CS992: Database Development – Lab 3

Making use of PL/SQL

Last week you were introduced to using SQL Persistent Stored Modules (in the form of Oracle's PL/SQL), including: blocks, cursors, procedures and functions. This week we will continue to use these approaches and will also outline some text retrieval tasks that you should attempt within your groups for submission.

A few general points to note:

(a) If your group has not completed the exercises from Week 2, you should complete those perhaps looking at the partial solutions on the MyPlace *Forum* page for hints;

(b) There are a number of points in this week's lab at which the word ***SUBMIT*** appears. Please generate a clearly labelled text file (e.g. "*Group_X_Lab_3.sql*") which includes the code that you have used for each part of the submission. You may wish to include a few comments within that text file to document its layout and/or any parts of the code that require explanation (e.g. how a particular function works, or even does not quite seem to work!);

(c) In the 'real world' you would likely tackle some of the tasks covered in this lab thought the use of specialised software/libraries. As such the exercises may seem a little 'unrealistic' but they are designed to allow you to increase your understanding of PL/SQL.

1. Practice your use of functions

These initial two examples do not require you to use SQL (or *cursors*, etc.), they are simply to get you into the frame of mind around creating functions and procedures.

1.a Create a very simple PL/SQL function (called "input_reversed") that takes an input string and outputs that same string but in reverse order.

i.e. If you had a variable named txt_in := "abcde" then when you call –

```
input_reversed(txt_in)
```

the value returned should be "edcba".

It may be useful to know that you can use the "`IN REVERSE`" order when looping through a `FOR` statement, and you should also probably look up the `SUBSTR` function.

SUBMIT: Please include your solution to **1.a** as part of this lab's submission.

1.b This second task is a bit more challenging in that it involves building a *recursive* function – or at least that is the most efficient way to build this function. (I'm guessing that you may already have done something like this in, say, Python as this is one of the 'classic' recursion examples… ***Hint:*** you should find some pseudo-code using a quick *Google* search ; -)

The task is to create a function that return the ***factorial*** of a number, which you may remember from high school maths is the product of all the integers from the given number down to 1. (That is important, if you go all the way to 0 then the factorial of any number would be zero!!) The sign in maths for factorial is "!", thus:

3! = 3 * 2 * 1 = 6
7! = 5045
1! = 1

Create a function (called "my_factorial") that takes a number and returns the factorial of that number. (You could of course do this using a 'for' loop, but I would like you to think about how you might write this using ***recursion***. A 'recursive' function is one which calls itself – so within your recursive version of "my_factorial" there will in fact be a call to "my_factorial". I realise that if you have not come across this idea before it takes a minute to get your head around it!!)

SUBMIT: Please include your solution to **1.b** (whether or not you managed to do this using recursion) as part of this lab's submission.

2. Generate some tables for text retrieval

A few scripts have been provided (on *MyPlace*) that will create a table ('SpamHam') with the structure shown below and populate it with data (initially just 150 records):

```
In_Ord (NUMBER, NOT NULL)
Target (VARCAHR2)
DateSent (VARCHAR2)
TimeSent (VARCHAR2)
Sender (VARCHAR2)
Message(VARCHAR2)
```

These data come from the SMS 'spam challenge' dataset in the UCI test collection. Each '`Message`' has been categorised (in the '`Target`' field) to be either of type "spam" or "ham". There are in fact two sets of 'matching' scripts, one generates a table with 150 rows, while the other generates 2,500 rows (there are over 5,000 rows in the original UCI test set) into a table called 'SpamHam2'. Students have found that *Live SQL* operates poorly on the larger data set so I would suggest (at least initially) you start with the smaller set.

A script is also provided to generate a *stop-word* list, which will be needed in 3.d below. You can either run this now or wait until you get to that part of the exercise.

3. Use functions/procedures for some text retrieval tasks

A few of the tasks below may be able to be solved using some sort of basic SQL DML query, but most of them require the creation of a procedure or function. When it comes to tackling 3.d you may wish to create additional tables or perhaps use views. Think about how you might go about the tasks in [3.d] and feel free to post questions to the 'chat' area or *Forum* if a particular approach is causing challenges.

As in Section 1 above, you are requested to ***SUBMIT*** various aspects of your solutions.

3.a We want to be able to enter a text string and see whether it is **present** in **any** of the messages in this collection.

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

SUBMIT: Please include your solution to **3.a** as part of this week's submission.

3.b Now be a bit more ambitious and build a procedure that lets you know **how many times a specific word appears** in the 'ham' and/or the 'spam' segments of the collection.

> You could use the function defined in [3.a] to pre-test whether it is even worth running the code within this procedure – i.e. if the word is not in the collection then there is not much point in checking how often it appears as ham or spam.

SUBMIT: Please include your solution to **3.b** as part of this week's submission.

3.c See whether there are any 'interesting' patterns between the 'sender' of a message and the number of ham/spam messages that she/he has sent.

> You may wish to tackle this simply using some SQL queries with some grouping functions, or you may find that it is easier to place these queries within a procedural setting to explore a variety of combinations between senders and their ham/spam messaging.

SUBMIT: Please provide some notes and any code used as part of **3.c** observations.

3.d It would be interesting to see what the **most frequently occurring** words (say the 'top 10') are in both the *ham* and *spam* categories. Think about a method that might be used to do this? You may wish to create some new tables (or views) to work on this problem – this may involve you in working with a cursor and running through the original table. There are a few things that you might want to think about in this context:

- What is a 'word'? In text retrieval we would normally talk about a 'token' which would involve doing things such as **regularising** words. You may also need to decide what to do with punctuation and other 'odd' characters.
- Had you already considered **lower** and **upper** case in the tasks above? (Perhaps, even in your initial 3.a function? If not, you should!)
- In most text retrieval contexts, you would typically have some sort of **'stop-word' list**, i.e. common words such as "the", "a", "in", etc. that you are not really interested in. These words would typically <u>not</u> be recorded in any frequency analyses, as they are not particularly 'interesting' and will likely appear fairly often in <u>both</u> the *spam* and *ham* messages. (A script has also been given to create a simple stop-word list tables that you can use as part of your solution to this task.)

SUBMIT: Please include all elements that you have explored to provide some interesting summaries of frequently occurring words as part of this **3.d** exercise. In contrast to the other parts of this week's submission, you will probably need to include some comments with your code to explain how you have gone about these tasks.