Database Programming in an Object-Relational SQL Procedural Programming Language

PL/pgSQL - Procedural Language/PostgreSQL

Dirk Van Gucht[1]

[1]Indiana University

March 30, 2019

Motivation

- Thus far, we have covered SQL and Object-Relational SQL (abbreviated as OR-SQL) as a language in which each statement correspond to a single query, a single update, a single declaration (definition), etc.
- That means that a client application must send each query to the database server, wait for it to be processed, receive and process the results, do some computation, then send further queries to the server.
- All this incurs interprocess communication and will also incur network overhead if the client is on a different machine than the database server

Motivation

- In this lecture, we want to address writing applications, i.e., programs, wherein multiple OR-SQL can be bundled and processed using control statements such as assignment statements, conditional statements, loop statements, etc
- This correspond to writing programs in an imperative programming languages with the ability to use OR-SQL statements as embedded code
- We will use the PostgreSQL's plpgsql language to write such programs

Overview

- We begin with a general overview of the programming environment of plpgsql
- We give a formal definition of a plpgsql program and show how to run it in the PostgreSQL interpreter
- We will illustrate most of this using examples

plpgSQL (Declaration Statements)

Type declarations	CREATE TYPE
Relation declarations	CREATE TABLE
View declarations	CREATE VIEW
Functions declarations	CREATE FUNCTION
Triggers declarations	CREATE TRIGGER
Program variable declarations	DECLARE
Iterator declaration	FOR LOOP, FOREACH
Cursor declaration	DECLARE CURSOR

plpgSQL Garbage collection statements

Type	DROP TYPE
Relation	DROP TABLE
View	DROP VIEW
Function	DROP FUNCTION
Trigger	DROP TRIGGER
Cursor declaration	CLOSE CURSOR
Program variable declarations	not required
Iterator declarations	not required

Expressions and Statements

Expression	Any valid OR-SQL expression including SELECT FROM WHERE expression
Assignment statement	variable_name := expression
Assignment statement	SELECT INTO variable_name
Update statement	INSERT, DELETE, UPDATE
Return statement	RETURN expression
Return query statement	RETURN QUERY query
Function call	SELECT function(parameters)
Block statement	BEGIN ··· END
Loop statement	LOOP, WHILE, FOR
Conditional statements	IF ELSE, CASE
Cursor operations	OPEN, FETCH

plpgsql (Program)

- The syntax of a plpgsql program is as follows:

```
CREATE OR REPLACE FUNCTION functionName (list of arguments)
   RETURNS return type AS
$$
<label>
<DECLARE declarations>
BEGIN
    sequence of statements;
END <label>;
$$ LANGUAGE plpgsql;
```

- A program is excecuted in the PostgreSQL interpreter using the call

```
SELECT functionName(parameters);
```

Program (Example with conditional if-else statement)

An example program with the IF statement

```
CREATE OR REPLACE FUNCTION convert(a char)
 RETURNS float AS
$$
BEGIN
  IF (a = 't') THEN RETURN 1;
  ELSE
    IF (a= 'f') THEN RETURN 0;
    ELSE
      IF (a = 'u') THEN RETURN 0.5;
      ELSE RETURN(2);
      END IF;
    END IF;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

→

SELECT convert('u');

convert
1/2

SELECT convert('z');

convert
2

Program (Example with conditional case statement)

An example program with the CASE statement

```
CREATE OR REPLACE FUNCTION convert(a char)
 RETURNS float AS
$$
BEGIN
  CASE WHEN (a = 't') THEN RETURN 1;
         WHEN (a = 'f') THEN RETURN 0;
         WHEN (a = 'u') THEN RETURN 0.5;
         ELSE RETURN 2;
  END CASE;
END;
$$ LANGUAGE plpgsql;
```

→

```
SELECT convert('u');
 convert
---------
     0.5
SELECT convert('z');
 convert
---------
       2
```

Program (with loop statement)

Iterative program for the factorial(n) function

```
CREATE OR REPLACE FUNCTION factorial_Iterative (n integer)
RETURNS integer AS
$$
DECLARE
  result integer;
  i integer;
BEGIN
  result := 1;
  FOR i IN 1..n
    LOOP
      result := i * result;
    END LOOP;
  RETURN result;
END;
$$ language plpgsql;
```

Program (with recursion)

Recursive program for the factorial(n) function

```
CREATE OR REPLACE FUNCTION factorial_Recursive (n integer)
RETURNS integer AS
$$
BEGIN
  IF n = 0 THEN
    RETURN 1;
  ELSE
    RETURN n * factorial_Recursive(n-1);
  END IF;
END;
$$ language plpgsql;
```

Functions that affect the database state

- Functions can be defined to affect (change) the database state
- Often such functions do not need to return values: they have the VOID return type

```
CREATE OR REPLACE FUNCTION change_db_state()
  RETURNS VOID AS
$$
BEGIN
  DROP TABLE foo_relation;
  CREATE  TABLE  foo_relation(a integer);
  INSERT INTO foo_relation VALUES (1), (2), (3);
  DELETE FROM foo_relation WHERE a=1;
END;
$$ language plpgsql;
```

→

select change_db_state();

change_db_state

select * from foo_relation;

a
2
3

Program with local functions

- You can also CREATE local functions
- Care must be taken with function delimiters

```
CREATE OR REPLACE FUNCTION globalFunction()
  RETURNS void AS
$proc$
BEGIN
  CREATE OR REPLACE FUNCTION localFunction()
    RETURNS integer AS
     $$
      SELECT 5;
     $$ language sql;
END;
$proc$ language plpgsql;
```

→

```
SELECT globalFunction();
 globalfunction
----------------

SELECT localFunction();
 localfunction
---------------
             5
```

- Notice that `localFunction()` persists after the SELECT `globalFunction()` call

Two kinds of assignment statements

- The typical assignment statement is of the form

  x := expression;

- An assignment to a variable can also be done with a query and the clause

  SELECT tuple component(s) INTO variable (s) FROM ... WHERE;

- The value of the tuple component (s) is (are) assigned to the variable(s)

```
CREATE OR REPLACE FUNCTION size_of_A()
  RETURNS integer AS
$$
DECLARE counter integer;
BEGIN
  SELECT INTO counter COUNT(*) from A;
  RETURN counter;
END;
$$ language plpgsql
```

→

SELECT * FROM A;

x
’A’
’B’

SELECT size_of_A();

size_of_A
2

Special alternative for SELECT INTO assignment statement

```
CREATE OR REPLACE FUNCTION size_of_A()
  RETURNS integer AS
$$
DECLARE counter integer;
BEGIN
  SELECT INTO counter COUNT(*) from A;
  RETURN counter;
END;
$$ language plpgsql
```

Since the expression $(\text{SELECT COUNT}(*)\ \text{FROM} A)$ evaluates to a single integer, this program can also be written as

```
CREATE OR REPLACE FUNCTION size_of_A()
  RETURNS integer AS
$$
DECLARE counter integer;
BEGIN
  counter := (SELECT COUNT(*) from A);
  RETURN counter;
END;
$$ language plpgsql
```

SELECT INTO (non-deterministic behavior)

- SELECT INTO can lead to non-deterministic (random) effects!
- This is because SELECT INTO chooses the first available tuple from the result of the query and assigns it to the INTO variable (in our case the variable element_from_A).[3]
- Of course, this can be useful when sampling data

```
CREATE OR REPLACE FUNCTION choose_one_from_A()
  RETURNS text AS
$$
DECLARE element_from_A text;
BEGIN
  SELECT INTO element_from_A a.x
  FROM (SELECT x from A ORDER BY random()) a;
  RETURN element_from_A;
END;
$$ language plpgsql
```

→

```
SELECT choose_one_from_A();
 choose_one_from_a
-------------------
 'B'

SELECT choose_one_from_A();
 choose_one_from_a
-------------------
 'A'
SELECT choose_one_from_A();
 choose_one_from_a
-------------------
 'A'
```

[3]If the query does not return any tuple, then the variable is set to NULL.

“Assignment" statements to relation variables

- “Assignment" statements to relation (table) variables are done using the INSERT INTO, DELETE FROM, and UPDATE statements, or using triggers

```
CREATE OR REPLACE FUNCTION relation_assignment()
  RETURNS void AS
$$
BEGIN
  CREATE TABLE IF NOT EXISTS AB(A integer, B integer);
  DELETE FROM AB;
  INSERT INTO AB VALUES (0,0);
  INSERT INT AB SELCT a1.x, a2.x FROM A a1, A a2;
  UPDATE AB SET A = A*A WHERE B = 2;
END;
$$ language plpgsql;
```

→

select * from A;

x
1
2

SELECT * FROM AB;
ERROR: relation "ab" does not exist

SELECT relation_assignment();
SELECT * from AB;

a	b
0	0
1	1
2	1
1	2
4	2

Iterators over collections

- Relations and arrays are collections
- Relations are unordered collections whereas arrays are ordered collections
- We consider iterator variables that slide (move; iterate) over such a collection one element at a time
- In SQL, an iterator variable over a relation (which may or may not be the result of a query) is often referred to as a CURSOR
- In SQL, it is frequently not necessary to use cursors as the following function illustrates

```
CREATE OR REPLACE FUNCTION there_is_book_that_cost_more_than(k integer)
  RETURNS boolean AS
$$
  SELECT EXISTS(SELECT * FROM book WHERE price > k);
$$ language sql
```

Iterators over collections (cursors)

```
CREATE OR REPLACE FUNCTION there_is_book_that_cost_more_than(k integer)
  RETURNS boolean AS
$$
BEGIN
   SELECT EXISTS(SELECT * FROM book WHERE price > k);
END;
$$ language sql;
```

The following function with the same semantics does use the iterator record variable (cursor) b

```
CREATE OR REPLACE FUNCTION there_is_book_that_cost_more_than(k integer)
  RETURNS boolean AS
$$
DECLARE exists_book boolean;
              b RECORD; – the structure will be defined during the program
BEGIN
  exists_book := false;
  FOR b IN SELECT * FROM book – RECORD b will have have the attribute structure of the book relation
    LOOP
      IF b.price > k
      THEN exists_book := true;
      EXIT;
      END IF;
    END LOOP;
  RETURN exists_book;
END; $$ language plpgsql;
```

Iterators over arrays

- Below is an example from the PostgreSQL manual illustrating iteration through an array using the FOREACH clause
- The function sum takes an integer array as input and returns the sum of its elements
- The variable x is the iterator which gets assigned, one at a time, to each element in the array
- Note in particular that x is not assigned to index positions of the array

```
CREATE FUNCTION sum(A int[])
  RETURNS int8 AS
$$ DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY A
  LOOP
    s := s + x;
  END LOOP; RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Iterators over arrays

On the right is an alternative version for the sum function. There an index variable i is used that iterates over the index positions of the array.

```
CREATE FUNCTION sum(A int[])
  RETURNS int8 AS
$$ DECLARE
  s int8 := 0;
  x int;
BEGIN
  foreach x IN ARRAY A
  LOOP
    s := s + x;
  END LOOP;
RETURN s;
END;
$$ LANGUAGE plpgsql;
```

⇔

```
CREATE FUNCTION sum(A int[])
  RETURNS int8 AS
$$ DECLARE
  s int8 := 0;
  i int;
BEGIN
  FOR i IN array_lower(A,1)..array_length(A,1)
  LOOP
    s := s + A[i];
  END LOOP;
RETURN s;
END;
$$ LANGUAGE plpgsql;
```