INFO 610 Fall 2020

Week 13, Nov 10, 2020

Sample Data - Subquery

chrisfauerbach.github.io/info610_fall_2020/

Upcoming Dates

- Final Exam: 12/1 @ 5:30P, 2 hours allotted
- Final Project submission: 11/11

Functions with PL/pgSQL

We've briefly covered functions for triggers. That's one use for them.

Functions can be written in a variety of languages, today we'll focus on PL/pgSQL

Loadable Procedural language for Postgesql

- Can be used for functions/trigger procedures
- Add control structure
- Perform complex computtions
- All the data types we need, including user defined
- (ha!) easy to use

Structure of PL/pgSQL

PL/pgSQL is a block-structured language. The complete text of a function definition must be a block. A block is defined as:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

Each declaration and each statement within a block is terminated by a semicolon. A block that appears within another block must have a semicolon after END, as shown above; however the final END that concludes a function body does not require a semicolon.

Example

```
CREATE OR REPLACE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50

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

Note: There is actually a hidden "outer block" surrounding the body of any PL/pgSQL function. This block provides the declarations of the function's parameters (if any), as well as some special variables such as FOUND (see Section 39.5.5). The outer block is labeled with the function's name, meaning that parameters and special variables can be qualified with the function's name.

Variable Declaration

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

Examples defaults:

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

Function Parameters

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

- Function has a named parameter
- Function has a return type of 'real'

This is much preferred to the old/outdated style:

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

In and Out Parameters

Instead of a function returning a single value, we can name the return value to potentially make code easier to read for some.

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales AS s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

Rows and Tables

```
DROP TABLE IF EXISTS table2, table1;

CREATE  TABLE table2(id serial primary key,
   f3 varchar,   f7 varchar);
INSERT INTO table2(  f3,   f7) values ('a',  'c' );

CREATE  TABLE table1(id serial primary key, f1 varchar,
                    f5 varchar );
INSERT INTO table1(f1,  f5 ) values ('1', '2' );


CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 LIMIT 1;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t  ;
```

Control flow:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION getAllFoo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN SELECT * FROM foo
    WHERE fooid > 0
    LOOP
        -- can do some processing here
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
    RETURN;
END
$BODY$
LANGUAGE 'plpgsql' ;

SELECT * FROM getallfoo();
```

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is null
    result := 'NULL';
END IF;
```