Lab V

Functions & Procedures

18/02/2025

Functions:

PostgreSQL functions allow you to carry out operations that would normally take several queries and round trips in a single function within the database. Functions allow database reuse as other applications can interact directly with your stored procedures instead of a middle-tier or duplicating code.

The basic syntax to create a function is as follows –

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $variable_name$
DECLARE
declaration;
BEGIN
   <function_body>
   RETURN { variable_name | value };
END;
$variable_name$
LANGUAGE plpgsql;
```

Example

1. Function to find the total number of employees in the EMPLOYEES table

```
CREATE OR REPLACE FUNCTION total_employees()
RETURNS integer AS $total$
DECLARE
total integer;
BEGIN
   SELECT count(*) INTO total FROM EMPLOYEES;
   RETURN total;
END;
$total$
LANGUAGE plpgsql;
```

2. Function to count products within a price range (from **Price_from** to **Price_to**)

```
CREATE FUNCTION count_products(Price_min int, Price_max int)
RETURNS int
LANGUAGE plpgsql
AS $$
DECLARE
   product_count integer;
BEGIN
   SELECT count(*) INTO product_count FROM PRODUCTS WHERE Price BETWEEN
Price_min AND Price_max;
   RETURN product_count;
END;
$$;
```

Reference : https://www.postgresql.org/docs/current/sql-createfunction.html

SQL Functions with Default Values for Arguments

Functions can be declared with default values for some or all input arguments. The default values are inserted whenever the function is called with insufficiently many actual arguments. Since arguments can only be omitted from the end of the actual argument list, all parameters after a parameter with a default value have to have default values as well. (Although the use of named argument notation could allow this restriction to be relaxed, it's still enforced so that positional argument notation works sensibly.)

```
CREATE FUNCTION add_numbers(a int, b int DEFAULT 5, c int
 DEFAULT 10)
RETURNS int
LANGUAGE SQL
AS $$
SELECT $1 + $2 + $3;
$$;
```

```
SELECT add_numbers(10, 20, 30);  -- Returns 60
SELECT add_numbers(10, 20);      -- Returns 40
SELECT add_numbers(10);          -- Returns 25
SELECT add_numbers();            -- Error: Missing required
 argument
```

SQL Functions with Variable Numbers of Arguments

SQL functions can be declared to accept variable numbers of arguments, so long as all the “optional” arguments are of the same data type. The optional arguments will be passed to the function as an array. The function is declared by marking the last parameter as `VARIADIC`; this parameter must be declared as being of an array type. For example

```
CREATE FUNCTION min_value(VARIADIC arr numeric[]) RETURNS
 numeric
AS $$
SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

```
SELECT min_value(12, 5, 7, 3);  -- Returns 3
```

SQL Functions on Composite Types

When writing functions with arguments of composite types, we must not only specify which argument we want but also the desired attribute (field) of that argument. For example, suppose that `emp` is a table containing employee data, and therefore also the name of the composite type of each row of the table. Here is a function `double_salary` that computes what someone's salary would be if it were doubled:

```
CREATE TABLE emp (
name text,
salary    numeric,
age  integer,
cubicle   point
 );

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

 CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
     SELECT $1.salary * 2 AS salary;
 $$ LANGUAGE SQL;

 SELECT name, double_salary(emp.*) AS dream
     FROM emp
WHERE emp.cubicle ~= point '(2,1)';
```

```
 name | dream
      +
 Bill | 8400
```

It is also possible to build a function that returns a composite type. This is an example of a function that returns a single `emp` row:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,
1000.0 AS salary,
25 AS age,
point '(2,2)' AS cubicle;
 $$ LANGUAGE SQL;
```

A different way to define the same function is:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

Procedures:

A procedure is similar to a function but does not necessarily return a value. Procedures can modify both data and relations. Functions can be called within procedures, but procedures cannot be called within functions.

Basic 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'
```

```
    | sql_body
  } ...
```

Examples

1.Procedure to transfer funds between accounts

```
CREATE OR REPLACE PROCEDURE transfer_funds(
   sender_id INT,
   receiver_id INT,
   amount DECIMAL
)
LANGUAGE plpgsql AS $$
BEGIN
   UPDATE accounts SET balance = balance - amount WHERE id =
sender_id;
   UPDATE accounts SET balance = balance + amount WHERE id =
receiver_id;
   COMMIT;
END;
$$;
```

2.Procedure to delete a product from the PRODUCTS table

```
CREATE OR REPLACE PROCEDURE delete_product(product_id INT)
LANGUAGE plpgsql AS $$
BEGIN
   DELETE FROM PRODUCTS WHERE ID = product_id;
END;
$$;
```

Collectively, functions and procedures are also known as *routines*. There are commands such as `ALTER ROUTINE` and `DROP ROUTINE` that can operate on functions and procedures without having to know which kind it is.

To rename the routine `foo` for type `integer` to `foobar`:

```
ALTER ROUTINE foo(integer) RENAME TO foobar;
```

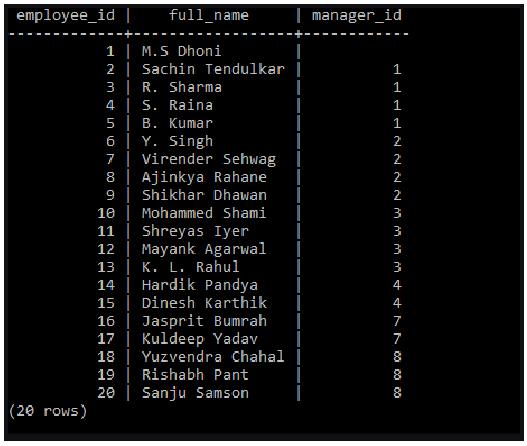
This command will work independent of whether `foo` is an aggregate, function, or procedure.

FOR-Loop

PostgreSQL provides the 'for' loop statements to iterate over a range of integers or over a result set or over the result set of a dynamic query.
Example:

1. The following code uses for loop to iterate over 10 numbers from 1 to 10 and display them in each iteration.

```
do $$
 begin
for cnt in 1..10 loop
     raise notice 'cnt: %', cnt;
    end loop;
end; $$
```



employee_id	full_name	manager_id
1	M.S Dhoni	
2	Sachin Tendulkar	1
3	R. Sharma	1
4	S. Raina	1
5	B. Kumar	1
6	Y. Singh	2
7	Virender Sehwag	2
8	Ajinkya Rahane	2
9	Shikhar Dhawan	2
10	Mohammed Shami	3
11	Shreyas Iyer	3
12	Mayank Agarwal	3
13	K. L. Rahul	3
14	Hardik Pandya	4
15	Dinesh Karthik	4
16	Jasprit Bumrah	7
17	Kuldeep Yadav	7
18	Yuzvendra Chahal	8
19	Rishabh Pant	8
20	Sanju Samson	8

(20 rows)

2. Consider the given relation and the code. The code uses for loop to iterate over largest 10 employee id

```
do
$$
declare
    f record;
begin
    for f in select employee_id, full_name
           from employees
           order by employee_id desc, full_name
           limit 10
    loop
    raise notice '% - % ', f.employee_id, f.full_name;
    end loop;
end;
$$;
```

Further Reference: https://www.geeksforgeeks.org/postgresql-for-loops/

IF-ELSE

In Postgres, the if statement checks a condition/criteria and returns true or false. In PostgreSQL, when a condition is false, the if statement does not handle it. Therefore, to handle the false conditions, the else statement is used in Postgres.

```
do $$
declare
   employee_salary INT := 50000;
begin
   IF employee_salary > 60000 THEN
      RAISE NOTICE 'High salary employee';
   ELSE
      RAISE NOTICE 'Average salary employee';
   END IF;
end $$;
```

Reference:

https://www.commandprompt.com/education/postgresql-if-else-statement-with-examples