**Batch: A3 Roll No.: 16010122083**

**Experiment / assignment / tutorial No. 6**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Queries based on Procedure,Function and Cursor** |

**Objective:** To be able to functions and procedures on the table.

**Expected Outcome of Experiment:**

CO 3 : Use SQL for Relational database creation, maintenance and query processing

**Books/ Journals/ Websites referred:**

1. Dr. P.S. Deshpande, SQL and PL/SQL for Oracle 10g.Black book, Dreamtech Press

2. www.db-book.com

3. Korth, Slberchatz, Sudarshan : “Database Systems Concept”, 5th Edition , McGraw

Hill

4. Elmasri and Navathe,”Fundamentals of database Systems”, 4th Edition,PEARSON

Education.

**Resources used:** Postgresql

**Theory**

**Procedures:**

A stored procedure is a set of Structured Query Language (SQL) statements with an assigned name, which are stored in a relational database management system as a group, so it can be reused and shared by multiple programs.

Stored procedures can access or modify data in a database, but it is not tied to a specific database or object, which offers a number of advantages.

**Benefits of using stored procedures**

A stored procedure provides an important layer of security between the user interface and the database. It supports security through data access controls because end users may enter or change data, but do not write procedures. A stored procedure preserves data integrity because information is entered in a consistent manner. It improves productivity because statements in a stored procedure only must be written once.

Use of stored procedures can reduce network traffic between clients and servers, because the commands are executed as a single batch of code. This means only the call to execute the procedure is sent over a network, instead of every single line of code being sent individually.

Syntax:

CREATE [ OR REPLACE ] PROCEDURE

***name*** ( [ [ ***argmode*** ] [ ***argname*** ] ***argtype*** [ { DEFAULT | = } ***default\_expr*** ] [, ...] ] )

{ LANGUAGE ***lang\_name***

| TRANSFORM { FOR TYPE ***type\_name*** } [, ... ]

| [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER

| SET ***configuration\_parameter*** { TO ***value*** | = ***value*** | FROM CURRENT }

| AS '***definition***'

| AS '***obj\_file***', '***link\_symbol***'

} ...

Parameters

**Name:** The name (optionally schema-qualified) of the procedure to create.

**Argmode:** The mode of an argument: IN, INOUT, or VARIADIC. If omitted, the default is IN. (OUT arguments are currently not supported for procedures. Use INOUT instead.)

**Argname:** The name of an argument.

**Argtype:** The data type(s) of the procedure's arguments (optionally schema-qualified), if any. The argument types can be base, composite, or domain types, or can reference the type of a table column.

Depending on the implementation language it might also be allowed to specify “pseudo-types” such as cstring. Pseudo-types indicate that the actual argument type is either incompletely specified, or outside the set of ordinary SQL data types.

The type of a column is referenced by writing table\_name.column\_name%TYPE. Using this feature can sometimes help make a procedure independent of changes to the definition of a table.

**default\_expr:** An expression to be used as default value if the parameter is not specified. The expression has to be coercible to the argument type of the parameter. All input parameters following a parameter with a default value must have default values as well.

**lang\_name** :The name of the language that the procedure is implemented in. It can be sql, c, internal, or the name of a user-defined procedural language, e.g. plpgsql. Enclosing the name in single quotes is deprecated and requires matching case.

**TRANSFORM { FOR TYPE type\_name } [, ... ] }**

Lists which transforms a call to the procedure should apply. Transforms convert between SQL types and language-specific data types; see CREATE TRANSFORM. Procedural language implementations usually have hardcoded knowledge of the built-in types, so those don't need to be listed here. If a procedural language implementation does not know how to handle a type and no transform is supplied, it will fall back to a default behavior for converting data types, but this depends on the implementation.

**[EXTERNAL] SECURITY INVOKER**

**[EXTERNAL] SECURITY DEFINER**

**SECURITY INVOKER** indicates that the procedure is to be executed with the privileges of the user that calls it. That is the default. SECURITY DEFINER specifies that the procedure is to be executed with the privileges of the user that owns it.

The key word EXTERNAL is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all procedures not only external ones.

A SECURITY DEFINER procedure cannot execute transaction control statements (for example, COMMIT and ROLLBACK, depending on the language).

**configuration\_parameter**

**value:** The SET clause causes the specified configuration parameter to be set to the specified value when the procedure is entered, and then restored to its prior value when the procedure exits. SET FROM CURRENT saves the value of the parameter that is current when CREATE PROCEDURE is executed as the value to be applied when the procedure is entered.

If a SET clause is attached to a procedure, then the effects of a SET LOCAL command executed inside the procedure for the same variable are restricted to the procedure: the configuration parameter's prior value is still restored at procedure exit. However, an ordinary SET command (without LOCAL) overrides the SET clause, much as it would do for a previous SET LOCAL command: the effects of such a command will persist after procedure exit, unless the current transaction is rolled back.

If a SET clause is attached to a procedure, then that procedure cannot execute transaction control statements (for example, COMMIT and ROLLBACK, depending on the language).

**Definition**

A string constant defining the procedure; the meaning depends on the language. It can be an internal procedure name, the path to an object file, an SQL command, or text in a procedural language.

It is often helpful to use dollar quoting to write the procedure definition string, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the procedure definition must be escaped by doubling them.

**obj\_file, link\_symbol**

This form of the AS clause is used for dynamically loadable C language procedures when the procedure name in the C language source code is not the same as the name of the SQL procedure. The string obj\_file is the name of the shared library file containing the compiled C procedure, and is interpreted as for the LOAD command. The string link\_symbol is the procedure's link symbol, that is, the name of the procedure in the C language source code. If the link symbol is omitted, it is assumed to be the same as the name of the SQL procedure being defined.

When repeated CREATE PROCEDURE calls refer to the same object file, the file is only loaded once per session. To unload and reload the file (perhaps during development), start a new session.

**Example:**

We will use the following accounts table for the demonstration:

CREATE TABLE accounts (

    id INT GENERATED BY DEFAULT AS IDENTITY,

    name VARCHAR(100) NOT NULL,

    balance DEC(15,2) NOT NULL,

    PRIMARY KEY(id)

);

INSERT INTO accounts(name,balance)

VALUES('Bob',10000);

INSERT INTO accounts(name,balance)

VALUES('Alice',10000);

The following example creates stored procedure named transfer that transfer specific amount of money from one account to another.

CREATE OR REPLACE PROCEDURE transfer(INT, INT, DEC)

LANGUAGE plpgsql

AS $$

BEGIN

    -- subtracting the amount from the sender's account

    UPDATE accounts

    SET balance = balance - $3

    WHERE id = $1;

    -- adding the amount to the receiver's account

    UPDATE accounts

    SET balance = balance + $3

    WHERE id = $2;

    COMMIT;

END;

$$;

CALL stored\_procedure\_name(parameter\_list);

CALL transfer(1,2,1000);

**Functions**

The basic syntax to create a function is as follows −

CREATE [OR REPLACE] FUNCTION function\_name (arguments)

RETURNS return\_datatype

language plpgsql

AS

$variable\_name$

declare

-- variable declaration

begin

-- stored procedure body

end; $$

**Explanation:**

**function-name** specifies the name of the function.

[OR REPLACE] option allows modifying an existing function.

The function must contain a return statement.

**RETURN** clause specifies that data type you are going to return from the function. The return\_datatype can be a base, composite, or domain type, or can reference the type of a table column.

**function-body** contains the executable part.

**The AS keyword** is used for creating a standalone function.

**plpgsql** is the name of the language that the function is implemented in. Here, we use this option for PostgreSQL,it Can be SQL, C, internal, or the name of a user-defined procedural language. For backward compatibility, the name can be enclosed by single quotes.

Example

CREATE TABLE employee (

id SERIAL PRIMARY KEY,

name VARCHAR(100),

date\_of\_joining DATE,

salary DECIMAL(10, 2)

);

INSERT INTO employee (name, dateofjoining, salary) VALUES

('John Doe', '2022-01-01', 50000),

('Jane Smith', '2022-02-15', 60000),

('Michael Johnson', '2022-03-20', 70000),

('Emily Davis', '2022-04-10', 45000),

('Christopher Wilson', '2022-05-05', 80000);

create a function named get\_employee\_count to calculate the number of records in the employee table:

CREATE OR REPLACE FUNCTION get\_employee\_count()

RETURNS INTEGER AS $$

DECLARE

total\_records INTEGER;

BEGIN

-- Execute a SQL query to count the number of records in the employee table

SELECT COUNT(\*) INTO total\_records FROM employee;

-- Return the total number of records

RETURN total\_records;

END;

$$ LANGUAGE plpgsql;

**calling the function using query:**

SELECT get\_employee\_count();

**Cursors**

Rather than executing a whole query at once, it is possible to set up a cursor that encapsulates the query, and then read the query result a few rows at a time. One reason for doing this is to avoid memory overrun when the result contains a large number of rows. (However, PL/pgSQL users do not normally need to worry about that, since FOR loops automatically use a cursor internally to avoid memory problems.) A more interesting usage is to return a reference to a cursor that a function has created, allowing the caller to read the rows. This provides an efficient way to return large row sets from functions.

Before a cursor can be used to retrieve rows, it must be opened. (This is the equivalent action to the SQL command DECLARE CURSOR.) PL/pgSQL has three forms of the OPEN statement, two of which use unbound cursor variables while the third uses a bound cursor variable.

**OPEN FOR query**

Syntax: OPEN unbound\_cursorvar [ [ NO ] SCROLL ] FOR query;

example:

OPEN curs1 FOR SELECT \* FROM foo WHERE key = mykey;

**OPEN FOR EXECUTE**

Syntax: OPEN unbound\_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query\_string

[ USING expression [, ... ] ];

example:

OPEN curs1 FOR EXECUTE 'SELECT \* FROM ' || quote\_ident(tabname)

|| ' WHERE col1 = $1' USING keyvalue;

**Opening a Bound Cursor**

Syntax: OPEN bound\_cursorvar [ ( [ argument\_name := ] argument\_value [, ...] ) ];

Examples (these use the cursor declaration examples above):

OPEN curs2;

OPEN curs3(42);

OPEN curs3(key := 42);

Because variable substitution is done on a bound cursor's query, there are really two ways to pass values into the cursor: either with an explicit argument to OPEN, or implicitly by referencing a PL/pgSQL variable in the query. However, only variables declared before the bound cursor was declared will be substituted into it. In either case the value to be passed is determined at the time of the OPEN. For example, another way to get the same effect as the curs3 example above is

DECLARE

key integer;

curs4 CURSOR FOR SELECT \* FROM tenk1 WHERE unique1 = key;

BEGIN

key := 42;

OPEN curs4;

**Using Cursors**

**FETCH**

Synatx: FETCH [ direction { FROM | IN } ] cursor INTO target;

Examples:

FETCH curs1 INTO rowvar;

FETCH curs2 INTO foo, bar, baz;

FETCH LAST FROM curs3 INTO x, y;

FETCH RELATIVE -2 FROM curs4 INTO x;

**MOVE**

MOVE [ direction { FROM | IN } ] cursor;

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only repositions the cursor and does not return the row moved to. As with SELECT INTO, the special variable FOUND can be checked to see whether there was a next row to move to.

Examples:

MOVE curs1;

MOVE LAST FROM curs3;

MOVE RELATIVE -2 FROM curs4;

MOVE FORWARD 2 FROM curs4;

**UPDATE/DELETE WHERE CURRENT OF**

UPDATE table SET ... WHERE CURRENT OF cursor;

DELETE FROM table WHERE CURRENT OF cursor;

When a cursor is positioned on a table row, that row can be updated or deleted using the cursor to identify the row. There are restrictions on what the cursor's query can be (in particular, no grouping) and it's best to use FOR UPDATE in the cursor. For more information see the DECLARE reference page.

An example:

UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;

**CLOSE**

CLOSE cursor;

CLOSE closes the portal underlying an open cursor. This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.

An example:

CLOSE curs1;

**Implementation Screenshots (Problem Statement, Query and Screenshots of Results):**

CREATE OR REPLACE PROCEDURE transfer(sender\_id INT, receiver\_id INT, amount DECIMAL)

LANGUAGE plpgsql

AS $$

BEGIN

UPDATE accounts

SET balance = balance - amount

WHERE id = sender\_id;

UPDATE accounts

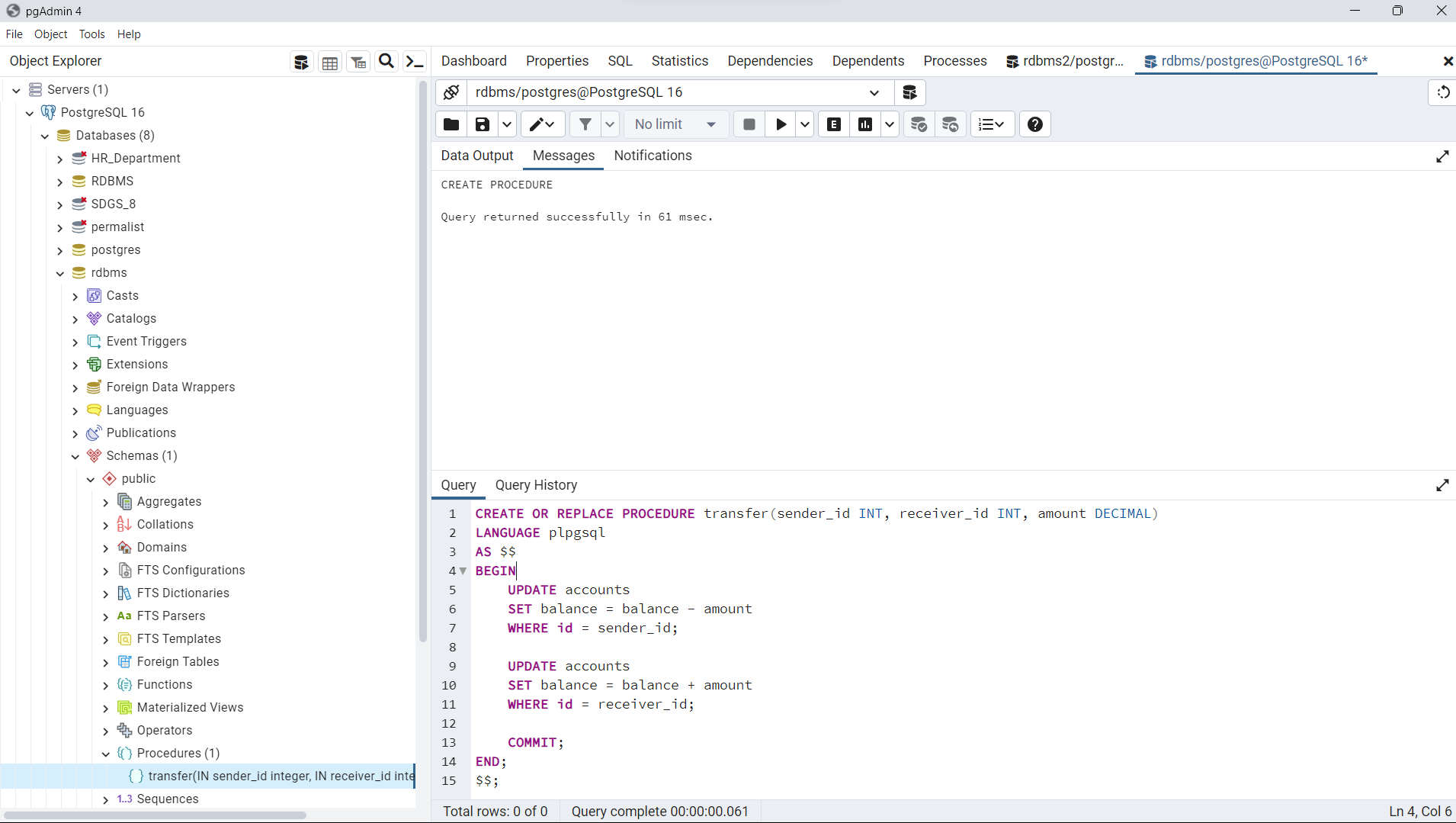
SET balance = balance + amount

WHERE id = receiver\_id;

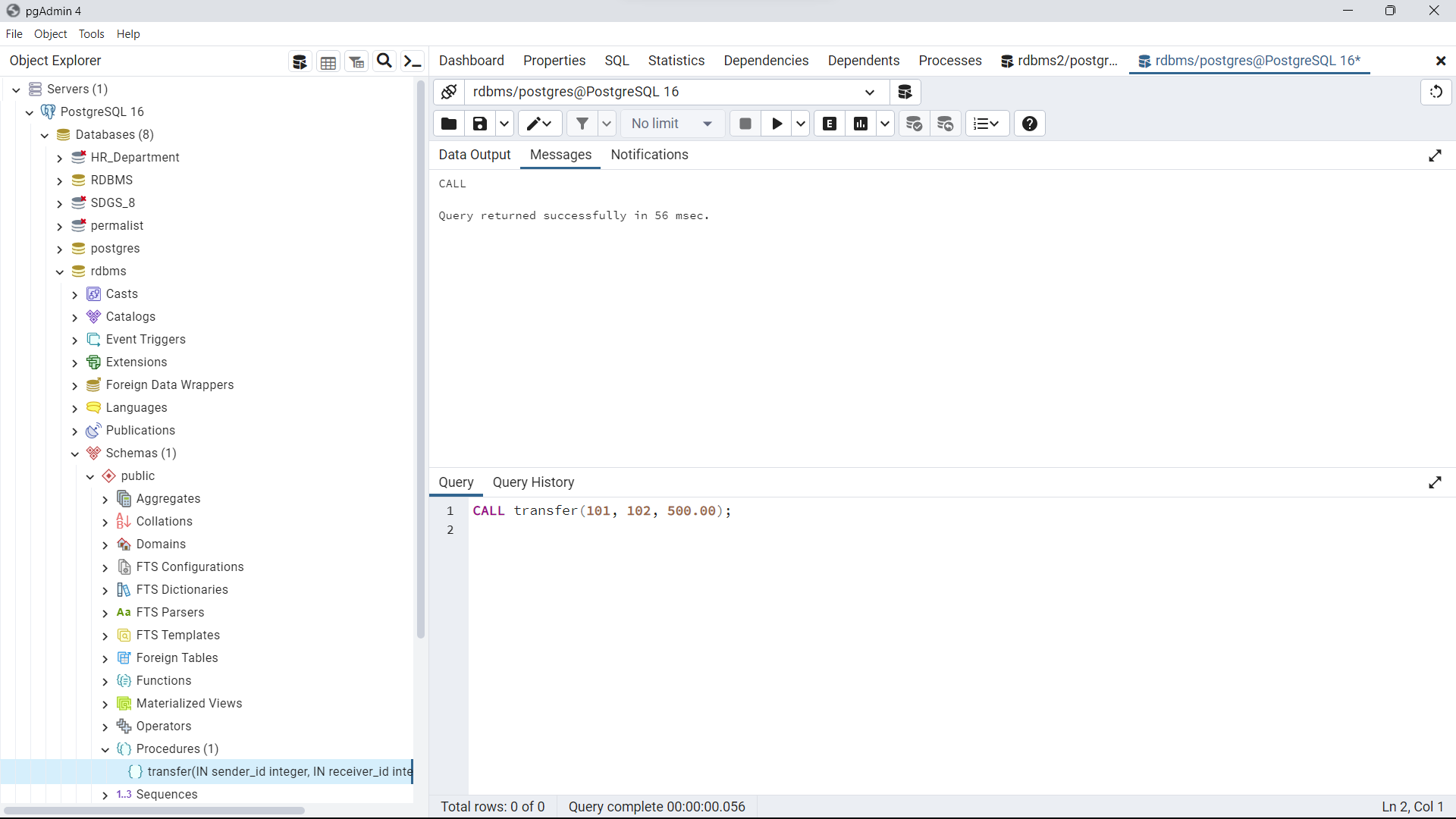
COMMIT;

END;

$$;



CALL transfer(101, 102, 500.00);



**CREATE OR REPLACE FUNCTION get\_employee\_count()**

**RETURNS INTEGER AS $$**

**DECLARE**

**total\_records INTEGER;**

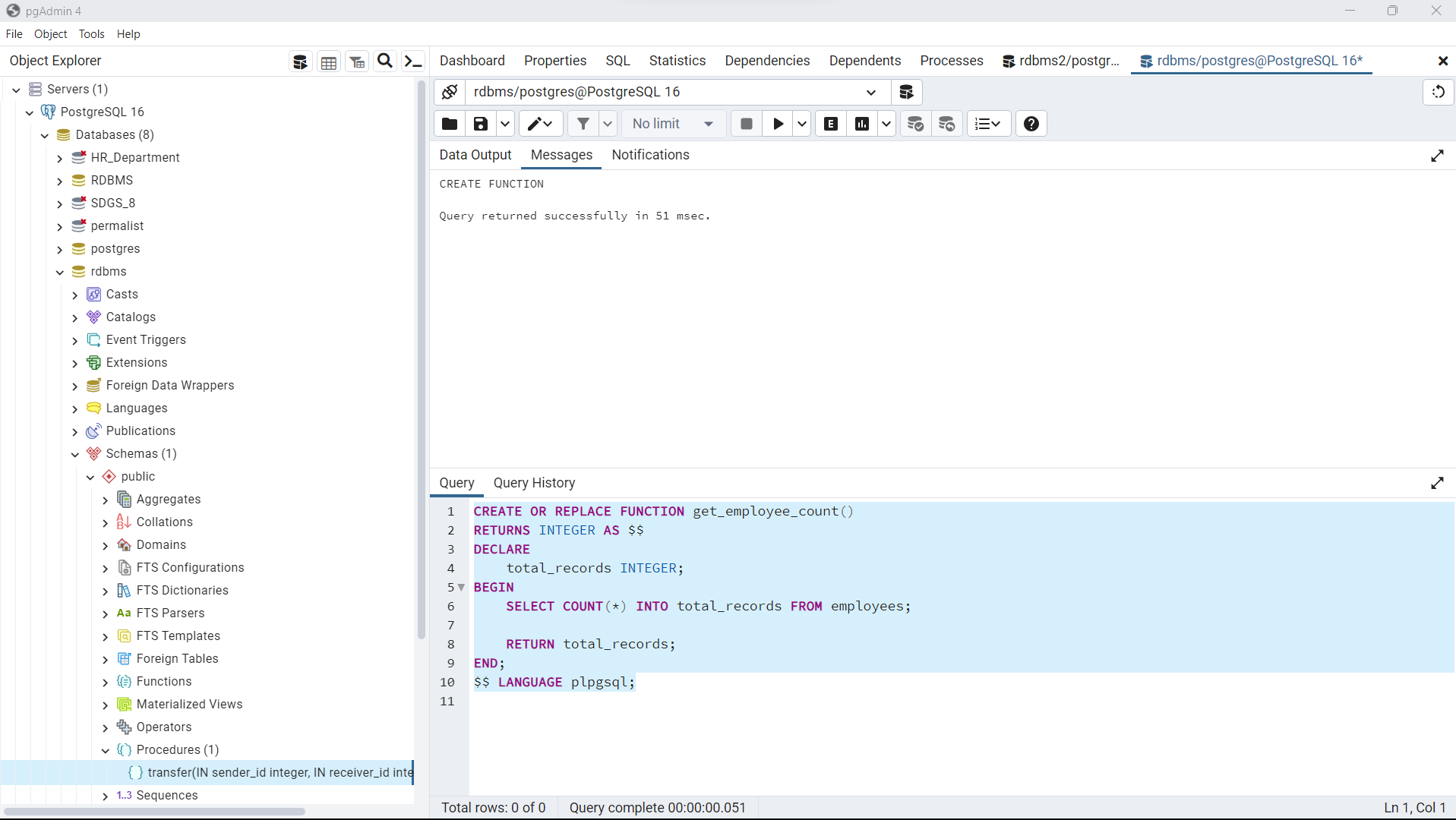
**BEGIN**

**SELECT COUNT(\*) INTO total\_records FROM employees;**

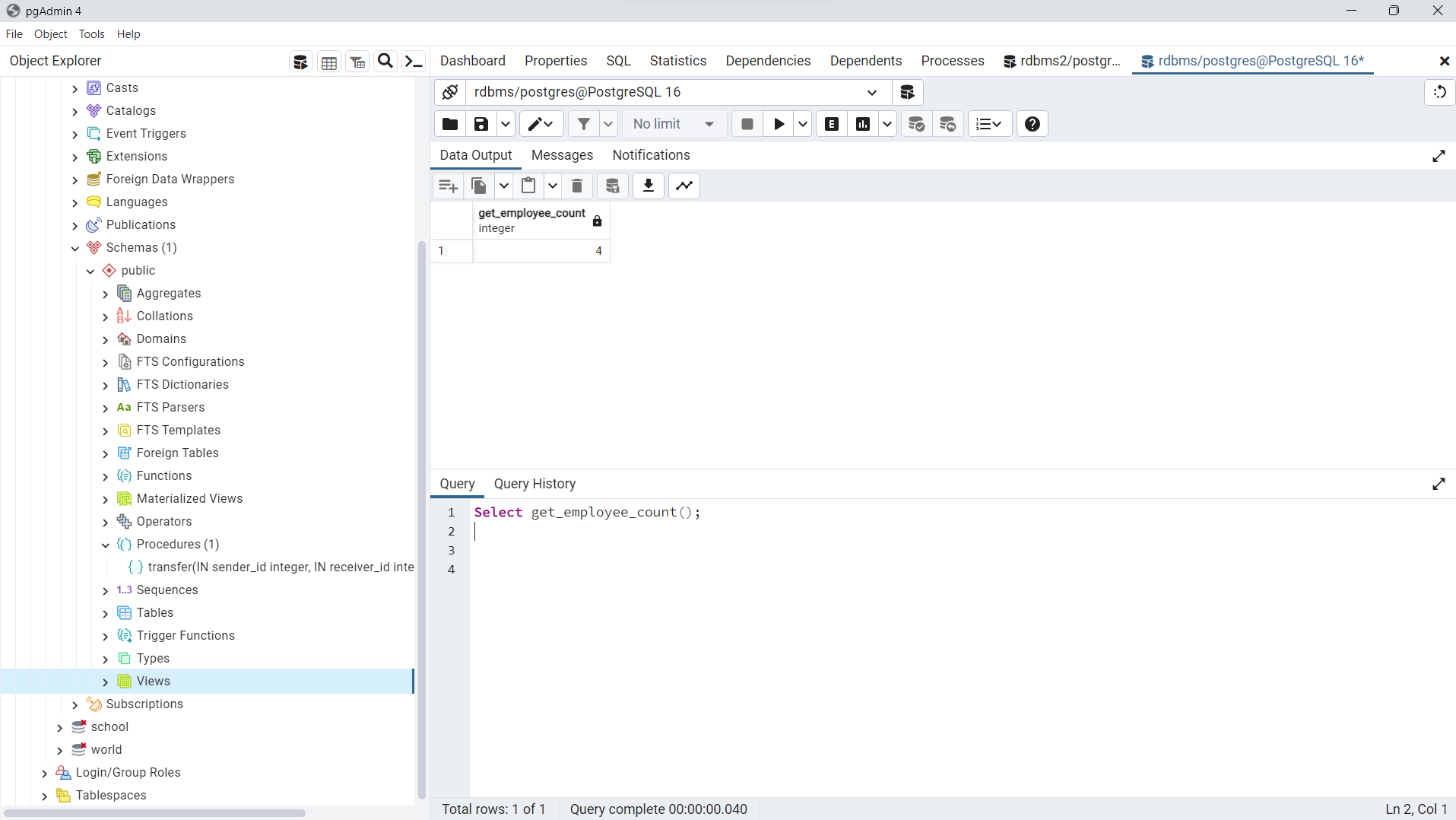
**RETURN total\_records;**

**END;**

**$$ LANGUAGE plpgsql;**

.

**Select get\_employee\_count();**



CREATE OR REPLACE FUNCTION get\_high\_salary\_jobs(min\_salary DECIMAL)

RETURNS REFCURSOR AS $$

DECLARE

job\_cursor REFCURSOR;

BEGIN

OPEN job\_cursor FOR

SELECT job\_title, job\_description, job\_location, salary

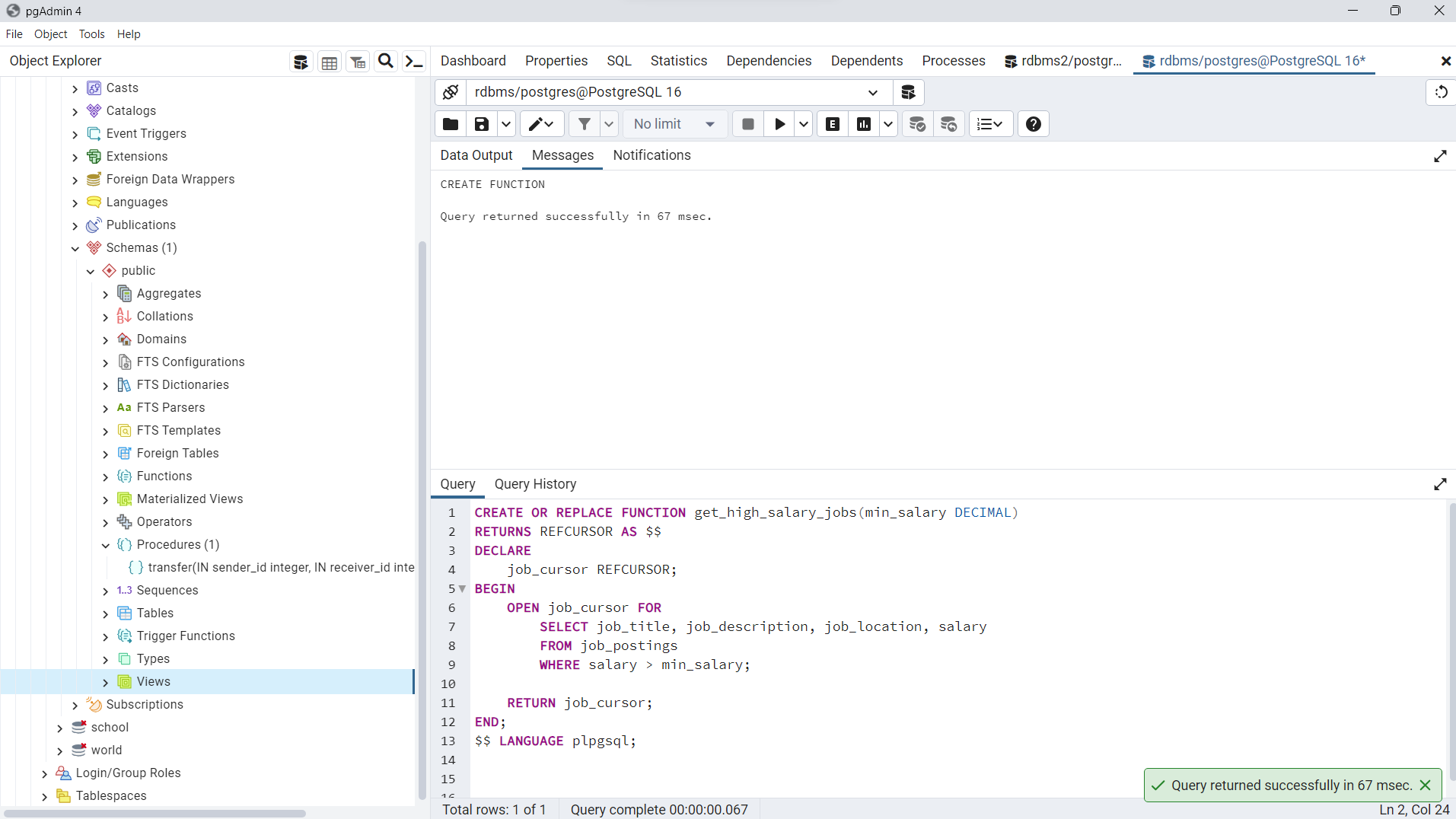
FROM job\_postings

WHERE salary > min\_salary;

RETURN job\_cursor;

END;

$$ LANGUAGE plpgsql;



# Conclusion:

# We learnt the concept of the **Procedure,Function and Cursor and implementation of Procedure,Function and Cursor.**

# Post Lab Questions:

# Does Storing Of Data In Stored Procedures Increase The Access Time? Explain?

Storing data in stored procedures does not inherently increase access time. Stored procedures are primarily used for encapsulating business logic and database operations, rather than storing data. However, the performance impact of stored procedures can depend on various factors such as the complexity of the logic, the efficiency of the SQL queries within the procedure, and the frequency of execution. In some cases, well-optimized stored procedures can enhance performance by reducing network traffic and promoting code reusability. However, poorly designed or inefficient stored procedures can potentially degrade performance.

# Explain the FETCH statement in SQL cursors.

The FETCH statement in SQL cursors is used to retrieve rows from the result set of a cursor. It fetches one or more rows from the cursor and assigns the fetched values to specified variables. There are different variations of the FETCH statement:

FETCH INTO: Fetches a single row from the cursor and assigns the column values to specified variables.

FETCH ALL INTO: Fetches all remaining rows from the cursor and assigns the column values to specified variables.

FETCH NEXT: Fetches the next row from the cursor.

FETCH PRIOR: Fetches the previous row from the cursor.

FETCH FIRST: Fetches the first row from the cursor.

FETCH LAST: Fetches the last row from the cursor.

FETCH ABSOLUTE n: Fetches the nth row from the beginning of the result set.

FETCH RELATIVE n: Fetches the row positioned relative to the current cursor position.

# What is the difference between a function and a stored procedure in PostgreSQL?

Function: A function is a named PL/pgSQL block that returns a value. It can accept input parameters and perform calculations or database operations based on those parameters. Functions can be used in SQL queries and can return single or multiple values. They are typically used for performing calculations or data manipulations.

Stored Procedure: A stored procedure is similar to a function but may not necessarily return a value. It is a reusable block of code that encapsulates a series of SQL statements or procedural logic. Stored procedures can accept input parameters, perform database operations, and include conditional logic and loops. They are commonly used for executing complex business logic or performing transactions.