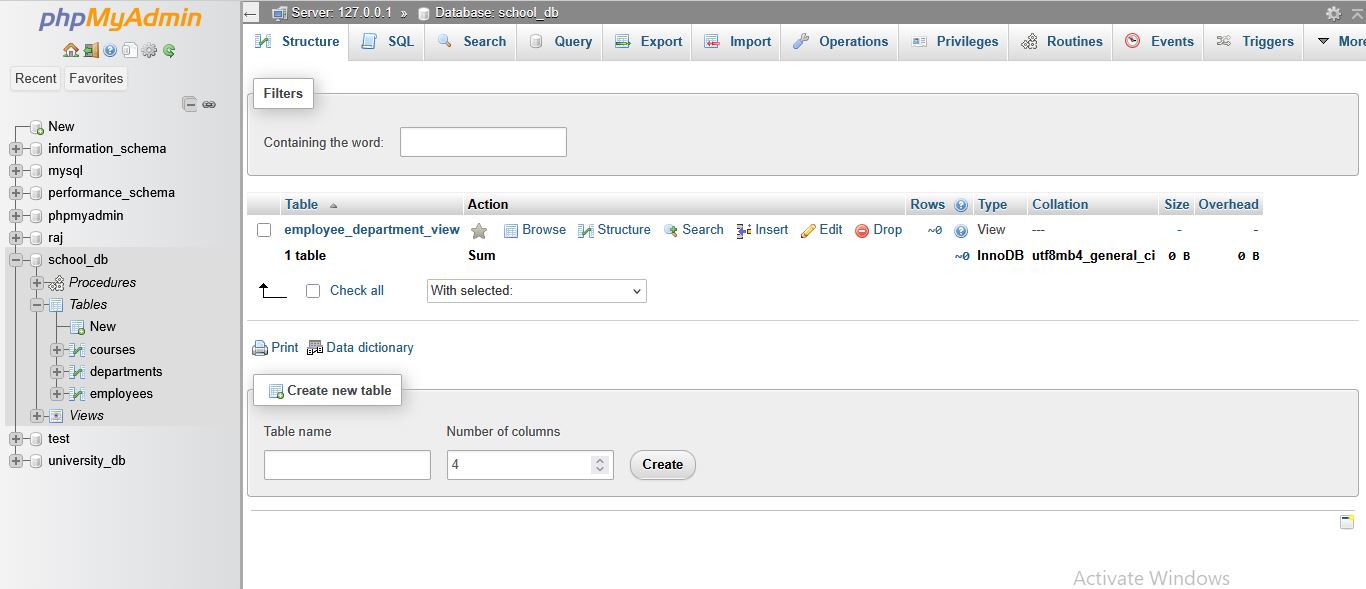
FROM employees e

INNER JOIN departments d

ON e.dept\_id = d.dept\_id;

* **Step 2: To see the view**

SELECT \* FROM employee\_department\_view;



**Lab 2: Modify the view to exclude employees whose salaries are below $50,000. Answer:**

**Query: Modify the View to Exclude Employees with Salary < 50,000**

* **Use OR REPLACE to update the view**

CREATE OR REPLACE VIEW employee\_department\_view AS

SELECT e.emp\_id, e.emp\_name, e.salary, d.dept\_name

FROM employees e

INNER JOIN departments d

ON e.dept\_id = d.dept\_id WHERE e.salary >= 50000;

* **To check updated view**

SELECT \* FROM employee\_department\_view;



# SQL Triggers

**Lab 1: Create a trigger to automatically log changes to the employees table when a new employee is added.**

**Answer:**

**Query: Trigger to Log New Employee Insertions**

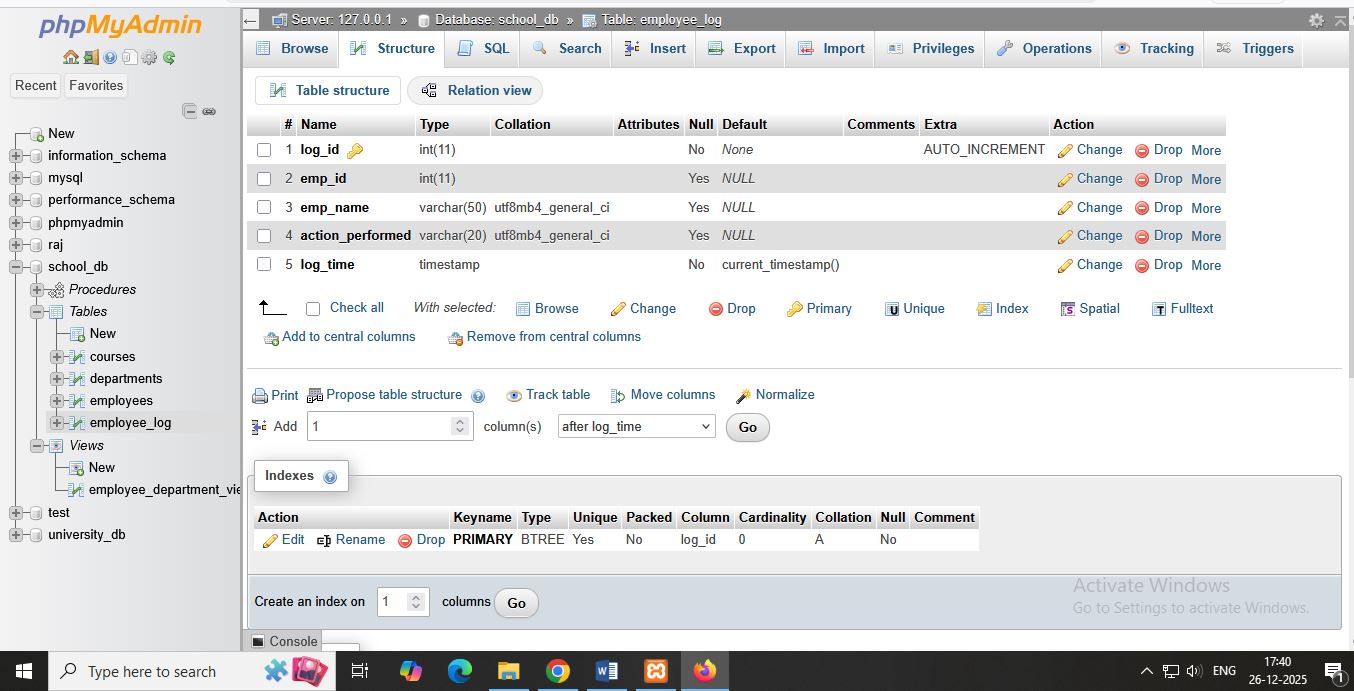
* **Step 1: Create a log table**

You need a table to store log records. CREATE TABLE employee\_log ( log\_id INT AUTO\_INCREMENT PRIMARY KEY,

emp\_id INT,

emp\_name VARCHAR(50), action\_performed VARCHAR(20), log\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);



* **Step 2: Create the trigger**

This trigger will run AFTER a new employee is inserted.

DELIMITER $$

CREATE TRIGGER log\_new\_employee

AFTER INSERT ON employees

FOR EACH ROW

BEGIN

INSERT INTO employee\_log (emp\_id, emp\_name, action\_performed)

VALUES (NEW.emp\_id, NEW.emp\_name, 'INSERT');

END $$

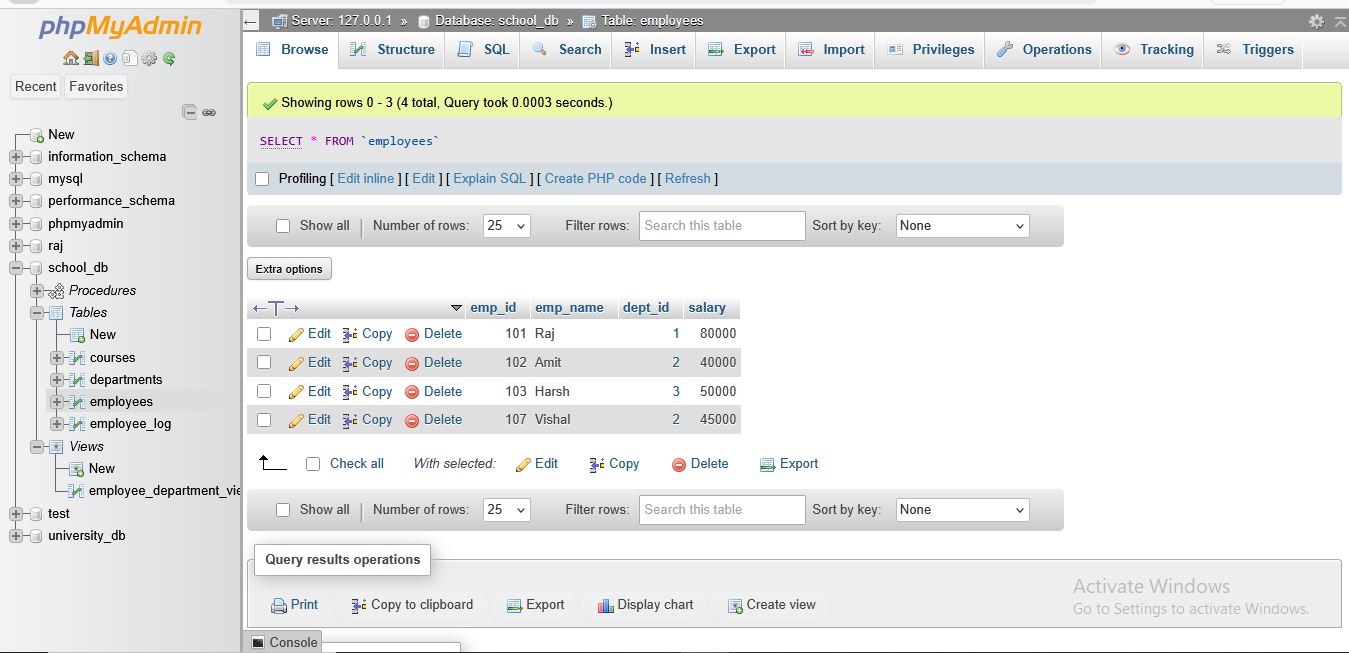
DELIMITER ;

* **Sample Output**

(After inserting a new employee)

INSERT INTO employees (emp\_id, emp\_name, dept\_id, salary)

VALUES (107, 'Vishal', 2, 45000);



* Running:

SELECT \* FROM employee\_log;

**Lab 2: Create a trigger to update the last\_modified timestamp whenever an employee record is updated.**

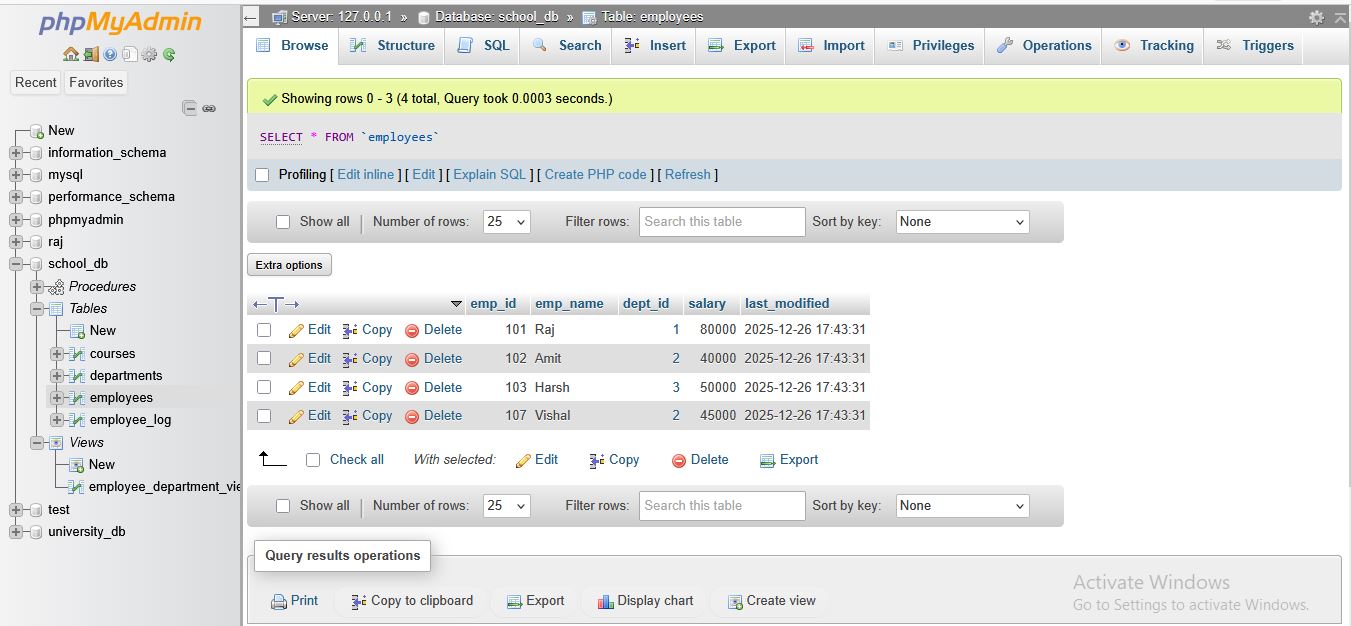
**Answer:**

**Query: Trigger to Automatically Update last\_modified Timestamp**

* **Step 1: Add a column to employees table**

ALTER TABLE employees

ADD last\_modified TIMESTAMP DEFAULT CURRENT\_TIMESTAMP;



* **Step 2: Create the UPDATE trigger**

This trigger updates the timestamp whenever an employee record changes.

DELIMITER $$

CREATE TRIGGER update\_last\_modified

BEFORE UPDATE ON employees

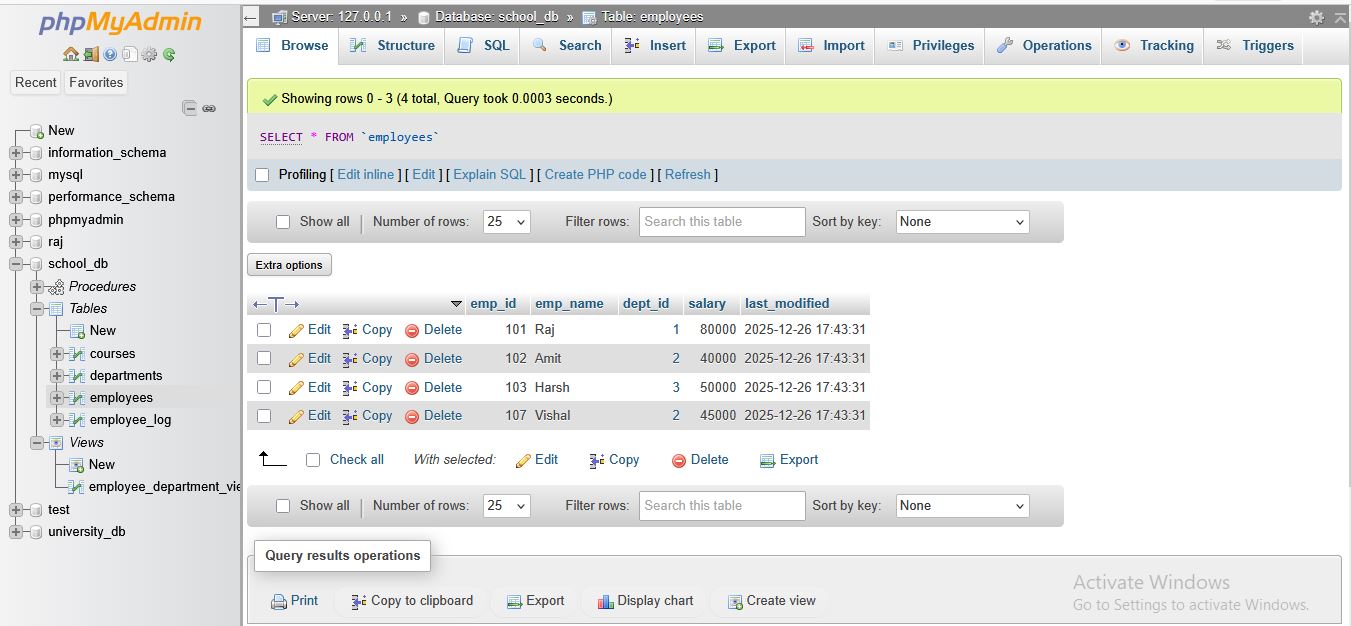
FOR EACH ROW

BEGIN

SET NEW.last\_modified = CURRENT\_TIMESTAMP;

END $$

DELIMITER ;

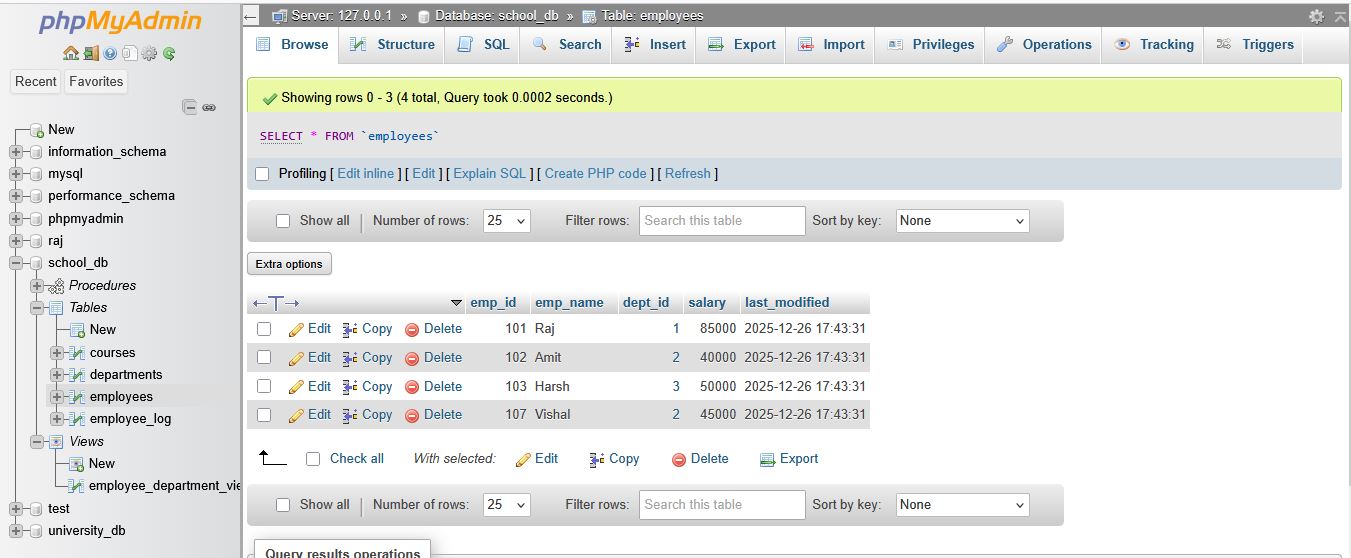


* **Example Update**

UPDATE employees

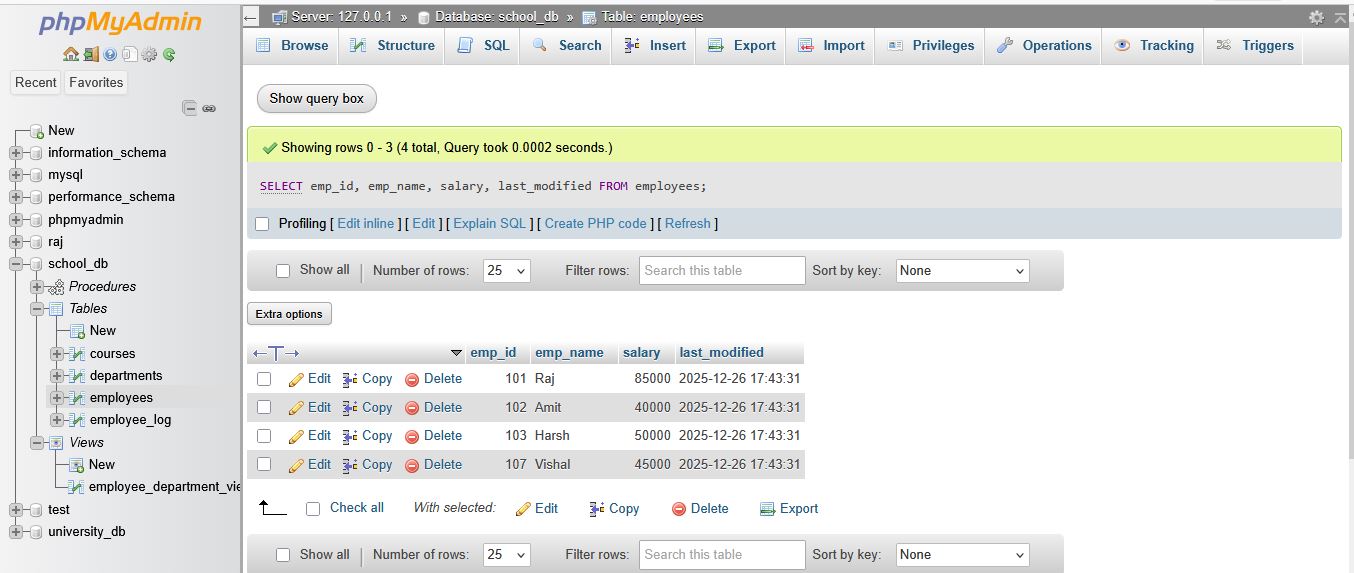
SET salary = 85000

WHERE emp\_id = 101;



* **Output (View the updated timestamp)**

SELECT emp\_id, emp\_name, salary, last\_modified FROM employees;



* Automatically updated — no need to manually set last\_modified.

# Introduction to PL/SQL

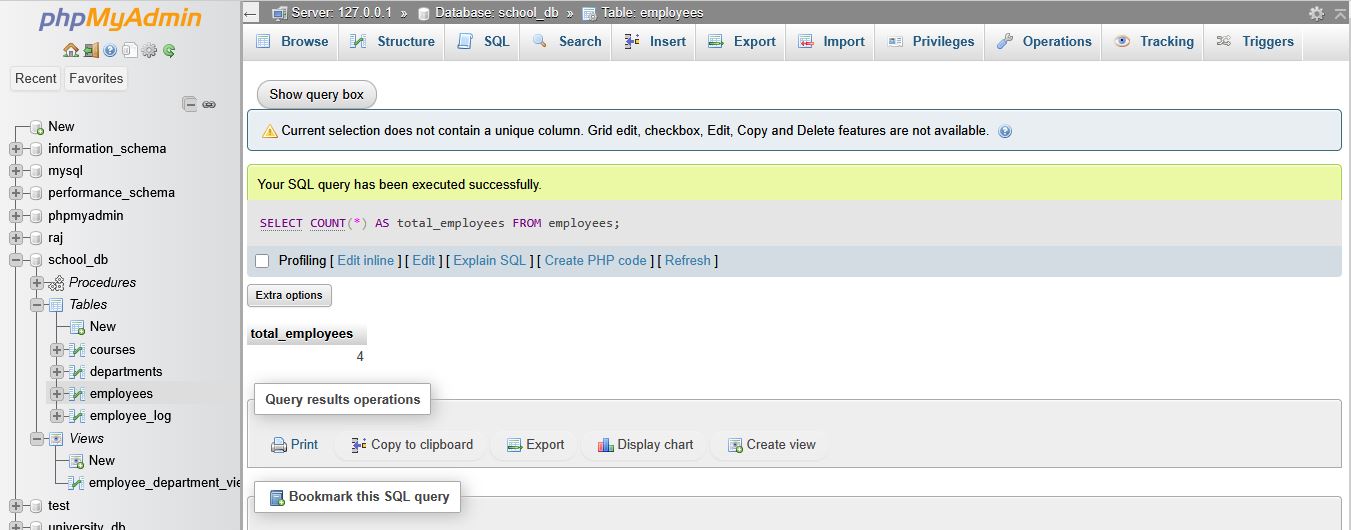
**Lab 1: Write a PL/SQL block to print the total number of employees from the employees**

**table.**

**Answer:**

**Query: Print total number of employees**

SELECT COUNT(\*) AS total\_employees FROM employees;

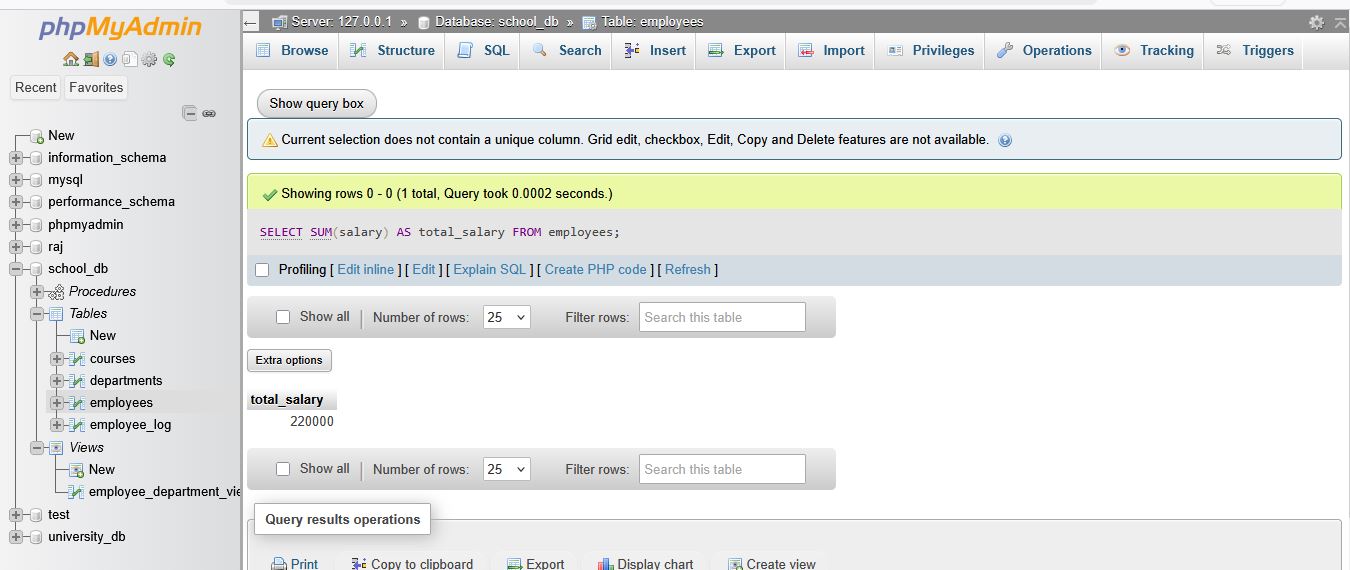


**Lab 2: Create a PL/SQL block that calculates the total salary from an orders table.**

**Answer:**

**Query: Calculate Total Salary**

SELECT SUM(salary) AS total\_salary FROM employees;



# PL/SQL Control Structures

**Lab 1: Write a PL/SQL block using an IF-THEN condition to check the department of an employee.**

**Answer:**

**Query 1: Stored Procedure**

DELIMITER $$

CREATE PROCEDURE check\_department(IN p\_emp\_id INT)

BEGIN

DECLARE v\_dept INT;

SELECT dept\_id INTO v\_dept

FROM employees

WHERE emp\_id = p\_emp\_id;

IF v\_dept = 2 THEN

SELECT 'Employee belongs to Department 2' AS message;

ELSE

SELECT 'Employee does NOT belong to Department 2' AS message; END IF;

END $$

DELIMITER ;

**Query 2: Call the Procedure**

CALL check\_department(102);



**Lab 2: Use a FOR LOOP to iterate through employee records and display their names.**

**Answer:**

**Query 1: Stored Procedure with FOR-LIKE LOOP (MySQL uses WHILE)**

DELIMITER $$

CREATE PROCEDURE show\_employee\_names()

BEGIN

DECLARE done INT DEFAULT 0;

DECLARE v\_name VARCHAR(50);

DECLARE emp\_cursor CURSOR FOR

SELECT emp\_name FROM employees;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

OPEN emp\_cursor;

read\_loop: LOOP

FETCH emp\_cursor INTO v\_name;

IF done = 1 THEN

LEAVE read\_loop;

END IF;

SELECT v\_name AS employee\_name;

END LOOP;

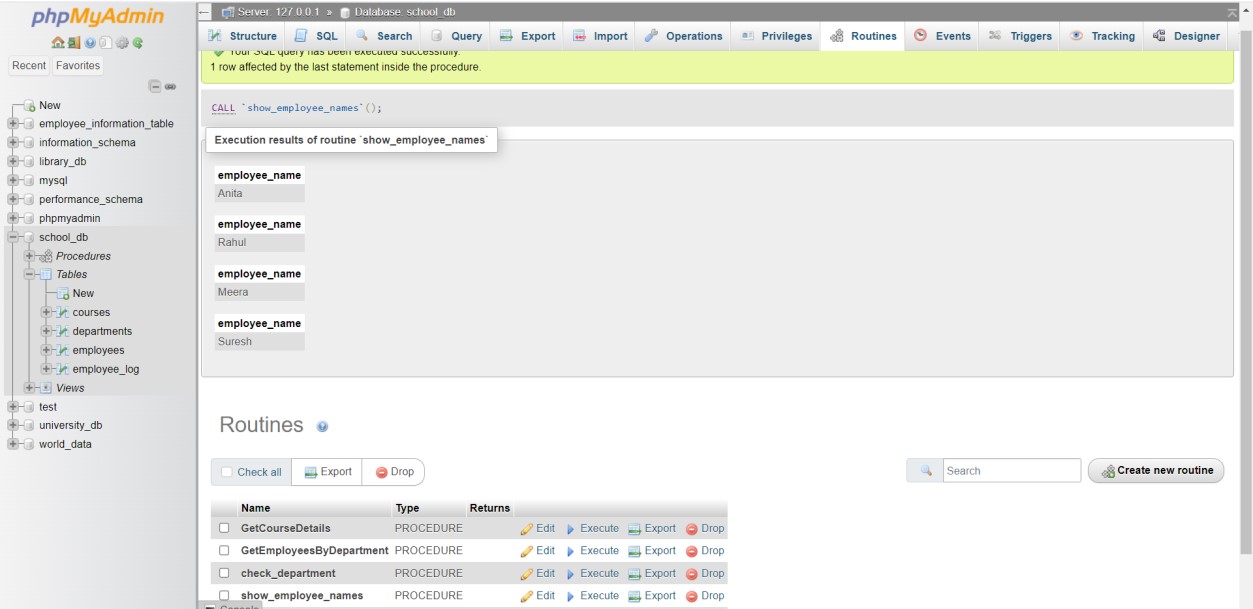
CLOSE emp\_cursor;

END $$

DELIMITER ;

**Query 2: Call the procedure**

CALL show\_employee\_names();



# SQL Cursors

**Lab 1: Write a PL/SQL block using an explicit cursor to retrieve and display employee details.**

**Answer:**

**Query 1: Stored Procedure**

DELIMITER $$

CREATE PROCEDURE show\_employee\_details()

BEGIN

DECLARE done INT DEFAULT 0;

DECLARE v\_id INT;

DECLARE v\_name VARCHAR(50);

DECLARE v\_dept INT;

DECLARE v\_salary INT;

DECLARE emp\_cursor CURSOR FOR

SELECT emp\_id, emp\_name, dept\_id, salary FROM employees;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

OPEN emp\_cursor;

read\_loop: LOOP

FETCH emp\_cursor INTO v\_id, v\_name, v\_dept, v\_salary;

IF done = 1 THEN

LEAVE read\_loop;

END IF;

SELECT CONCAT(

'ID: ', v\_id,

', Name: ', v\_name,

', Dept: ', v\_dept,

', Salary: ', v\_salary

) AS Employee\_Details;

END LOOP;

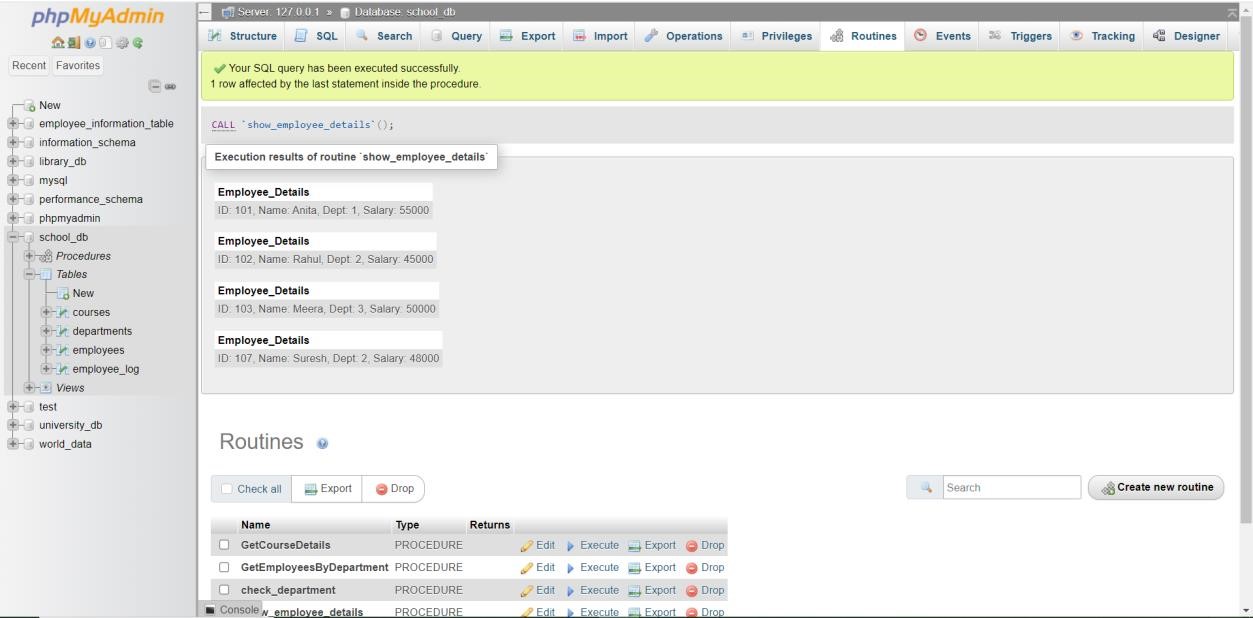
CLOSE emp\_cursor;

END$$

DELIMITER ;

Query 2: **Run the procedure**

CALL show\_employee\_details();



**Lab 2: Create a cursor to retrieve all courses and display them one by one.**

**Answer:**

**Query 1: Stored Procedure**

DELIMITER $$

CREATE PROCEDURE show\_courses()

BEGIN

DECLARE done INT DEFAULT 0;

DECLARE v\_id INT;

DECLARE v\_name VARCHAR(100);

DECLARE v\_duration VARCHAR(50);

DECLARE course\_cursor CURSOR FOR

SELECT course\_id, course\_name, course\_duration FROM courses;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

OPEN course\_cursor;

course\_loop: LOOP

FETCH course\_cursor INTO v\_id, v\_name, v\_duration;

IF done = 1 THEN

LEAVE course\_loop;

END IF;

SELECT CONCAT(

'Course ID: ', v\_id,

', Name: ', v\_name,

', Duration: ', v\_duration

) AS Course\_Details;

END LOOP;

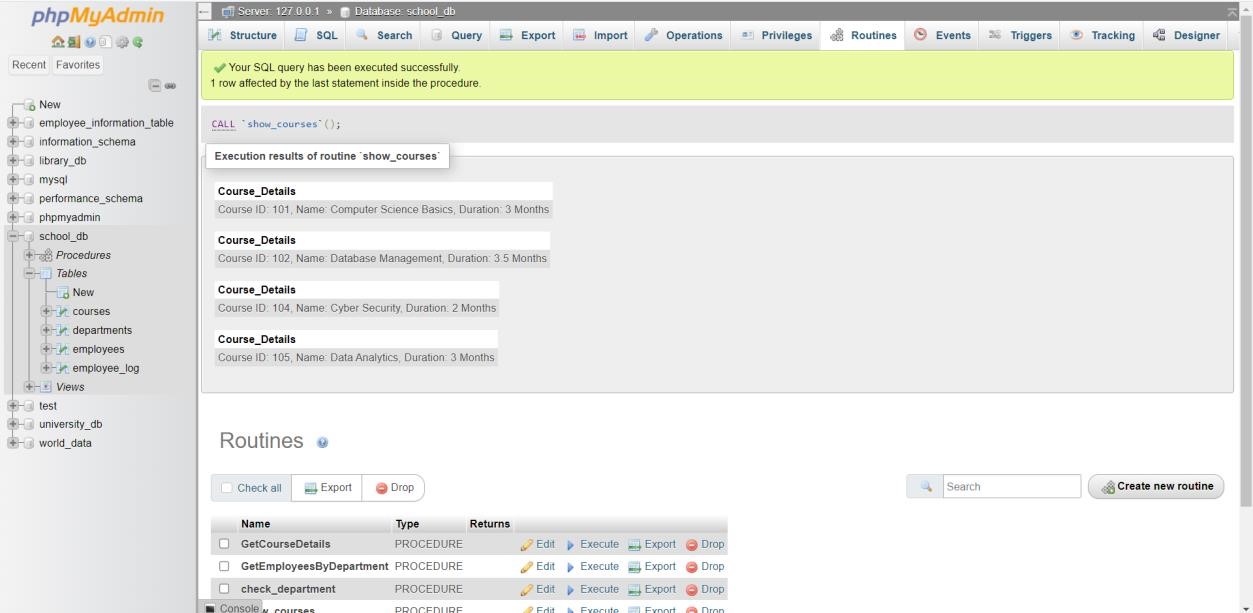
CLOSE course\_cursor;

END$$

DELIMITER ;

**Query 2: Run the procedure**

CALL show\_courses();



# Rollback and Commit Savepoint

**Lab 1: Perform a transaction where you create a savepoint, insert records, then rollback to the savepoint.**

**Answer:**

**Query: Create Savepoint → Insert Records → Rollback to Savepoint**

-- Start the transaction

START TRANSACTION;

-- Insert 1st record

INSERT INTO employees (emp\_id, emp\_name, dept\_id, salary)

VALUES (110, 'User1', 1, 20000);

-- Create savepoint

SAVEPOINT sp1;

-- Insert 2nd record

INSERT INTO employees (emp\_id, emp\_name, dept\_id, salary)

VALUES (111, 'User2', 2, 35000);

-- Rollback to savepoint (means the 2nd insert will be removed)

ROLLBACK TO sp1;

-- Commit remaining changes

COMMIT;



**Lab 2: Commit part of a transaction after using a savepoint and then rollback the remaining changes.**

**Answer:**

**Query: Commit part of a transaction → Rollback remaining changes**

-- Start transaction

START TRANSACTION;

-- Insert two new employees

INSERT INTO employees (emp\_id, emp\_name, dept\_id, salary)

VALUES (120, 'P User1', 1, 36000);

INSERT INTO employees (emp\_id, emp\_name, dept\_id, salary)

VALUES (121, 'P User2', 2, 40000);

-- Create savepoint

SAVEPOINT sp2;

-- Insert another record

INSERT INTO employees (emp\_id, emp\_name, dept\_id, salary)

VALUES (122, 'RollbackUser', 3, 37000);

-- Commit changes BEFORE savepoint (120, 121)

RELEASE SAVEPOINT sp2;

COMMIT;

-- Now rollback remaining uncommitted changes

ROLLBACK;

