Relational Programming in SQL
PL/pgSQL - SQL Procedural Language

Dirk Van Gucht[1]

[1]Indiana University

October 26, 2018

PL/pgSQL Relational Programming Environment

- Types
- Variables
  - Simple-type variables; int, text, boolean etc
  - Complex-type variables; composite, array, JSON, etc
  - Relation and view variables
  - Cursors
- Functions and Triggers
- Declaration statements
  - Type declarations (CREATE TYPE)
  - Simple and complex-type variable declarations (DECLARE)
  - Relation and view variable declaration (CREATE TABLE and CREATE VIEW)
  - Functions and triggers declarations (CREATE FUNCTION and CREATE TRIGGER)
  - Cursor declaration (DECLARE CURSOR)

Garbage collection statements

- Garbage collection for types (DROP TYPE)
- Garbage collection for relation and view variables (DROP TABLE, DROP VIEW)
- Garbage collection for functions (DROP FUNCTION)
- Garbage collection for triggers (DROP TRIGGER)
- Garbage collection for cursors (DROP CURSOR)

Control statements

- Return statement (RETURN *expressions*;)
- PERFORM query
- Block statements (BEGIN $\cdots$ END)
- Loop statements (LOOP, WHILE, FOR)
- Conditional statements (IF THEN ELSE, CASE)
- Cursor operations (OPEN, FETCH, CLOSE)

Relational Programs

The skeleton of a relational program is as follows:

```
CREATE OR REPLACE FUNCTION functionName (list of arguments)
   RETURNS return type AS
$$
(Optional Declaration section;)
BEGIN
    sequence of statements;
END;
$$ LANGUAGE plpgsql;
```

Example: Relational Program

An example with the IF statement

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

→

SELECT convert('u');

convert
1/2

SELECT convert('z');

convert
2

Example: Relational Program

An example with the CASE statement

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

$\rightarrow$

SELECT convert('u');

convert
1/2

SELECT convert('z');

convert
2

Example: Iterative Program

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;
```

Example: Iterative Program

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 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

Example: Local functions

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

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

→

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

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

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 INTO variable query;

- The value of the query is assigned to the variable

```
CREATE OR REPLACE FUNCTION size_of_ A FUNCTION foo ()
  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

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).[1]
- Of course, this can be useful when sampling data

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

[1] 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

```
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
$$
BEGIN
   SELECT EXISTS(SELECT * FROM book WHERE price > k);
END;
$$ 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
$$
DECLARE exists_book boolean;
              b  RECORD;
BEGIN
  exists_book := false;
  FOR b IN SELECT * FROM book
    LOOP
      IF b.price > k
      THEN exists_book := true;
      EXIT;
      END IF;
    END LOOP;
  RETURN exists_book;
END; $$ language plpgsql;
```

Iterators over arrays

- Below is an example from the PostgreSQL manual illustrating iteration through an array using the FOR EACH 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
$$ DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOR EACH x IN ARRAY A
  LOOP
    s := s + x;
  END LOOP; RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Iterators over arrays

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

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

The Ancestor-Descendant Relation

- Assume that we are given a parent-child relation

  `PC(parent,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

  `ANC(ancestor,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

- It can be shown that there does not exist any Pure SQL query that can compute the `ANC` relation
- However, if we add an iteration construct to Pure SQL then this becomes possible

A recursive definition of the ANC relation

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

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

- 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 $\texttt{ANC}$ relation in stages

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

- These two rules allow us to compute the $\texttt{ANC}$ relation in stages:
  Stage 1: Start with the (parent,child) pairs in $\texttt{PC}$
  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 $\texttt{PC}$ relation is assumed to be a finite relation
- We will show how we can compute these stages using a simple iterative process

Computing the $\mathtt{ANC}$ relation in stages with RA expressions

- We will specify the computation at each stage using an RA expression
- We assume that $\mathtt{PC}$ has attributes $\mathtt{P}$ and $\mathtt{C}$ and that for each i, $\mathtt{ANC}_i$ has attributes $\mathtt{A}$ and $\mathtt{D}$

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

- Clearly, this sequence of computation suggest a simple loop

Computing `ANC` incrementally

- To develop our code, we first specify a function

  `new_ANC_pairs()`

- This function computes, given an approximation for the ancestor relation `ANC`, additional (ancestor, descendent)-pairs that should be present in `ANC`

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

Computing $\mathtt{ANC}$ incrementally

- More precisely, at stage i ($i \geq 1$), the function `new_ANC_pairs()` computes $(\mathtt{ANC}_i - \mathtt{ANC}_{i-1})$
- Observe that to compute $\mathtt{ANC}_i$, it then suffices to insert into $\mathtt{ANC}_{i-1}$ 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 $\mathtt{ANC}_{n+1} = \mathtt{ANC}_n$ and, if fact, then $\mathtt{ANC}_n = \mathtt{ANC}$

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
$$
  (SELECT A, C
   FROM   ANC, PC
   WHERE D = P)
  EXCEPT
  (SELECT A, D
   FROM   ANC);
$$ language sql;
```

The `Ancestor_Descendant()` program

We can now specify a function (a program) that computes 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;
```

Alternative version using SQL Recursive View

- Observe that the `ANC` relation can also be computed using a recursive view as follows:

```
WITH RECURSIVE ANC(A,D) AS
  (SELECT P, C FROM PC
   UNION
   SELECT A, C
   FROM    ANC INNER JOIN PC ON (D = P) )

SELECT A, D FROM ANC;
```

Illustration

PC

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

SELECT Ancestor_Descendant();

$\rightarrow$

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

Paths and Connectivity in Graphs

- A graph is a pair $G = (V, E)$ where V is a set of vertices (nodes) V and E is a set of edges
- The set of edges E is a subset of $V \times V$. Consequently, $|E| \leq |V|^2$
- A path in G is a sequence of its vertices $(v_0, v_1, \ldots, v_n)$ such that
  1. $n \geq 1$, and
  2. for each $i \in [1, n-1]$, (v_i, v_{i+1}) is an edge in E
- n is called the length of the path.

Paths and Connectivity in Graphs

- Let G be a graph let s and t be a pair of its vertices.
- We say that s and t are connected in G if there exists a path $(v_0, v_1, \ldots, v_n)$ in G such that $s = v_0$ and $t = v_n$.
- We say that G is connected if each of its pairs of vertices is connected.

Path Connectivity in a Graph

- Assume that we are given a graph $G = (V, E)$ where E is represented by the relation `Edge(source,target)`
- In this relation, a pair (s, t) indicates that $(s, t) \in E$.
- We want to compute the connectedness relation

$$\texttt{Connected}(\textit{source}, \textit{target})$$

- In this relation, a pair (s, t) indicates that there is a path in G from vertex s to vertex t

Path Connectivity in a Graph

- We can define the (`Connected`) recursively using the following two rules:

  Rule 1: if `Edge(s,t)` then `connected(s,t)`
  Rule 2: if `connected(s,u)` and `Edge(u,t)` then `connected(s,t)`

- Rule 1 states that if there is an edge from `s` to `t` then `s` is connected to `t`
- Rule 2 states that if `s` is connected to a vertex `u` and if there is an edge from `u` to `t`, then `s` is connected to `t`

Path Connectivity in a Graph

- Notice that these are exactly the same rules as those for the Ancestor-Descendant relation!
- So if we replace in the previous program the `PC(P,C)` relation by `Edge(source,target)` and the relation `ANC(A,D)` by the relation `Connected(source,target)`, then we have program to compute the set of all pairs of vertices in G that are connected
- The relation `Connected` is often called the transitive closure of `G` and is denoted by `TC`.[2]

[2] We will assume that the scheme of `TC` is `(source,target)`.

Path Connectivity in a Graph (Transitive Closure)

```
CREATE OR REPLACE FUNCTION new_TC_pairs()
   RETURNS TABLE (source integer, target integer)AS
$$
BEGIN
   (SELECT TC.source, Edge.target
    FROM     TC INNER JOIN Edge ON (TC.target = Edge.source)
   EXCEPT
   (SELECT source, target
    FROM TC);
END;
$$ language plpgsql;
```

Path Connectivity in a Graph (Transitive Closure)

We can now specify a function that computes the `TC` relation

```
CREATE OR REPLACE FUNCTION TC()
  RETURNS void AS
$$
BEGIN
  DROP TABLE IF EXISTS TC;
  CREATE TABLE TC(source integer, target integer);

  INSERT INTO TC SELECT * FROM Edge;

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

Illustration

Edge

source	target
1	2
1	3
2	3
3	2
3	4
3	5
6	7
6	6
7	8
4	9
9	5

Initialization of TC (Phase 1)
Paths of length 1
$\rightarrow$

TC

source	target
1	2
1	3
2	3
3	2
3	4
3	5
6	7
6	6
7	8
4	9
9	5

Illustration

length 1 paths
TC

source	target
1	2
1	3
2	3
3	2
3	4
3	5
6	7
6	6
7	8
4	9
9	5

Phase 2
new pairs
$\rightarrow$

length 2 paths

new_TC_pairs

source	target
1	4
1	5
2	2
2	4
2	5
3	3
3	9
4	5
6	8

Phase 2
UNION
$\rightarrow$

TC

source	target
1	2
1	3
1	4
1	5
2	2
2	3
2	4
2	5
3	2
3	3
3	4
3	5
3	9
4	5
4	9
6	6
6	7
6	8
7	8
9	5

Illustration

length 1 and 2 paths

TC

source	target
1	2
1	3
1	4
1	5
2	2
2	3
2	4
2	5
3	2
3	3
3	4
3	5
3	9
4	5
4	9
6	6
6	7
6	8
7	8
9	5

Phase 3
new pairs
$\rightarrow$

length 3 paths

new_TC_pairs

source	target
1	9
2	9

Phase 3
UNION
$\rightarrow$

TC

source	target
1	2
1	3
1	4
1	5
1	9
2	2
2	3
2	4
2	5
2	9
3	2
3	3
3	4
3	5
3	9
4	5
4	9
6	6
6	7
6	8
7	8
9	5

Illustration

Edge

source	target
1	2
1	3
2	3
3	2
3	4
3	5
6	7
6	6
7	8
4	9
9	5

SELECT TC();
$\rightarrow$
SELECT * FROM TC;

TC

source	target
1	2
1	3
1	4
1	5
1	9
2	2
2	3
2	4
2	5
2	9
3	2
3	3
3	4
3	5
3	9
4	5
4	9
6	6
6	7
6	8
7	8
9	5

Transitive Closure (non-linear recursion)

- Observe that `connected` can also be specified by the following two rules:

Rule 1	if `Edge(s,t)` then `connected(s,t)`
Rule 2	if `connected(s,u)` and `connected(u,t)` then `connected(s,t)`

- Rule 1 states that if there is an edge in *E* from `s` to `t` then `s` and `t` are connected.
- Rule 2 states that if `s` is connected to a vertex `u` and if `u` is connected to `t`, then `s` is connected to `t`.

Transitive Closure (non-linear recursion)

- The nice aspect of this recursive definition is that $O(log(n))$ applications of these rules suffice to compute the transitive closure (i.e., the connectedness relation).
- Here n denotes the length of the longest path in the graph
- $O(n)$ number of applications may be required when we use the original rules for defining the connectedness relation

Path Connectivity in a Graph (Transitive Closure) Non-linear recursion

Notice how the new_TC_Pairs function uses non-linear recursion the FROM TC p1 INNER JOIN TC p2 clause

```
CREATE OR REPLACE FUNCTION new_TC_pairs()
  RETURNS TABLE (source integer, target integer)AS
$$
BEGIN
  (SELECT p1.source, p2.target
   FROM    TC p1 INNER JOIN TC p2 ON (p1.target=p2.source)
  EXCEPT
  (SELECT source, target
   FROM     TC);
END;
$$ language plpgsql;
```

Path Connectivity in a Graph (Transitive Closure) Non-linear version

The TC function remains the same

```
CREATE OR REPLACE FUNCTION TC()
  RETURNS void AS
$$
BEGIN
  DROP TABLE IF EXISTS TC;
  CREATE TABLE TC(source integer, target integer);

  INSERT INTO TC SELECT * FROM Edge;

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