

# Stored Procedures and Triggers

# Content

1. PostgreSQL Procedural Language
2. Types of functions
3. The PL/ pgSQL language

# 1. PostgreSQL Procedural Language (PL)

- PL = programming languages used by the database engine → manipulate and extract data
- PL **execute directly inside the database engine**, instead of remotely in a separate application program.
  - greatly **improve performance** by reducing the execution time of your application
  - provide a standard place to store functions that is **accessible to anyone who uses the database**

# Procedural languages

- Standard PostgreSQL supports:
  - PL/pgSQL
  - PL/Tcl
  - PL/Perl
  - PL/Python
- additional procedural languages available but not included in the core distribution: PL/Java, PL/PHP, PL/sh, ...
- To use = installed PL on the system
- Default: PL/pgSQL is installed

# PL/ pgSQL

- **NOT** a standard language - it is specific to PostgreSQL
- **NOT** portable to other database engines
- **HOWEVER:**
  - very close to Oracle's default procedural language, **PL/SQL**
  - very little effort is required to port functions created in PL/SQL to PL/pgSQL, and visa versa

<file:///C:/Program%20Files/PostgreSQL/9.4/doc/postgresql/html/plpgsql.html>

## 2. Types of functions

- 2 types of functions:
  - Stored procedures
  - Triggers
- Stored procedures:
  - instead of using individual SQL commands create a specialized stored procedure that performs **multiple SQL commands as one function**.
  - Each user can then **execute the stored procedure** as a **single function within a SQL command**

## 2. Types of functions

- Triggers:
  - are functions that are **executed directly by the database engine**, **based on an event** (insert, update, delete... ) that occurs in a table.
  - often **used to update related data** in tables automatically when a table value is inserted or updated in a single table → allows you to **maintain data relations automatically**, **without having to worry about that in your application code**.

### 3. The PL/ pgSQL language

- Uses **standard SQL commands** to build programs (query, insert, modify, and delete) → easy to learn
- Uses **standard programming** statements:
  - define and evaluate variables
  - accept input values for the function
  - supply output values from the function
  - use control logic, such as conditional loops and IF/THEN statements
- *interpreted* PL = **NOT** compiled into a binary form, **EXECUTES** the text program lines one by one as the function is executed

<file:///C:/Program%20Files/PostgreSQL/9.4/doc/postgresql/html/plpgsql.html>



### 3. The PL/ pgSQL language

- The first time a PL/pgSQL program is run, an *execution plan* is prepared:
  - *keeps the plan* and *uses it every time* the function is called (during the PostgreSQL server session) → increases performance, as the PL/pgSQL code does not have to be interpreted each time the function is run
- **CAREFUL** when making dynamic changes to functions:
  - PostgreSQL detects if a function's code has changed, and creates a new execution plan when necessary
  - A function *A* call a function *B*, *B is changed*, **BUT** execution plan of *A* is not changed → **re-create A**

# Creating a PL/pgSQL Function

- CREATE [OR REPLACE] FUNCTION *functionname* ([ [*argmode*] [*argname*] *argtype* ,...] ) [RETURNS *returntype*] AS \$\$

DECLARE <variable declarations>

BEGIN

<code section>

END;

\$\$

LANGUAGE *language*

[ IMMUTABLE | STABLE | VOLATILE

| CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT |  
STRICT

| SECURITY INVOKER | SECURITY DEFINER ];

# Creating PL/pgSQL Function - Explain

- *language name*: procedural language used to create the function (eg. PL/pgSQL = plpgsql)
- *function name*: unique name used to identify the function
- input and output arguments: *argmode argname argtype*
  - *argmode*: IN, OUT, INOUT
  - may define as many input and output variables as you need (each separated by a comma)
- for compatibility with previous versions, PostgreSQL 8 also **supports defining a single output variable** using the RETURNS keyword → The last line of code in the function must be a RETURN statement

# Creating PL/pgSQL Function - Explain

- **\$\$**: signify the *start and end of the function code* text
- **DECLARE** section of the code is used to declare variables used within the function:
  - *variablename datatype [ := value];*
- Output:
  - RETURN → must RETURN in code
  - OUT

# Simple example

- **Definition:**

```
CREATE FUNCTION store.test(IN val1 int4, IN val2 int4, out result  
    int4) AS
```

```
$$DECLARE vmultiplier int4 := 3;
```

```
BEGIN
```

```
    result := val1 * vmultiplier + val2;
```

```
END; $$
```

```
LANGUAGE plpgsql;
```

- **Execute:** `select store.test(10, 5);`

# Function parameters

- **IMMUTABLE:** indicates that the function cannot modify the database and always returns the same result when given the same argument values. Any call of the function with all-constant arguments can be immediately replaced with the function value.
- **STABLE:** indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same argument values, but that its result could change across SQL statements.
- **VOLATILE** indicates that the function value can change even within a single table scan. DEFAULT

# Function parameters

- **CALLED ON NULL INPUT:** the function will be called normally when some of its arguments are null. DEFAULT
- **RETURNS NULL ON NULL INPUT / STRICT:** indicates that the function always returns null whenever any of its arguments are null.
- **SECURITY INVOKER** indicates that the function is to be executed with the privileges of the user that calls it. DEFAULT
- **SECURITY DEFINER** specifies that the function is to be executed with the privileges of the user that created it.

# Notification

- Must use the **GRANT SQL** command and grant the **EXECUTE** privilege to **Group or Login Roles** before other users can use the new stored procedure function
- When creating complex functions, create a text file that contains the complete **CREATE FUNCTION** command
- When rewriting a function, use the **CREATE OR REPLACE FUNCTION** format when creating your function code in the text file → any code updates will replace the function without you having to manually delete the function
- Must end the **CREATE FUNCTION** command with a semicolon
- When creating a function, use a graphical development environment that allows you to alter program code within an editing window



# Creating a Stored Procedure Using pgAdmin III

- **Functions** Define functions that use the RETURNS keyword to return a single value
- **Procedures** Define functions that use the OUT keyword to return one or more values → **Functions** *in new version pgAdmin III*
- **Triggers** Define functions that are used to manipulate tables based on a table event
- pgAdmin III → graphic → ok
- Query Tool? psql?

# PL/pgSQL Function Code

- Assigning Values to Variables:
  - *variable := expression;*
- The format of the SELECT INTO statement:
  - *SELECT INTO variable [, ... ] column [, ... ] clause;*
  - The variable must be declared using the table's %ROWTYPE table attribute
  - *customervar store."Customer"%ROWTYPE*  
*SELECT INTO customervar \* FROM store."Customer" where*  
*"CustomerID" = 'BLU001';*  
*result := customervar."FirstName" || ' ' || customervar."LastName";*

# Condition Statements

- IF *condition* THEN  
    <*statements1*>  
ELSE  
    <*statements2*>  
END IF;

- IF *condition1* THEN  
    <*statements1*>  
ELSE IF *condition2* THEN  
    <*statements2*>  
ELSE  
    <*statements3*>  
END IF;

# Loop Statements

- LOOP

*<statements>*

EXIT [ WHEN *expression*];

END LOOP;

- WHILE *condition* LOOP

*<statements>*

END LOOP;

- FOR *variable* IN *select\_clause* LOOP

*<statements>*

END LOOP;

```
customervar store."Customer"%ROWTYPE;  
FOR customervar IN SELECT * FROM store."Customer" LOOP  
    <statements>  
END LOOP;
```

# Trapping errors

**BEGIN**

*statements*

**EXCEPTION**

WHEN *condition* [ OR *condition* ... ] THEN  
*handler\_statements*

[ WHEN *condition* [ OR *condition* ... ] THEN  
*handler\_statements* ... ]

**END;**

The *condition* names can be any of those shown in

[Appendix A](#)

# Trapping errors

- When an error is caught by an **EXCEPTION** clause, the local variables of the PL/pgSQL function remain as they were when the error occurred, but all changes to persistent database state within the block are rolled back.

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

# Examples

- CREATE FUNCTION add(integer, integer) RETURNS integer AS

'select \$1 + \$2;'

LANGUAGE SQL

IMMUTABLE

RETURNS NULL ON NULL INPUT;

# Examples

- CREATE OR REPLACE FUNCTION increment (i integer)  
RETURNS integer AS

```
$$ BEGIN RETURN i + 1; END; $$
```

```
LANGUAGE plpgsql;
```



# Examples

- `CREATE FUNCTION dup(in int, out f1 int, out f2 text) AS  
$$ SELECT $1, CAST($1 AS text) || ' is text' $$  
LANGUAGE SQL;`
- `SELECT * FROM dup(42);`

# Examples

- `CREATE TYPE dup_result AS (f1 int, f2 text);`
- `CREATE FUNCTION dup(int) RETURNS dup_result AS  
$$ SELECT $1, CAST($1 AS text) || ' is text' $$  
LANGUAGE SQL;`
- `SELECT * FROM dup(42);`

# Examples

- ```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price
                    FROM sales AS s
                    WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

# Examples

- ALTER FUNCTION sqrt(integer) RENAME TO square\_root;
- ALTER FUNCTION sqrt(integer) OWNER TO joe;
- ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
- DROP FUNCTION sqrt(integer);

# Other examples

- <http://www.java2s.com/Code/PostgreSQL/CatalogPostgreSQL.htm>
- <file:///C:/Program%20Files/PostgreSQL/9.4/doc/postgresql/html/plpgsql.html>