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 (PSM) mechanism within the SQL framework. We use PL/SQL (more generally PSM) when it is not possible to carry out certain actions using only the declarative SQL syntax. Due to the limits of 'artificial' laboratory exercises, a few examples below 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) [1] and cursors [2], then build some stored procedures / functions [3 and 4].

In Lab 3 we will continue these approaches and word towards a more 'applied' scenario.

1. Simple PL/SQL programs

1.a Let's create a very simple table to use as part of our 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 can enter the code into a text file with a ".sql" extension and run this as a script, or you can enter the text below directly into the *SQL Worksheet* window.

```
/* 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;
```

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

Arguably this may not have been what we intended to happen – i.e. we have a new 'corrected' row but still have the old 'incorrectly ordered' row in the table. (I a more realistic setting we would likely use the "UPDATE" DML command rather than an INSERT.)

1.c What about if we had wanted to work with multiple rows? (I 'cheated' a little as I knew that only one of the rows in our table met the condition above.)

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

You should receive a "01422" error from Oracle, indicating that you are returning more than one row. To achieve this type of operation 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 We will now look at using 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 mini-lecture videos, so make sure that you understand what is going on in this code block.

You may see in the Oracle documentation a discussion of *implicit* and *explicit* cursors. When you run a SELECT statement, the returned data are placed into an *implicit* (kind of system-wide) cursor, even though none has been explicitly defined. In this situation an operation such as the FOR loop can be run on the data in that implicit cursor. However, if you plan to refer to the data returned in more than one place (i.e. to 're-use' the cursor) then it is better to explicitly define and name a cursor. Also we can use various 'cursor attributes' (such as %FOUND or %ROWCOUNT) when working with *explicit* cursors.

In actual fact your table (Lab2A) only has one row that satisfies the condition of being in the 'incorrect' order so you may wish to insert a couple more rows with this characteristic – though the code will also work fine even if it is operating only on one record (or indeed none!)

(This is actually pretty *BAD* coding, as in reality you would use an UPDATE command to achieve what we are doing here, using a single SQL statement!! But the whole point of this exercise is to illustrate the use of a cursor ; -)

```
/* 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 Lab2ACursor 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; it will abort if it cannot get a 'lock' (NOWAIT) */

BEGIN
      OPEN Lab2ACursor;
      LOOP
            /* Get each row from the query defined in the cursor into the PL/SQL variables */
            FETCH Lab2ACursor INTO a, b, xyz;
            EXIT WHEN Lab2ACursor%NOTFOUND;
            /* If you didn't 'exit' then there is a row to take action on */
            DELETE FROM Lab2A WHERE CURRENT OF Lab2ACursor;
            INSERT INTO Lab2A VALUES (b, a, CONCAT(xyz,' - now correct
order'));
      END LOOP;
      CLOSE Lab2ACursor;
END;
```

2.b Look at the table contents to make sure that your code from above appears to have changed all of the rows that you expected it to operate on.

> Enter a few new rows into your table with N1 being larger than N2, but where the difference between them is sometimes less than and sometimes more than 5.

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

> As before, generate alternative rows with 'switched' (ordered) values where N1 is the larger of (N1, N2) and delete the old/wrong row, except in cases 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 'useful', we have simply set up a procedure that takes a value and uses it when we are inserting 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) */
/* There is just one parameter of type NUMBER which is defined as a read-only ('IN') parameter */

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

You should get a "Procedure created" message in the SQL Worksheet window…

If you happened to make some mistake and try to re-run the code that contains the procedure definition you may get a "name is already used by an existing object" error. One way to avoid getting this error is to use the "CREATE OR REPLACE PROCEDURE" option.

3.b So once you have created your procedure (and made sure that it compiled without any errors) how do you ***use*** it?! Well, it exists within your current environment, so you could call it using a very simply program block such as this:

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

BEGIN addRow(123);
END;
```

> Call your procedure and check that the 'not very useful row' was added…
> (and you can of course call this procedure multiple times…)

3.c Now make your procedure a bit more useful:

> Create a new procedure – say "addRow_to_Lab2a" – that accepts **three** parameters and allows you to insert a row with complete information into Lab2A.

Assuming that this was correctly created you should be able to call as, for example:

```
BEGIN addRow_to_Lab2a (5, 10, 'five and its double');
END;
```

Hopefully this also begins to provide a bit of a practical example as to how a 'Thick DB' approach could operate… i.e. in these examples we could obviously simply have used the "INSERT" command directly – or have used that SQL syntax in some external program that had been given write access permission to the tables. However, if we restricted adding rows to our Lab2a table through only a set of defined procedure calls (such as those illustrated here), then we could carry out a range of quality/validity checks before the data were entered into the actual tables.

You may have heard of "SQL injection" which is a way to place malicious code into SQL statements before execution. While there are many ways to tackle the potential problems of SQL injection, running various 'pre-checks' (such as using insertion procedures like those introduced here in the *Thick DB* model) before actually sending the SQL statement to the database engine for execution are among the most useful.

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. Then use this function. */

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


DECLARE
    getResult NUMBER(10);

BEGIN
  getResult := mySum(5, 7);
  dbms_output.put_line('Result of summation was: ' || getResult);
END;
```

Here we have used a slightly different form in the script to "run" (/) the function definition code first and then the second block of code that calls the function. Once the function exists in a given context then of course we can use it at any point from the time after which it has been successfully created.

3.e You can find a list of <u>all</u> the procedures and functions you have created and are available in a given context, by using the following SQL query:

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

This is of course the same as the syntax that you may already have been using to obtain a list of other objects in the Oracle environment (e.g. Tables or Views). And, as is the case for other object types, you can drop a stored procedure or function by using:

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

Try this with one of your procedures… (I'm confident you will have saved your code, so you can always re-create it again if you need too!)

4. Using functions in queries

4.a Write a function that will apply VAT (a value defined within the function) to a cost and return the price to be charged to the customer (rounded to 2 decimal places). Use this function in a query with the `Products` table in the `Customer Orders` schema (aliased as "`CO`") within LiveSQL. i.e. If your function name was **`Price_with_VAT`** then you would be able to carry out the query below:

```
SELECT Product_ID, Product_Name, Unit_Price, Price_with_VAT(Unit_Price)
AS Price_to_Customer
FROM CO.Products;
```

Which should give a table in the following format:

PRODUCT_ID	PRODUCT_NAME	UNIT_PRICE	PRICE_TO_CUSTOMER
36	Women's Trousers (Blue)	29.51	35.41
37	Boy's Jeans (Blue)	22.98	27.58

4.b Write a function to check if a given year is a leap-year, you can ignore the 100/400 'special' case if that seems a bit complex (i.e. just assume that all you need to do is to divide by 4). As in the example given in the mini-lecture, the best way to do this (in my view) is to have a function that returns a Boolean value. Once you have written your function, check that it works for various submitted years.

Now with the `OLYM` schema in *LiveSQL* and use your function to check whether any of the summer game (`OLYM_Games` table) were held in years that were NOT leap-years. You will encounter a rather annoying fact, that ORACLE (and indeed the ANSI standard) do not allow columns in a database to be of type Boolean, and as such you cannot use functions that return a Boolean in SELECT or WHERE clauses of a query. So, to get your leap-year function to work you will need to alter it to return a NUMBER (say 0 or 1), or you may prefer a single character (i.e. "F" or "T"). You will then be able to use your function to look for leap-year or non leap-year entries in the `OLYM_Games` table. In my example below, I have used a 0/1 'number' version of a year-leap function. (Mine also uses the full definition of a leap-year; if you have simply check for division by 4 then you will get NO records, as 1900 is divisible by 4 even though it was not a leap-year.)

```
1  SELECT *
2  FROM OLYM.Olym_Games
3  WHERE fLeapYear2(Year) = 0;
```

ID	YEAR	SEASON	CITY
2	1900	Summer	Paris