Tutorial – PL/pgSQL (programming with psql)

Prerequisite: You must create and populate tables S, P, J and SPJ in order to test the programs in this tutorial. You may already have them in your account from lab4. If not, use scripts create_spj.sql and insert_spj.sql on the course webpage to create and populate them.

Basic unit in PL/pgSQL

A basic unit in all PL/pgSQL programs is a block. A block can be anonymous or named.

A PL/pgSQL block has the following structure (note that – indicates the beginning of a comment)

```
$$
DECLARE –optional
BEGIN
EXCEPTION  --optional
END
$$;
```

1. Connect to psql

a. Connect to linux.socs.uoguelph.ca. Change your folder to 3530 or whichever folder you want to save your contents in.
b. psql –h db
c. (Optional step) Assuming that the schema in which you work is called practice,
   SET search_path = practice;

2. Creating an anonymous block in PL/pgSQL:

Example1: Create an anonymous block that prints a message 'Welcome to 3530'.

```
DO
$$
BEGIN
     RAISE NOTICE 'Welcome to 3530';
END
$$;
```

Notes:

1. You may use command line to type the above block (or any block in general). But it is more convenient to use an editor and type the block in the editor. To use an editor, type \e on the psql prompt. When you save your work and exit the editor, the block runs and you get back the psql prompt.
2. If you prefer to save all work as a sql file, so that you can run it at a later time, do the following.
    a. Use \e to launch the editor, type the block and save your work as a file with an extension of sql (e.g. welcome.sql). This file is saved in your current folder (the folder from where you started psql –h db).
    b. To run this file, type \i welcome.sql
    c. To edit this file, type \e welcome.sql
3. RAISE NOTICE is used to print messages.

    For more on RAISE statement, refer to https://www.postgresql.org/docs/8.3/static/plpgsql-errors-and-messages.html

Example 2: SELECT statements in PL/pgSQL

Assume table Test has the following structure and rows. To create table Test, you may use the following command (provided you have created table S in your account): CREATE TABLE Test (SELECT * FROM S);

```
 sno | sname | status |  city
-----+-------+--------+--------
 S1  | SMITH |     20 | LONDON
 S2  | JONES |     10 | PARIS
 S3  | BLAKE |     30 | PARIS
 S4  | CLARK |     20 | LONDON
 S5  | ADAMS |     30 | ATHENS
(5 rows)
```

```
DO
$$
DECLARE
     vsname VARCHAR(30);
BEGIN
     SELECT sname INTO vsname FROM test;
     RAISE NOTICE 'Welcome to 3530 %', vsname;
END
$$;
```

1. \e (type in the editor)
2. Save the file as ex2.sql
3. Run the block as \i ex2.sql

Notes:

1. The syntax of SELECT inside a BEGIN – END block requires an additional keyword (INTO).
2. SELECT must return only 1 and only row – if SELECT returns more than one row – only the 1st one is used and discards the rest.

Example 3: SELECT with STRICT option

To ensure that SELECT inside a BEGIN END block returns only 1 row (it only makes sense for it to return 1 row – the other rows get discarded anyways), SELECT INTO must be used with the STRICT option.

```
DO
$$
DECLARE
    vsname VARCHAR(30);
BEGIN
    SELECT sname INTO STRICT vsname FROM test;
    RAISE NOTICE 'Welcome to 3530 %', vsname;
END
$$;
```

1. \e (type in the editor)
2. Save the file as ex3.sql
3. Run the block as \i ex3.sql

Notes:

SELECT INTO throws an error or exception if the SELECT returns more than 1 row or 0 rows. We can catch these exceptions – although this tutorial doesn't cover exceptions. To learn more on exceptions, please visit https://www.postgresql.org/docs/8.3/static/plpgsql-errors-and-messages.html.

Example 4: Cursors

Examples 2 and 3 demonstrate that SELECT used in the BEGIN END block handles only 1 row. To process 1 or more rows returned by an SQL, we use cursors or inline queries. An example of an inline query is shown on slide 23 of Week 9 lecture slides on our course webpage. This tutorial explains the creation and use of explicit cursors only.

Notes:

Typically, a cursor is

- declared in the declarative section of a PL/pgSQL block, by naming it and defining the structure of the query to be associated with it.

- OPENed in the begin-end block. The OPEN statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.
- Rows from the cursor are fetched -mostly inside a loop.
- CLOSED after all rows are processed

```
DO $$
DECLARE
      C1 CURSOR FOR SELECT * FROM Test;

      VC1 RECORD;
      result TEXT;

BEGIN
     OPEN C1;

     LOOP

          FETCH C1 INTO VC1;
          EXIT WHEN NOT FOUND;

          result :=   ' SNAME = ' || VC1. sname;

          RAISE NOTICE 'REsult is % ', result;

     END LOOP;
     close C1;
End;
```

A FOR LOOP can replace OPEN, FETCH, EXIT and CLOSE statements. The above block can be written using a FOR loop as shown below.

```
DO
$$
DECLARE
      C1 CURSOR FOR SELECT * FROM Test;
      VC1 RECORD;
      result TEXT;

BEGIN
    FOR vc1 in C1 LOOP
          result :=   ' SNAME = ' || VC1. sname;

          RAISE NOTICE '****** % ', result;
     END LOOP;
End;
$$;
```

1. \e (type in the editor)
2. Save the file as ex4.sql
3. Run the block as \i ex4.sql

Example 5: Cursor

Assume that there is a table called new_j1andj2 that has 1 attribute called sname. You can create this table using the following command: CREATE TABLE new_j1andj2 (sname varchar);

Question: Write a PL/pgSQL block using cursors that finds and displays names of suppliers who supply both J1 and J2. It also inserts these results into table new_j1andj2.

```
DECLARE
    /* find and displays names of suppliers who supply both J1 and J2.
    It also inserts these results into table new_j1andj2.
    */

    c1 CURSOR FOR SELECT sname
              FROM s NATURAL JOIN spj WHERE jno = 'J1'
              INTERSECT
              SELECT sname
              FROM s NATURAL JOIN spj WHERE jno ='J2';

    vc1 S.SNAME%TYPE;

BEGIN

    FOR i IN c1 LOOP

        RAISE NOTICE 'Supplier % supplies both J1 and J2', i.sname;
            INSERT INTO NEW_J1ANDJ2 VALUES(i.sname);
    END LOOP;
END
$$;
```

1. \e (type in the editor)
2. Save the file as ex5.sql
3. Run the block as \i ex5.sql

Example 6: Cursor with parameters

Cursors may take parameters as shown in this example.

Question: Write a PL/pgSQL block using cursors that finds and displays names of suppliers who supply to both J1 and J2. It then displays the parts supplied by those suppliers.

```
Sample Output:

NOTICE:  Supplier JONES supplies both J1 and J2
NOTICE:  Parts supplied by JONES
NOTICE:  P3    17.00
NOTICE:  P5    12.00
NOTICE:  Supplier BLAKE supplies both J1 and J2
NOTICE:  Parts supplied by BLAKE
NOTICE:  P3    17.00
NOTICE:  P4    14.00
```

```
DECLARE
    /* finds and displays names of suppliers who supply to both J1 and
    It then displays the parts supplied by those suppliers
    */

    c1 CURSOR FOR SELECT sname
                  FROM s NATURAL JOIN spj WHERE jno = 'J1'
                  INTERSECT
                  SELECT sname
                  FROM s NATURAL JOIN spj WHERE jno ='J2';

    vc1 S.SNAME%TYPE;

    c2 CURSOR (sn VARCHAR) FOR SELECT DISTINCT P.pno, P.color, p.weight
                               FROM S, P, SPJ
                               WHERE S.sno = SPJ.sno
                               AND   P.pno = SPJ.pno
                               AND   S.sname = sn;
BEGIN

    FOR i IN c1 LOOP
        RAISE NOTICE 'Supplier % supplies both J1 and J2', i.sname;

        RAISE NOTICE 'Parts supplied by %', i.sname;
        FOR n IN c2(i.sname) LOOP
            RAISE NOTICE '%    %', n.pno, n.weight ;
        END LOOP;
    END LOOP;
END
$$;
```

1. \e (type in the editor)
2. Save the file as ex6.sql
3. Run the block as \i ex6.sql

Part II – stored procedures and functions

Example1:

Write a plpgsql stored function that

a. finds and displays the supplier number and name of suppliers who supply all parts.
b. It also inserts their data into table *all_parts*.
c. Finally, display the projects that these suppliers work on.

Notes:

1. Both stored procedures and user-defined functions are created with CREATE FUNCTION statement. A function in plpgsql must include RETURNS keyword in its header and must include RETURN in the BEGIN END block. A stored procedure must include RETURNS void and may not include a RETURN keyword in its BEGIN END block.
2. It is recommended that you first write the SELECT statement for the question, run it on command-line mode (using psql) and convince yourself of the results before you put the SELECT in the function.
3. To create functions and compile them, I use the same method that I used for creating anonymous blocks in part I of tutorial 2.
    a. Open the editor \e
    b. Type your function header and block in the editor.
    c. Save it with an extension of sql (e.g. find_all_parts.sql)
    d. Compile the function by running the sql file \i find_all_parts.sql
    e. If it compiles with errors, then you can open the sql file using \e find_all_parts.sql , fix errors, save the file and recompile until free of errors. Once it is free of errors, it will display a message CREATE FUNCTION. This indicates that the function is now stored in the database for use.
    f. You can use \sf functionName at the psql prompt to see the function's code.

I tested my SQL query on the prompt – one that displays the number and name of those suppliers that supply all parts. And yes – it needs a divide ☺

```
SELECT sno, sname
FROM S
WHERE NOT EXISTS (SELECT * FROM p
                  WHERE NOT EXISTS (SELECT * FROM spj
                                    WHERE s.sno = spj.sno
                                    AND p.pno = spj.pno));
```

Result of this query is:

```
sno | sname
-----+-------
 S5  | ADAMS
(1 row)
```

Note that it is a coincidence that this SQL query returns only 1 row – but we may have instances where we may have more than 1 supplier in the result. Therefore, using a cursor makes sense.

Now use a different query for finding Adams's project numbers and names. We can get this info from table SPJ and J. Note that the cursor for this query will take n input parameter of S5 (in this case, Adam's supplier number).

```
CREATE OR REPLACE FUNCTION  all_parts()
RETURNS VOID
AS $$
DECLARE
    c1 CURSOR FOR   SELECT *
                    FROM S
                    WHERE NOT EXISTS (SELECT * FROM p
                    WHERE NOT EXISTS (SELECT * FROM spj
                    WHERE s.sno = spj.sno
                    AND p.pno = spj.pno));

    c2 CURSOR (snum S.sno%TYPE) FOR SELECT DISTINCT J.jno, jname
                        FROM J, SPJ
                        WHERE J.jno = SPJ.jno
                        AND   SPJ.sno = snum
                        ORDER BY j.jno;
BEGIN
    FOR vc1 IN c1 LOOP

        RAISE NOTICE 'Supplier %, % supplies all parts', vc1.sno, vc1.sname;
        INSERT INTO all_parts values (vc1.sno, vc1.sname, vc1.status, vc1.city);

        RAISE NOTICE '% works on the following projects', vc1.sname;
        FOR vc2 in c2(vc1.sno) LOOP
            RAISE NOTICE '%       %', vc2.jno, vc2.jname ;
        END LOOP;
    END LOOP;
END;
$$
LANGUAGE 'plpgsql';
```

After this procedure compiles successfully, it can be invoked with a SELECT such as

SELECT all_parts();

Sample Output:

```
NOTICE:  Supplier S5, ADAMS supplies all parts
NOTICE:  ADAMS works on the following projects
NOTICE:  J2      DISPLAY
NOTICE:  J4      CONSOLE
```

```
NOTICE:  J5      RAID
NOTICE:  J7      TAPE


 all_parts
-----------

(1 row)
```

Example 2:

Write a function called todays_date that returns the current date.

```
CREATE OR REPLACE FUNCTION todays_date()
 RETURNS date
AS $$
BEGIN
  RETURN current_date;
END
$$
LANGUAGE 'plpgsql';
```

Note the use of RETURN inside the BEGIN – END block.

After the function is compiled, it can be invoked as:

```
todays_date
-------------
 2017-11-24
(1 row)
```

Example 3:

Write a function called find_max_qty that returns the max quantity supplied by a supplier, given the supplier's number.

```
CREATE OR REPLACE FUNCTION find_max_qty(snum VARCHAR)
RETURNS INTEGER
AS $$
DECLARE
  mqty  INTEGER;

BEGIN
   SELECT MAX(qty)
   INTO mqty
   FROM spj
   WHERE sno = SNUM;

   RETURN mqty;
END
$$
LANGUAGE 'plpgsql'
```

Invoke it using SELECT:

SELECT find_max_qty('S4');

```
find_max_qty
--------------
      300
(1 row)
```

Part III: Triggers

Example 1:

```
-- Create the table log_transaction:

CREATE TABLE log_transaction (supp_no INTEGER, transaction_date  DATE, count_supp INTEGER);

-- Compile the function below before creating the trigegr on line 25

CREATE OR REPLACE FUNCTION log_count_supp()
RETURNS trigger AS
$$
DECLARE
    vcnt    INTEGER;
BEGIN
    SELECT count(*) INTO vcnt FROM S;

    INSERT INTO log_transaction
           VALUES(NEW.sno,current_date, vcnt);

    RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';


-- before insert trigger on S
CREATE TRIGGER before_insert_S
BEFORE INSERT ON S
FOR EACH ROW
EXECUTE PROCEDURE log_count_supp();

/*

if number of rows in table S is 6, then

insert into S (sno) values ('S9');

will insert 1 row in table S for 'S9' AND

will insert 1 row in table log_transaction

ritu=> select * from log_transaction;
 supp_no | transaction_date | count_supp
---------+------------------+------------
 S9      | 2017-11-21       |       7

 */
```