Module 4 – 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.

Query -1

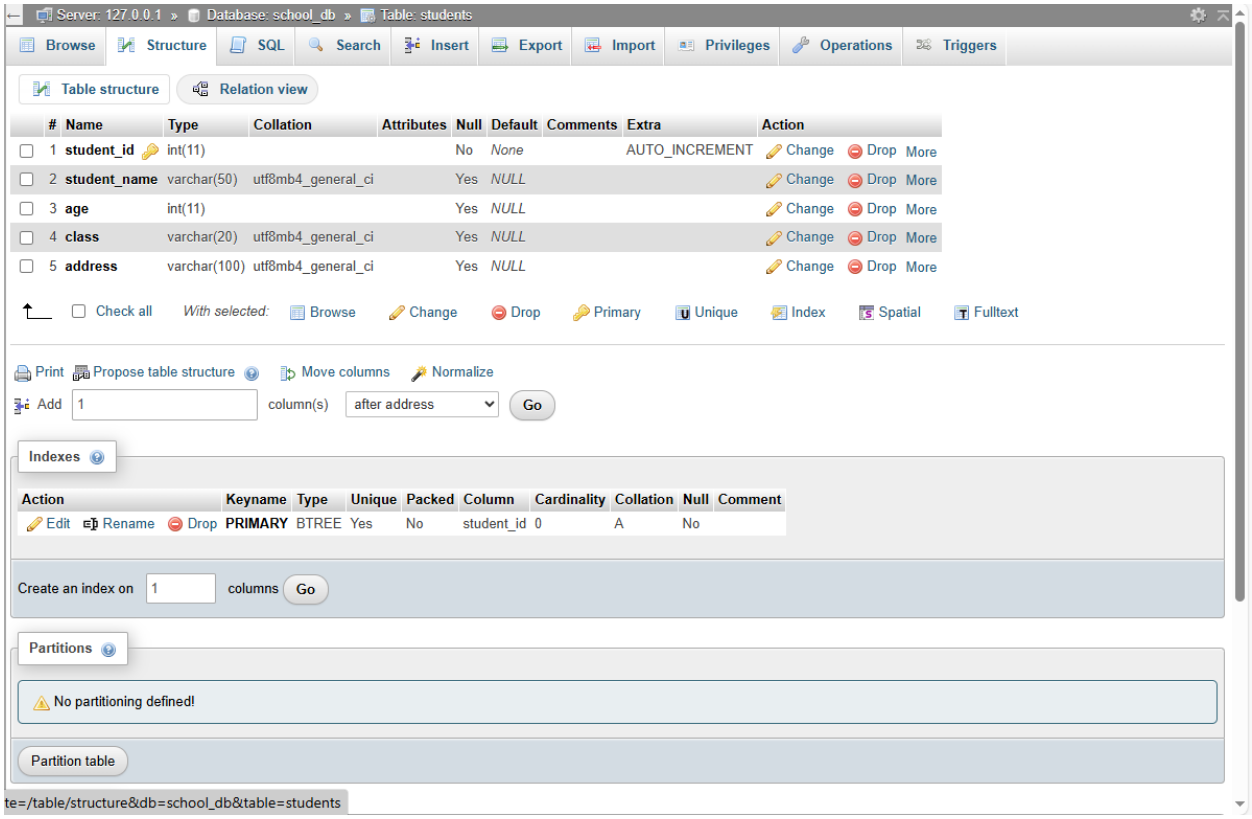
Database Query

```
CREATE DATABASE school_db;
```

Query – 2

students Table Query

```
CREATE TABLE students (
   student_id INT PRIMARY KEY AUTO_INCREMENT,
   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.

Query -1

5 Records Insert Query

INSERT INTO students (student_name, age, class, address) VALUES

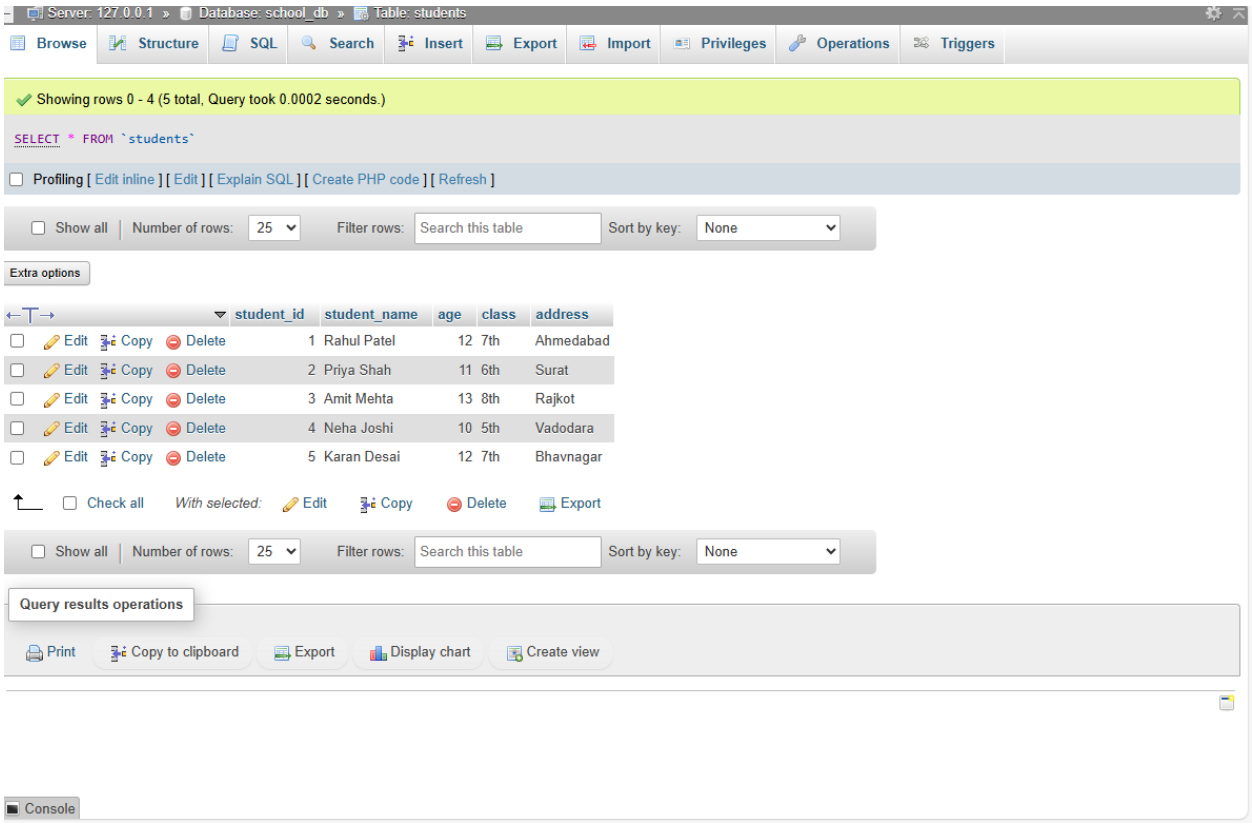
('Rahul Patel', 12, '7th', 'Ahmedabad'),

('Priya Shah', 11, '6th', 'Surat'),

('Amit Mehta', 13, '8th', 'Rajkot'),

('Neha Joshi', 10, '5th', 'Vadodara'),

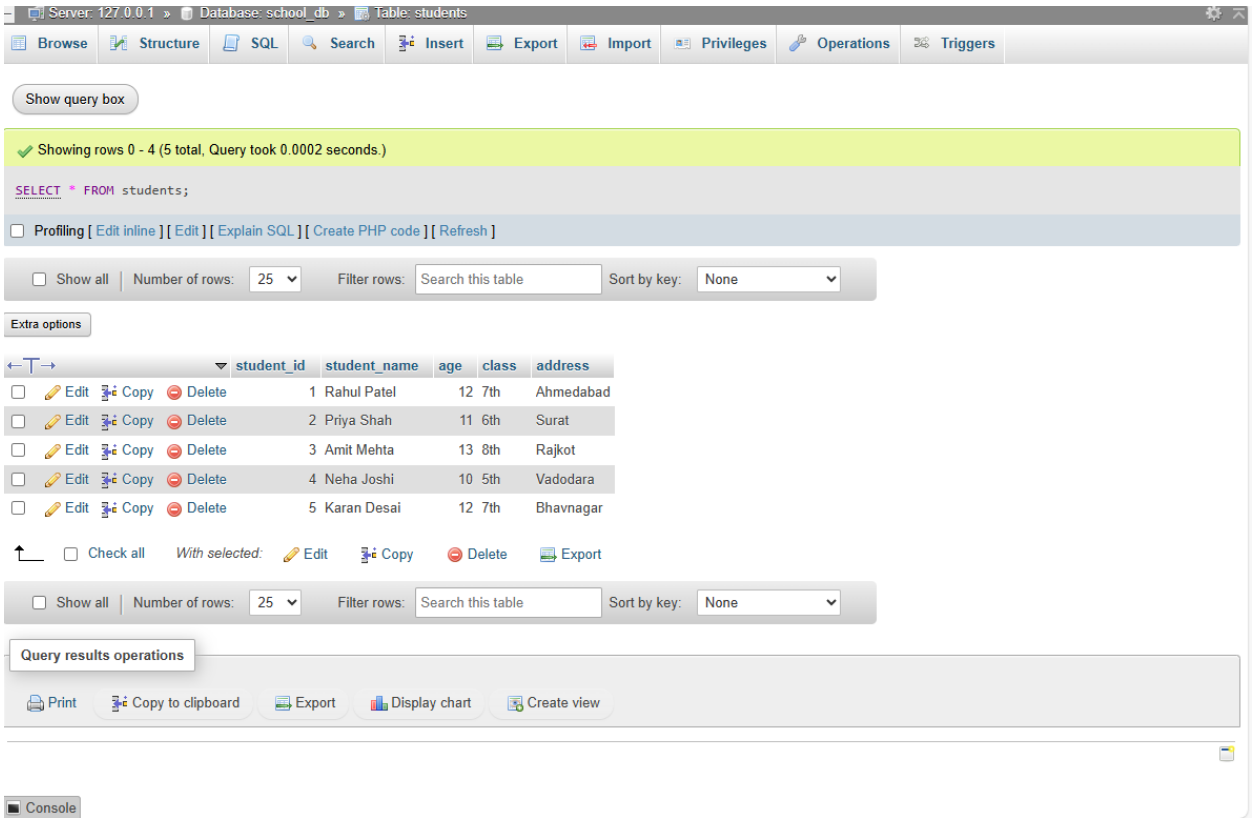
('Karan Desai', 12, '7th', 'Bhavnagar');



Query – 2

All Records SELECT Query

SELECT * FROM students;



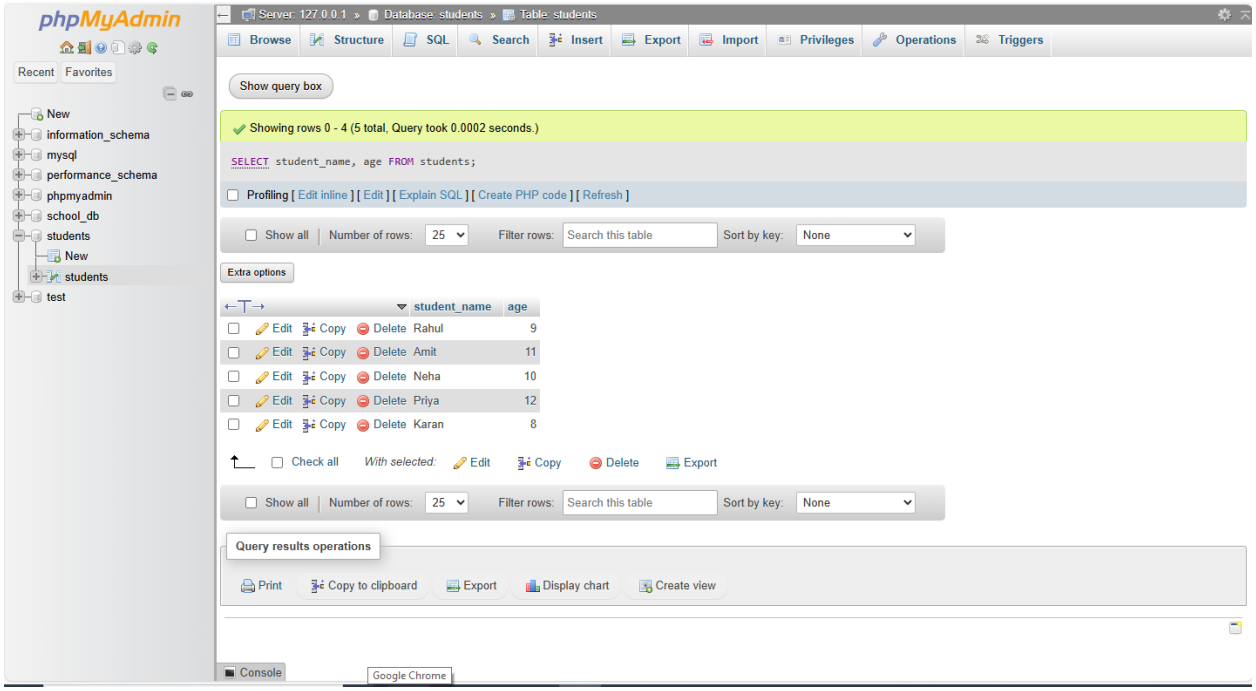
2. SQL Syntax

Lab 1: Write SQL queries to retrieve specific columns (student_name and age) from the students table.

Query – 1

student_name and_age

SELECT student_name, age FROM students;

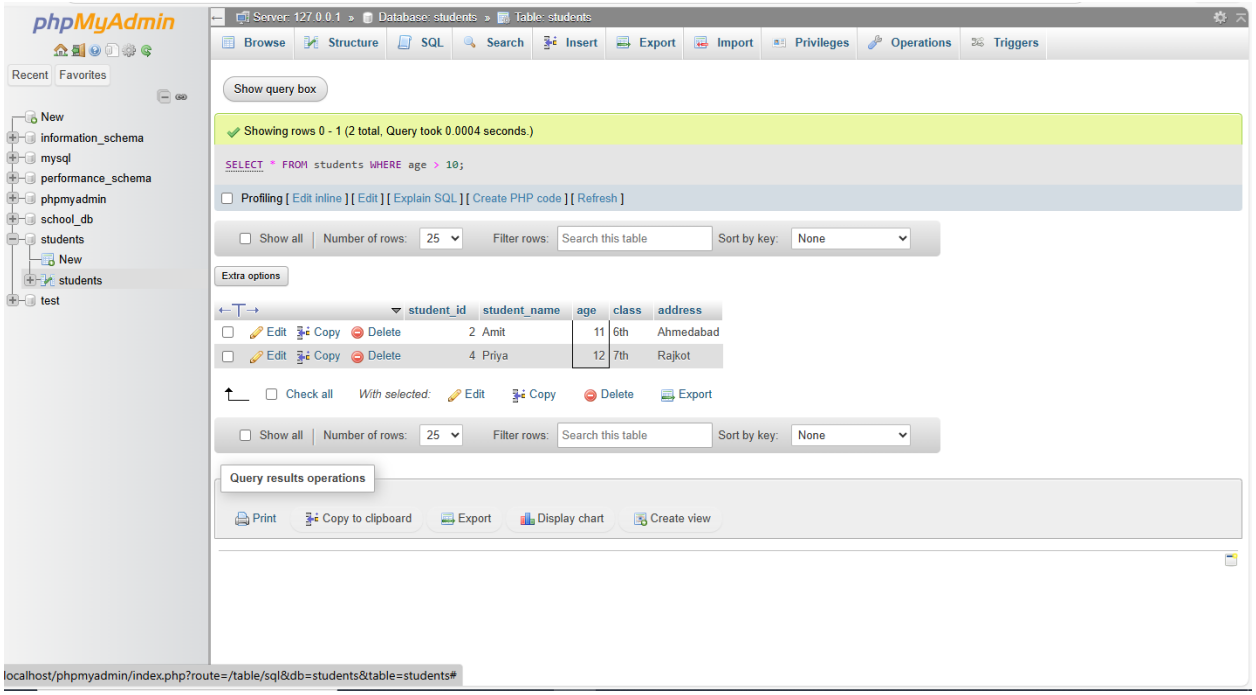


Lab2:Retrieveallstudentswhoseageisgreaterthan 10

SELECT *

FROM students

WHERE age > 10;



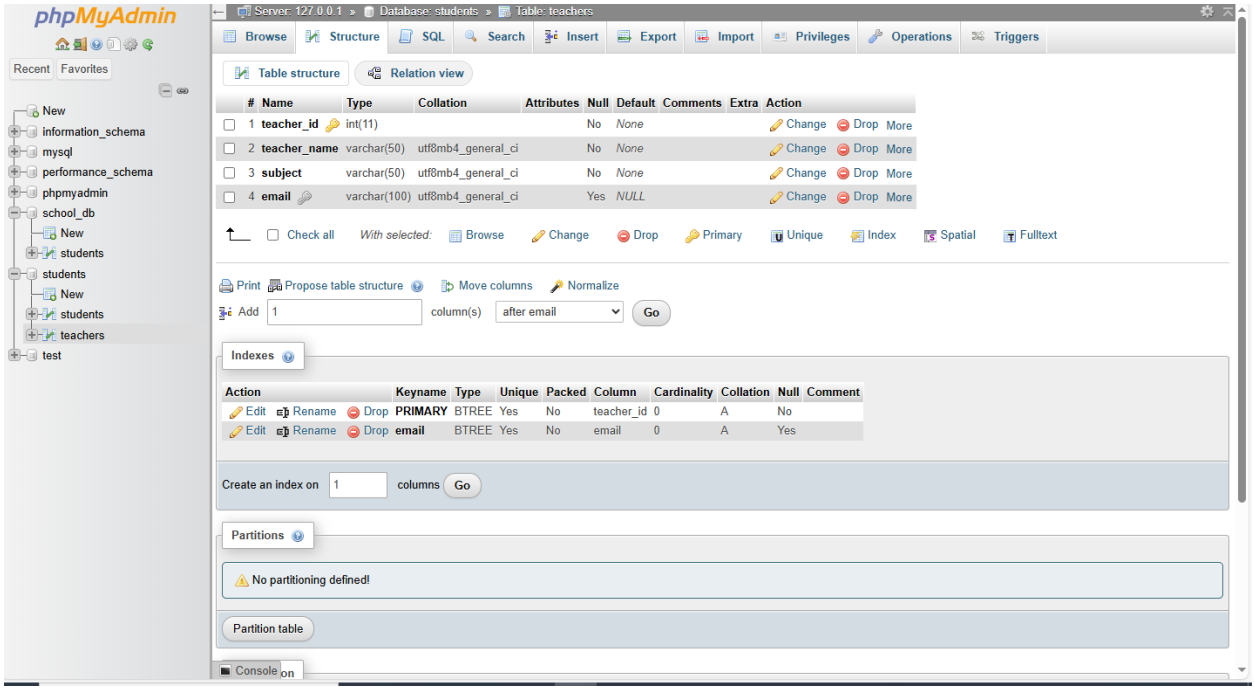
3.SQLConstraints

Lab 1: Create a table teachers with the following columns: teacher_id (Primary Key), teacher_name (NOT NULL), subject (NOT NULL), and email (UNIQUE).

Query – 1

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
);
```

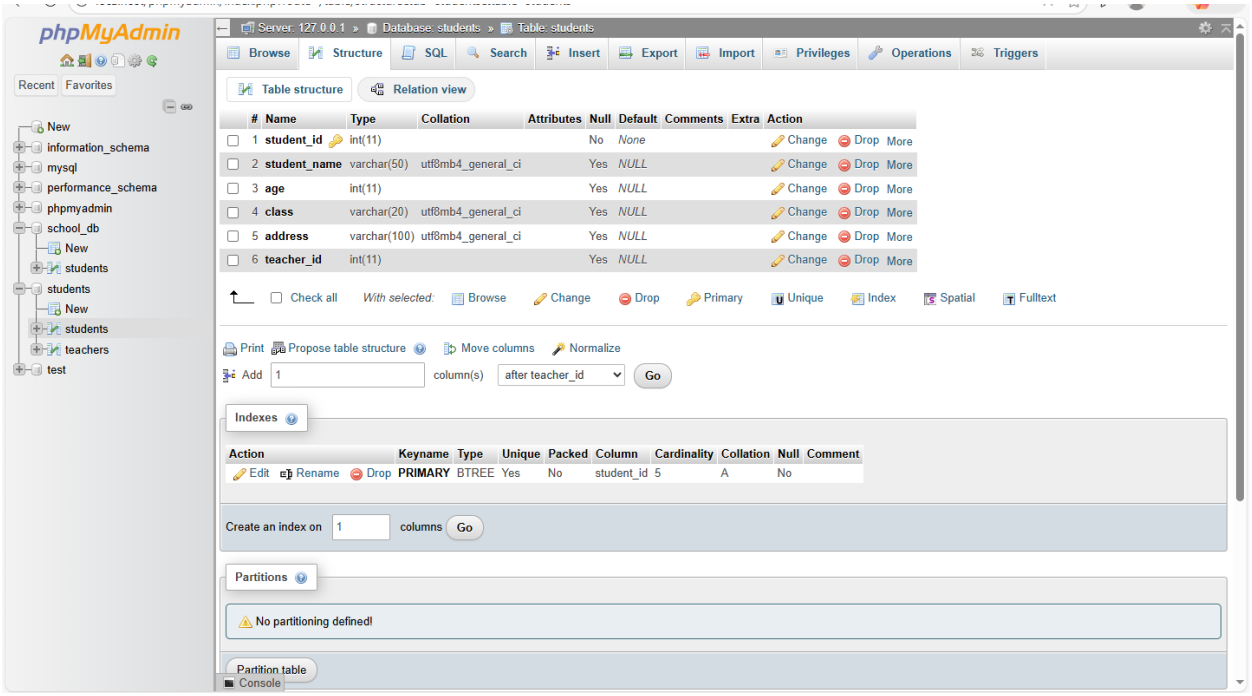


Lab2:ImplementaFOREIGNKEYconstrainttorelateteacher_idwith students table

Query 1: Add teacher_id column to students table

ALTER TABLE students

ADD teacher_id INT;



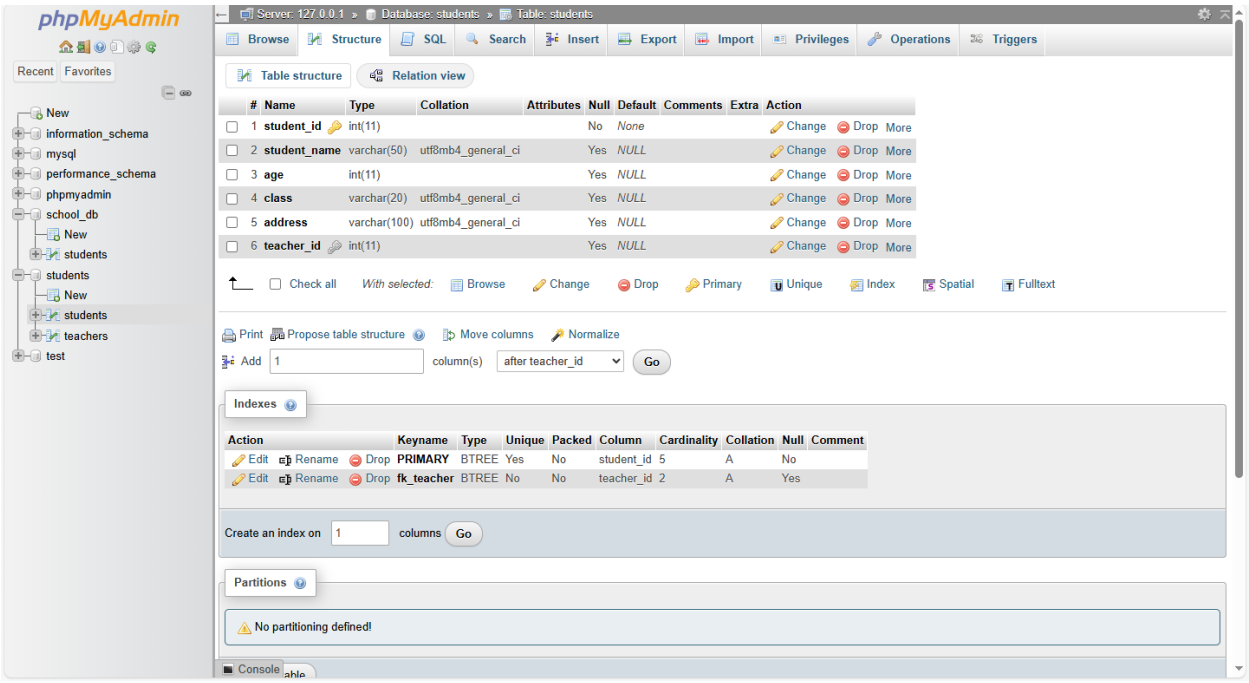
Query 2: Add FOREIGN KEY Constraint

ALTER TABLE students

ADD CONSTRAINT fk_teacher

FOREIGN KEY (teacher_id)

REFERENCES teachers(teacher_id);

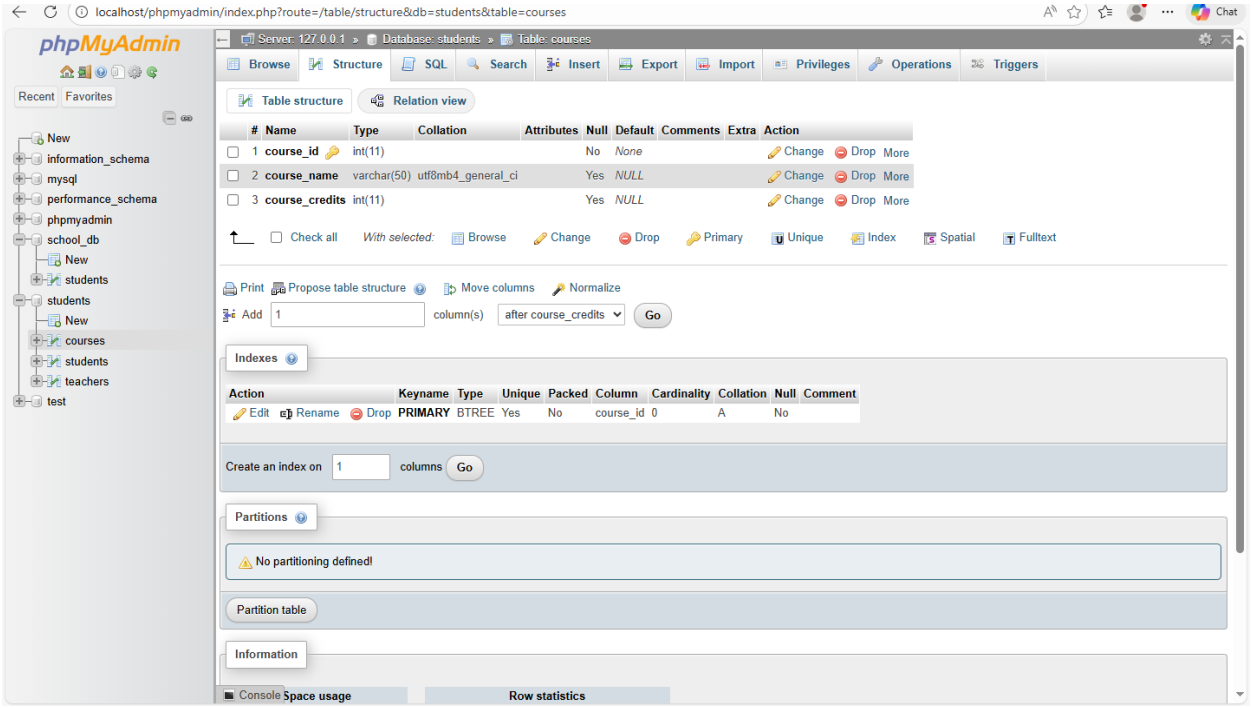


4. Main SQL Commands and Sub-commands (DDL)

Lab 1: Create a table courses with columns: course_id, course_name, and course_credits. Set the course_id as the primary key.

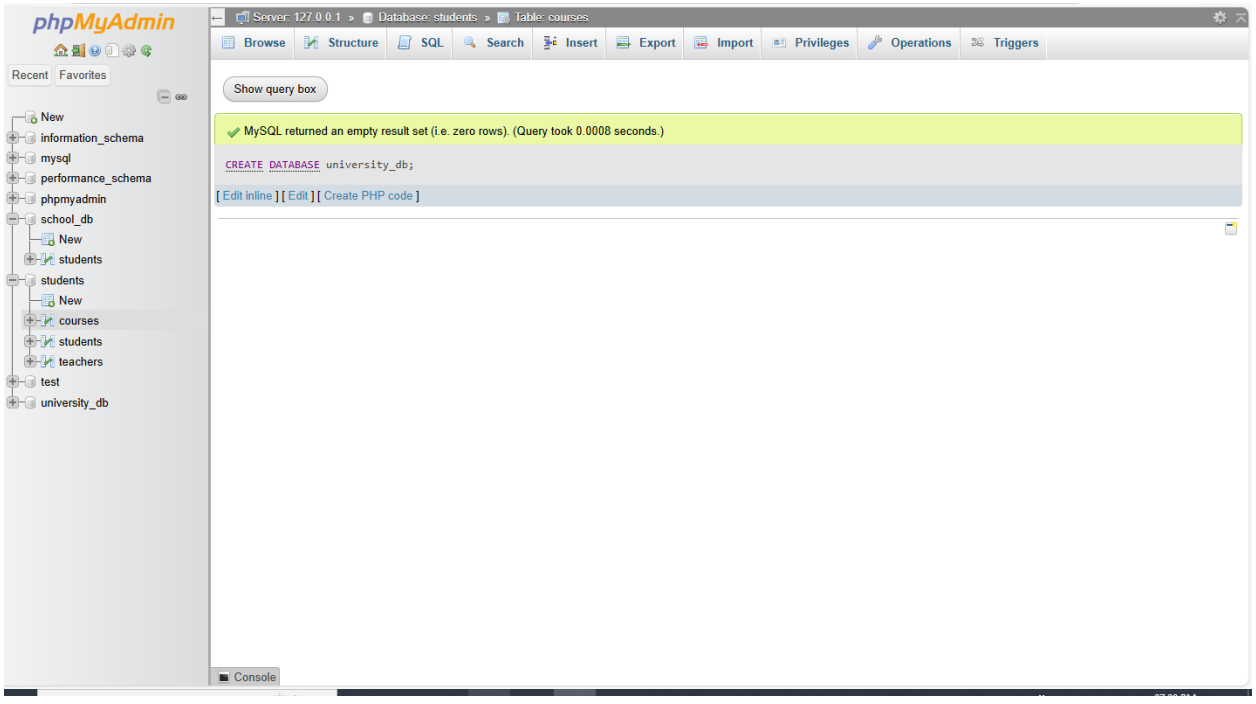
Query: Create a table courses

```
CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(50),
    course_credits INT
);
```



Lab2:Createadatabasenameduniversity_db

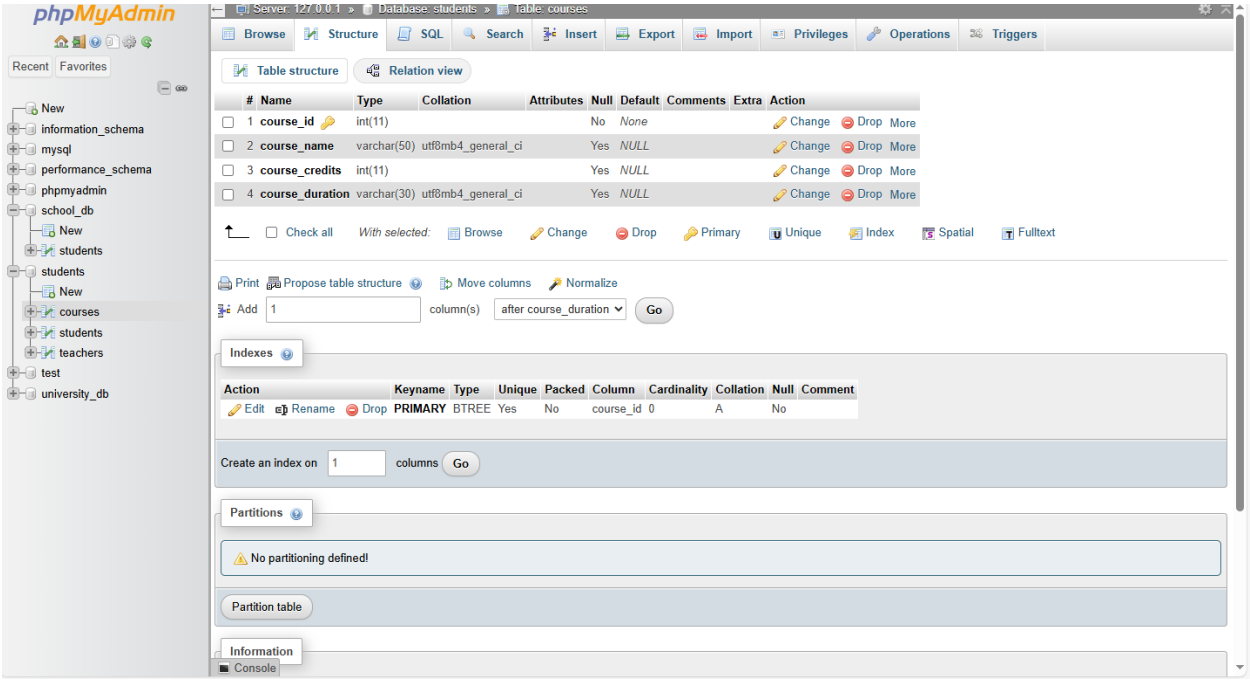
CREATE DATABASE university_db;



5. ALTER Command

Lab 1: Modify the courses table by adding a column course_duration using the ALTER command.

Query: Add a New Column (course_duration) to courses Table

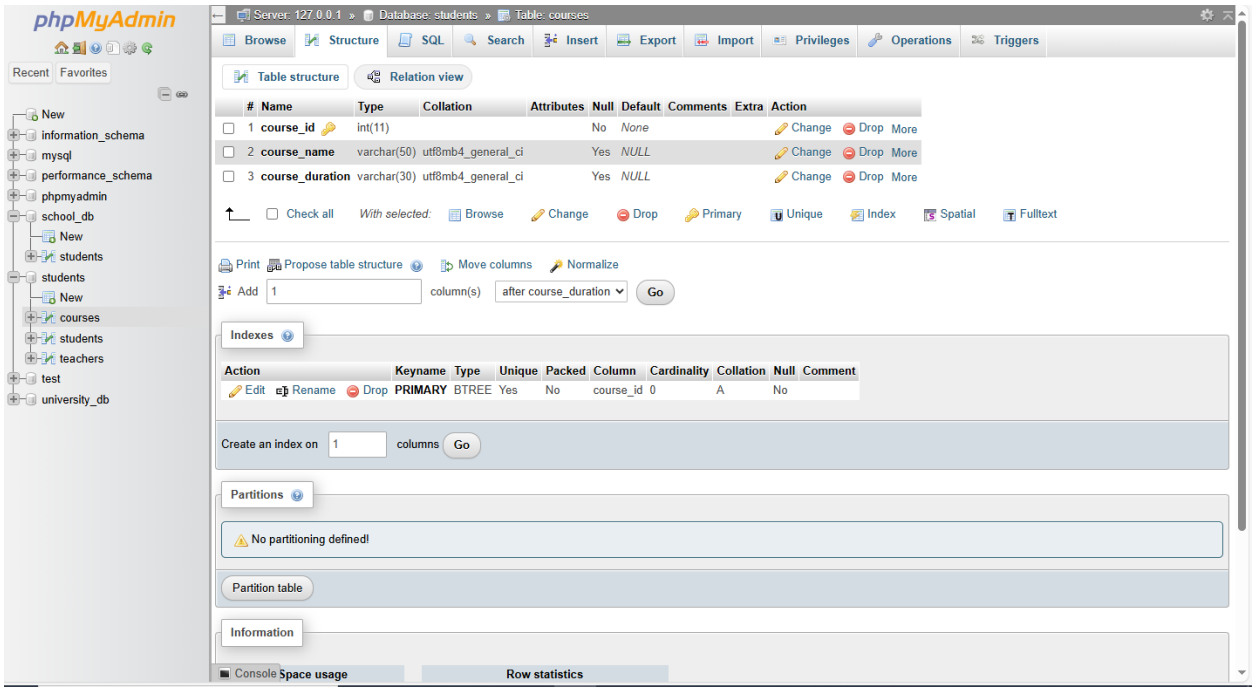


Lab 2: Drop the course_credits column from the courses table.

Query: Drop the Column (course_credits) from courses Table

ALTER TABLE courses

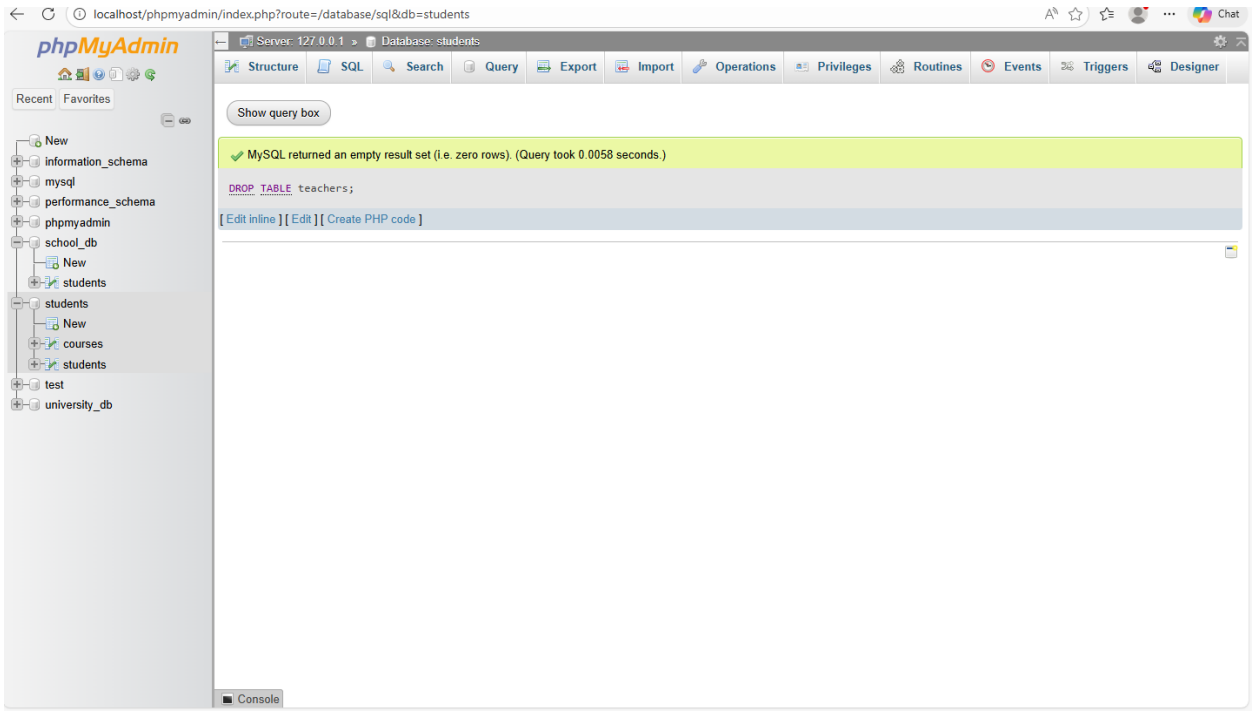
DROP course_credits;



6. DROP Command

Lab 1: Drop the teachers table from the school_db database.

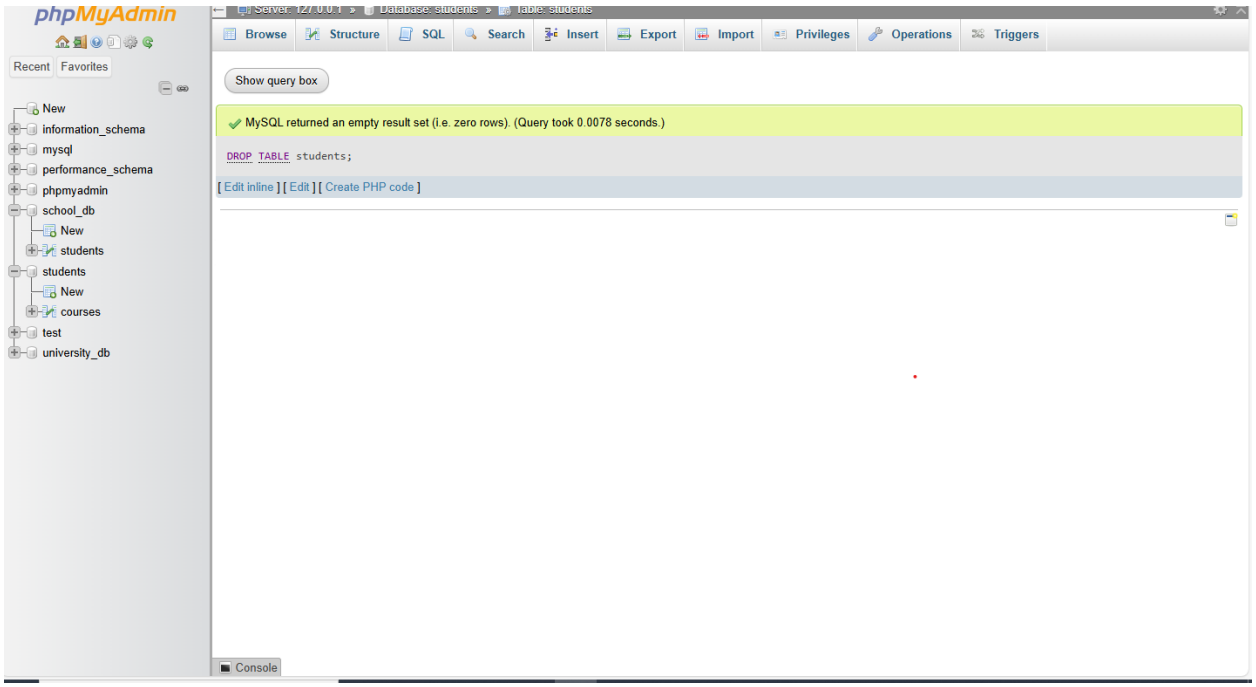
DROP TABLE teachers;



Lab 2: Drop the students table from the school_db database and verify that the table has been removed.

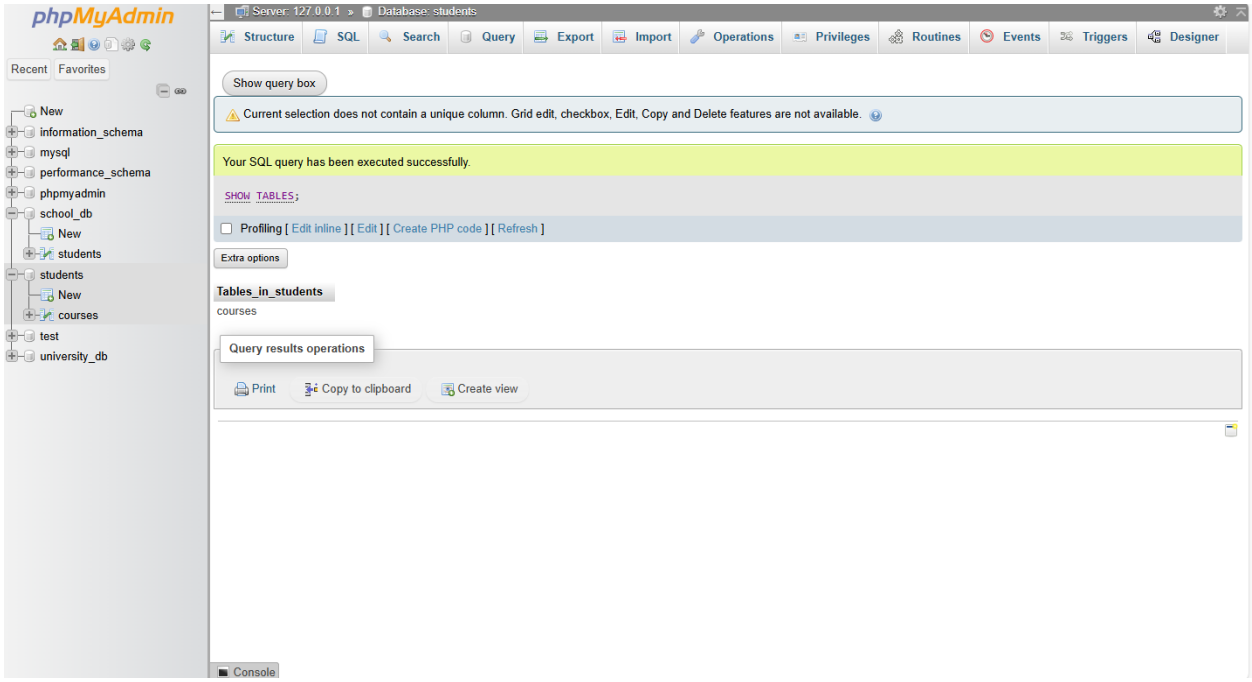
Query 1: Drop the students table and verify removal

DROP TABLE students;



Query 2: Verify that the table is removed

SHOW TABLES;

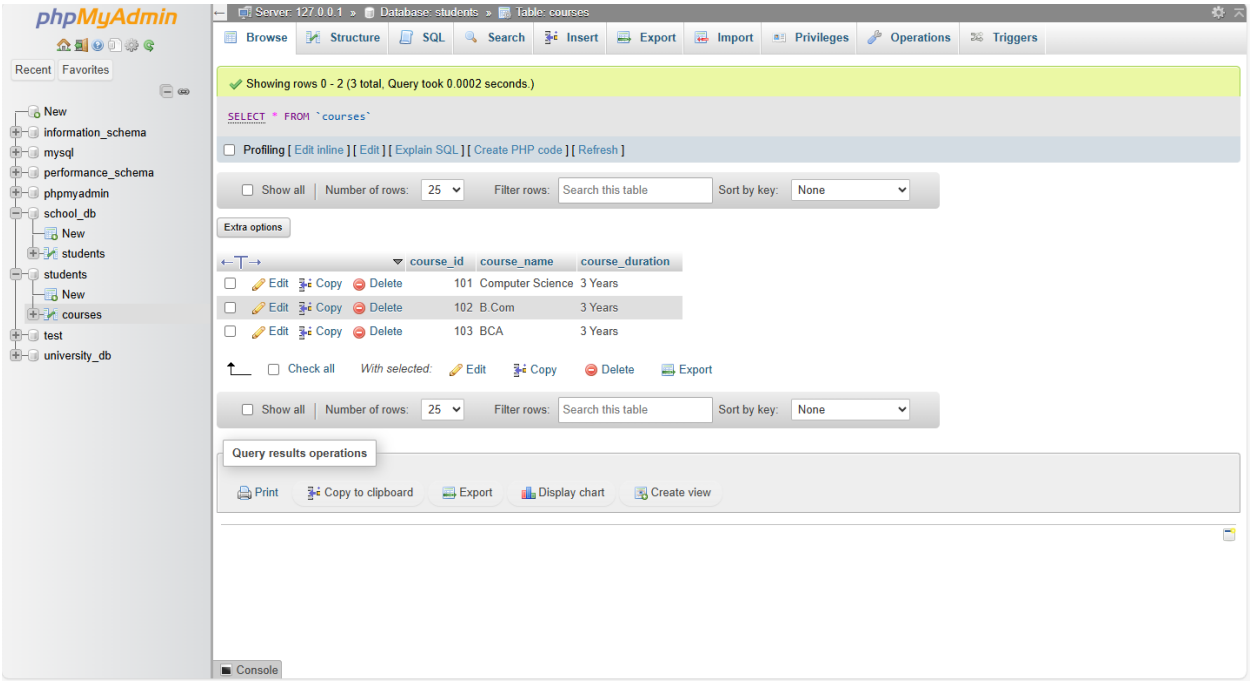


7. Data Manipulation Language (DML)

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

Query: Insert 3 Records into courses Table

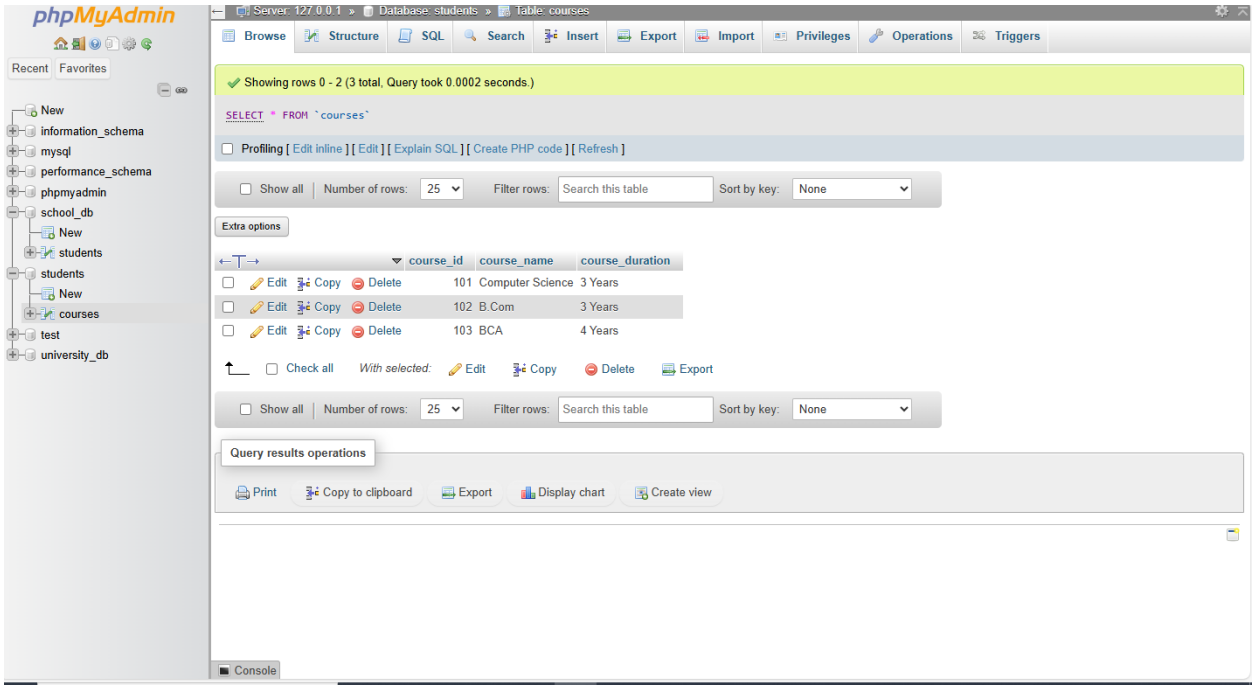
```
INSERT INTO courses (course_id, course_name, course_duration)
VALUES
(101, 'Computer Science', '3 Years'),
(102, 'B.Com', '3 Years'),
(103, 'BCA', '3 Years');
```



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

<u>Query: Update Course Duration of a Specific Course</u>

```
UPDATE courses

SET course_duration = '4 Years'

WHERE course_id = 103;
```

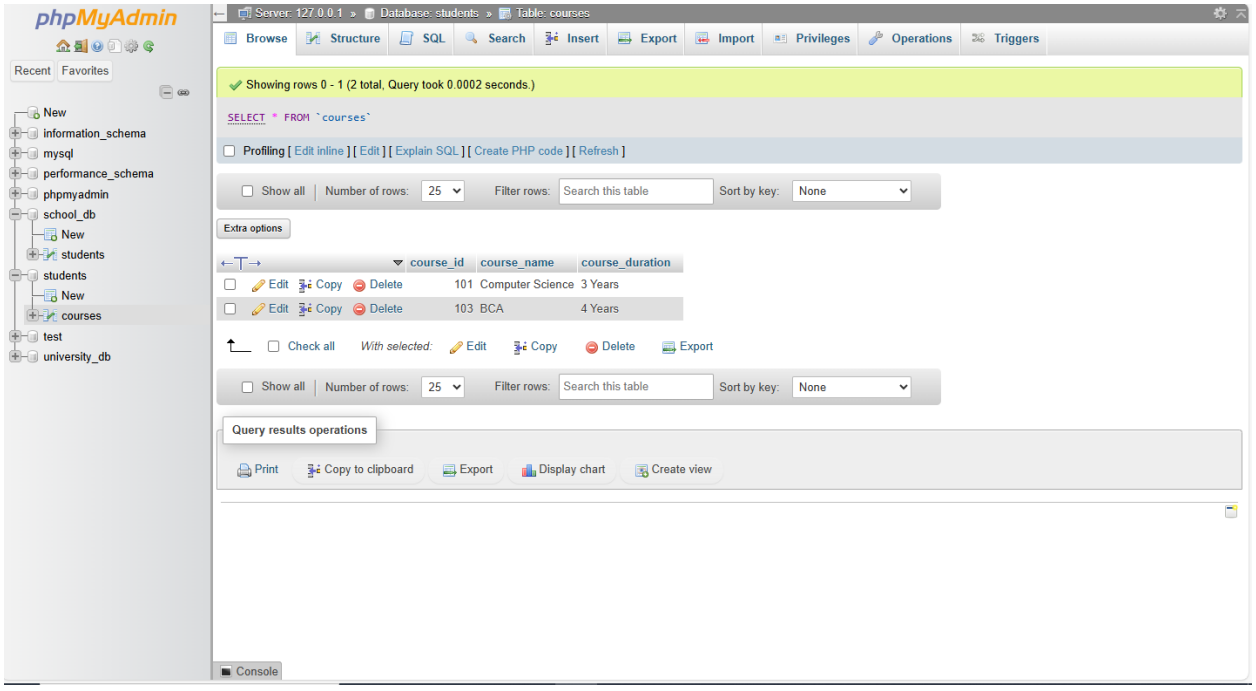


Lab 3: Delete a course with a specific course_id from the courses table using the DELETE command.

<u>Query: Delete a course using DELETE Command</u>

DELETE FROM courses

WHERE course_id = 102;

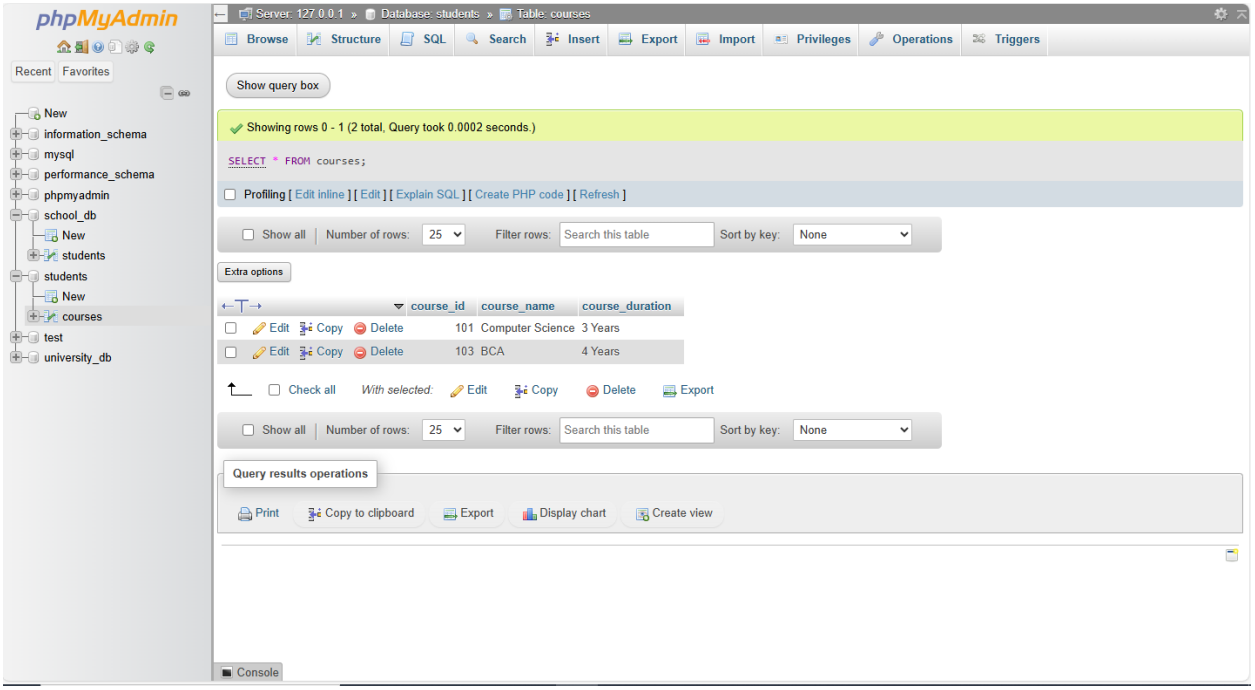


8. Data Query Language (DQL)

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

Query: Retrieve all courses

SELECT * FROM courses;



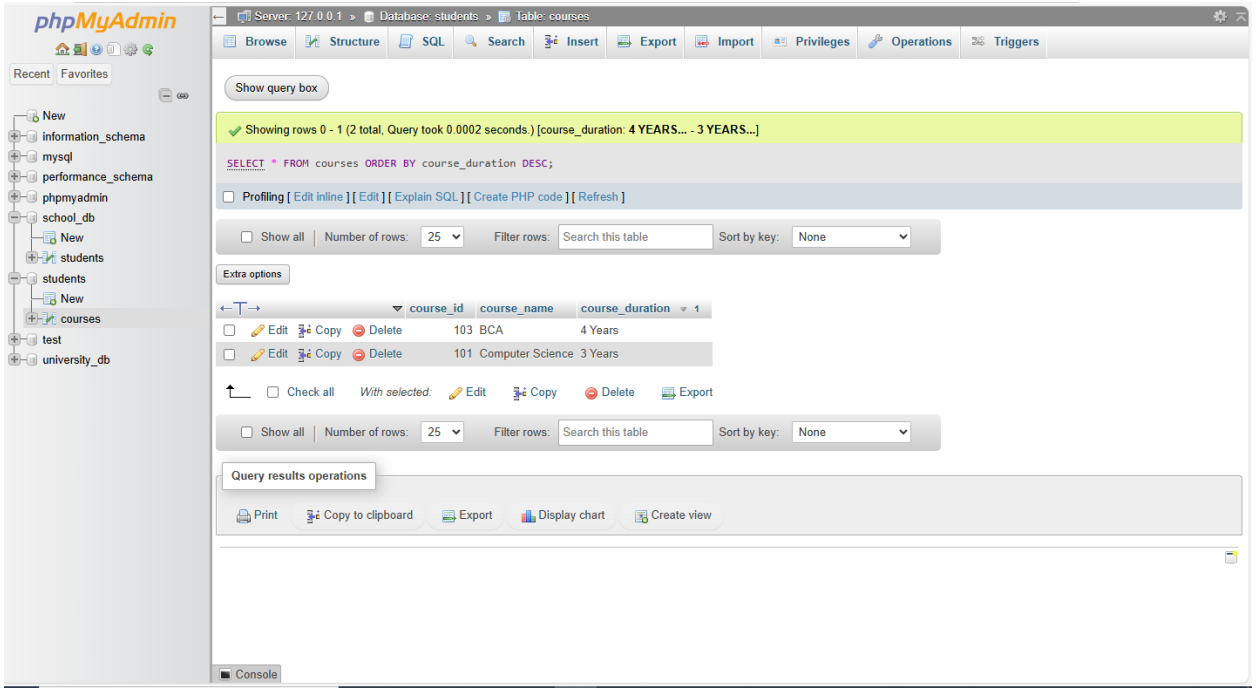
Lab 2: Sort the courses based on course_duration in descending order using ORDER BY.

Query: Sort Courses by course_duration (Descending Order)

SELECT *

FROM courses

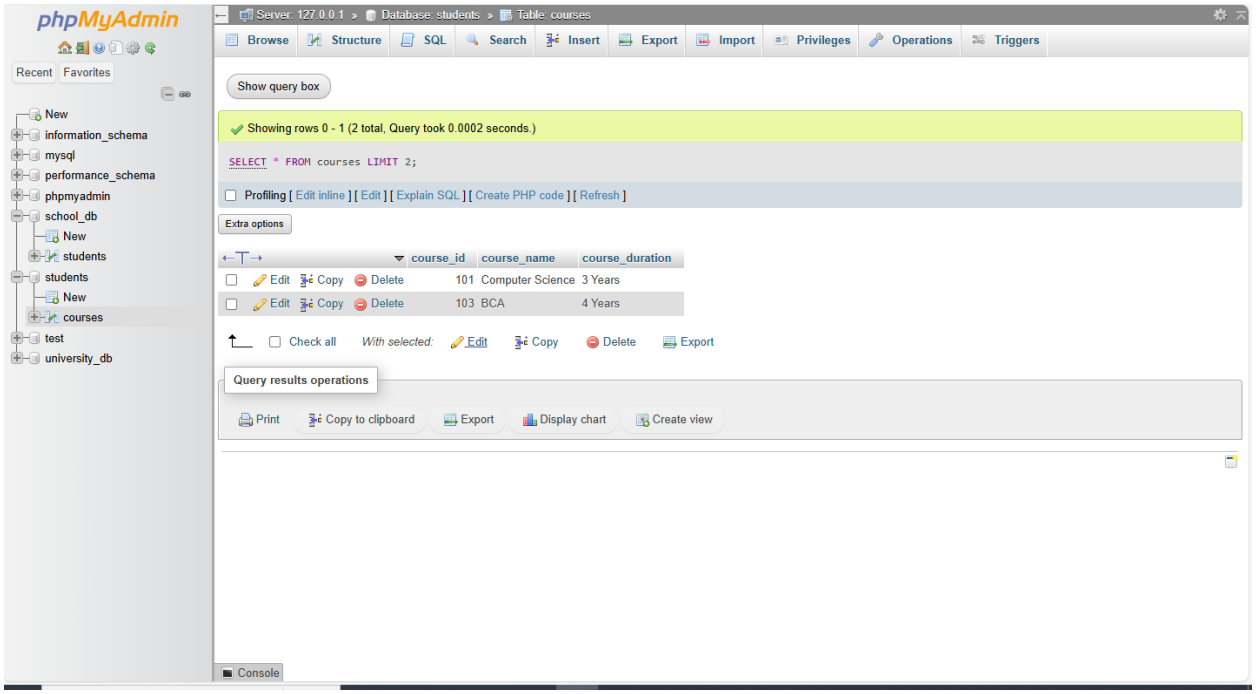
ORDER BY course_duration DESC;



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

<u>Query: Show only top 2 courses using LIMIT</u>

```
SELECT *

FROM courses

LIMIT 2;
```



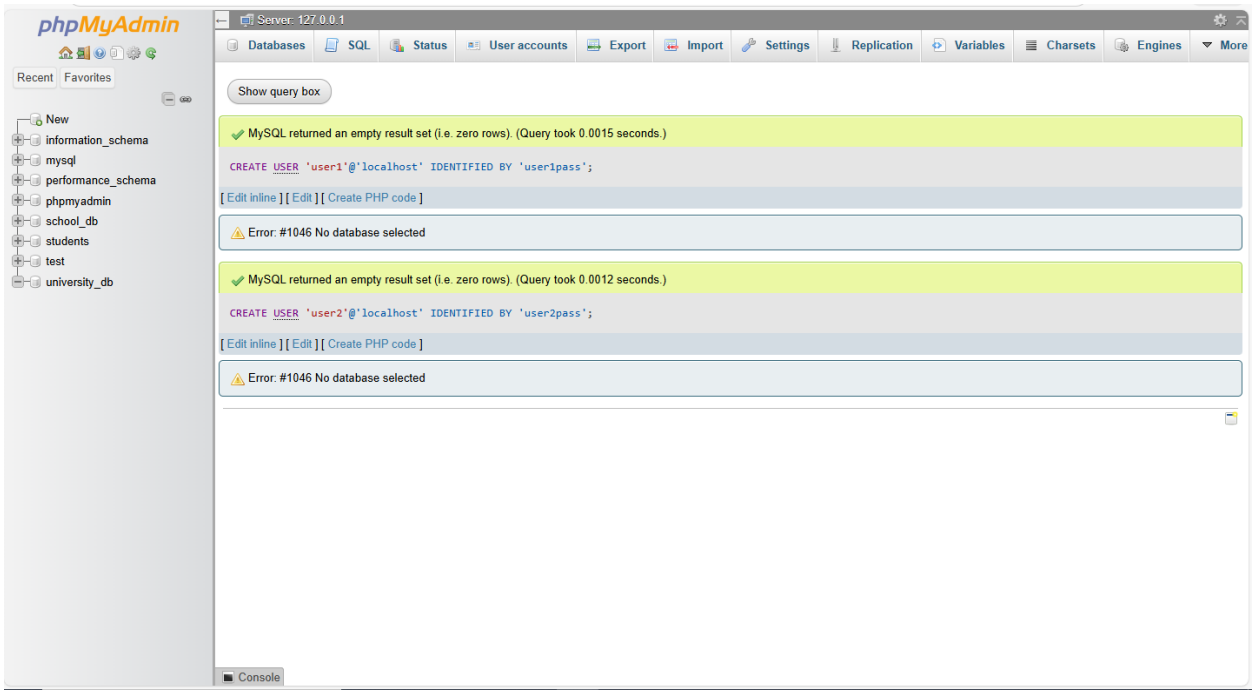
9. Data Control Language (DCL)

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

Query 1: Create two new users user1 and user2

CREATE USER 'user1'@'localhost' IDENTIFIED BY 'user1pass';

CREATE USER 'user2'@'localhost' IDENTIFIED BY 'user2pass';

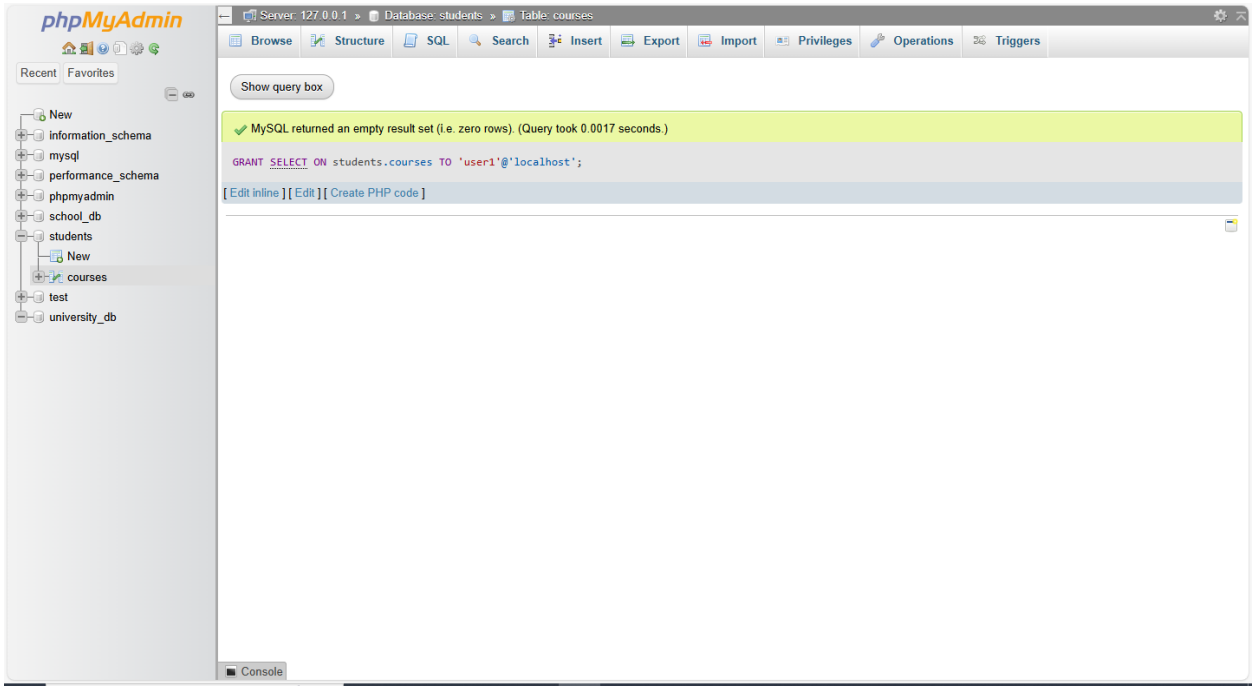


Query 2: Grant SELECT permission on courses table to user1

GRANT SELECT

ON students.courses

TO 'user1'@'localhost';



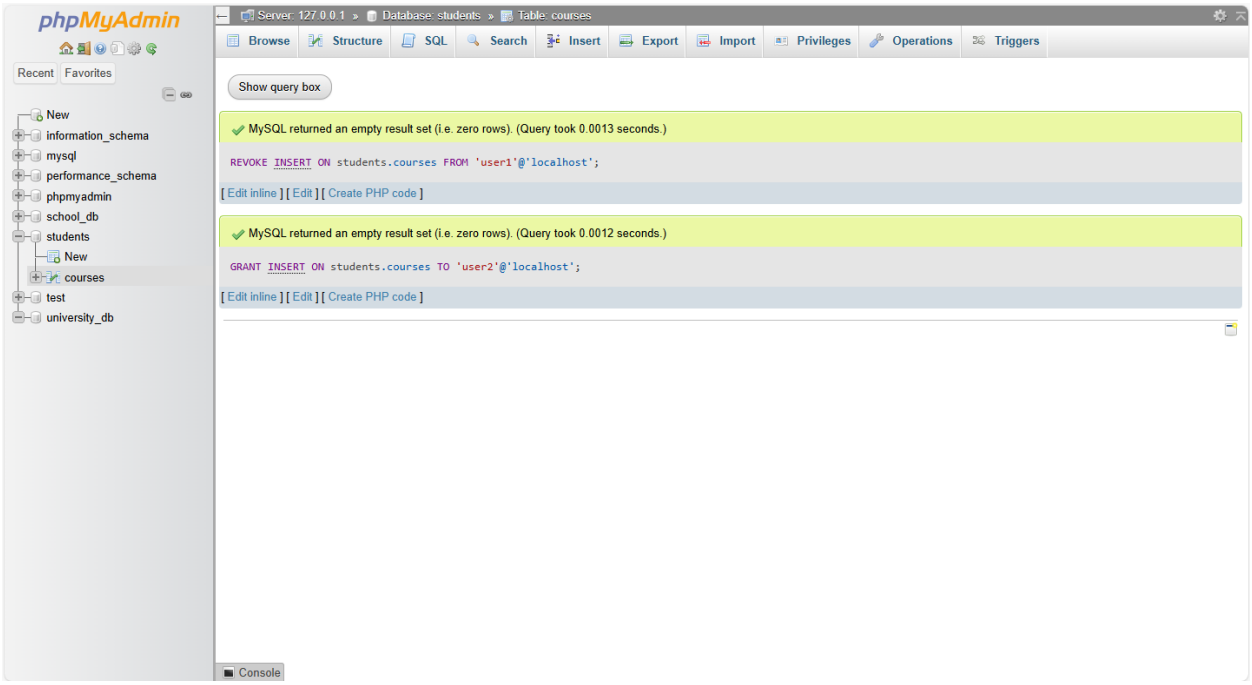
Lab 2: Revoke the INSERT permission from user1 and give it to user2.

Query 1: Revoke INSERT permission from user1

REVOKE INSERT students.courses FROM 'user1'@'localhost';

Query 2: Grant INSERT permission to user2

GRANT INSERT ON students.courses TO 'user2'@'localhost';



10. Transaction Control Language (TCL)

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

Query: Insert a few rows + COMMIT

START TRANSACTION;

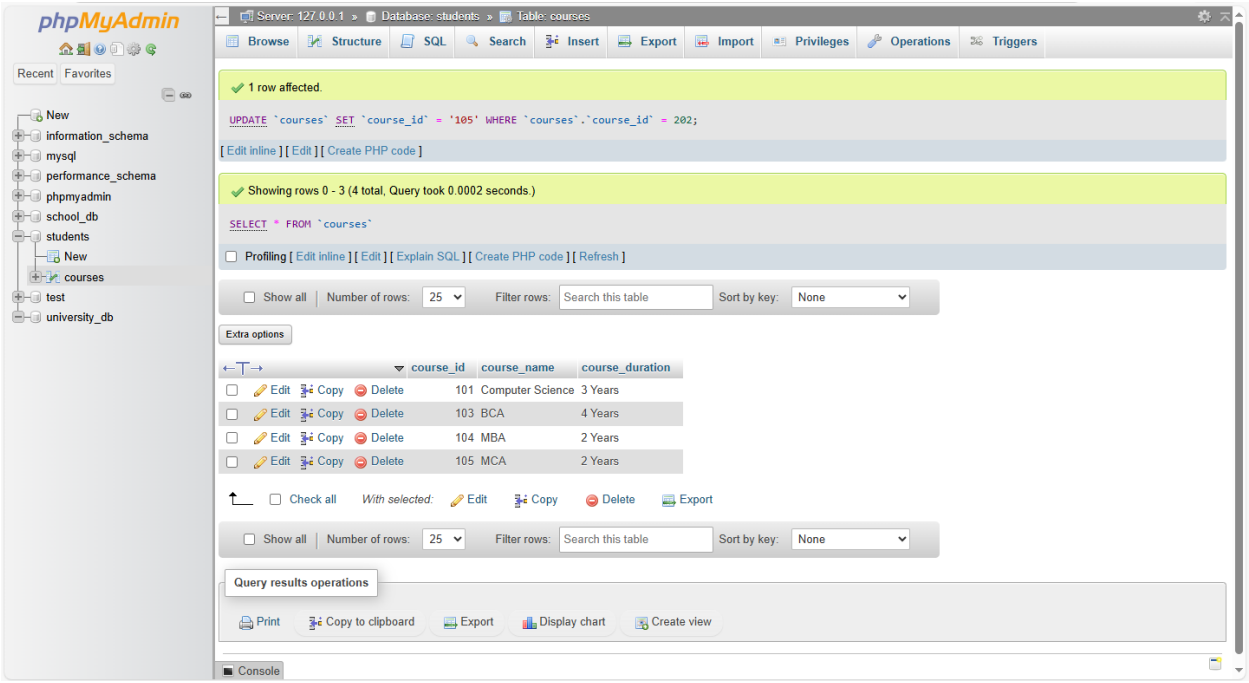
INSERT INTO courses (course_id, course_name, course_duration)

VALUES

(104, 'MBA', '2 Years'),

(105, 'MCA', '2 Years');

COMMIT;



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

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

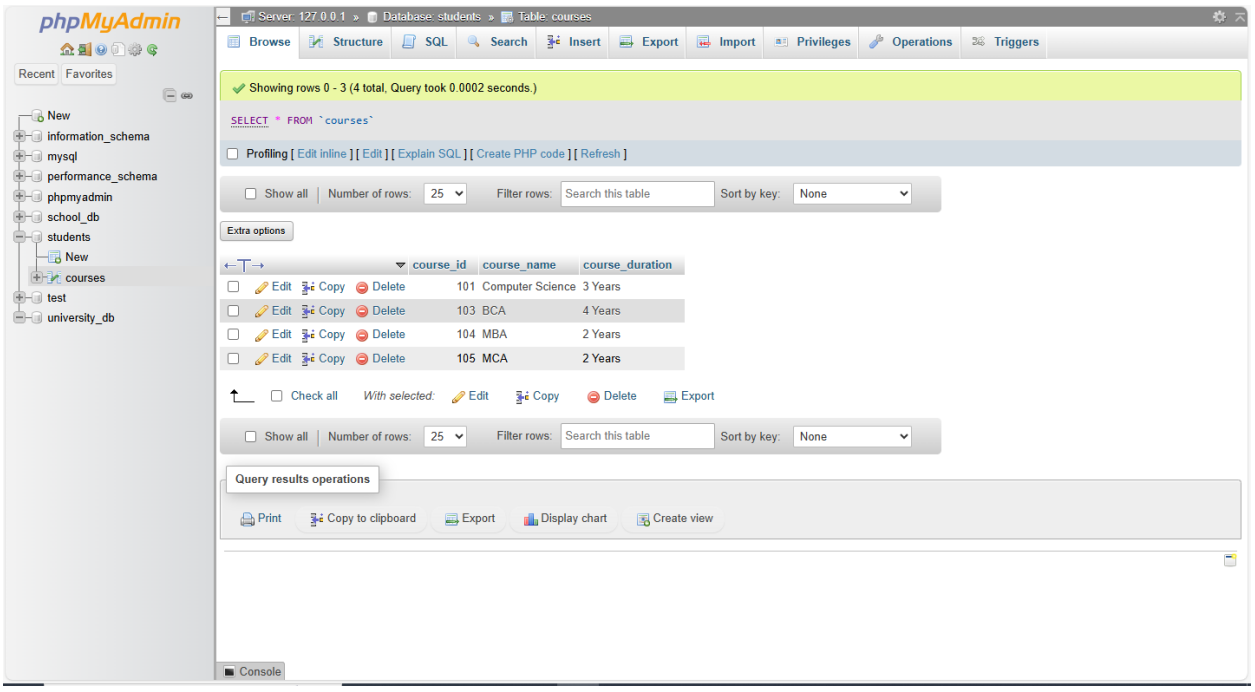
<u>Query: Insert additional rows + ROLLBACK</u>

START TRANSACTION;

INSERT INTO courses (course_id, course_name, course_duration)

VALUES (106, 'BBA', '3 Years');

ROLLBACK;

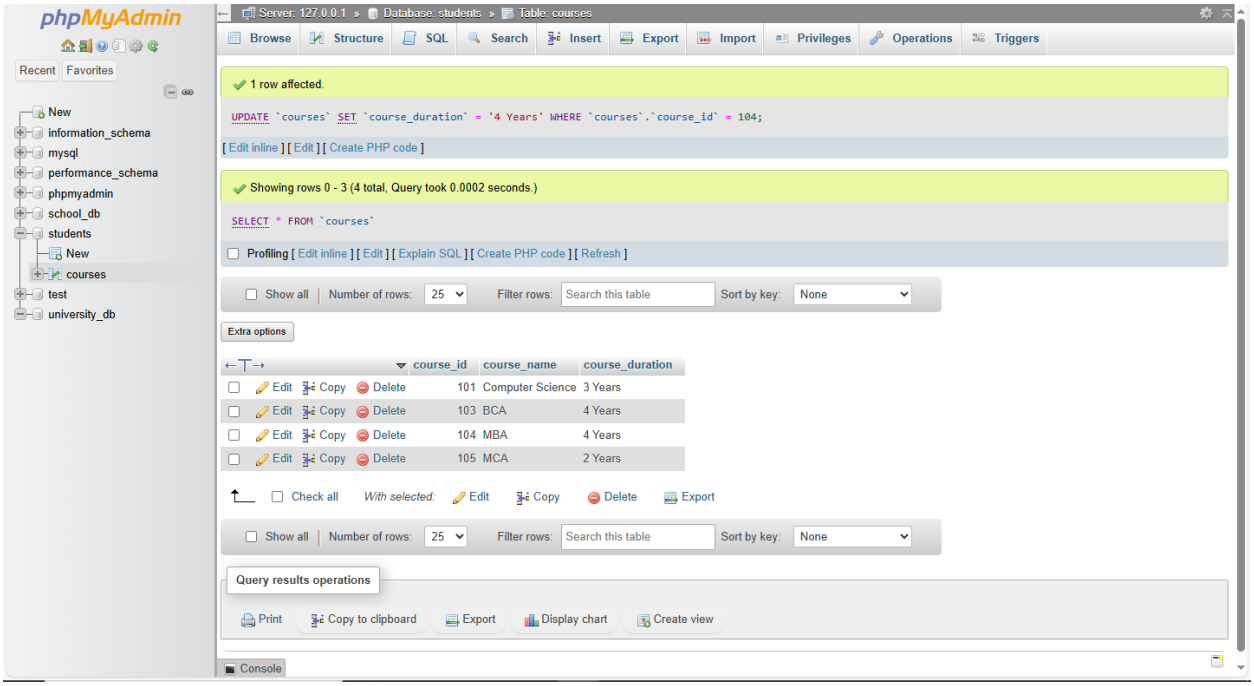


• **Course_id 106** will **NOT appear** because rollback cancelled the insert.

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

Query: SAVEPOINT + ROLLBACK TO SAVEPOINT

```
START TRANSACTION;

-- Create savepoint
SAVEPOINT before_update;

-- Update a course
UPDATE courses SET course_duration='4 Years' WHERE course_id=104;

-- Rollback only this update
ROLLBACK TO before_update;

-- Commit remaining changes
COMMIT;
```



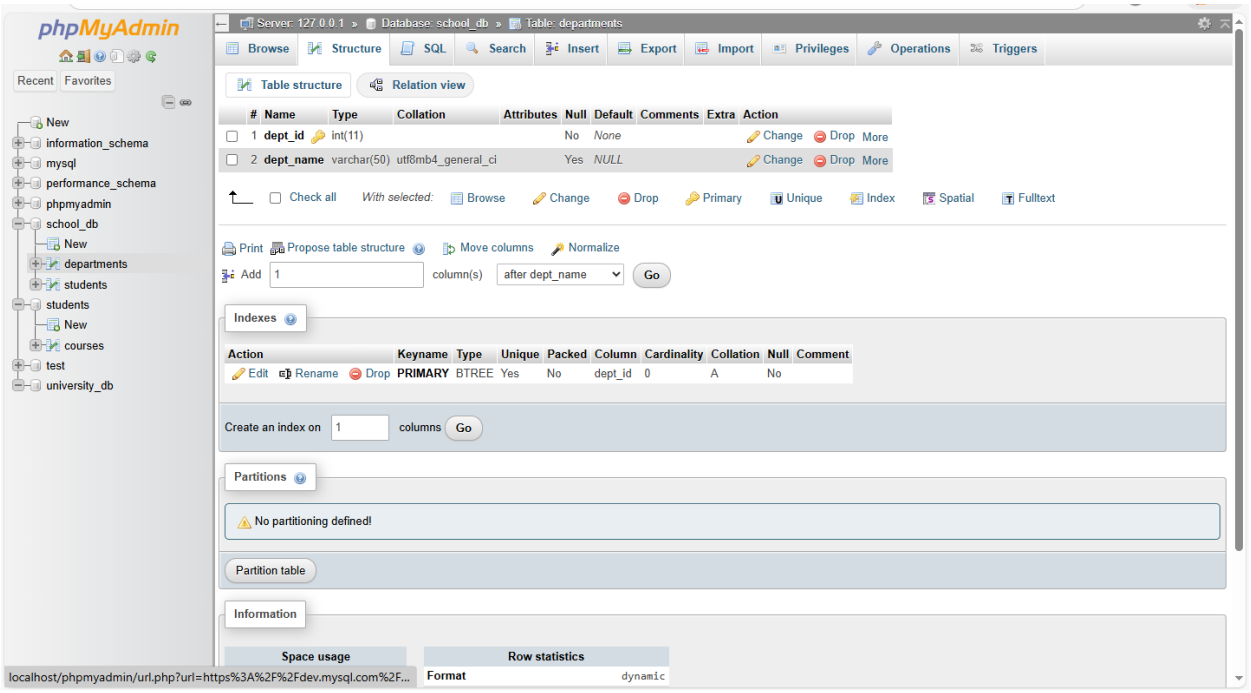
- Course_id 104 duration will remain ‘4 years’ (rollback undo the update)

11. SQL Joins

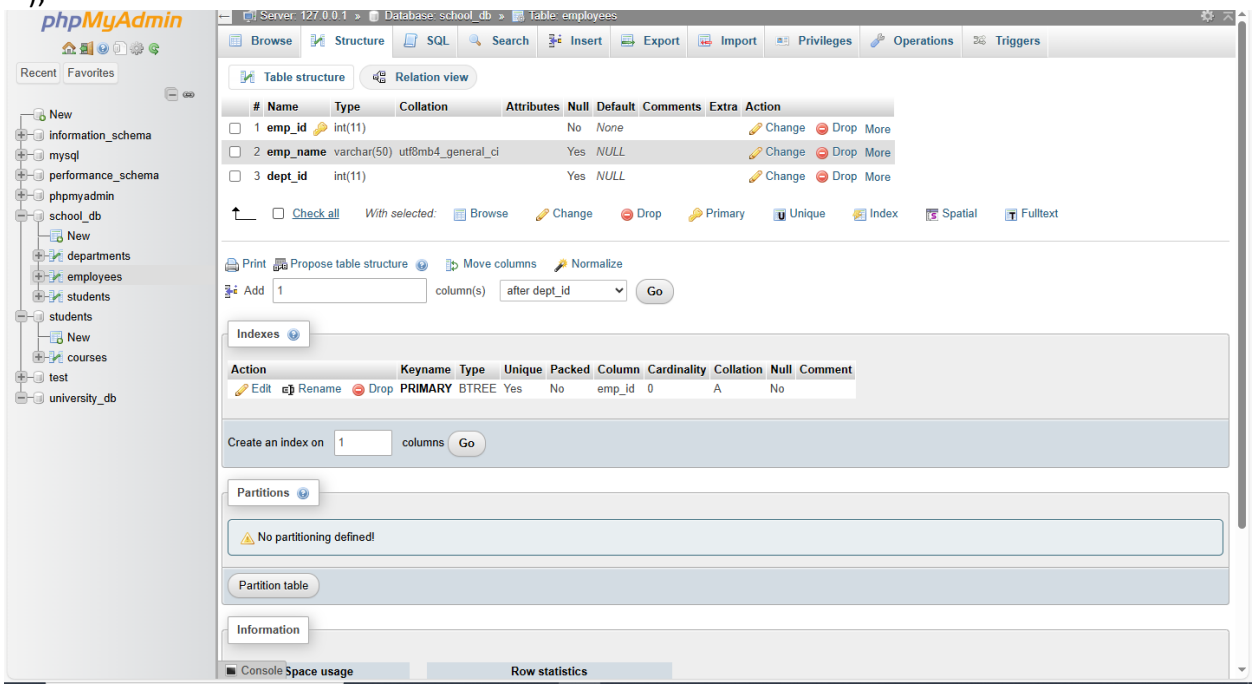
Lab 1: Create two tables: departments and employees. Perform an INNER JOIN to display employees along with their respective departments.

<u>Query 1: Create two tables</u>

1. CREATE TABLE departments (
   dept_id INT PRIMARY KEY,
   dept_name VARCHAR(50)
   );

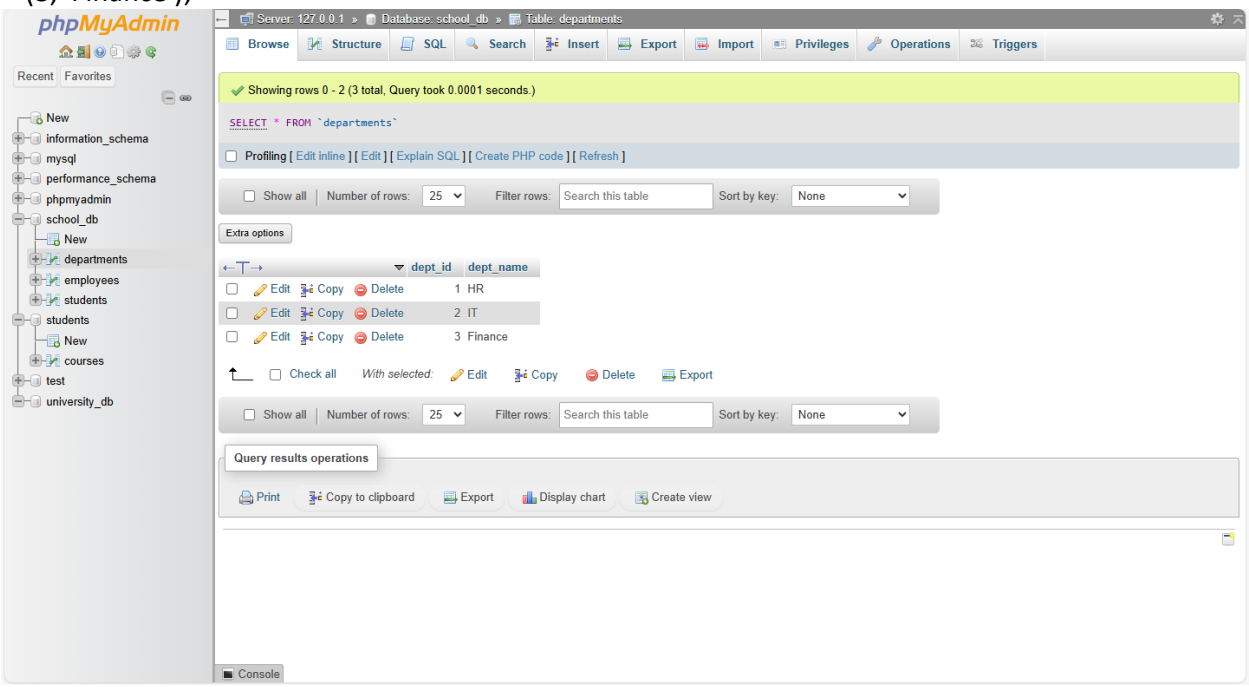


2. CREATE TABLE employees (
   emp_id INT PRIMARY KEY,
   emp_name VARCHAR(50),
   dept_id INT
   );

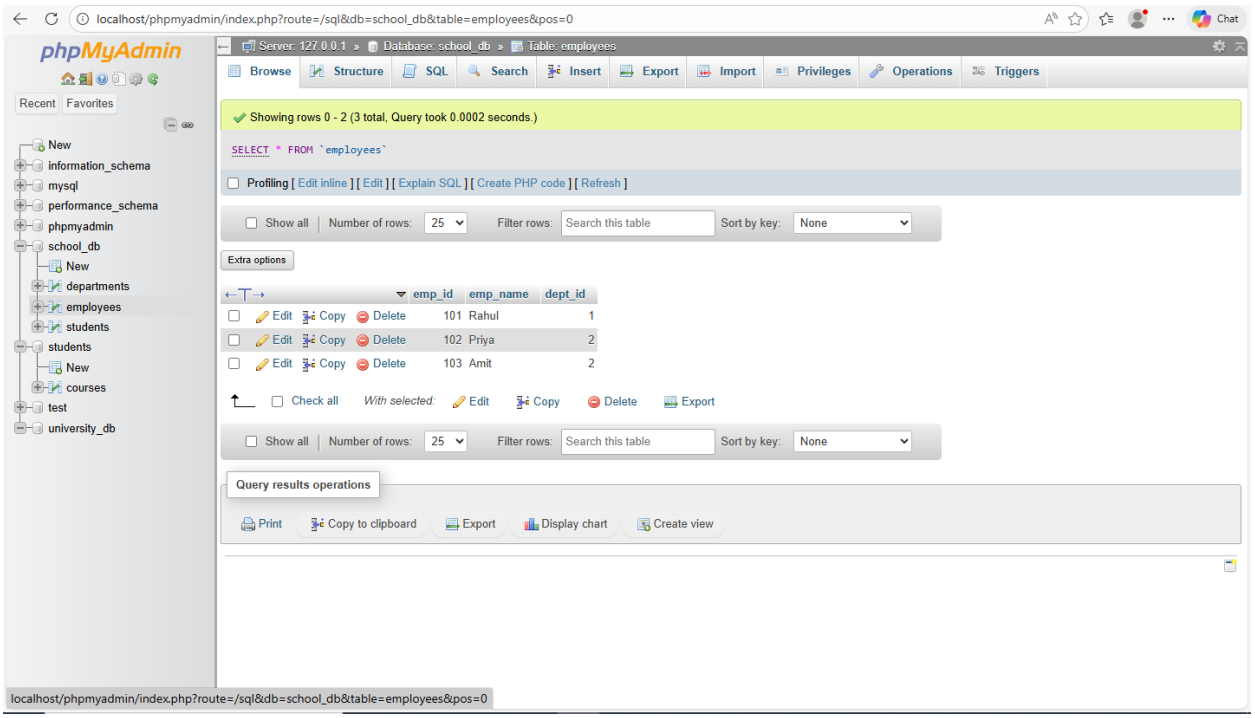


Query 2: Insert sample records

1. INSERT INTO departments VALUES
   (1, 'HR'),
   (2, 'IT'),
   (3, 'Finance');

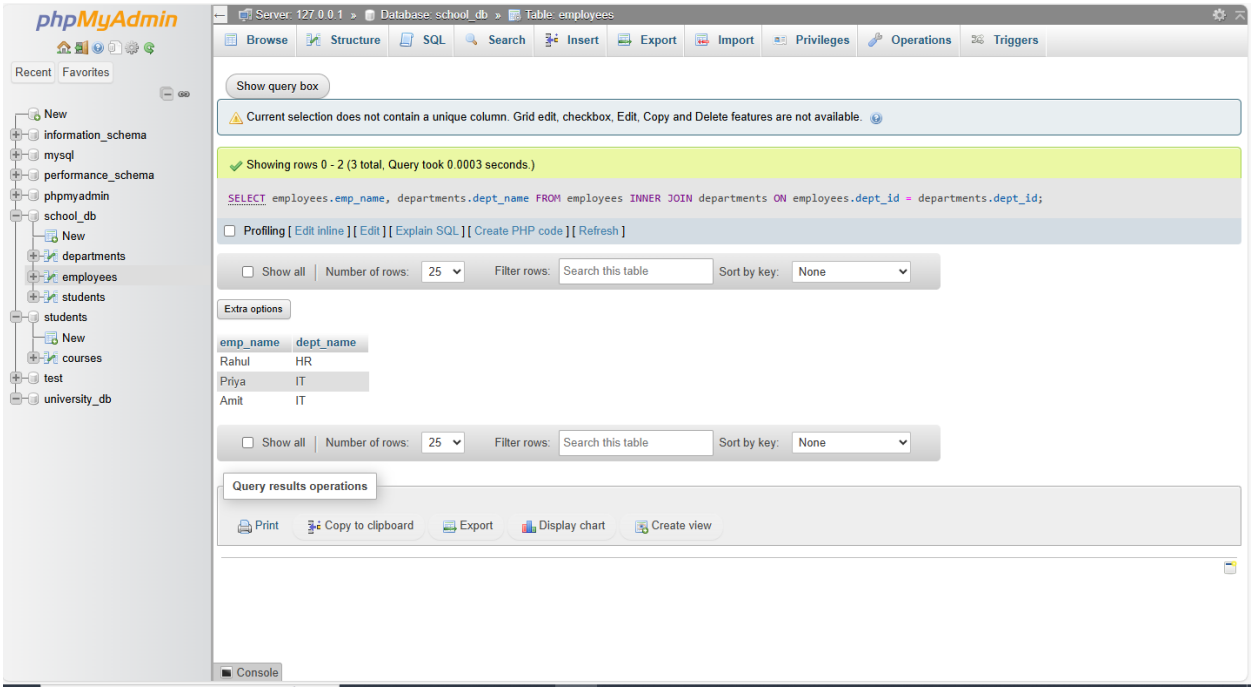


2. INSERT INTO employees VALUES
(101, 'Rahul', 1),
(102, 'Priya', 2),
(103, 'Amit', 2);



Query 3: Perform INNER JOIN

SELECT employees.emp_name, departments.dept_name
FROM employees
INNER JOIN departments
ON employees.dept_id = departments.dept_id;

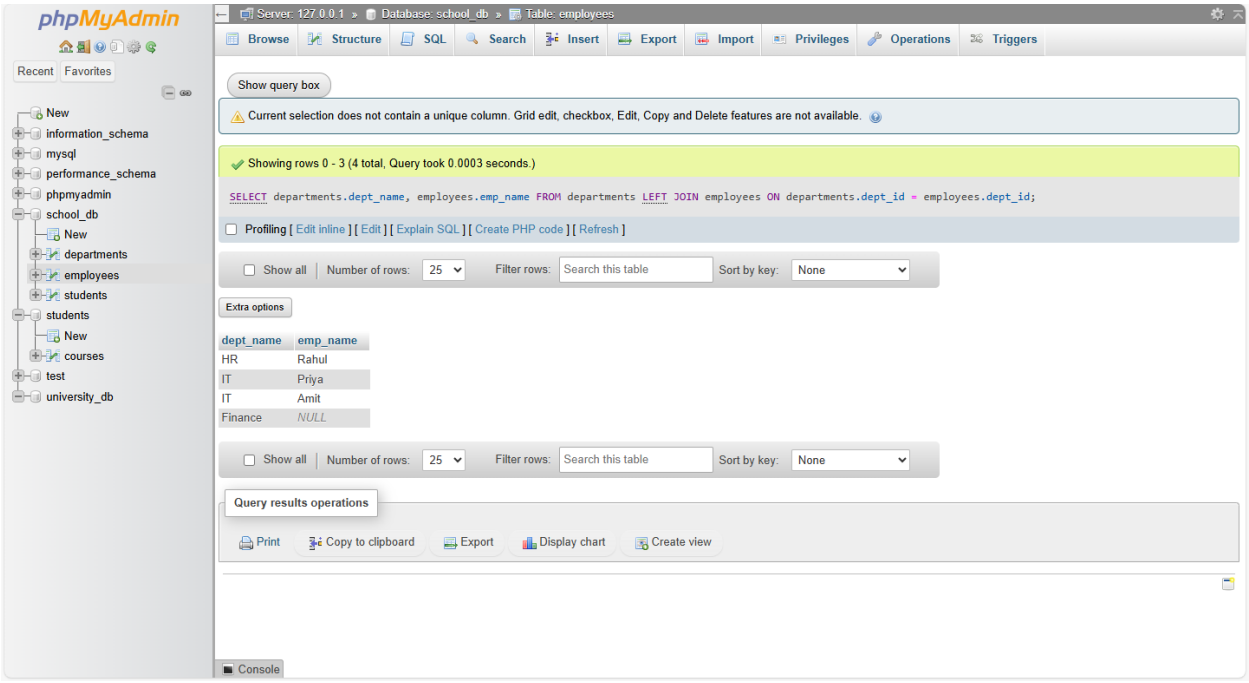


• INNER JOIN shows **only matching** employee – department records.

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

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

```
SELECT departments.dept_name, employees.emp_name
FROM departments
LEFT JOIN employees
ON departments.dept_id = employees.dept_id;
```

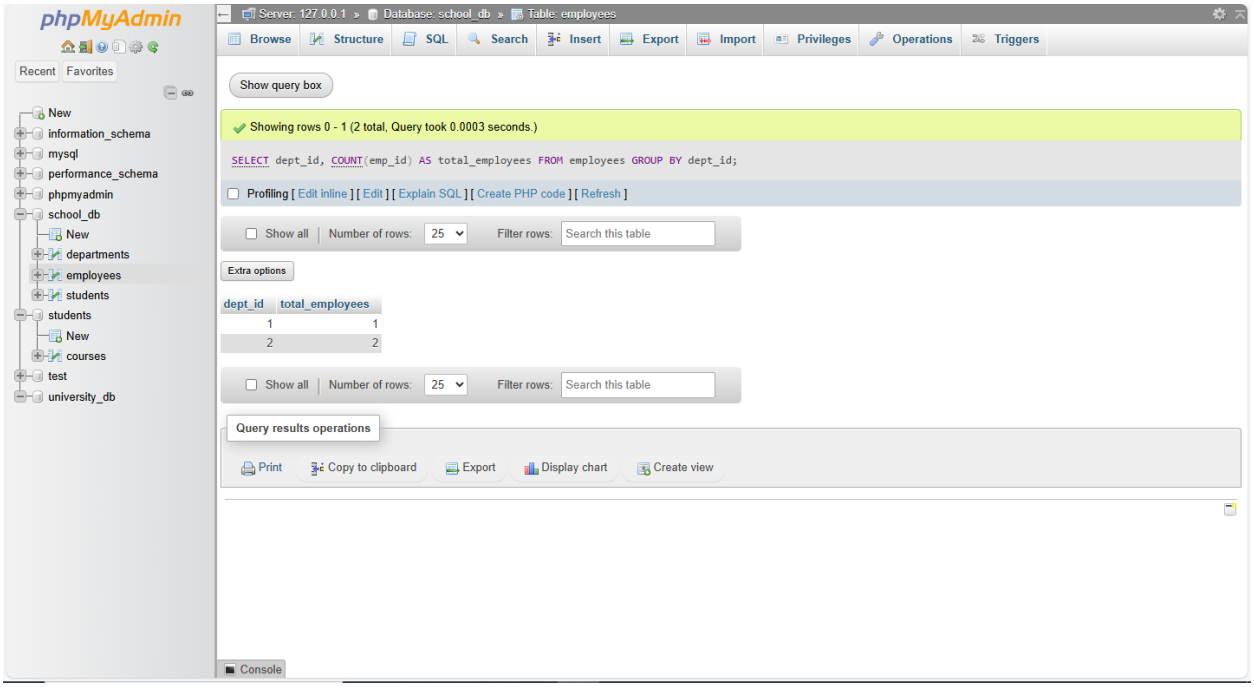


12. SQL Group By

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

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.

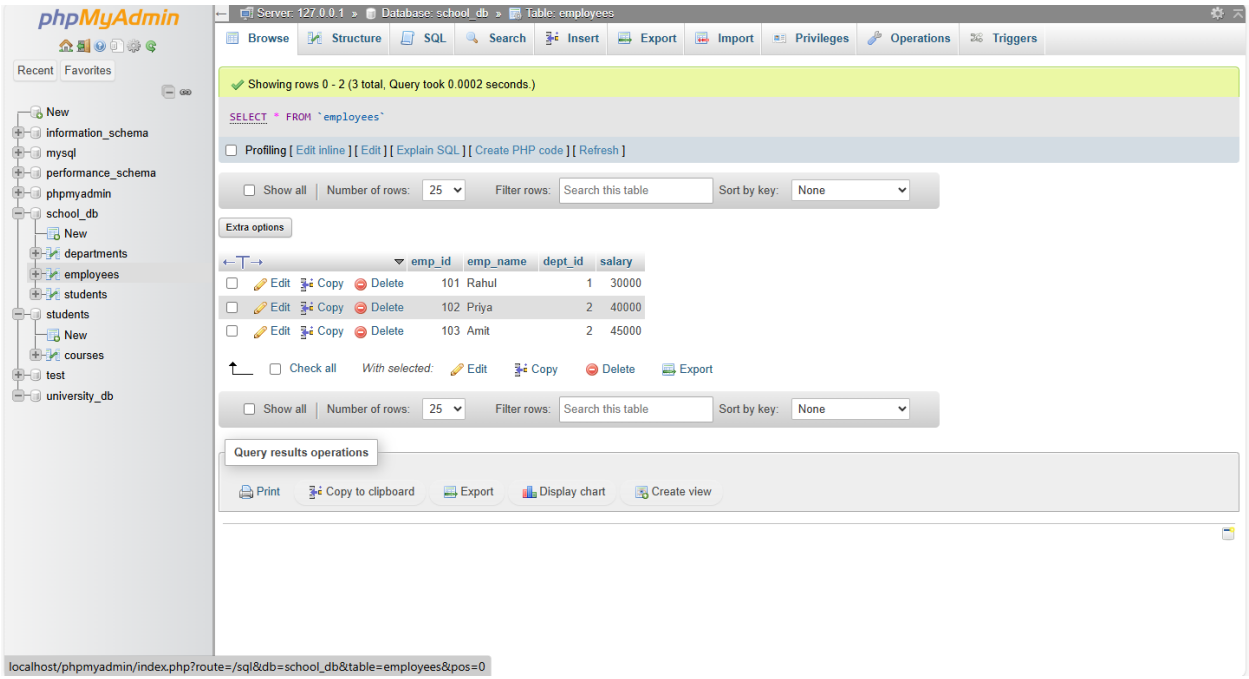
Important note:

. 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 = 30000 WHERE emp_id = 101;
UPDATE employees SET salary = 40000 WHERE emp_id = 102;
UPDATE employees SET salary = 45000 WHERE emp_id = 103;

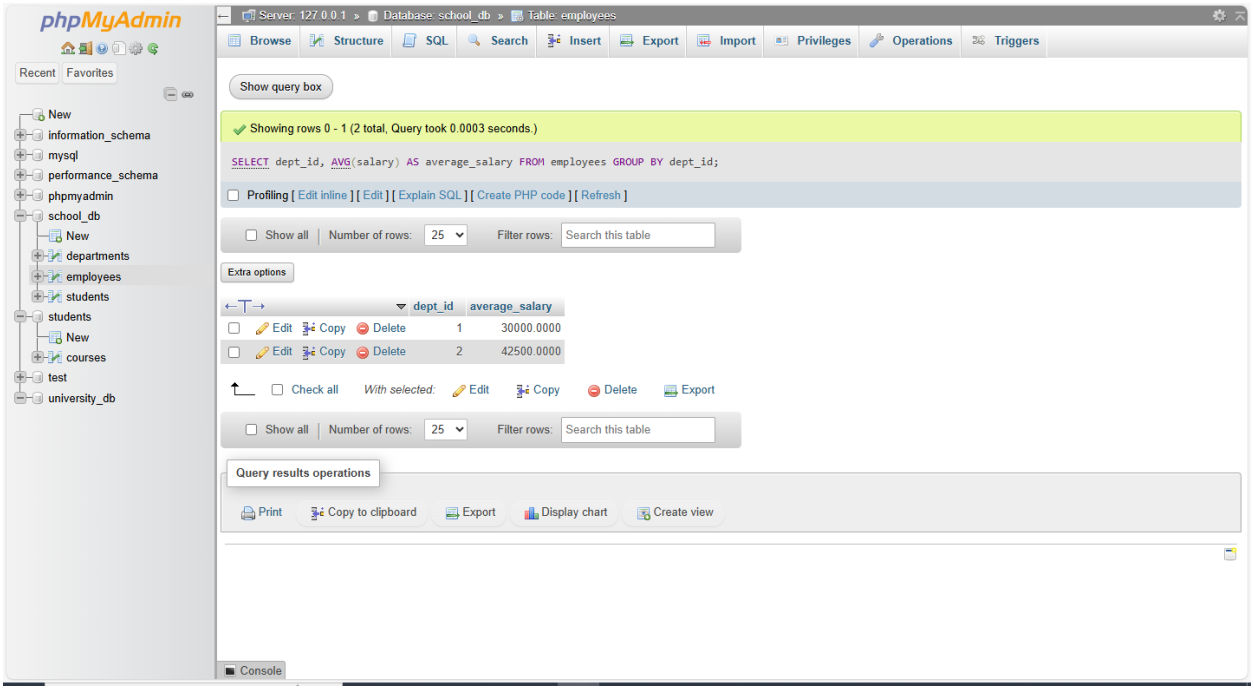


Query: Find Average Salary per Department (Using AVG)

SELECT dept_id, AVG(salary) AS average_salary

FROM employees

GROUP BY dept_id;



13. SQL Stored Procedure

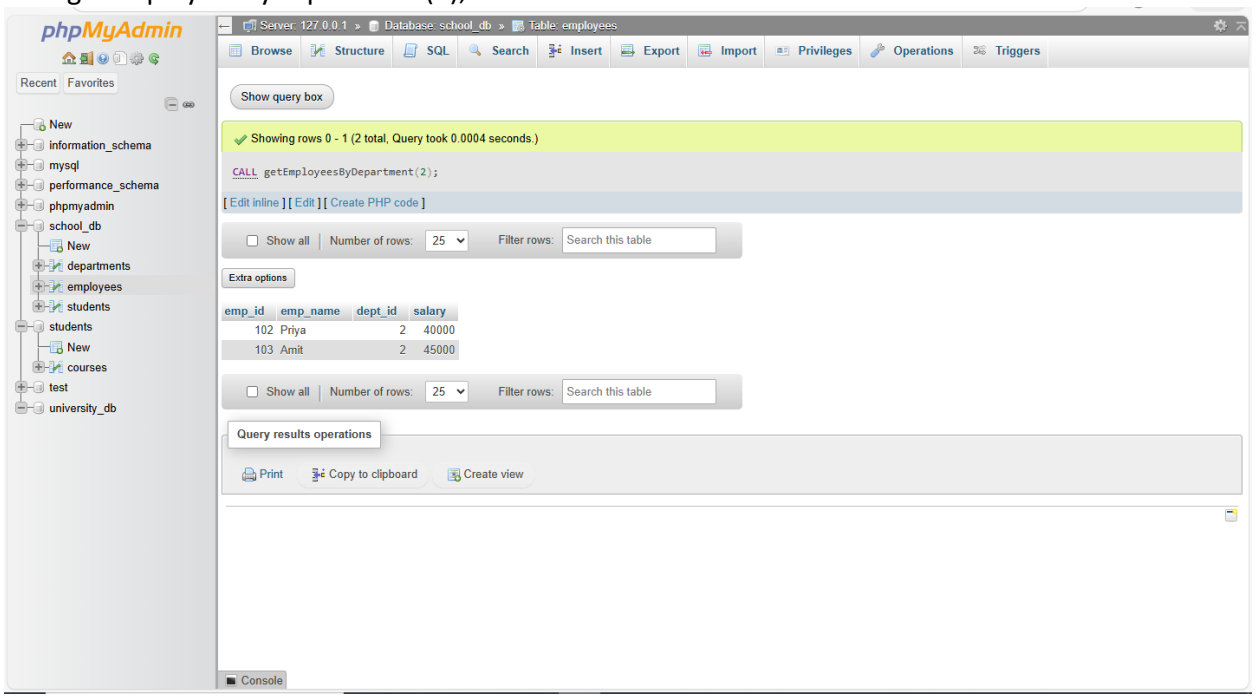
Lab 1: Write a stored procedure to retrieve all employees from the employees table based on department.

Query: Stored Procedure to Retrieve Employees by Department

```
DELIMITER $$


CREATE PROCEDURE getEmployeesByDepartment(IN d_id INT)

BEGIN

   SELECT *

   FROM employees

   WHERE dept_id = d_id;

END $$

DELIMITER ;
```

- **How to Execute the Procedure**

```
CALL getEmployeesByDepartment(2);
```



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

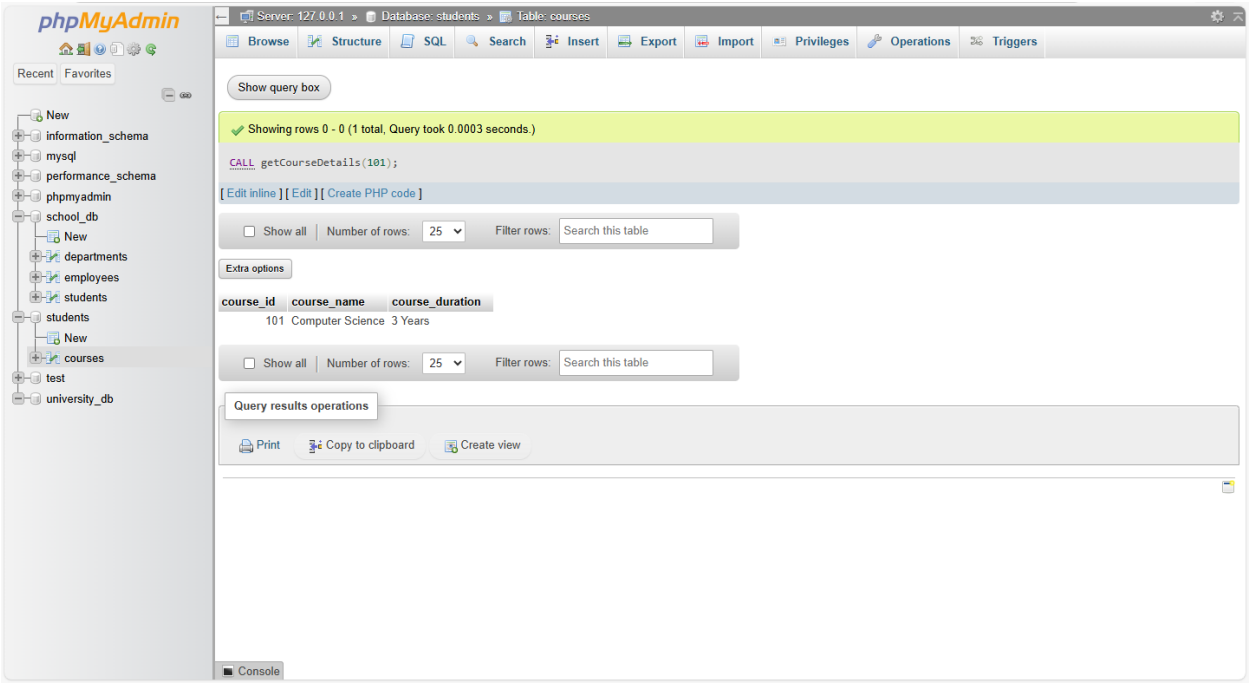
Lab 2: Write a stored procedure that accepts course_id as input and returns the course details.

Query: Stored Procedure to Retrieve Course Details by course_id

```
DELIMITER $$

CREATE PROCEDURE getCourseDetails(IN c_id INT)
BEGIN
  SELECT *
  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.

14. SQL View

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

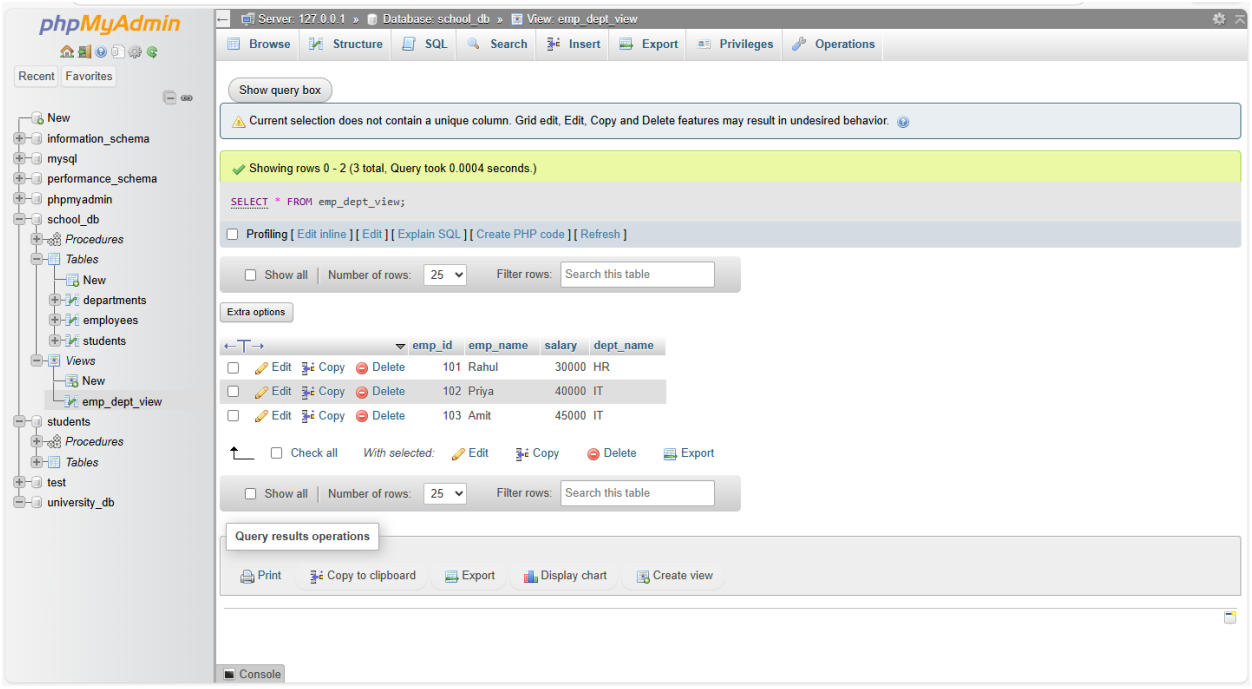
Query: Create a View to Show Employees with Department Names

Step 1: Create a VIEW using JOIN

```
CREATE VIEW emp_dept_view AS
SELECT
   employees.emp_id,
   employees.emp_name,
   employees.salary,
   departments.dept_name
FROM employees
INNER JOIN departments
ON employees.dept_id = departments.dept_id;
```

Step 2: To see the view

```
SELECT * FROM emp_dept_view;
```



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

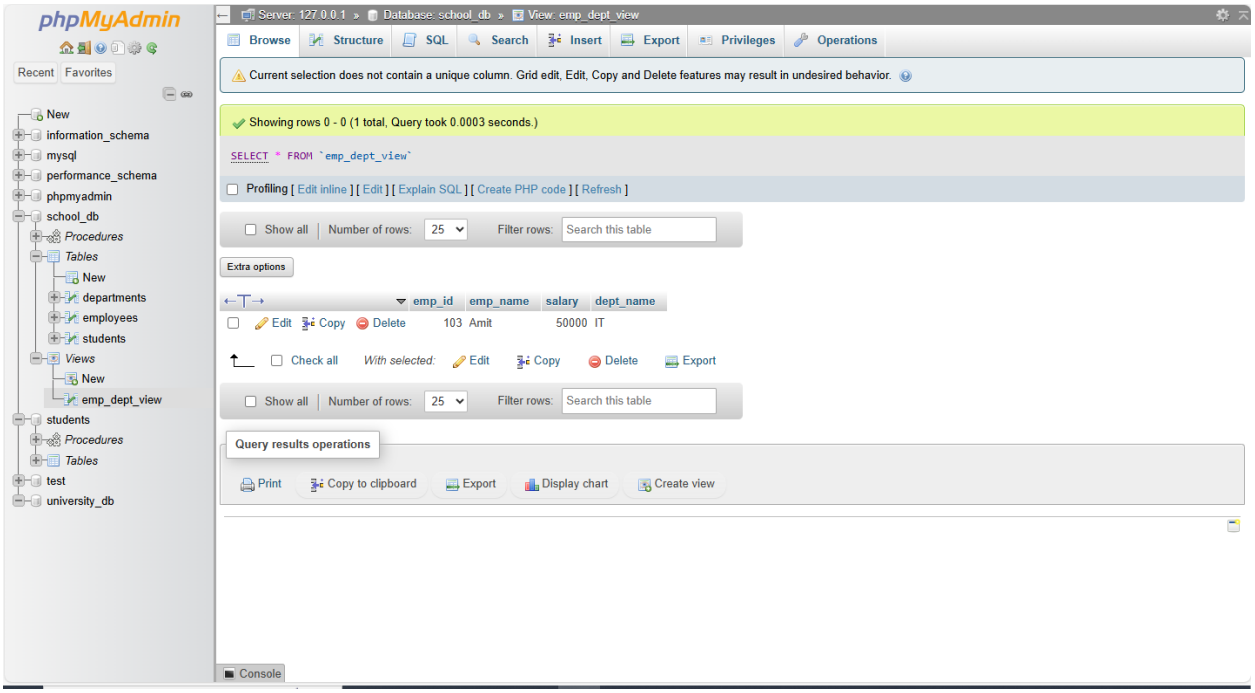
```
CREATE OR REPLACE VIEW emp_dept_view AS

SELECT

   employees.emp_id,

   employees.emp_name,

   employees.salary,

   departments.dept_name

FROM employees

INNER JOIN departments

ON employees.dept_id = departments.dept_id

WHERE employees.salary >= 50000;
```

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

```
Use OR REPLACE to update the view
CREATE OR REPLACE VIEW emp_dept_view AS
SELECT
   employees.emp_id,
   employees.emp_name,
   employees.salary,
   departments.dept_name
FROM employees
INNER JOIN departments
ON employees.dept_id = departments.dept_id
WHERE employees.salary >= 50000;
```

To check updated view

```
SELECT * FROM `emp_dept_view`
```



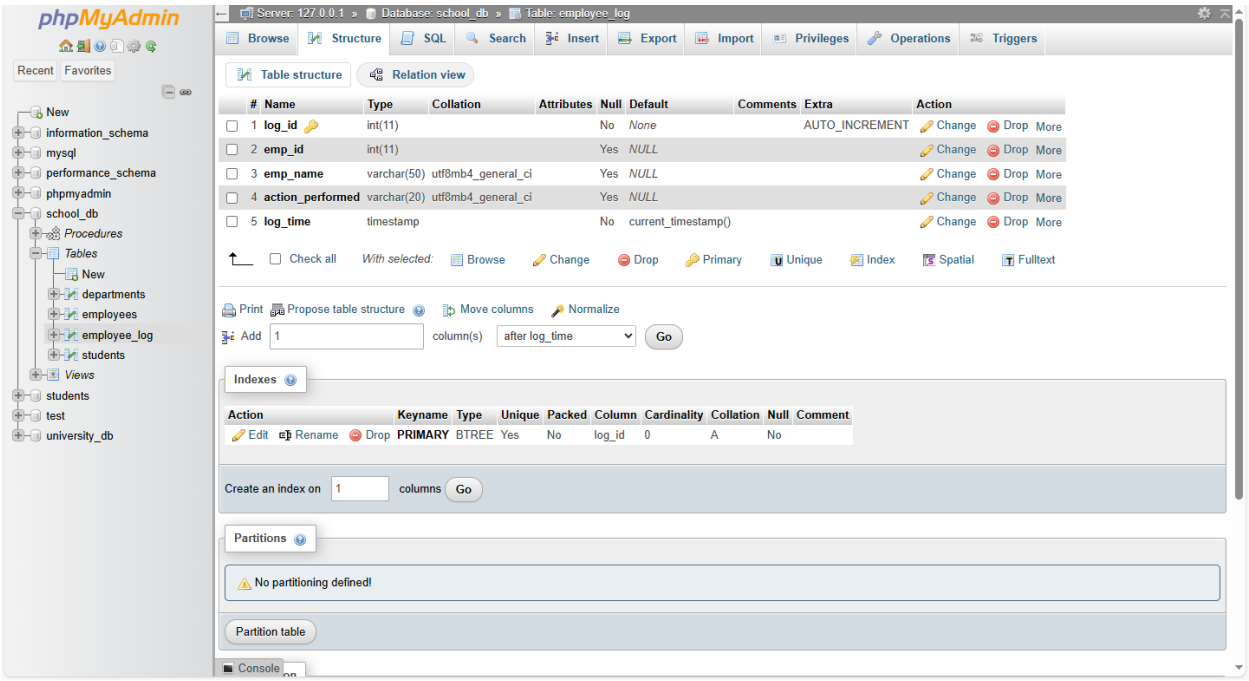
15. SQL Triggers

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

<u>Query: Trigger to Log New Employee Insertions</u>

Step 1: Create a log table

```
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
);
```



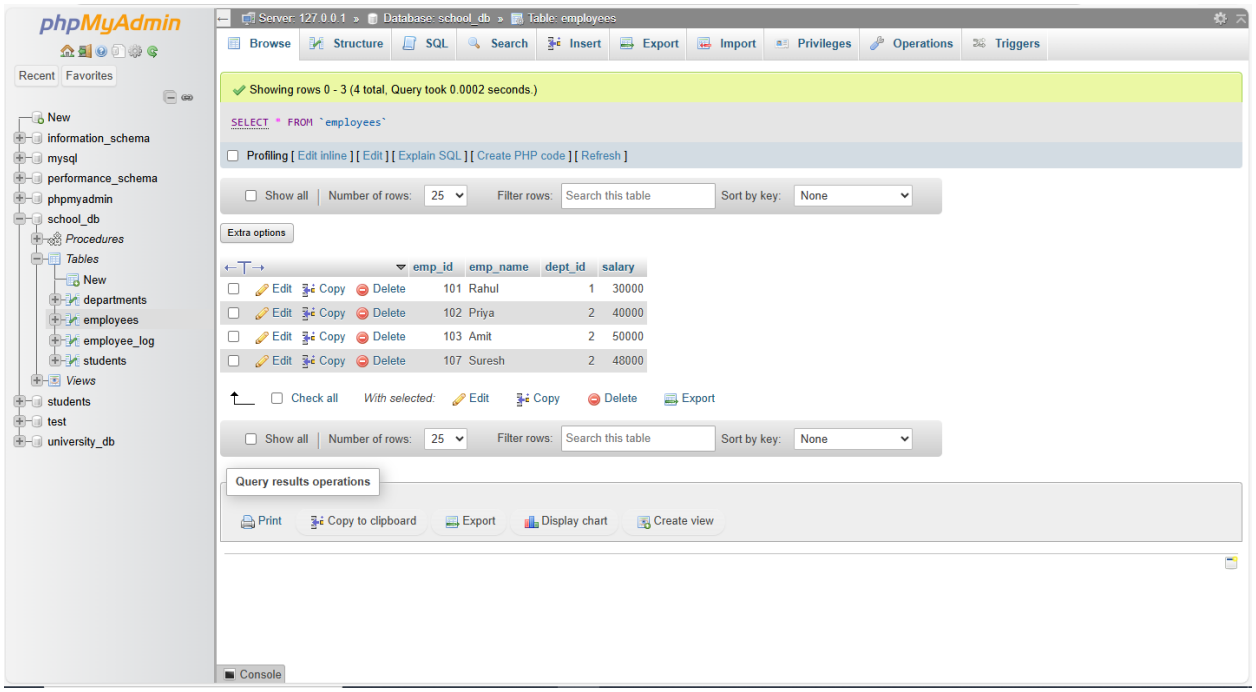
- **Step 2: Create the trigger**

```
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, 'Suresh', 2, 48000);
```



• Running:

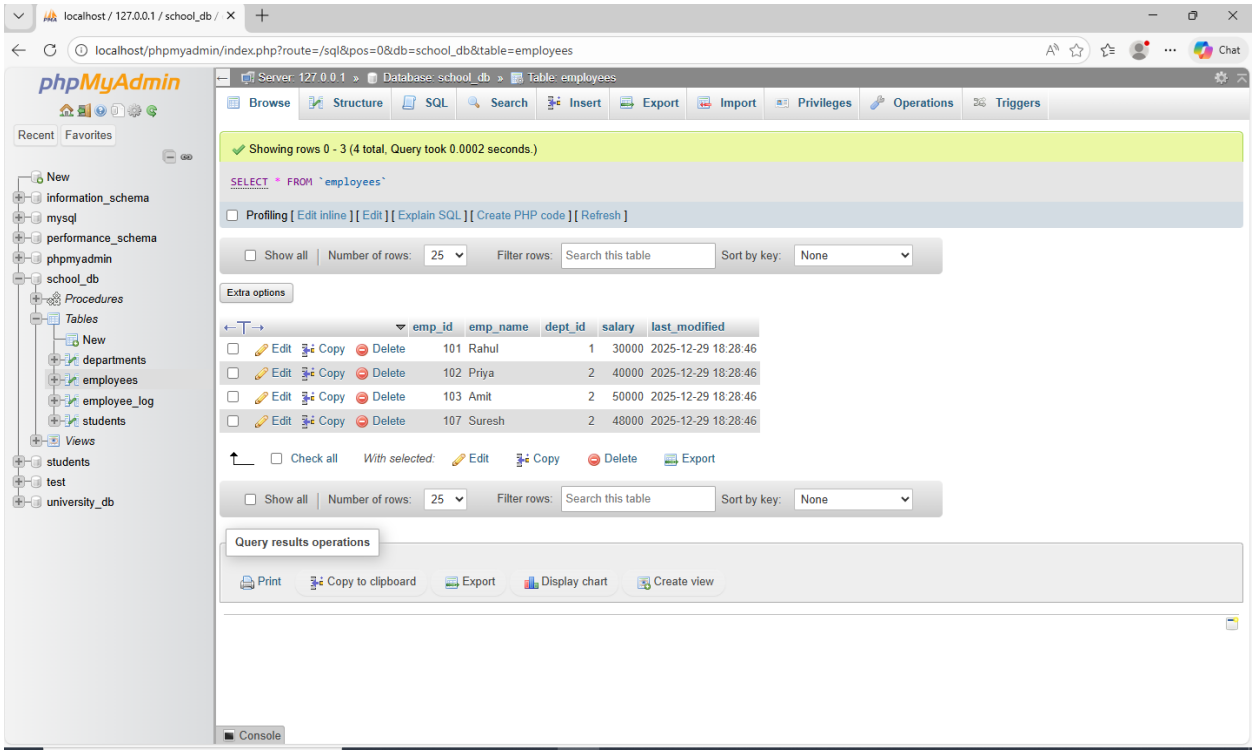
SELECT * FROM employee_log;

Lab 2: Create a trigger to update the last_modified timestamp whenever an employee record is updated.

Query: Trigger to Automatically Update last_modified Timestamp

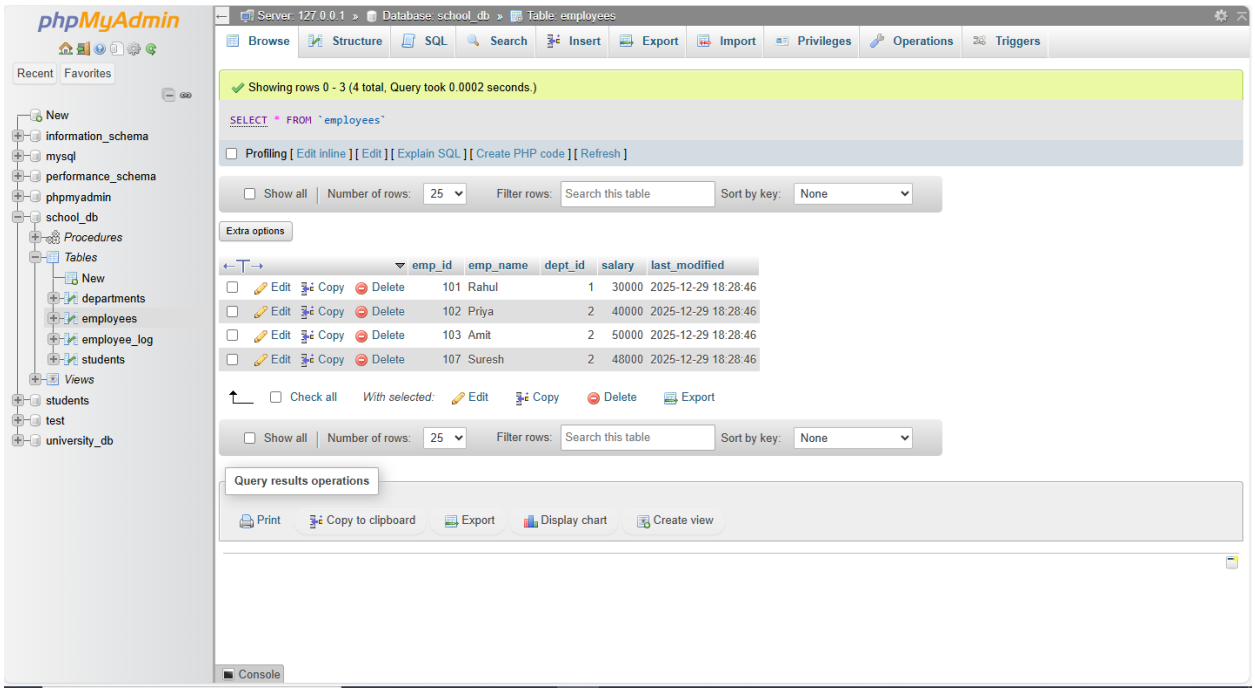
• Step 1: Add a column to employees table

ALTER TABLE employees
ADD last_modified TIMESTAMP;



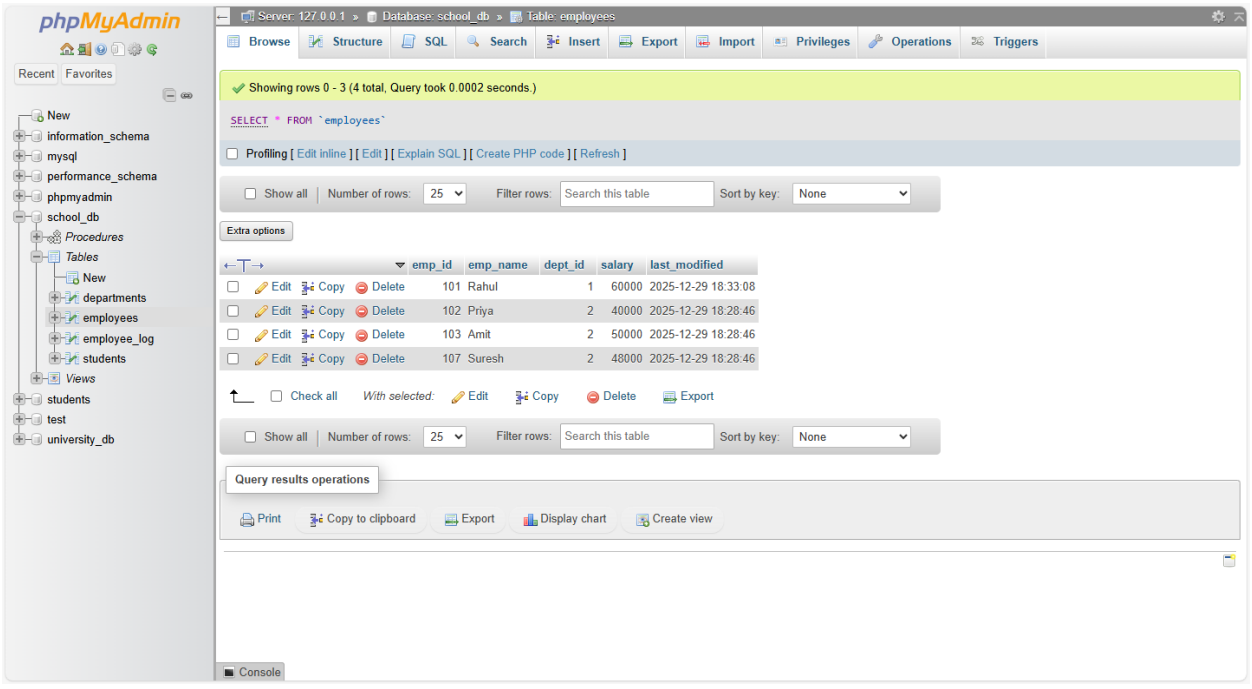
- Step 2: Create the UPDATE trigger

```
DELIMITER $$

CREATE TRIGGER before_employee_update
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
   SET NEW.last_modified = CURRENT_TIMESTAMP;
END $$

DELIMITER ;
```



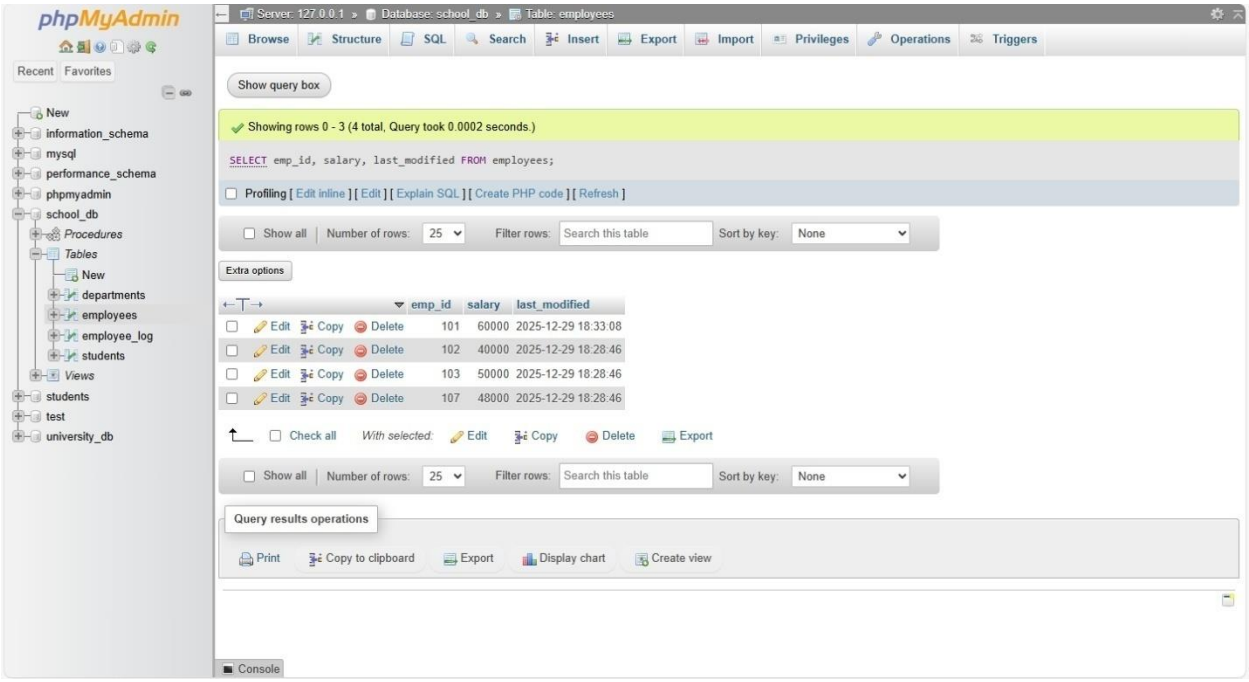
- Example Update

UPDATE employees
SET salary = 60000
WHERE emp_id = 101;



• Output (View the updated timestamp)

SELECT emp_id, salary, last_modified FROM employees;



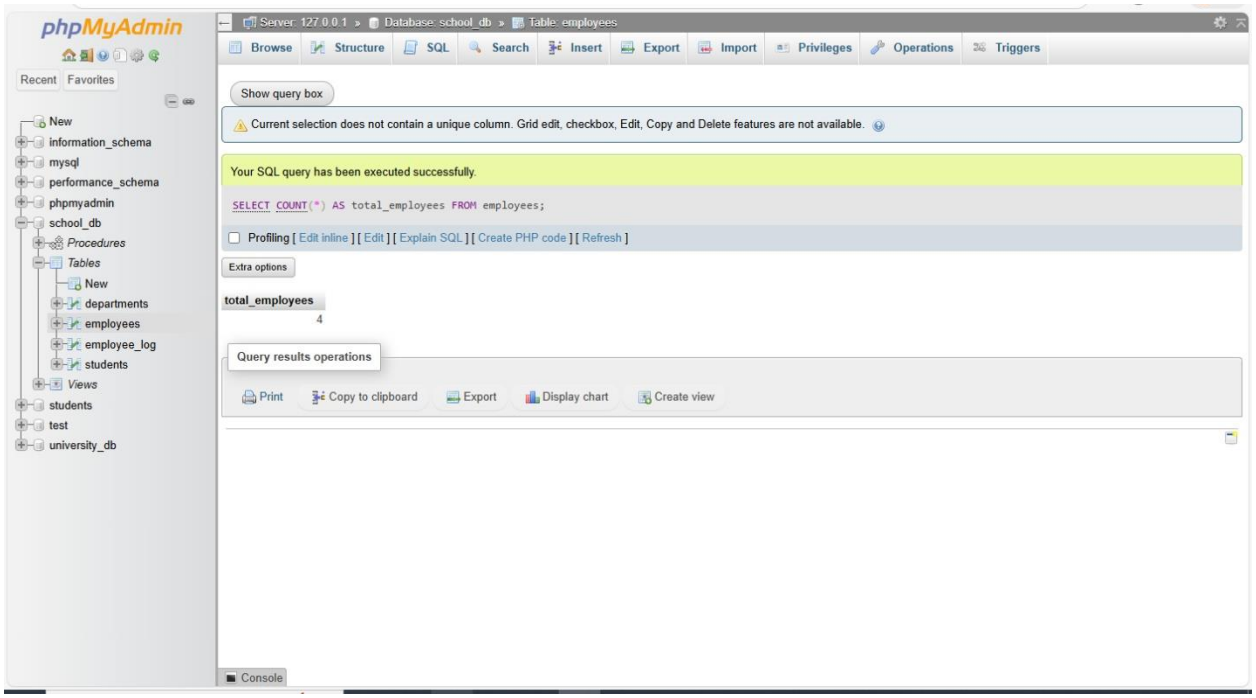
• Automatically updated — no need to manually set last_modified.

16. Introduction to PL/SQL

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

Query: Print total number of employees

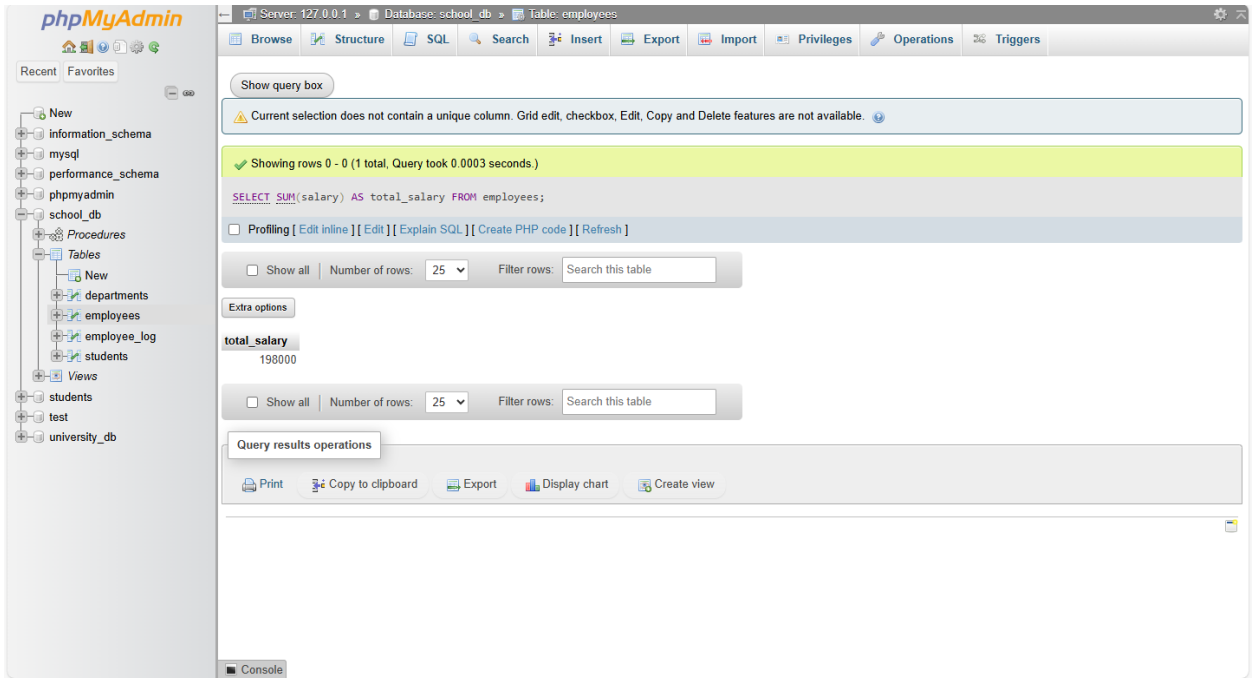
SELECT COUNT(*) AS total_employees FROM employees;



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

Query: Calculate Total Salary

SELECT SUM(salary) AS total_salary FROM employees;



17. PL/SQL Control Structures

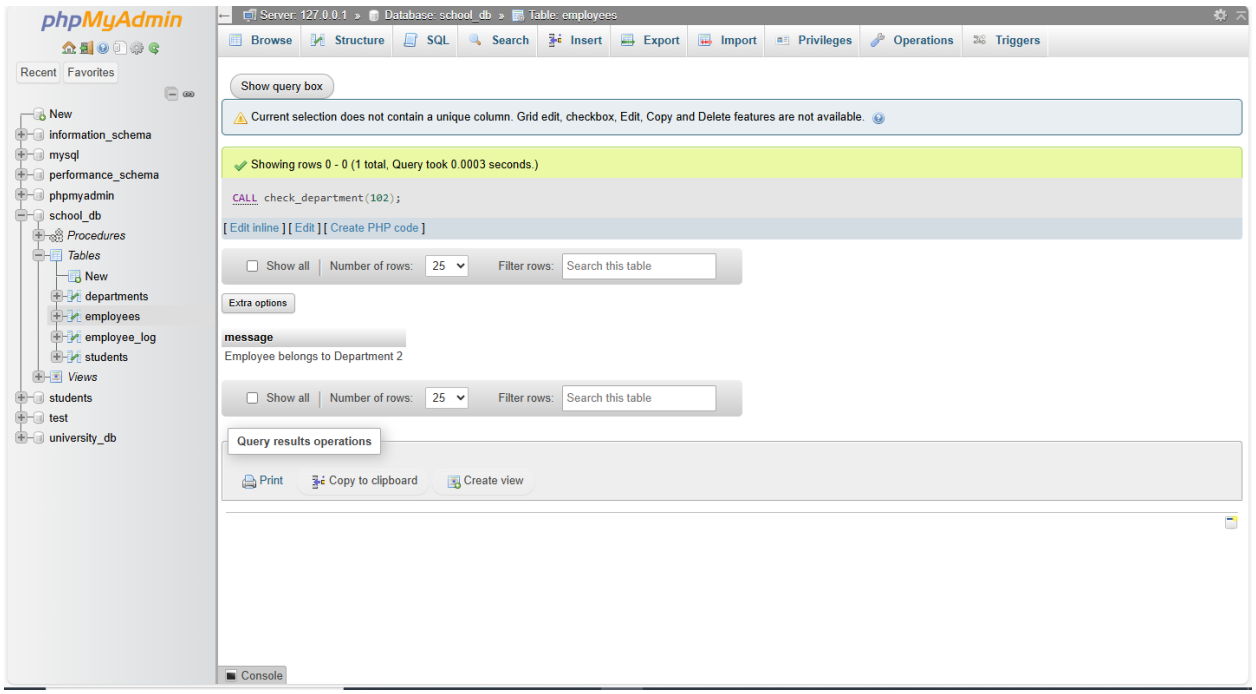
Lab 1: Write a PL/SQL block using an IF-THEN condition to check the department of an employee.

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);
```



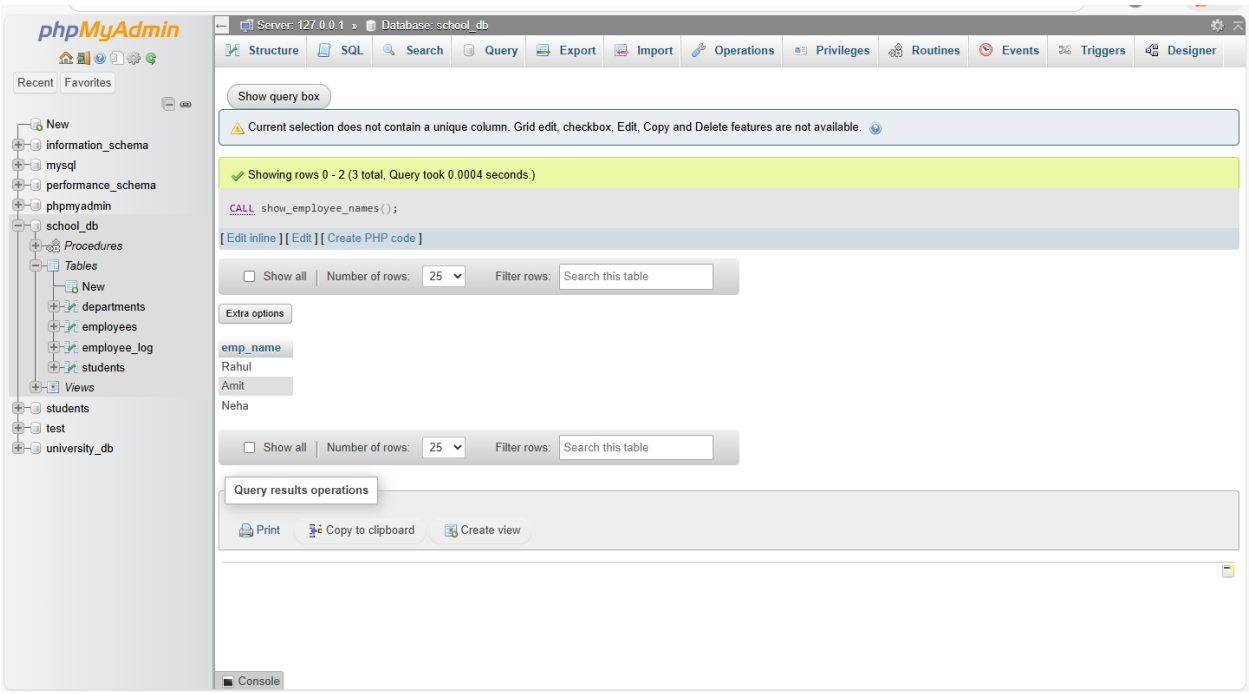
Lab 2: Use a FOR LOOP to iterate through employee records and display their names.

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

```
DELIMITER $$

CREATE PROCEDURE DisplayEmployeeNames()
BEGIN
   DECLARE done INT DEFAULT 0;
   DECLARE ename VARCHAR(50);

   DECLARE emp_cursor CURSOR FOR
      SELECT emp_name FROM employee;

   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

   OPEN emp_cursor;

   read_loop: LOOP
      FETCH emp_cursor INTO ename;

      IF done = 1 THEN
         LEAVE read_loop;
      END IF;

      SELECT ename AS Employee_Name;
   END LOOP;

   CLOSE emp_cursor;
END $$

DELIMITER ;
```

Query 2: Call the procedure

```
CALL show_employee_names();
```



18. SQL Cursors

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

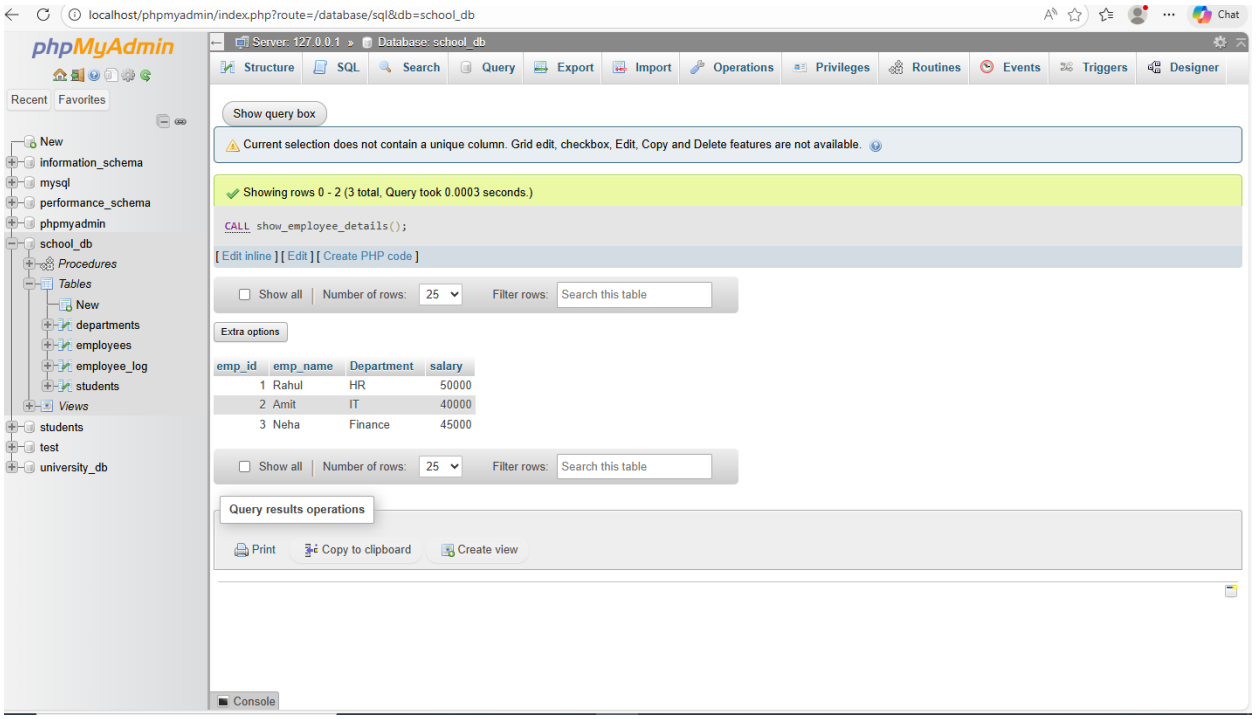
<u>Query 1: Stored Procedure</u>

```
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.

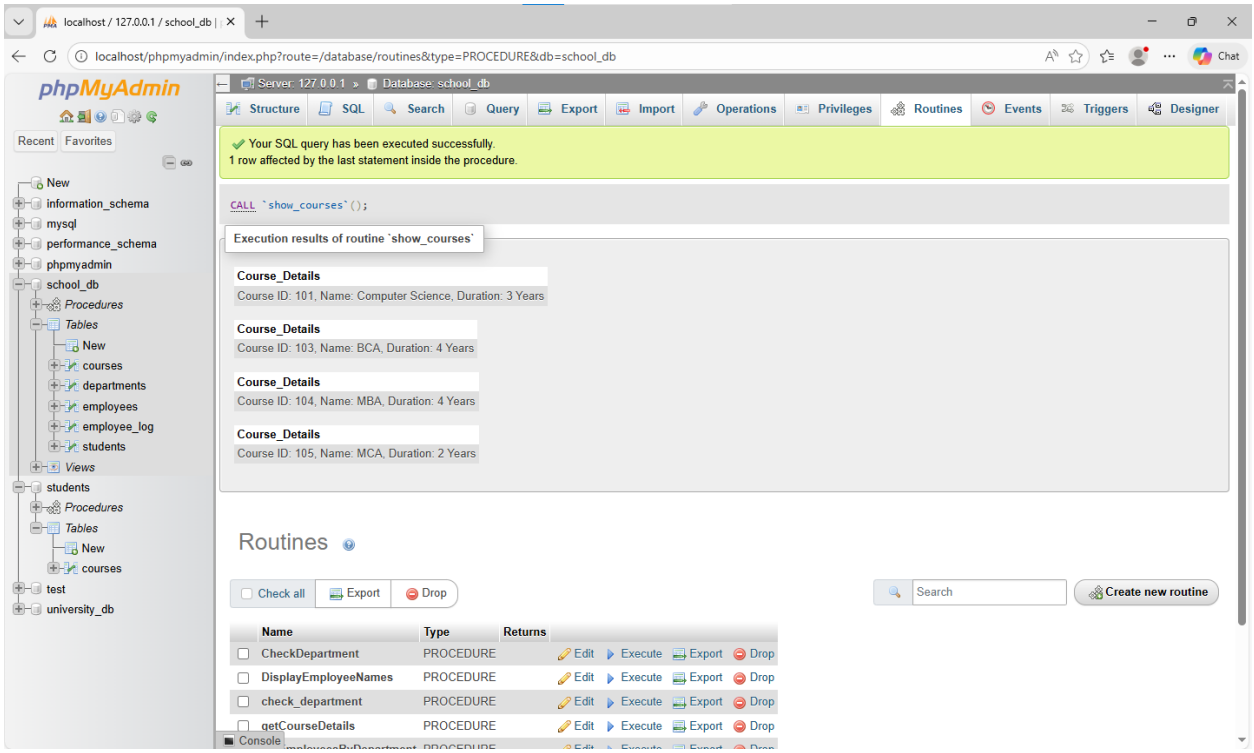
<u>Query 1: Stored Procedure</u>

```
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();
```



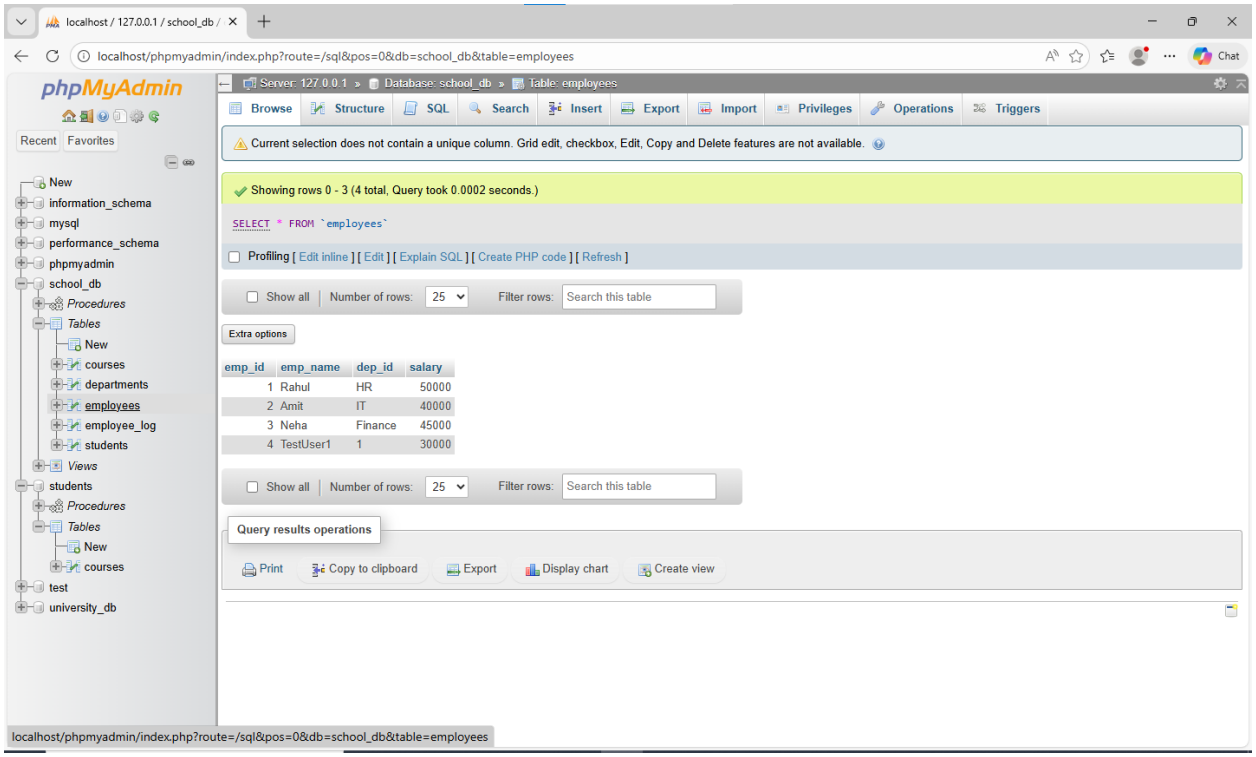
19. Rollback and Commit Savepoint

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

<u>Query: Create Savepoint → Insert Records → Rollback to Savepoint</u>

```
-- Start the transaction
START TRANSACTION;

-- Insert 1st record
INSERT INTO employees (emp_id, emp_name, dept_id, salary)
VALUES (4, 'TestUser1', 1, 30000);

-- Create savepoint
SAVEPOINT sp1;

-- Insert 2nd record
INSERT INTO employees (emp_id, emp_name, dept_id, salary)
VALUES (5, 'TestUser2', 2, 40000);

-- 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.

<u>Query: Commit part of a transaction → Rollback remaining changes</u>

-- Start transaction

START TRANSACTION;

-- Insert two new employees

INSERT INTO employees (emp_id, emp_name, dept_id, salary)

VALUES (5, 'PartialUser1', 1, 35000);

INSERT INTO employees (emp_id, emp_name, dept_id, salary)

VALUES (6, 'PartialUser2', 2, 36000);

-- Create savepoint

SAVEPOINT sp2;

-- Insert another record

INSERT INTO employees (emp_id, emp_name, dept_id, salary)

VALUES (7, 'RollbackUser', 3, 37000);

-- Commit changes BEFORE savepoint (5,6)

RELEASE SAVEPOINT sp2;

COMMIT;

-- Now rollback remaining uncommitted changes

ROLLBACK;