**Experiment-1**

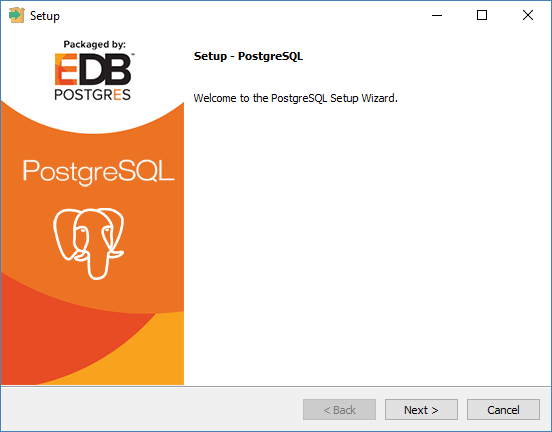
**Aim** - a. Learn to install PostgreSql.

b. Compare different Databases such as PostgreSQL, Oracle, IBM DB2, MySql.

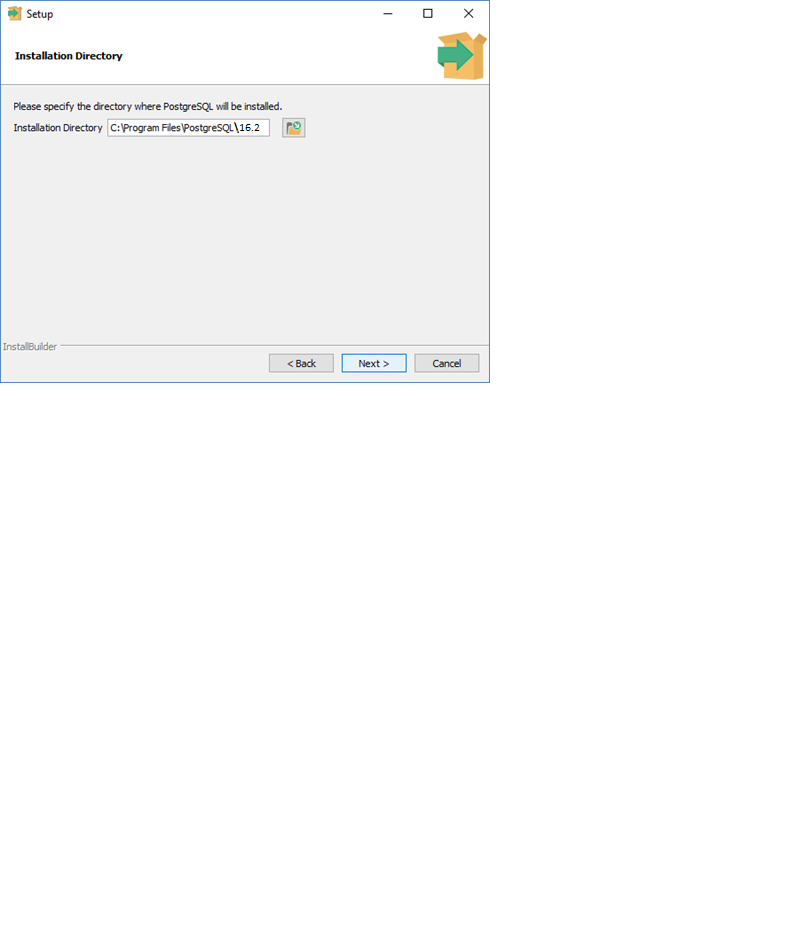
**a) Learn to install PostgreSql**

Following steps were to install PostgreSQL on Windows machine. Pick the version number of PostgreSQL you want and, as exactly as possible, the platform you want from a Enterprise DB

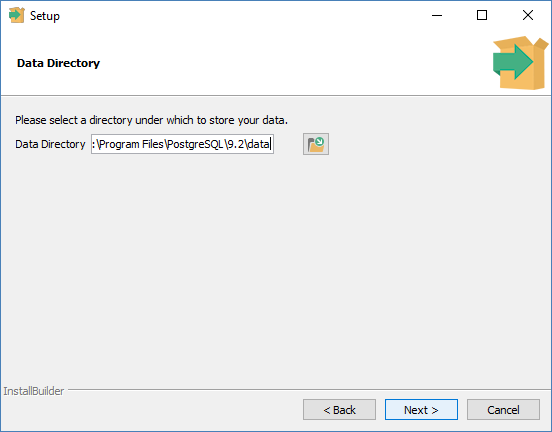
1. Download postgresql-16.1 for Windows PC running in 64 bit mode.
2. Run postgresql-16.1-1-windows-x64.exe as administrator and click next.



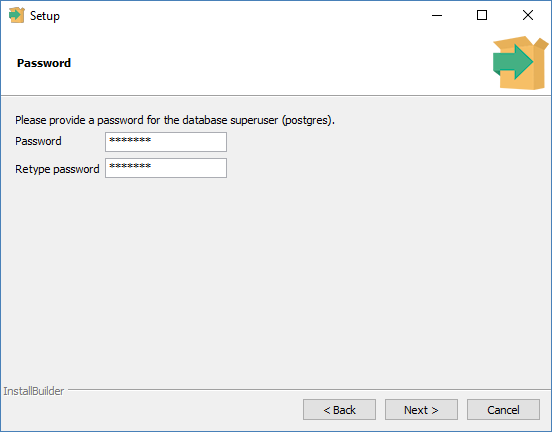
1. To install PostgreSQL. Select the location where you want to install it. By default it is installed within Program Files folder.



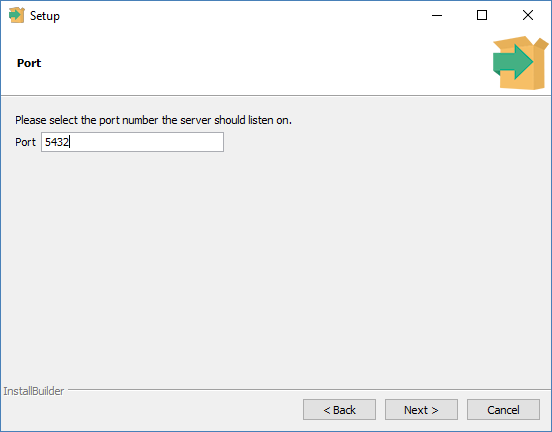
1. The next step of the installation process would be to select the directory where data would be stored, by default it is stored under "data" directory.



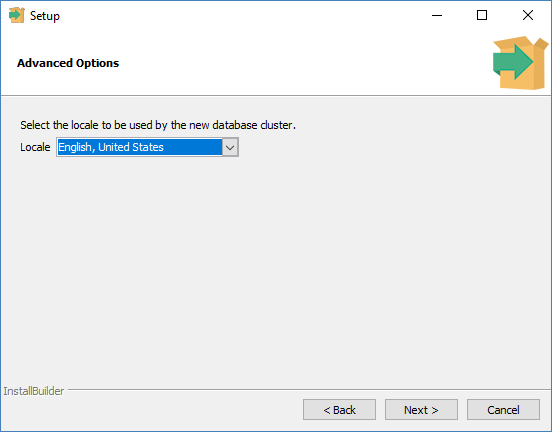
1. The next step, setup asks for password, so you can use your favorite password



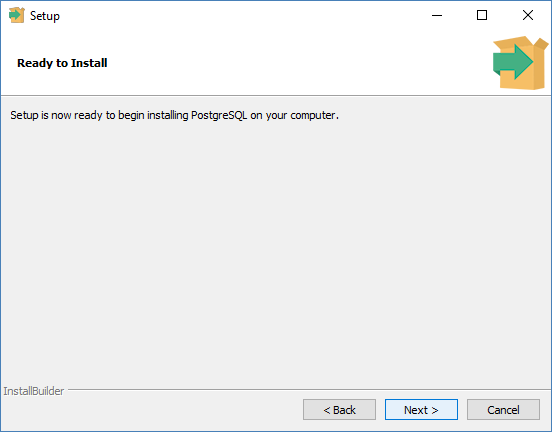
1. The next step, keep the port as default

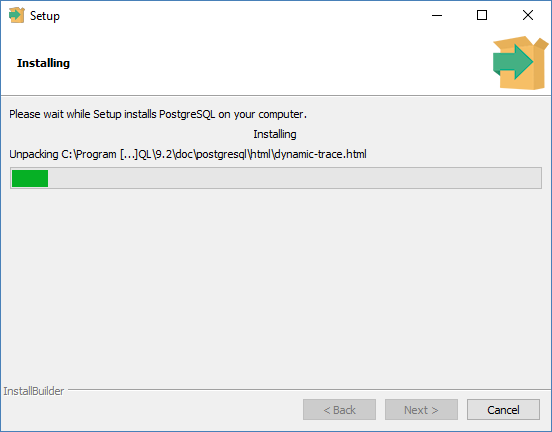


1. The next step, when asked for "Locale", "English, United States" has been selected.



1. It takes a while to install PostgreSQL on your system. On completion of the installation process, you will get the following screen. Uncheck the checkbox and click on Finish button.







**b) Compare different Databases such as PostgreSQL, Oracle, IBM DB2, MySql.**

PostgreSQL, Oracle, IBM DB2, and MySQL are all popular relational database management systems (RDBMS), but they have some differences in terms of features, performance, licensing, and use cases. Below is a comparison of these databases:

**PostgreSQL:**

* Licensing: PostgreSQL is open-source and released under the PostgreSQL License, which is similar to the MIT License.
* Performance: Known for its extensibility, PostgreSQL supports various indexing methods and has good performance for read-heavy workloads. It has strong support for complex queries and advanced data types.
* Features: Supports advanced data types like JSON, hstore, and array. Has a rich set of features including support for stored procedures, triggers, and views.
* Community: Has a strong open-source community that contributes to its development and maintenance.

**Oracle:**

* Licensing: Oracle Database is a commercial product, and its licensing can be expensive. Oracle offers various editions with different feature sets and pricing.
* Performance: Oracle is known for its robustness and is often used in enterprise-level applications. It excels in handling complex transactions and high-concurrency environments.
* Features: Offers a wide range of features including advanced security options, data warehousing, and support for complex data types. Oracle also has a comprehensive set of management tools.
* Community: Oracle has a large user base and provides extensive documentation and support, but it's not an open-source community.

**IBM DB2:**

* Licensing: IBM DB2 is a commercial RDBMS, and its licensing model can vary based on the edition and deployment scenario.
* Performance: IBM DB2 is known for its performance and scalability. It is often used in large enterprises and supports various deployment options, including on-premises and cloud.
* Features: Offers features like robust security, data compression, and support for complex data types. It has a reputation for high availability and reliability.
* Community: IBM DB2 has a strong user community, and IBM provides support and documentation for its users.

**MySQL:**

* Licensing: MySQL is open-source and is dual-licensed under the GNU General Public License or a commercial license for proprietary use.
* Performance: MySQL is known for its speed and is often used in web applications. It is particularly well-suited for read-heavy workloads.
* Features: Supports a wide range of storage engines, including InnoDB and MyISAM. It has a simple configuration and is easy to set up.
* Community: MySQL has a large and active open-source community. It is widely used in web development, and there are many resources available for support.

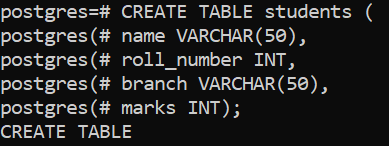
**Experiment-2**

**Aim** - Write SQL queries on DDL (eg create, alter, drop, rename, truncate), DML (eg. Insert, update, delete etc.), DCL (eg. Grant, revoke etc.) and Built-in Functions (eg Sum, min, max, avg, count, lower, upper, trim, len etc.) in PostgreSQL.

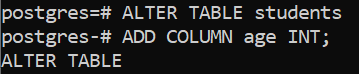
**Code:**

1. **DDL – Data Definition Language**

**Create query** – create table students (name varchar(50), roll\_number INT, branch varchar(50), marks INT);



**Alter query** – alter table students add column age INT;



**Drop query** – drop table students;



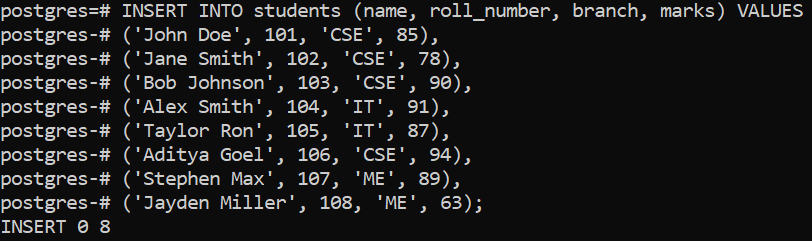
1. **DML – Data Manipulation Language**

**Insert query** – INSERT INTO students (name, roll \_ number, branch, marks) values

('John Doe', 101, 'CSE', 85), ('Jane Smith', 102, 'CSE', 78), ('Bob Johnson', 103, 'CSE', 90),

('Alex smith', 104, 'IT', 91), ( 'Taylor Ron', 105,'IT', 87), ('Aditya Goel', 106, 'CSE', 94),

('Stephen max', 107, 'ME', 89), ('Jayden Miller', 108, 'ME', 63);



**Update query** – update students set marks = 98 where roll\_number = 106;

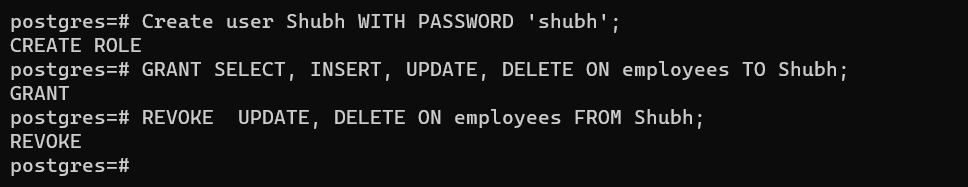


**Delete query** – delete from students where roll\_number = 108;

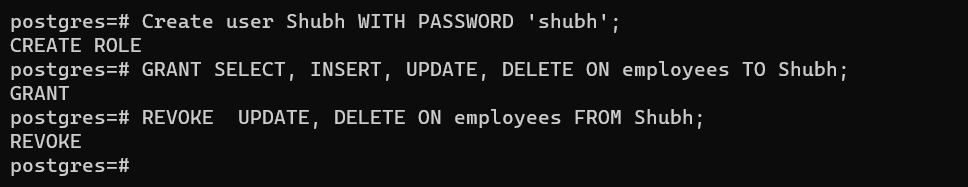


1. **DCL – Data Control Language**

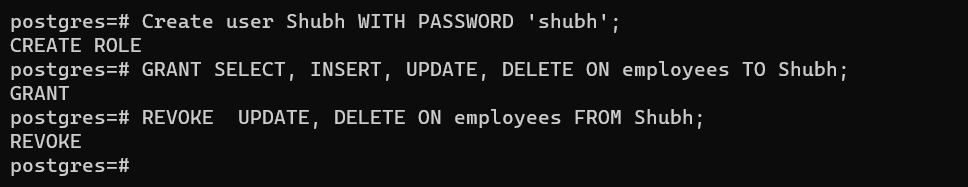
**Create user -** CREATE USER username WITH PASSWORD 'password';



**Grant permission -** GRANT SELECT, INSERT, UPDATE ON table\_name TO rolename;

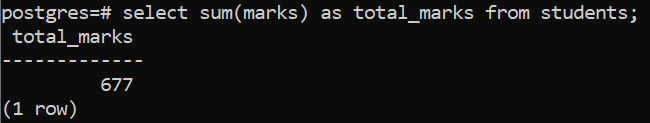


**Revoke Permission -** REVOKE SELECT, INSERT, UPDATE ON tablename FROM role;

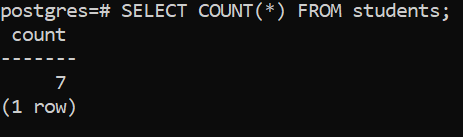


1. **Built in Functions**

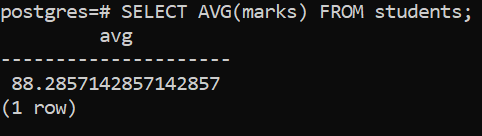
**Sum** – select sum(marks) as total\_marks from students;



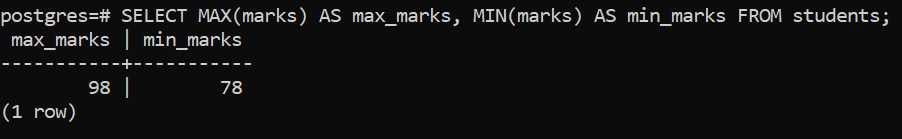
**Count** – select count(\*) from students;



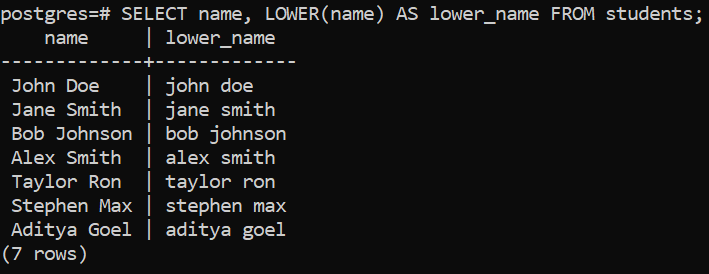
**Average** – select avg(marks) from students;



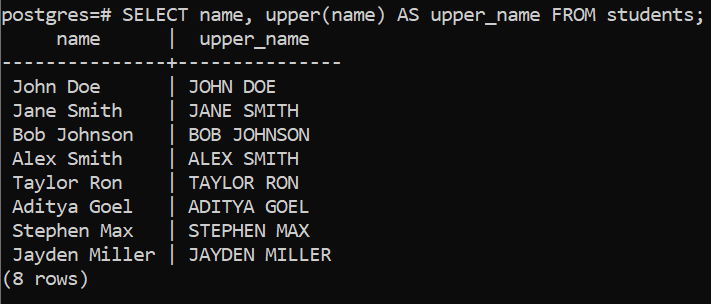
**Min Max** – select max(marks) as max\_marks, min(marks) as min\_marks from students;



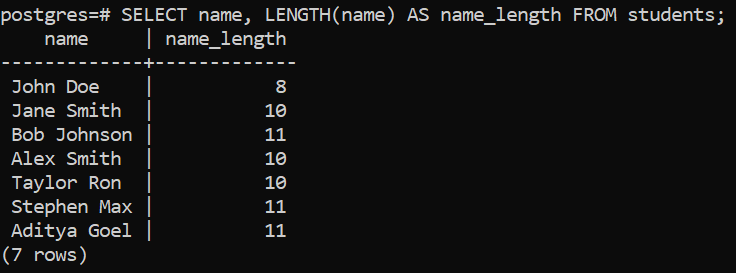
**Lower** – select name, lower(name) as lower\_name from students;



**Upper** – select name, upper(name) as upper\_name from students;



**Length** – select name, length(name) as name\_length from students;

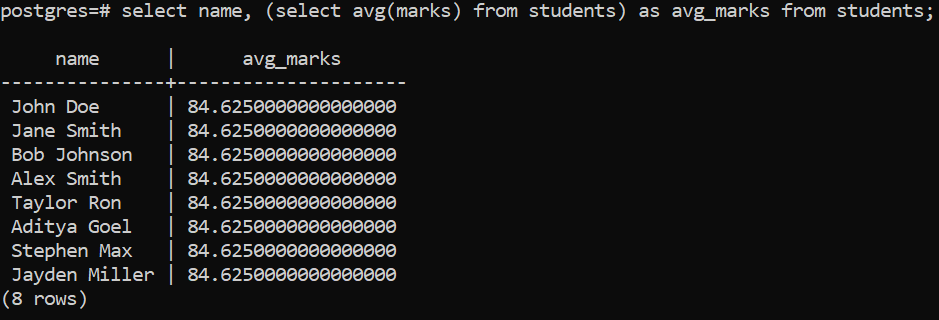


**Experiment-3**

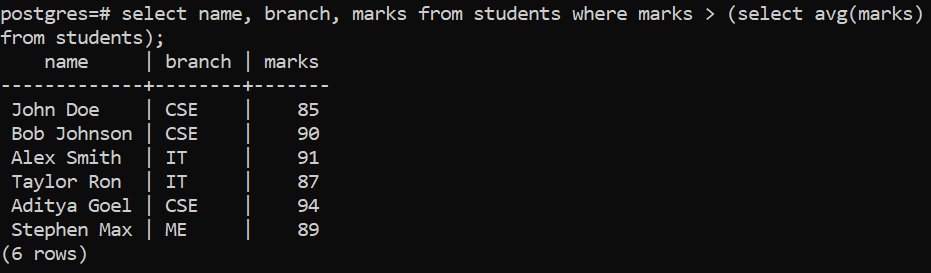
**Aim** - Write SQL queries on Nested queries and Join.

Code:

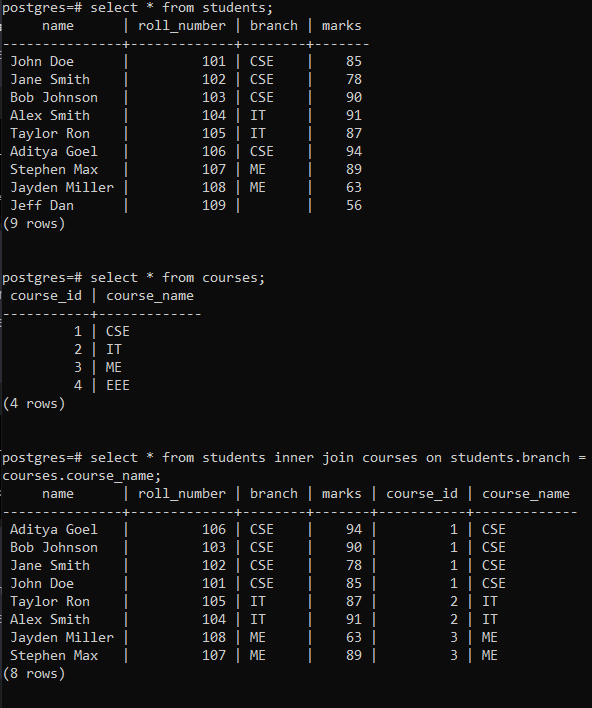
1. **Nested Query 1 -** Select name, (select ( avg(marks) from students) as avg\_marks from students;



1. **Nested Query 2 -** Select name, branch, marks from students where marks > (select avg(marks) from students);

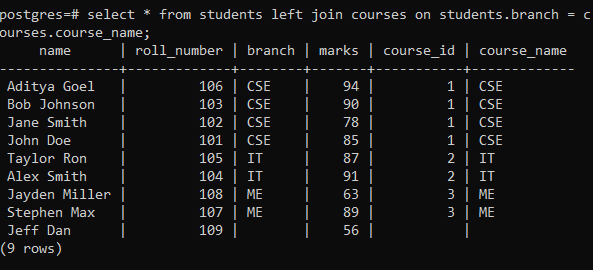


1. **Inner Join -** Select \* from students inner join courses on students.branch = courses.course\_name;



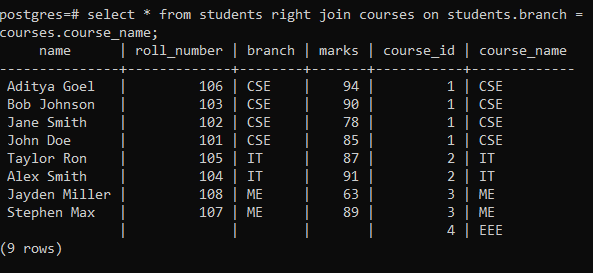
1. **Left Join**

Select \* from students left join courses on students.branch = courses.course\_name;



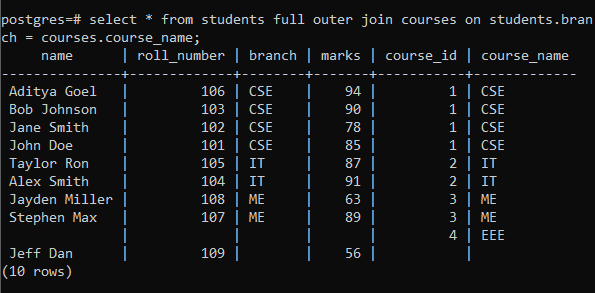
1. **Right Join**

Select \* from students right join courses on students.branch = courses.course\_name;



1. **Full Outer Join**

Select \* from students full outer join courses on students.branch = courses.course\_name;



**Experiment-4**

**Aim** - a. Create an index

b. Drop index

c. Create view with condition

d. Drop the view

**Code:**

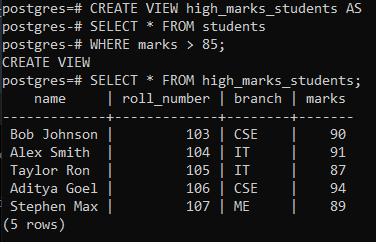
1. **Create index -** Create index idx\_marks on students(marks);



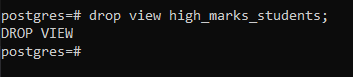
1. **Drop index -** Drop index idx\_marks;



1. **Create view -** Create view high\_marks\_students as select \* from students where marks > 85;



1. **Drop view -** Drop view high\_marks\_students;



**Experiment-5**

**Aim** - a. Write a program for addition of two numbers

b. Write a program to check the given number is even or odd

c. Write a program to inverse a number

d. Write a program to calculate the average of n numbers

**Code**

1. **Addition of two numbers.**

CREATE OR REPLACE FUNCTION add\_numbers(a INTEGER, b INTEGER)

RETURNS INTEGER AS $$

DECLARE

result INTEGER;

BEGIN

result := a + b;

RETURN result;

END;

$$ LANGUAGE plpgsql;

SELECT add\_numbers(5, 7) AS sum\_result;

1. **Check if number is even or odd**

CREATE OR REPLACE FUNCTION check\_even\_odd(num INTEGER)

RETURNS VARCHAR AS $$

DECLARE

result VARCHAR;

BEGIN

IF num % 2 = 0 THEN

result := 'Even';

ELSE

result := 'Odd';

END IF;

RETURN result;

END;

$$ LANGUAGE plpgsql;

SELECT check\_even\_odd(10) AS number\_result;

1. **Inverse a number**

CREATE OR REPLACE FUNCTION reverse\_digits(num INTEGER)

RETURNS INTEGER AS $$

DECLARE

reversed INTEGER := 0;

remainder INTEGER;

BEGIN

WHILE num > 0 LOOP

remainder := num % 10;

reversed := reversed \* 10 + remainder;

num := num / 10;

END LOOP;

RETURN reversed;

END;

$$ LANGUAGE plpgsql;

SELECT reverse\_digits(12345) AS reversed\_result;

1. **Calculate average of n numbers**

CREATE OR REPLACE FUNCTION calculate\_average(numbers INTEGER[])

RETURNS NUMERIC AS $$

DECLARE

total NUMERIC := 0;

count INTEGER := 0;

avg NUMERIC;

BEGIN

FOR i IN 1..array\_length(numbers, 1) LOOP

total := total + numbers[i];

count := count + 1;

END LOOP;

IF count > 0 THEN

avg := total / count;

RETURN avg;

ELSE

RAISE EXCEPTION 'Array is empty.';

END IF;

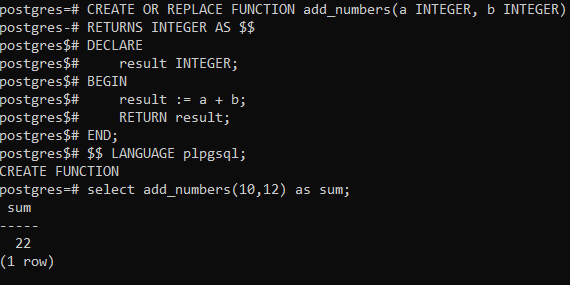
END;

$$ LANGUAGE plpgsql;

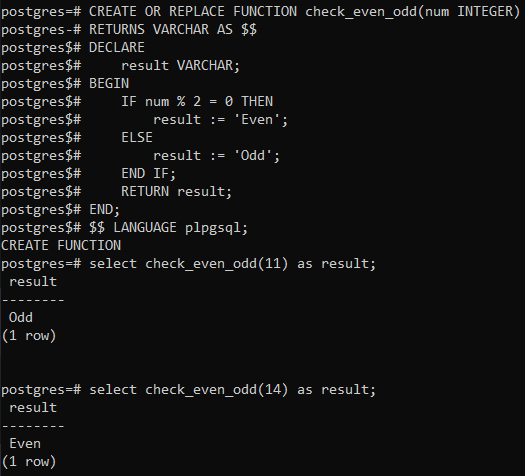
SELECT calculate\_average(ARRAY[5, 10, 15]) AS avg\_result;

**Output**

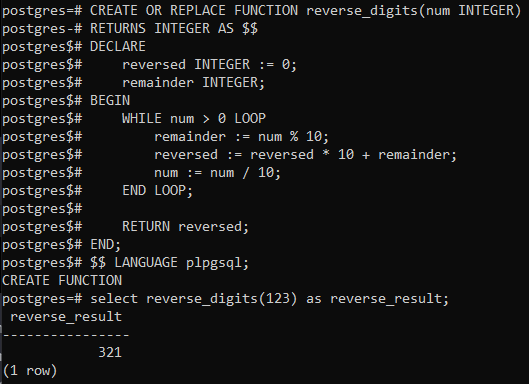
1. **Sum of two numbers**



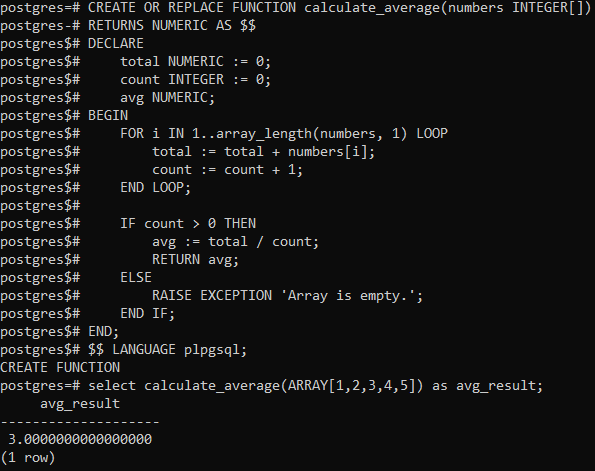
1. **Check number odd or even**



1. **Inverse a number**



1. **Average of n numbers**



**Experiment-6**

**Aim** - Write the cursor to increase the salary by 50%

**Code** :-

DO $$

DECLARE

emp\_record RECORD;

BEGIN

FOR emp\_record IN SELECT id, salary FROM employees LOOP

UPDATE employees

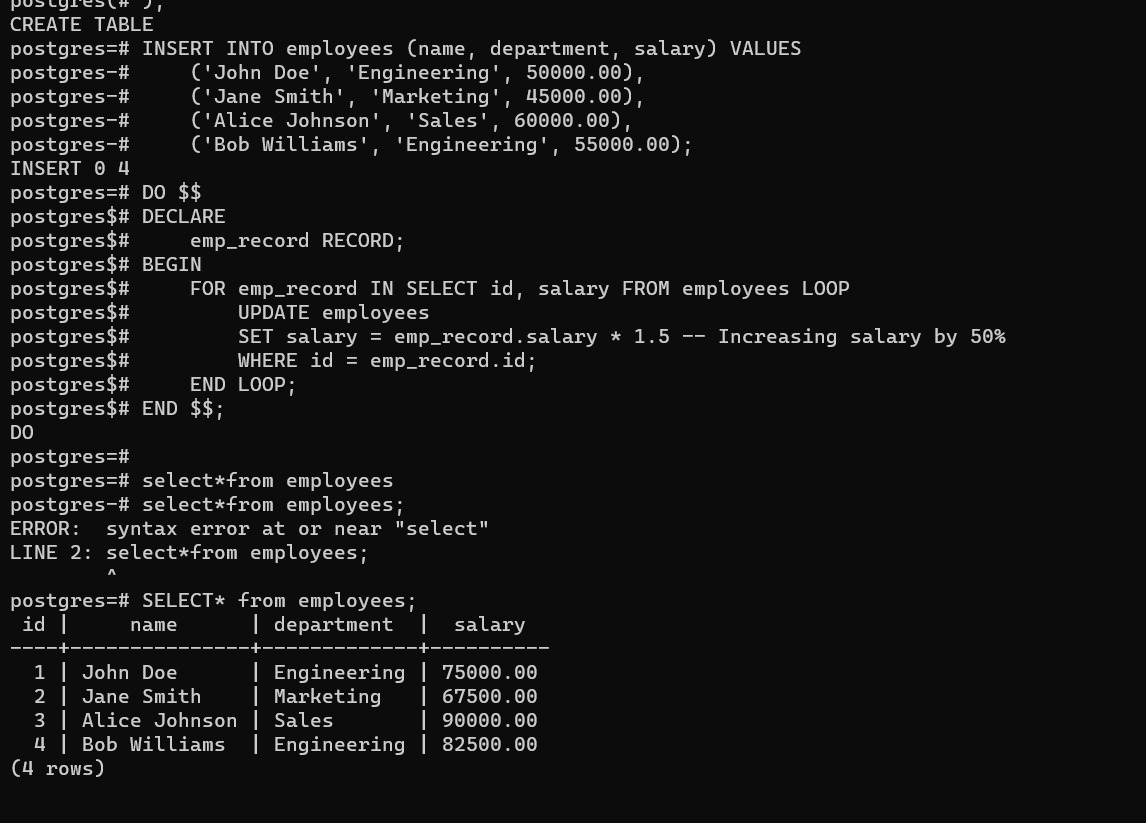
SET salary = emp\_record.salary\*1.5 -- Increasing salary by 50%

WHERE id = emp\_record.id;

END LOOP;

END $$;

select\* from employees



**Experiment-7**

**Aim** - Write a program to create exceptions according and raise these exceptions explicitly by using raise command.

**Code:**

CREATE OR REPLACE FUNCTION example\_exception\_handling()

RETURNS VOID AS $$

DECLARE

v\_condition BOOLEAN := true; -- Change this condition to test different exceptions

BEGIN

-- Raise custom\_exception\_1

IF v\_condition THEN

RAISE EXCEPTION 'Custom Exception 1 Raised';

END IF;

-- Raise custom\_exception\_2

IF NOT v\_condition THEN

RAISE EXCEPTION 'Custom Exception 2 Raised';

END IF;

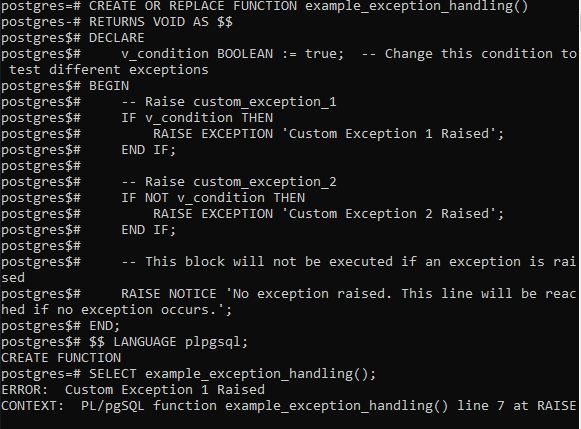
-- This block will not be executed if an exception is raised

RAISE NOTICE 'No exception raised. This line will be reached if no exception occurs.';

END;

$$ LANGUAGE plpgsql;

**Output**



**Experiment-8**

**Aim** - a. Write the procedure to get the average marks of students for branch “CSE”.

b. Write a function that accepts department name and returns the total no. of classes of the department. Also write a function to call the function.

**Code :**

**8a) Procedure to get average marks of students :-**

CREATE OR REPLACE FUNCTION get\_average\_marks\_cse()

RETURNS NUMERIC AS $$

DECLARE

avg\_marks NUMERIC;

BEGIN

SELECT AVG(marks) INTO avg\_marks

FROM students

WHERE branch = 'CSE';

RETURN avg\_marks;

END;

$$ LANGUAGE plpgsql;

SELECT get\_average\_marks\_cse();

**8b) Function to get number of classes of departments**

**Code 1)**

CREATE OR REPLACE FUNCTION get\_total\_classes\_by\_department(dept\_name VARCHAR)

RETURNS INT AS $$

DECLARE

total\_classes INT;

BEGIN

SELECT SUM(class\_count) INTO total\_classes

FROM classes

WHERE department = dept\_name;

RETURN COALESCE(total\_classes, 0);

END; $$ LANGUAGE plpgsql

**Code : 2)**

CREATE OR REPLACE FUNCTION call\_get\_total\_classes()

RETURNS INT AS $$

DECLARE

dept\_name VARCHAR := 'CSE'; -- Change this to the desired department

total\_classes INT;

BEGIN

total\_classes := get\_total\_classes\_by\_department(dept\_name);

RETURN total\_classes;

END;

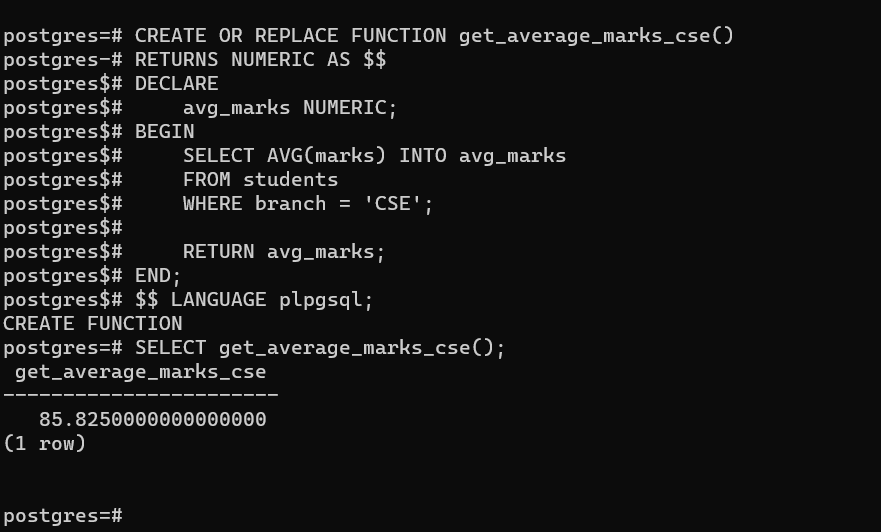
$$ LANGUAGE plpgsql;

Call this function :-

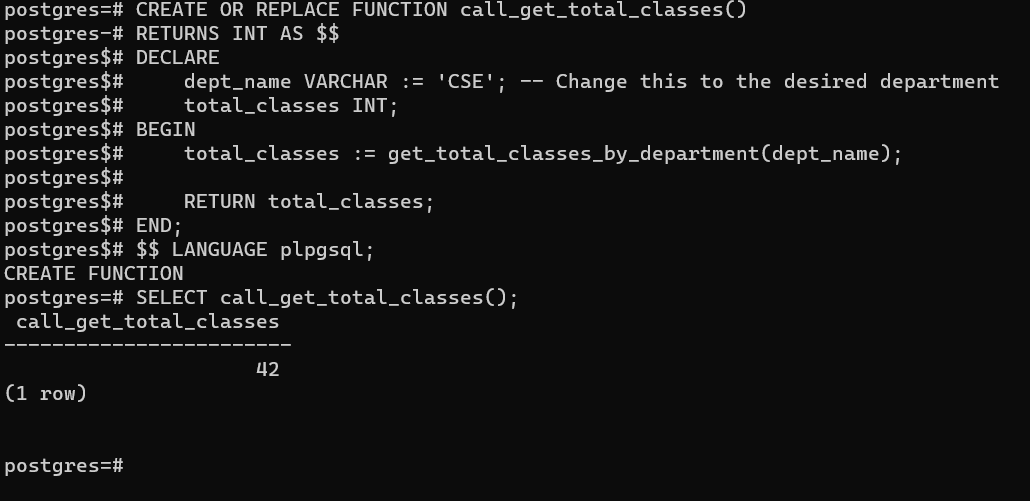
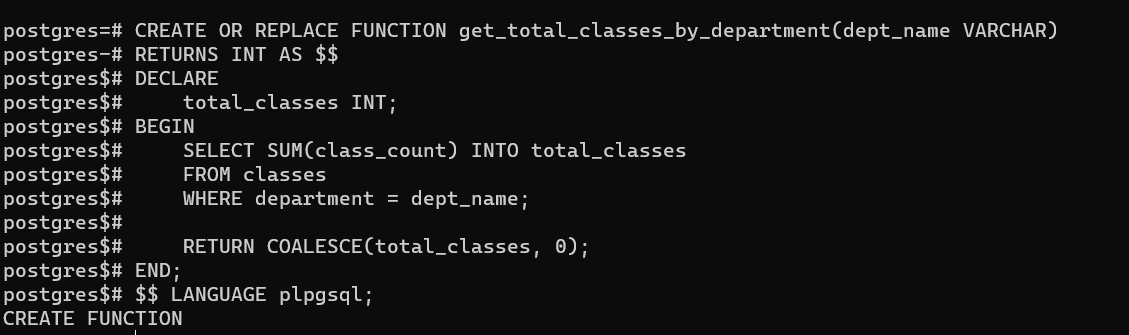
SELECT call\_get\_total\_classes();

**Output**

**8a) average marks of CSE**



**8b) Get number number of classes of department**



**Experiment-9**

**Aim** - a. Create a trigger on table after inserting a row into table.

b. Create a trigger on table after updating a row into table.

c. Write a row trigger to insert the existing values of the student table in to a new table when the marks of student are updated.

**Code :**

**9a) Trigger after inserting a row.**

CREATE OR REPLACE FUNCTION after\_student\_insert\_trigger()

RETURNS TRIGGER AS $$

BEGIN

-- Perform an action after insertion (e.g., print a message)

RAISE NOTICE 'A new student has been inserted with ID: %', NEW.id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

//Create the trigger to execute the function after insertion//

CREATE TRIGGER after\_student\_insert

AFTER INSERT ON students

FOR EACH ROW

EXECUTE FUNCTION after\_student\_insert\_trigger();

**9b) Trigger after updaating a row.**

CREATE OR REPLACE FUNCTION after\_student\_update\_trigger()

RETURNS TRIGGER AS $$

BEGIN

-- Perform an action after update (e.g., print a message)

RAISE NOTICE 'Student with ID % has been updated', NEW.id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

-- Create the trigger to execute the function after update

CREATE TRIGGER after\_student\_update

AFTER UPDATE ON students

FOR EACH ROW

EXECUTE FUNCTION after\_student\_update\_trigger()

UPDATE students SET marks = 95.0 WHERE id = 11;

**9c) trigger to insert the existing values of the student table in to a new table when the marks of student are updated.**

CREATE TABLE IF NOT EXISTS student\_marks\_history (

history\_id SERIAL PRIMARY KEY,

student\_id INT,

name VARCHAR(100),

branch VARCHAR(100),

old\_marks NUMERIC(5, 2),

new\_marks NUMERIC(5, 2),

change\_timestamp TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE OR REPLACE FUNCTION on\_student\_marks\_update\_trigger()

RETURNS TRIGGER AS $$

BEGIN

IF NEW.marks <> OLD.marks THEN -- Check if the marks have actually changed

INSERT INTO student\_marks\_history (student\_id, name, branch, old\_marks, new\_marks)

VALUES (NEW.id, NEW.name, NEW.branch, OLD.marks, NEW.marks);

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER student\_marks\_update

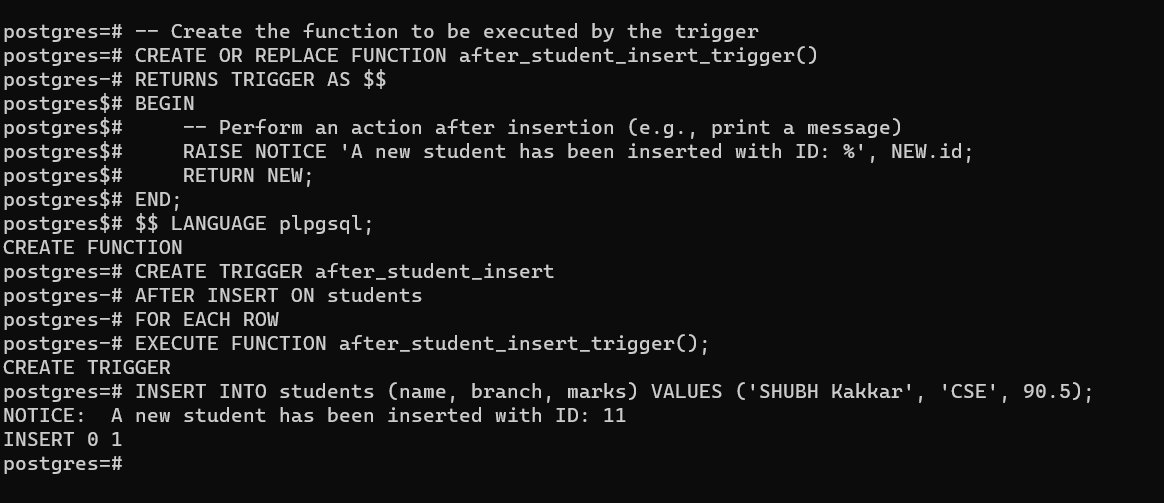
AFTER UPDATE OF marks ON students

FOR EACH ROW

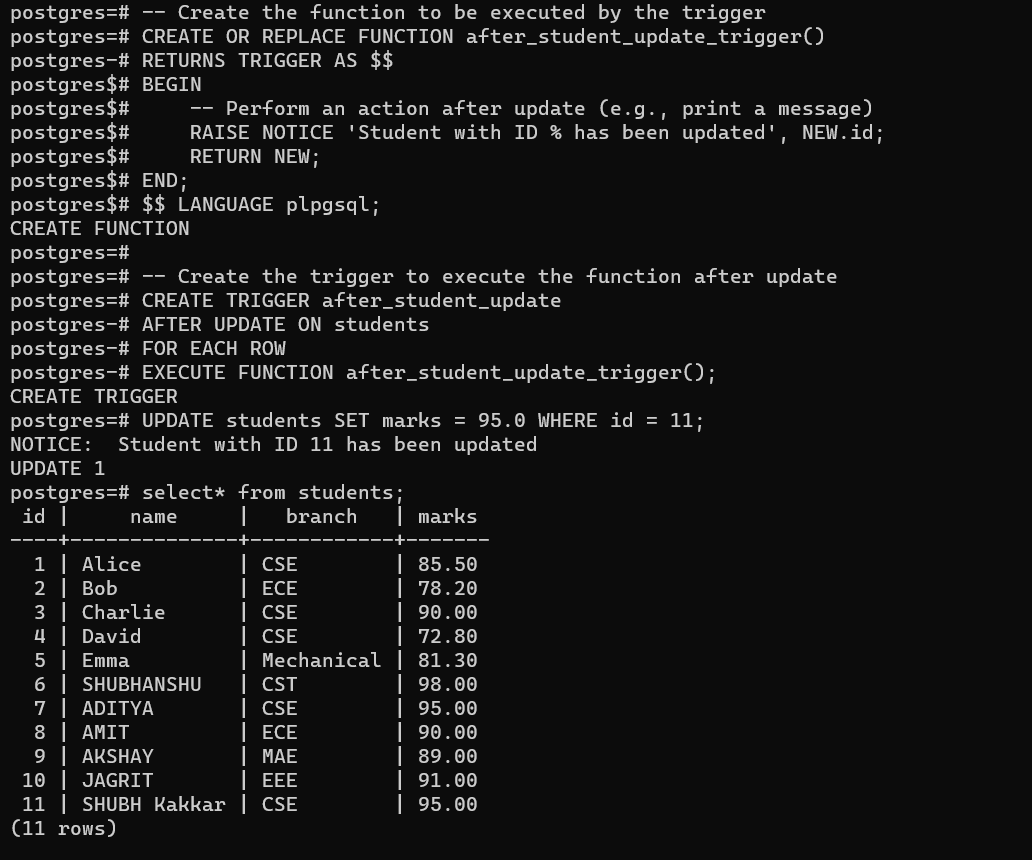
EXECUTE FUNCTION on\_student\_marks\_update\_trigger();

**Output**

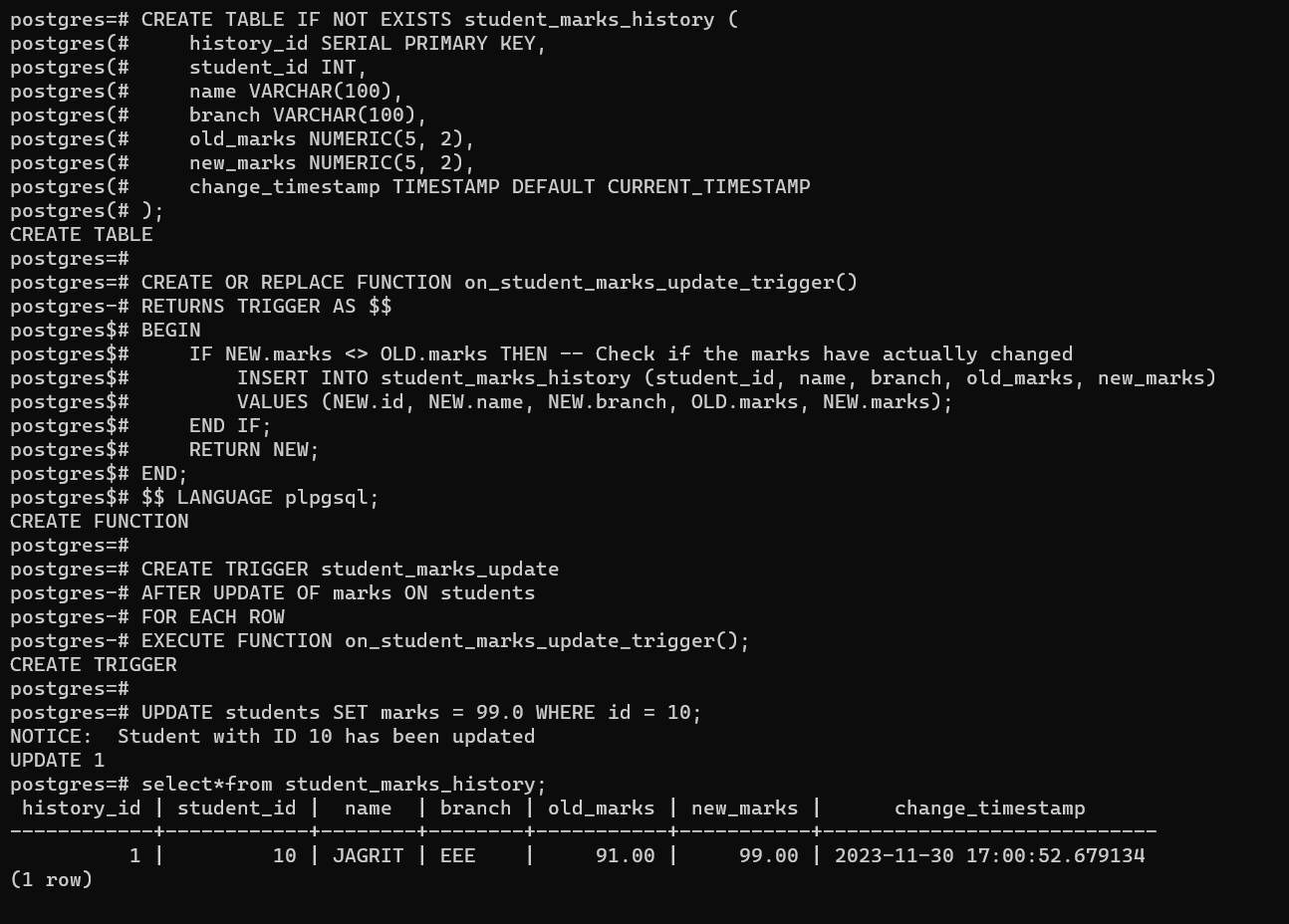
**9a) Trigger after inserting a row**

****

**9b) Trigger after updating a row**



**9c) trigger to insert the existing values of the student table in to a new table when the marks of student are updated.**

****