Database Programming in an Object-Relational SQL Procedural Programming Language

PL/pgSQL - Procedural Language/PostgreSQL

Presented by: Muazzam Siddiqui

Original Slides: Dirk Van Gucht

Motivation

- Thus far, we have covered 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.
- As it turns out, OR-SQL is not a computationally complete programming language.[1]
- This implies that many processes that manipulate a database can not be formulated in OR-SQL
- For example, it is not possible to write an OR-SQL statement that computes the shortest distances between nodes in a weighted graph, etc.

[1] It can be shown that each OR-SQL (without recursive view) statement can be evaluated in polynomial time and space in the size of the database.

Motivation

- In this lecture, we want to address writing applications, i.e., programs, wherein multiple OR-SQL statements 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
- We will then write programs that compute queries which can not be expressed in OR-SQL
- These queries are of utmost importance in graph databases

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
$$
<DECLARE declarations>
BEGIN
     sequence of statements;
END;
$$ 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 (SELECT COUNT(*) 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;
```

Working with Cursors

- A PL/pgSQL cursor allows you to encapsulate a query and process each individual row at a time

- Works in four steps
  - Declare a cursor
  - Open a cursor
  - Fetch result rows one by one till there are no more rows
  - Close the cursor

Declare Cursor Variables

- All access to cursors in PL/pgSQL goes through cursor variables, which are always of the special data type refcursor

```
cursor_name [ [no] scroll ] cursor [(
arguments)] for query;

DECLARE
      curs1 refcursor;
      curs2 CURSOR FOR SELECT * FROM tenk1;
      curs3 CURSOR (key integer) FOR SELECT *
      FROM tenk1 WHERE unique1 = key;
```

- curs1 is an unbound cursor variable
- curs2 is bound to a query
- curs3 is is bound to a parameterized query

Opening cursors

- Before a cursor can be used to retrieve rows, it must be *opened*.

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ]
FOR query;

OPEN curs1 FOR SELECT * FROM foo WHERE key
= mykey;

OPEN bound_cursorvar [ ( [ argument_name :=
] argument_value [, ...] ) ];

OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

Using Cursors

```
FETCH [ direction { FROM | IN } ] cursor INTO
target;
```

- FETCH retrieves the next row from the cursor into a target, which might be a row variable, a record variable, or a comma-separated list of simple variables
- The **direction** can be NEXT, PRIOR, FIRST, LAST, ABSOLUTE **count**, RELATIVE **count**, FORWARD, or BACKWARD

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

- Other cursor manipulation statements include MOVE, UPDATE, DELETE and CLOSE

The Ancestor-Descendant Relation

- Assume that we are given a parent-child relation

$$\mathrm{PC}(\mathrm{parent}, \mathrm{child})$$

  In this relation, a pair (p, c) indicates that person p is the parent of person c

- Starting from the information in the PC relation, we want to compute the ancestor-descencant

$$\mathrm{ANC}(\mathrm{ancestor}, \mathrm{descendant})$$

  relation.

- In this relation, a pair (a, d) indicates that person a is an ancestor of person d.

Computing the Ancestor-Descendant relation in Pure SQL

- We saw a similar problem earlier with a recursive solution
- We will solve the present at iterative solution using plpgsql

A recursive definition of the ANC relation

- The ANC relation can be recursively defined using the following rules:

  Rule 1: if $\mathrm{PC}(p, c)$ then $\mathrm{ANC}(p, c)$
  Rule 2: if $\mathrm{ANC}(a, p)$ and $\mathrm{PC}(p, c)$ then $\mathrm{ANC}(a, c)$

- Rule 1 states that if p is a parent of c then p is an ancestor of c
- Rule 2 states that if a is an ancestor of p and if p is the parent of c, then a is an ancestor of c

Computing the ANC relation in stages

Rule 1: if $\mathrm{PC}(p, c)$ then $\mathrm{ANC}(p, c)$
Rule 2: if $\mathrm{ANC}(a, p)$ and $\mathrm{PC}(p, c)$ then $\mathrm{ANC}(a, c)$

- These two rules allow us to compute the ANC relation in stages:
  - Stage 1: Start with the (parent,child) pairs in PC
  - Stage 2: Add to these the (grandparent,grandchild) pairs
  - Stage 3: Add to these the (great-grandparent, great-grandchild) pairs
  - etc
- This computation in stages will terminate because the PC relation is assumed to be a finite relation
- We will show how we can compute these stages using a plpgsql program that use iteration

Computing the ANC relation in stages with RA expressions

- We will specify the computation at each stage using an RA expression
- We assume that PC has attributes P and C and that for each *i*, ANC_i has attributes A and D

$$
\begin{array}{lcl}
\mathrm{ANC}_0 & \leftarrow & \mathrm{PC} \\
\mathrm{ANC}_1 & \leftarrow & \mathrm{ANC}_0 \cup \pi_{\mathrm{A},\,\mathrm{C}}(\mathrm{ANC}_0 \bowtie_{\mathrm{D}=\mathrm{P}} \mathrm{PC}) \\
\mathrm{ANC}_2 & \leftarrow & \mathrm{ANC}_1 \cup \pi_{\mathrm{A},\,\mathrm{C}}(\mathrm{ANC}_1 \bowtie_{\mathrm{D}=\mathrm{P}} \mathrm{PC}) \\
\ldots & & \\
\mathrm{ANC}_{i+1} & \leftarrow & \mathrm{ANC}_i \cup \pi_{\mathrm{A},\,\mathrm{C}}(\mathrm{ANC}_i \bowtie_{\mathrm{D}=\mathrm{P}} \mathrm{PC}) \\
\ldots & & \\
 & \text{until} & \mathrm{ANC}_{n+1} = \mathrm{ANC}_n
\end{array}
$$

- Clearly, this sequence of computations suggest a simple loop

Computing $\mathcal{ANC}$ incrementally

- To develop our code, we first specify a function

$$\text{new\_ANC\_pairs()}$$

- This function computes, given an approximation for the ancestor relation $\mathcal{ANC}$, additional (ancestor, descendent)-pairs that should be present in $\mathcal{ANC}$

- Such pairs can be discovered by joining (i.e., composing) the current approximation for ANC with the PC relation

Computing ANC incrementally

- More precisely, at stage i ($i \geq 1$), the function `new_ANC_pairs()` computes ($\mathrm{ANC}_i - \mathrm{ANC}_{i-1}$)
- Observe that to compute ANC_i, it then suffices to insert into ANC_{i-1} this new set of new pairs
- Further observe that when the function `new_ANC_pairs()` at some stage $n + 1$ returns no new pairs, then $\mathrm{ANC}_{n+1} = \mathrm{ANC}_n$ and, if fact, then $\mathrm{ANC}_n = \mathrm{ANC}$

The new_ANC_pairs() function in SQL

Recall that ANC has schema (*A*, *D*) and PC has schema (*P*, *C*)

```
CREATE OR REPLACE FUNCTION new_ANC_pairs()
  RETURNS TABLE (A integer, D integer) AS
$$
  (SELECT A, C
   FROM     ANC JOIN PC ON (D = P))
  EXCEPT
  (SELECT A, D
   FROM     ANC);
$$ language sql;
```

The `Ancestor_Descendant()` program

We can now write a `plpgsql` program to compute the ANC relation

```
CREATE OR REPLACE FUNCTION Ancestor_Descendant()
  RETURNS void AS
$$
BEGIN
  DROP TABLE IF EXISTS ANC;
  CREATE TABLE ANC(A integer, D integer);

  INSERT INTO ANC SELECT * FROM PC;

  WHILE EXISTS (SELECT * FROM new_ANC_pairs())
  LOOP
     INSERT INTO ANC SELECT * FROM new_ANC_pairs();
  END LOOP;
END;
$$ language plpgsql;
```

Illustration of the result of the Ancestor_Descendant program

PC

P	C
1	2
1	3
1	4
2	5
2	6
3	7
5	8

SELECT Ancestor_Descendant();

→

SELECT * FROM ANC;

ANC

A	D
1	2
1	3
1	4
1	5
1	6
1	7
1	8
2	5
2	6
2	8
3	7
5	8