Databases

Seminar 8
Triggers

Triggers

Stored code is a database object that is a set of SQL procedures (and not only them) that is compiled once and stored on the server.

Why do I need stored procedures?

- hide data processing algorithms;
- expand programming capabilities, allowing you to implement complex logic;
- support data security and integrity functions, exception handling.

PostgreSQL allows you to develop your own stored procedures in languages other than SQL (also in C, but we don't consider it as part of the course).

These other languages in general are usually called procedural languages (PL, Procedural Languages).

Triggers

Procedural languages are not built into the PostgreSQL server; they are offered by downloadable modules:

- PL/pgSQL
- PL/Tcl
- PL/Perl
- PL/Python

PL/pgSQL is a programming language used to implement stored procedures for PostgreSQL. With this PostgreSQL extension, you can implement executable blocks, functions, and a special database object – triggers.

In addition to PL/pgSQL, we will also implement triggers in PL/Python.

Functions

PostgreSQL provides a large number of functions for embedded data types.

For example, the string repetition function:

repeat (text, integer) → text

(the right arrow indicates the result of the function)

repeat('Pg', 4) → 'PgPgPgPg'

Native functions are defined on the server by CREATE FUNCTION commands. Such a command usually looks, for example, like this:

CREATE FUNCTION somefunc(integer, text) RETURNS integer

AS 'function body'

LANGUAGE [SQL|plpgsql|plpython|...];

Note!
If we consider CREATE FUNCTION, the body of the function is just a text string. Often, to write the body of a function, it is more convenient to enclose this line in dollars, rather than in ordinary apostrophes.
If you do not end the line in dollars, all apostrophes or backslashes in the body of the function will have to be accounted for by duplicating them.

String constants in functions

String constants in dollars can be nested into each other by choosing different tags at different levels of nesting. This is most often used when writing function definitions. For example:

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

Functions in SQL

The simplest version of functions in the query language (functions written in SQL).

Just look at a few examples:

```
CREATE FUNCTION add(integer, integer)
RETURNS integer
AS 'SELECT $1 + $2;'
LANGUAGE SQL;
---- Request
SELECT add(1, 3);
--- Answer
| add |
-------
|  4  |
```

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

---- Request
SELECT * FROM dup(42);

--- Answer
| f1   | f2         |
---------------------
|   42 | 42 is text |
```

Functions in PL/pgSQL

PL/pgSQL is a block-structured language. The text of the function body must be a block.
Block Structure:

[<<label>>]

[DECLARE

initialization]

BEGIN

operators

END [label];

Functions in PL/pgSQL

Each declaration and each statement in the block must end with a ";" (semicolon). A block nested in another block must have a semicolon after the END, as shown above. However, the final END terminating the function body does no require a semicolon.

The label is required only when it is necessary to identify a block in the EXIT statement, or to supplement the names of variables declared in this block. If the label is specified after the END, then it must match the label at the beginning of the block.

Keywords are case insensitive. As in normal SQL commands, identifiers are implicitly converted to lowercase unless they are enclosed in double quotes.

Comments in PL/pgSQL code work the same way as in regular SQL. Double dash -- starts a comment that ends at the end of the line. A block comment starts with /* and ends with */. Block comments can be nested.

Functions in PL/pgSQL

Any statement in the executed section of the block can be a nested block. Nested blocks are used to logically group multiple operators or localize the scope of variables for a group of operators. During the execution of a nested block, variables declared in it hide variables of external blocks with the same names. To access external variables, you need to add a block label to their names.

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
quantity integer := 30;
BEGIN
RAISE NOTICE 'Now quantity = %', quantity; -- Output 30
quantity := 50;
--
-- Nested block
--
DECLARE
quantity integer := 80;
BEGIN
RAISE NOTICE 'Now quantity = %', quantity; -- 80
RAISE NOTICE ' is outputIn the outer block quantity = %', outerblock.quantity; -- 50 is output
END;

RAISE NOTICE 'Now quantity = %', quantity; -- 50 is output

RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

It is important not to confuse the use of BEGIN/END for grouping operators in PL/pgSQL with the SQL commands with the same name for transaction management.
BEGIN/END in PL/pgSQL only serve to group sentences; they do not start or end transactions.
You can read about transaction management in PL/pgSQL in the documentation.
In addition, the block with the EXCEPTION clause essentially creates a nested transaction that can be canceled without affecting the external transaction.
Error handling is described in more detail in the documentation.

Installation

Before you can using a package, you need to install it. If you are using a postgres container on top of apline download the packages:

```
apk add --no-cache --virtual .plpython3-deps --repository
http://nl.alpinelinux.org/alpine/edge/testing \

    postgresql-plpython3 \

    && ln -s /usr/lib/postgresql/plpython3.so
/usr/local/lib/postgresql/plpython3.so \

    && ln -s /usr/share/postgresql/extension/plpython3u.control
/usr/local/share/postgresql/extension/plpython3u.control \

    && ln -s /usr/share/postgresql/extension/plpython3u--1.0.sql
/usr/local/share/postgresql/extension/plpython3u--1.0.sql \

    && ln -s /usr/share/postgresql/extension/plpython3u--unpackaged--
1.0.sql /usr/local/share/postgresql/extension/plpython3u--unpackaged--
1.0.sql
```

After you need to install PL/Python in a specific database, run the command:

```
CREATE EXTENSION plpython3u;
```

Functions in PL/Python are declared in a standard way using the CREATE FUNCTION command:

```
CREATE FUNCTION funcname (argument-list)

  RETURNS return-type

AS $$

  # functions body  PL/Python

$$ LANGUAGE plpython3u;
```

PL/Python Functions

The function body contains just a Python script. When a function is called, its arguments are passed as args list items; named arguments are also passed to the Python script as regular variables. With the use of named arguments, the script is usually better read. The result from the Python code is returned in the usual way, with the return or yield command (in the case of a function that returns a set). If the return value is undefined, Python returns None. The PL/Python executor converts the Python None to the SQL NULL value.

For example, a function that returns the larger of two integers can be defined as:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpython3u;
```

The values of the arguments are set in global variables. As a result, according to Python variable scope rules, an argument variable cannot be assigned an expression inside a function that includes the name of this variable itself, unless this variable is declared global in the current block. For example, the following code will not work:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip()  # error
  return x
$$ LANGUAGE plpython3u;
```

Since assigning a value to x makes x a local variable for the entire block, and at the same time, x in the right part of the assignment turns out to be an undefined local variable x, and not a parameter of the PL/Python function. By adding the global operator, this can be fixed:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip()  # no error
  return x
$$ LANGUAGE plpython3u;
```

Useful links

About types and their mutual conversion

Anonymous code blocks

Accessing database data

Auxiliary functions

Function Volatility Categories

Each function has its own degree/category of volatility: VOLATILE, STABLE, IMMUTABLE.

VOLATILE is the default value in the CREATE FUNCTION in the command. The degree of volatility is a contract for the optimizer about the behavior of the function:

- The VOLATILE function does not give any guarantees about its behavior: the behavior can be non-deterministic with the same arguments, modify the internal state of the database, ... The optimizer does not make any assumptions about the behavior of such a function. Queries using VOLATILE functions will recalculate the value of the function on each applied row of the table.
- The STABLE function guarantees that it does not modify the internal state of the database and guarantees deterministic behavior on the same arguments within the same query. Therefore, this degree of volatility allows you to optimize multiple calls by caching. In particular, such functions can be used to search by index (this is prohibited for VOLATILE).
- The IMMUTABLE function gives the same guarantees as STABLE, but removes the restriction on the scope of a single request. Therefore, the optimizer can pre-"warm up the cache" on constant arguments.

For better operation of the optimizer, you should use the acceptable category that gives more guarantees about its behavior. Functions that modify the state of the database should be designated as VOLATILE. Functions like random(), currval(), timeofday() should also be designated VOLATILE.

Procedures

A procedure is a database object similar to a function, but with the following differences:

- Procedures are defined by the CREATE PROCEDURE command, not CREATE FUNCTION.
- Procedures, unlike functions, do not return a value; therefore, there is no RETURNS clause in the CREATE PROCEDURE. However, procedures can output data to the calling code via output parameters.
- Functions are called as part of a request or a DML command, and procedures are called separately by the CALL command.
- A procedure, unlike a function, can commit or roll back transactions during its execution (and then automatically start a new transaction) if the calling CALL command is not in an explicit transaction block.
- Some function attributes (for example, STRICT) are not applicable to procedures. These attributes affect function calls in queries and are not related to procedures.

```
CREATE PROCEDURE insert_data(a integer, b integer)

LANGUAGE SQL --  may use other procedural languages

AS $$

  INSERT INTO tbl VALUES (a);

  INSERT INTO tbl VALUES (b);

$$;

CALL insert_data(1, 2);
```

Triggers

A trigger is a stored procedure of a special type that the user does not call directly, but whose execution is conditioned by an action to modify data:
adding INSERT, deleting DELETE rows in a given table, or changing UPDATE data in a certain column of a given relational database table.

They can be performed before/instead of/after the main action and for the entire row /entire expression of data modification.

When	Event	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
AFTER	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	—
	TRUNCATE	—	—

Trigger creation algorithm:

1. Creating a stored function that returns a special TRIGGER type
2. Creating a trigger directly that triggers the function from point 1

Functions for triggers:

- Do not accept anything at the entrance
- Return the trigger type
- They can use special attributes of the type TG_variable

Special variables to use in triggers

Special variables to use in triggers:

- NEW is a RECORD type variable containing a new string of UPDATE/INSERT operations. NULL for DELETE
- OLD is a RECORD type variable containing the old string of UPDATE/DELETE operations. NULL for INSERT
- TG_WHEN is a variable of type TEXT indicating the trigger time: BEFORE, AFTER, INSTEAD OF
- TG_LEVEL is a variable of the TEXT type corresponding to the trigger type by the trigger level: ROW, STATEMENT
- TG_OP is a TEXT type variable corresponding to the type of operation that the trigger was called for: INSERT, DELETE, UPDATE, TRUNCATE
- TG_TABLE_NAME is a NAME type variable corresponding to the table that triggered the trigger
- TG_TABLE_SCHEMA is a variable of type NAME corresponding to the schema in which the table that triggered

the trigger is stored

Trigger creation syntax

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF }
{ event [ OR ... ] }
    ON table_name
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
```

where event can be one of:

```
    INSERT
    UPDATE [ OF column_name [, ... ] ] DELETE
    TRUNCATE
```

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE FUNCTION log_account_update();
```

Triggers are created outside of schemas, bound to a specific database table. They are also deleted with an indication of the table on which the trigger was created:

```
DROP TRIGGER [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```