Databases

Lecture 7
SQL extensions. Triggers, functions, procedures

I. PL/pgSQL. Functions and procedures

PL/pgSQL

- PL/pgSQL - a procedure language for DBMS PostgreSQL
- PL/pgSQL:
  - used to create functions, procedures and triggers
  - adding control structures to SQL language
  - can perform complex calculations
- PL/pgSQL functions can be used wherever built-in functions are allowed
- Technologically, PL/pgSQL is a default loaded module that can be removed from the layout of a specific DBMS by the administrator
- PL/pgSQL allows you to group a block of calculations and a sequence of queries within a database server

PL/pgSQL

Types of stored procedures:

- Functions
  - Return result
  - Triggers are defined through function syntax
  - PL/pgSQL functions are used in expressions with SQL statements in the same way as built-in functions
  - Control statements cannot be used in the function body
  - transactions
- Procedures
  - Doesn't return result
  - To make a call you must use the CALL statement
  - can control transactions (unless the procedure call statement is itself inside a transaction)

PL/pgSQL

- You can use SQL statements as operators in stored functions and procedures written in PL/pgSQL
- Neither functions nor procedures can store any values in internal variables between different calls to the same or different functions (procedures). All data that is passed between calls must be passed through function or procedure parameters, database server configuration parameters, or written to the database

PL/pgSQL and other programming languages

- In the PostgreSQL system, the body of a subroutine can be written in any of the programming languages known to the database server at the time of execution of the statement that creates the function or procedure
- By default these are C and SQL - they are considered internal Postgres languages
- Also in the basic version of Postgres there are Python, Perl, Tcl processors
- There are extensions for other languages

PL/pgSQL: functions: syntax

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'body of the function'
LANGUAGE plpgsql;

CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS $$body of the function$$
LANGUAGE plpgsql;
```

PL/pgSQL: functions: syntax
block structure

The function body can also be written in the form of a block:

```
AS$$
[ DECLARE
            definition ]
BEGIN
   operators
END;
$$;
```

PL/pgSQL: functions: syntax
block structure

- DECLARE
  - optional section describing local variables used in this block
- BEGIN… END
  - section containing executable statements

- An additional EXCEPTION section is also allowed,
- EXCEPTION describes the handling of “exceptions”

PL/pgSQL: functions: syntax

The conditional operator is written in one of the following ways:

```
IF condition THEN
      operator; ...
ELSE
      operator; ...
END IF;
```

```
IF condition THEN
     operator; ...
END IF;
```

PL/pgSQL: functions: syntax

The operator for selecting alternative calculation options can be written in two forms: with separate conditions for each alternative or with a list of possible expression values:

```
CASE
    WHEN condition-1 THEN
        operator; ...
    WHEN condition-2 THEN
        operator; ...
        ...
    ELSE
        operator; ...
END CASE;
```

```
CASE expression
    WHEN condition-1 THEN
        operator; ...
    WHEN condition-2 THEN
        operator; ...
        ...
    ELSE
        operator; ...
END CASE;
```

PL/pgSQL: functions: syntax

Creating a loop:

```
LOOP
    operator; ...
END LOOP;
```

The number of repetitions of the loop is determined using the following headers:

- WHILE condition
- FOR variable IN start .. end BY step
- FOREACH variable IN ARRAY array

Loop without header repeats endlessly

PL/pgSQL: functions: example

```
CREATE OR REPLACE FUNCTION hello(p text) RETURNS text
    LANGUAGE plpgsql AS $$
        DECLARE
            v text;
        BEGIN
            v := 'Hello, ‘;
            RETURN v || p || '!’;
        END;
    $$;

SELECT hello('world');
```

PL/pgSQL: functions: example

```
CREATE OR REPLACE FUNCTION example_fixed_loop()
RETURNS VOID AS $$
DECLARE
    i INT;
BEGIN
    RAISE NOTICE 'Loop with a fixed number of repetitions:';
    FOR i IN 1..5 LOOP
        RAISE NOTICE 'Iteration %', i;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT example_fixed_loop();
```

PL/pgSQL: functions: example

```
CREATE OR REPLACE FUNCTION example_array_loop()
RETURNS VOID AS $$
DECLARE
    j INT;
    numbers INT[] := ARRAY[1, 2, 3, 4, 5];
BEGIN
    RAISE NOTICE 'Loop through array elements:';
    FOR j IN ARRAY_LOWER(numbers, 1)..ARRAY_UPPER(numbers, 1) LOOP
        RAISE NOTICE 'Элемент %: %', j, numbers[j];
    END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT example_array_loop();
```

PL/pgSQL: functions: example

```
CREATE OR REPLACE FUNCTION example_if_else(num INT)
RETURNS TEXT AS $$
BEGIN
    IF num > 0 THEN
        RETURN 'Positive number';
    ELSIF num < 0 THEN
        RETURN 'Negative number';
    ELSE
        RETURN 'Number is equal to zero';
    END IF;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT
example_if_else(5);
-- returns 'Positive
number’'
```

PL/pgSQL: functions: example

```
SELECT example_case(95);  -- returns 'Excellent'
```

```
CREATE OR REPLACE FUNCTION example_case(score INT)
RETURNS TEXT AS $$
BEGIN
    CASE
        WHEN score >= 90 THEN
            RETURN 'Excellent';
        WHEN score >= 80 THEN
            RETURN 'Good';
        WHEN score >= 70 THEN
            RETURN 'Satisfactory';
        ELSE
            RETURN 'Unsatisfactory';
    END CASE;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL: procedures: example

```
CREATE OR REPLACE PROCEDURE create_user(new_email TEXT)
LANGUAGE plpgsql AS $$
BEGIN
    IF EXISTS (SELECT 1 FROM users WHERE email = new_email) THEN
        RAISE EXCEPTION 'A user with this email already exists';
    END IF;

    INSERT INTO users(email)  VALUES (new_email)
    COMMIT;
    RAISE NOTICE ‘E-mail address % is added', new_email;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END;
$$;
```

PL/pgSQL

But there are also potential negative consequences of using functions and procedures

- Separate execution of subqueries. Nested queries can be converted to a join operation. However, a subquery placed inside a function is optimized and executed separately from the main query that uses the function.
- Side effects of functions or procedures. The database may change in ways that are not obvious or unpredictable to the user.
- Unavailability of cost estimates. The optimizer cannot always calculate the cost of performing a specific function or procedure, and therefore optimization algorithms simply will not work.

III. Triggers

Triggers

- This is a special type of stored procedure that is not directly called by the user, but whose execution is conditioned by a data modification action: INSERT, UPDATE [OF column_name [, ...]], DELETE, TRUNCATE
- Essentially, a trigger is a function that performs actions in response to a specific set of events

Triggers

- There are also event triggers that do not connect to a specific table, but work at the database level and allow you to create conditions for executing DDL commands
- Essentially, a trigger is a function that performs actions in response to a specific action.
- Triggers can significantly complement or change the semantics of standard SQL statements. For example, triggers can be used to test integrity conditions that cannot be described using standard SQL tools, or to log changes made by an application to another table.

Triggers

- To create a trigger function, you need to specify a special type when defining a stored procedure:

```
CREATE OR REPLACE FUNCTION log_user_updates()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$...$$;
```

Triggers: syntax

```
CREATE [ OR REPLACE ] [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referencing_table ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [WHEN (condition)]
    EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
```

Triggers: syntax

```
CREATE [ OR REPLACE ] [ CONSTRAINT ]
TRIGGER name { BEFORE | AFTER |
INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referencing_table ]
    [ NOT DEFERRABLE | [ DEFERRABLE ]
[ INITIALLY IMMEDIATE | INITIALLY
DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE
[ AS ] transition_relation_name } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [WHEN (condition)]
    EXECUTE { FUNCTION | PROCEDURE }
function_name ( arguments )
```

```
Valid events:
    INSERT
    UPDATE [ OF
column_name [, ... ] ]
    DELETE
    TRUNCATE
```

Triggers: work conditions

- The trigger function must return either NULL or a record/row corresponding to the structure of the table for which the trigger was fired
- The FOR EACH ROW and FOR EACH STATEMENT clauses determine whether the trigger function will fire once per row or per SQL statement. If nothing is specified, FOR EACH STATEMENT is implied. For constraint triggers, you can only specify FOR EACH ROW

Triggers: work conditions

When	Event	Row level	Operator level
BEFORE	INSERT/UPDATE/DELETE	Tables and third-party tables	Tables, views, and third-party tables
	TRUNCATE	—	Tables and third-party tables
AFTER	INSERT/UPDATE/DELETE	Tables and third-party tables	Tables, views, and third-party tables
	TRUNCATE	—	Tables and third-party tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	—
INSTEAD OF	TRUNCATE	—	—

Triggers: special variables

When a PL/pgSQL function fires as a trigger, several special variables are automatically created in the top-level block (function):

- NEW - new database row for INSERT/UPDATE commands in row level triggers. In statement level triggers and for the DELETE command, this variable has the value NULL
- OLD is the old database row for UPDATE/DELETE commands in row level triggers. In statement level triggers and for the INSERT command, this variable has the value NULL
- TG_NAME - name of the trigger
- TG_WHEN - BEFORE, AFTER or INSTEAD OF depending on the trigger definition

Triggers: special variables

When a PL/pgSQL function fires as a trigger, several special variables are automatically created in the top-level block (function):

- TG_LEVEL - ROW or STATEMENT depending on trigger definition
- TG_OP - operation for which the trigger was fired: INSERT, UPDATE, DELETE or TRUNCATE
- TG_TABLE_NAME - table for which the trigger was fired
- TG_TABLE_SCHEMA - table schema for which the trigger was fired
- TG_NARGS - the number of arguments in the CREATE TRIGGER command that are passed to the trigger function

Triggers: examples

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

Triggers: examples

```
CREATE OR REPLACE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

Triggers: examples

The trigger in the following example, whenever a row is added or changed in a table, stores information about the current user and a timestamp in that row. In addition, it requires that the employee's name be specified and the salary be specified as a positive number.

Step 1: create a table:

```
CREATE TABLE emp (
  empname       text,
  salary        integer,
  last_date     timestamp,
  last_user     text
);
```

Triggers: examples

Step 2: create a trigger function:

```
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that the employee's name and salary are indicated
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        ENDIF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        ENDIF;

        -Who will work if they have to pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        ENDIF;

        -- Remember who changed the entry and when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;
```

Triggers: examples

Step 3: create a trigger:

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE
ON emp
    FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```