CS992: Database Development – Lab 2

SQL/PSM (Persistent Stored Modules) – PL/SQL

As you are now aware, PL/SQL is Oracle's implementation of the Persistent Stored Module mechanism within the SQL framework. We use PL/SQL when it is not possible to carry out certain actions within the declarative/'raw' SQL syntax. Due to the limits of laboratory exercises a number of the examples may in fact be actions that could (and probably should!) be carried out with pure SQL. The purpose of introducing them here is to illustrate the structure and execution of PL/SQL code, rather than elucidate the more complex situations in which their use may actually be justified (or indeed required).

We begin with some simple PL/SQL code (blocks), then build some stored procedures and functions, before finishing off with an example on the use of triggers.

1. Simple PL/SQL programs

1.a Let's create a very simple table to try out some coding.

> Create a table called "Lab2A" with two integer fields, "N1" and "N2" and a text field called "Comments".
>
> Insert the following rows into your Lab2A table:
>
> (3, 7, 'First')
> (13, 6, 'Second')
> (9, 17, 'Third')

1.b We can now use the code below to add an extra row into the table for the instance ('Second') in which the values in N1 and N2 are not in ascending order. As in the case of SQL code, you are best to enter the code into a text file with a ".sql" extension and run this from the SQLPlus prompt using the "@" command.

Note that after the "END;" of the block you need a "." on a line by itself, followed by the "run;" statement for this to run in response to a "@" call from SQLPlus.

```
/* A little program to enter a new row when the values in two integer
columns are not in ascending order */

DECLARE
    x NUMBER(10);
    y NUMBER(10);
      abc VARCHAR2(40);

BEGIN
    SELECT N1,N2,Comments INTO x,y,abc FROM Lab2A WHERE N1>N2;
    INSERT INTO Lab2A VALUES(y, x, CONCAT(abc,' - Altered entry'));
END;
.
run;
```

> You can use "select * from Lab2A" to see the effect of running this code…

1.c What about if we have multiple rows? (I 'cheated' a little as I knew that only one of the rows we entered met the condition.)

> Modify the code above to look for rows where you switch in the 'opposite' condition (i.e. if N1 and N2 are already in ascending order, enter a row that places in a reverse entry). What happened?

You should have received a "01422" error from Oracle, indicating that you are returning more than one row. This means that we need to use a **cursor**…

1.d However, before moving to cursors, how about revising your code to test whether only one row is returned (as happened in 1.b) before carrying out the INSERT command, and if this is not allowed then give the user a slightly more 'friendly' message?

> Alter the code from (1.c) to include a conditional IF THEN ELSE statement.
>
> Check the link below for the syntax (where you are also see how to use the DBMS_OUTPUT command to provide some feedback to the user).
>
> https://oracle-base.com/articles/misc/introduction-to-plsql#branching-and-conditional-control

2. Using cursors

2.a Now we will use a cursor to allow us to alter multiple rows at the same time. The code for this is a bit more 'involved' and some of the concepts were only touched on briefly in the lecture, so make sure that you understand what is going on in this block.

In actual fact your 'test' table (Lab2A) only has one row that satisfies the condition of being 'incorrect' so you may wish to insert a couple more rows with that characteristic – though the code will also work fine with just one record.

```
/* Program switches rows to be (N2, N1) for instances where N1 > N2 in the initial query. */
/* It does this by deleting any 'incorrect' row(s) and then inserting new 'correct' row(s). */

DECLARE
    a Lab2A.N1%TYPE;
    b Lab2A.N2%TYPE;
    xyz Lab2A.Comments%TYPE;

      /* Cursor declaration */
      CURSOR LabCursor IS
            SELECT N1, N2, Comments
            FROM Lab2A
            WHERE N1 > N2
            FOR UPDATE NOWAIT;
      /* Last line will lock the active set when the cursor is opened, and will abort if it cannot get a 'lock' (NOWAIT) */

BEGIN
      OPEN LabCursor;
```

```
	LOOP
		/* Get each row from the query defined in the cursor into the PL/SQL variables */
		FETCH LabCursor INTO a, b, xyz;
		EXIT WHEN LabCursor%NOTFOUND;
		/* If you didn't 'exit' then there is a row or take action on */
		DELETE FROM Lab2A WHERE CURRENT OF  LabCursor;
		INSERT INTO Lab2A VALUES (b, a, xyz);
	END LOOP;
	CLOSE LabCursor;
END;
.
run;
```

2.b Make sure that you have tested the code above where it is changing more than one row.

Enter a couple of rows with N1 and N2 being larger or smaller than each other but where the difference between them is less than 5 and re-run the code above.

2.c Alter the code from 2.a to do the following:

Enter new rows with 'switched' values where N2 is the larger of (N1, N2) except in the case where the difference between N1 and N2 is less than 5, in which case leave the row as is.

3. Stored Procedures and Functions

3.a The following procedure is not very 'interesting', we are simply taking a value and using that when we insert a new row.

```
/* Create a procedure to take a value and insert a row into Lab2A */
/* Each parameter is followed by a 'mode' and 'type' definition. The mode can be IN (read-only), OUT (write-only) or INOUT (both). */
/* Unlike type specifications in variable declarations, here the type must be unconstrained - i.e. VARCHAR2 rather than VARCHAR2(40) */
/* Here is have just one parameter of type NUMBER which is action as a read-only/input parameter */

CREATE PROCEDURE addRow(x IN NUMBER) AS
BEGIN
	INSERT INTO Lab2A VALUES (x, 99, 'Not very useful row');
END addRow;
.
run;
```

If you happen to make any mistakes and try to re-run your SQL file that contains the procedure definition you will get a "name already used by an existing object" error. One way to avoid this is to use the "CREATE OR REPLACE PROCEDURE" option.

3.b So once you have created your procedure (and made sure that it compiled correctly) how do you ***use*** it?! Well you could call it in a very simply program block such as this:

```
/* Simple call to the procedure you just created */

BEGIN addRow(123); END;
.
Run;
```

3.c Let's make your procedure a bit more useful:

> Create a new procedure – say "addRow2" – that accepts **three** parameters and allows you to insert a row with complete information into Lab2A.

3.d The main difference between a procedure and a **function** is that the latter MUST return a value. The format is:

```
CREATE FUNCTION <fctName>(<parameter_list>) RETURN <return_type> AS
```

```
/* Create a function to accept two parameters and return their sum. */

CREATE OR REPLACE FUNCTION mySum (n1 IN NUMBER, n2 IN NUMBER) RETURN NUMBER
AS
BEGIN
   RETURN N1*N2;
END mySum;
/

VARIABLE getResult NUMBER
BEGIN
  :getResult := mySum(5, 7);
END;
/

PRINT getResult
```

Here we have used a slightly different form to the ". run" option to avoid us having to create two separate files. We also call the "Print" function for the first time.

3.e You can find a list of all the procedures and functions you have created, use the following SQL query:

```
SELECT object_type, object_name
FROM user_objects
WHERE object_type = 'PROCEDURE'
   OR object_type = 'FUNCTION';
```

You can drop a stored procedure or function using:

```
DROP PROCEDURE <procedure_name>;
DROP FUNCTION <function_name>;
```

Try this with one of yours…. (you have the code so can always get it back!)

4. Triggers

As mentioned in class there is a fair bit of debate as to the value of triggers – particularly in an 'introductory' setting I feel that simply knowing that they can exist (and cause problems!) may be sufficient). Rather than waste time on these in the lab we will look a bit more at using functions and procedures in a more 'realistic' setting. (Of course you should feel free to read about and indeed try out the use of triggers.)

5. Use functions/procedures for some simple text retrieval tasks

Access to a table has been provided with the following structure:

```
Target (VARCAHR2)
DateSent (DATE)
Sender (VARCHAR2)
Message(VARCHAR2)
```

This is from a famous SMS 'spam challenge' dataset loaded from the UCI test collection. Each message has been marked (in the 'Target' field) to be either "spam" or "ham".

5.a We want to be able to enter a word and see whether it is present in the collection.

> Build a function that accepts a text input and then returns a value indicating whether or not this string was found in any of the 'Message' entries.

5.b Now let's get a little bit more ambitious and build a procedure that lets us know how often a word appears in the 'ham' and/or the 'spam' segments of our collection.

> You could use the function defined in [5.a] to pre-test whether it is even worth running the code within the procedure – i.e. if the word is not even in the collection then there is not much point in checking how often it appears in ham/spam.