RMIT UNIVERSITY

School of Computing Technologies

ISYS1101/ 1102 Database Applications

Week 6: Tute/Lab – Stored Procedures

Semester 2 2023

1 Objective

This week we introduce a new programming language – PL/SQL. PL/SQL is Oracle's *procedural extension to SQL*. *Procedural* refers to a series of ordered steps that the computer should follow to produce a result. Programs written in such a language use its sequential, conditional, and iterative constructs to express algorithms. In that sense, PL/SQL is in the same family of languages as Python, C, or PHP.

A language extension is a set of features that somehow enhance an existing language. In that sense, it implies that PL/SQL can be considered as an enhanced version of SQL. It is not exactly correct. PL/SQL is a programming language in its own right; it has its own syntax, its own rules, and its own compiler. You can write PL/SQL programs with or without any SQL statements. However, practically in most of the situations, you use PL/SQL for database programming by means of embedding SQL statements.

In this lab session, we introduce building blocks of a PL/SQL program using a set of small activities. In later activities, we will develop more complete PL/SQL blocks and will discuss more advanced concepts.

2 Preparation Tasks

Most of these activities are based on the (small) Movies database we used in Week 1 revision lab activities. If you haven't installed it in your Oracle account, you must get it built and populated using the sample data provided on Canvas.

Complete this section ONLY IF YOU HAVEN'T BUILT the (small) Movies database yet!

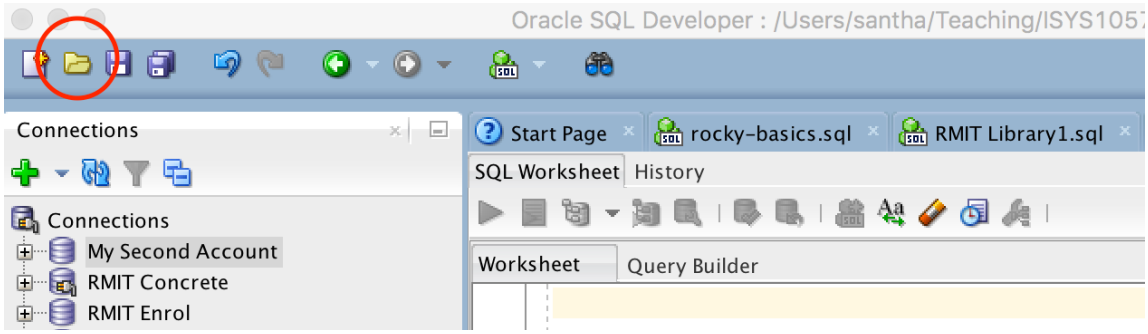
Log in to your Canvas shell, go into "Course Resources" → "Sample Databases" → "Movies Database" area. There are a bunch of SQL script files in that folder.
Download them into a folder in your computer (or a folder on MyDesktop).

Your Lab assistant will demonstrate you how to copy files between your desktop computer and MyDesktop.

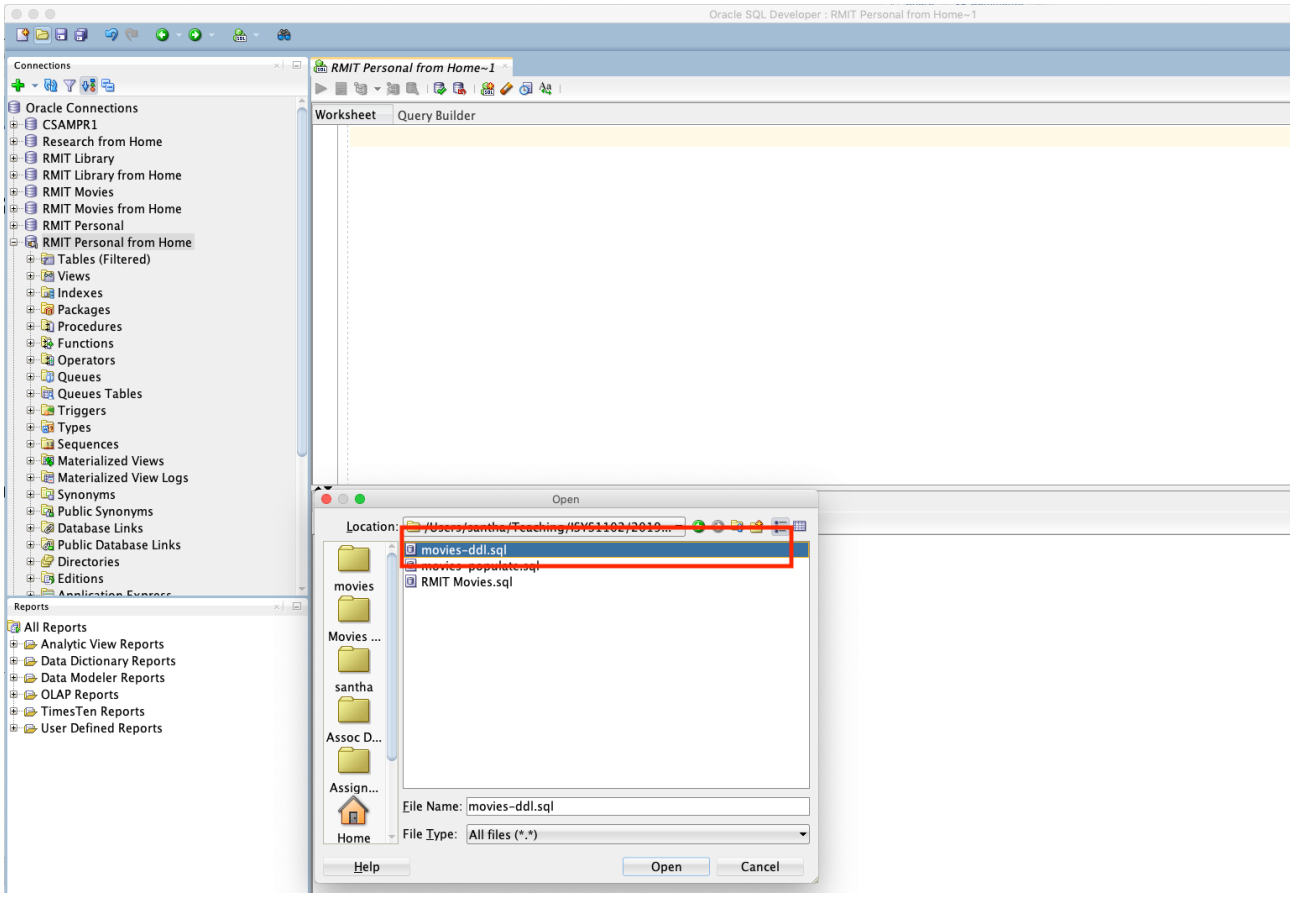
In this activity, you familiarize yourself with running a SQL DDL script in Oracle SQL Developer to create following four tables.

- Movie
- Director
- Star
- Movstar
- Member
- Borrow

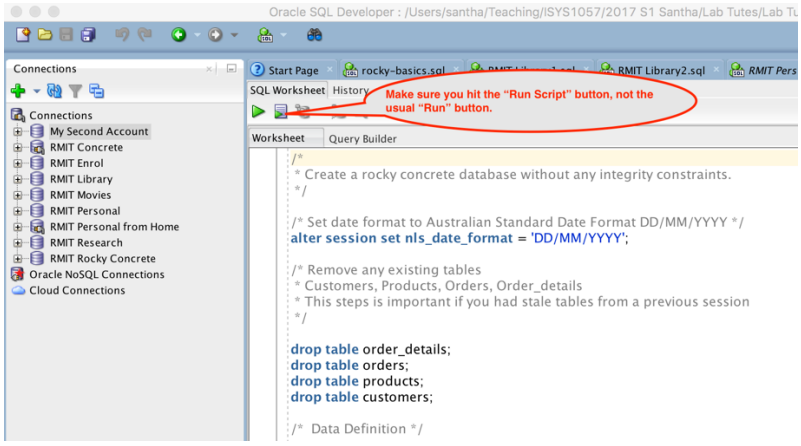
On Oracle SQL Developer, Click on “Open Script” button as shown on the following screenshot, or go to File → Open.



On Open Script dialog window, choose the “movies-ddl.sql” file.



Once the script appears on your SQL Editor, hit the “Run Script” button on the top. Do not use “Run” button this time.



To populate tables in the Movies database, follow these steps.

On Oracle SQL Developer, click on “Open Script” button as shown above, or go to File → Open.

On Open Script dialog window, choose the “movies-populate.sql” file. Then, run the script in tha same as above.

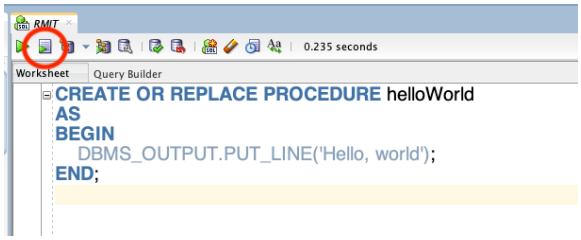
3 Tutorial Activities on PL/SQL

3.1 Running your first PL/SQL program

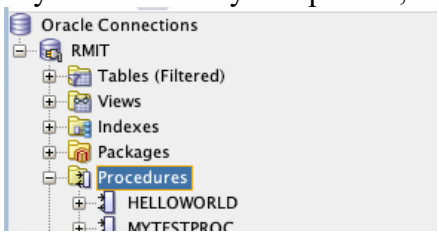
To partake in a grand tradition of beginning programmers, the first program to write in a new language will merely print out the message "Hello, world". PL/SQL can display this archetypal greeting with only four lines of code:

```
CREATE OR REPLACE PROCEDURE helloWorld
AS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello, world');
END;
```

Enter the above on SQL Developer and compiled it, using “Run Script” option on the Tool Bar.



If you successfully compiled it, it should appear on your left-hand pane, under "Procedures".



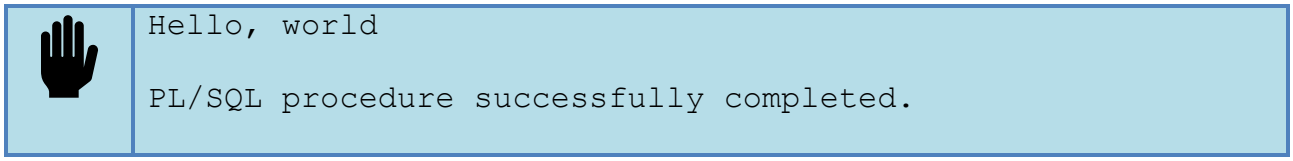
To run a stored procedure from SQL Developer, you should do two things:

1. Direct the output of the procedure to the SQL Developer results pane;
2. Execute the procedure.

Enter the following two commands:

```
SET SERVEROUTPUT ON
EXECUTE helloWorld;
```

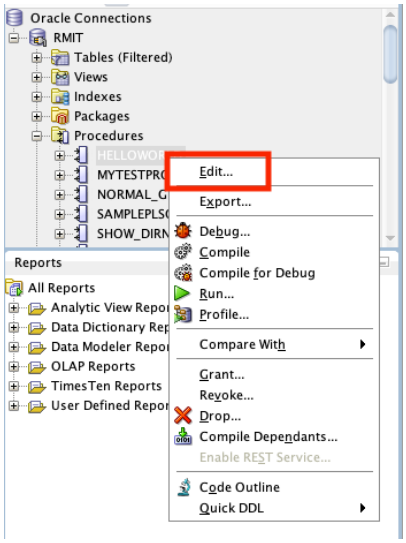
If successful, theis result will appear as follows:



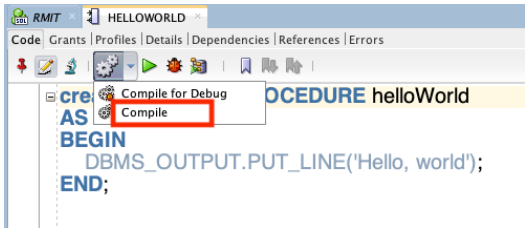
```
Hello, world

PL/SQL procedure successfully completed.
```

Now, it is the time to read the PL/SQL Introduction tutorial or other suitable PL/SQL reference to find out other interesting features in the PL/SQL language.

If you require further editing with your PL/SQL Stored Procedure, you can use "Edit" option. Right-click on the procedure name and choose "Edit".



A new edit window appears with the current contents of the procedure. After making the required changes, you can re-compile it using "Compile" option on the tool bar.



After the recompilation, you may try it again using the same "EXECUTE helloWorld;" command on the main SQL Editor pane.

The following activities introduce you a few of very useful features in PL/SQL.

3.2 Using Variables in a PL/SQL program

PL/SQL supports a wide variety of data types. Read the above tutorial to investigate more on PL/SQL data types. In this activity, we declare a few variables in the DECLARE section of a PL/SQL block and manipulate the declared variable within the body of the block.

```
CREATE OR REPLACE PROCEDURE updateMarks
AS
      name CHAR(40);
      marks INT;
BEGIN
      name := 'Intro to Database Systems';
      marks := 48;
      DBMS_OUTPUT.PUT_LINE(name);
      DBMS_OUTPUT.PUT_LINE(marks);
      -- Adjust marks by 2 points
```

```
        -- Note the assignment operator in PL/SQL
        -- (= compares two operands for equalancy ,
        -- := assigns a value to a variable)
        marks := marks + 2;
        DBMS_OUTPUT.PUT_LINE('Marks after scaling up');
        DBMS_OUTPUT.PUT_LINE(marks);
END;
```

Compile and execute the above stored procedure.

Did you get the expected output? Now try yourself with other data types.

3.3 Embedding a simple SELECT statement

As we mentioned at the beginning of the discussion, one of the mostly common usage of a PL/SQL program is to embed one or more SQL statement(s) and execute them procedurally. PL/SQL allows us to declare variables and use them as vehicles to input and output data to and from SQL queries.

The following example demonstrates how we can embed an SQL statement, pass some values into it, execute it, and store the results in another variable.

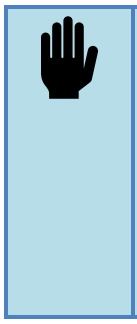
```
CREATE OR REPLACE PROCEDURE displayDirector
AS
     director_name CHAR(20);
     movie_name CHAR(20);
BEGIN
     movie_name := 'North by Northwest';
     SELECT dirname INTO director_name
           FROM movie, director
           WHERE movie.dirnumb = director.dirnumb AND
                              movie.mvtitle = movie_name;
      DBMS_OUTPUT.put_line('The director of the movie is:');
      DBMS_OUTPUT.put_line(director_name);
END;
```

Note that, the special INTO clause in the SQL statement, which is used to assign the result into a PL/SQL variable.

We can embed SQL statements in this way, under some specific conditions. The SQL statement produces only one tuple as its output. If it produces more than one row, we have to use a different technique, called a cursor. We discuss the declaration and the use of cursor to fetch results from a SQL statement in the last activity.

In the above program, we have hard-coded the variable types for the movie_name and director variables. However, this is somewhat dangerous programming practice. If you are unsure about the type attributes you deal with, the associated variables can be defined used the type of the corresponding attribute (in the database). The following example demonstrates how movie_name

and director_name variables can be defined using the corresponding attributes: movie.mvtitle and director.dirname.

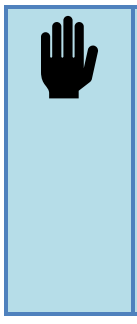


```
CREATE OR REPLACE PROCEDURE displayDirector
AS
      --director_name CHAR(20);
     --movie_name CHAR(20);
     director_name director.dirname%TYPE;
     movie_name movie.mntitle%TYPE;

BEGIN
...
```

3.4 Input Parameters

In the previous exercises, we have had the movie title (North by Northwest) hard-coded into the program. You can make your program much more versatile, by allowing the user to enter the movie title that they wish to retrieve other information.

That can be achieved by including an input parameter. So, the user, when they execute the procedure, will feed in the value for the input parament.



```
CREATE OR REPLACE PROCEDURE displayDirector
                              (movie_name IN movie.mvtitle%type)

AS
     director_name director.dirname%TYPE;
     movie_name movie.mntitle%TYPE; -- not required

BEGIN
...
```

Run the Execute command as follows:

```
SET SERVEROUTPUT ON
EXECUTE displayDirector('North by Northwest');
```

3.5 Conditional logic – if statement

Almost every piece of code you write will require conditional control: the ability to direct the flow of execution of your program based on a condition.

For example, you may want to implement requirements such as:

- if salary is between 10000 and 20000, then apply a bonus of $1500
- if salary is between 20000 and 40000, then apply a bonus of $1500
- if the salary is over $40000, give the employee a bonus of $500.

The IF-THEN-ELSE statement in PL/SQL allows you to do such branching.

The general syntax of IF-THEN-ELSE statement is given below:

IF condition
    THEN
        statements to execute when condition is TRUE;
    ELSE
        statements to execute when condition is FALSE;
    END IF;

The following example demonstrates the use of IF statement to determine a suitable comment on a movie, retrieved from the movie relation.

```
CREATE OR REPLACE PROCEDURE displayDirectorWithComment
AS
    director_name director.dirname%type;
    movie_name movie.mvtitle%type;
    awards INT;
BEGIN
      movie_name := 'North by Northwest';
      SELECT dirname,awrd INTO director_name,awards
            FROM movie, director
            WHERE movie.dirnumb = director.dirnumb AND
                  movie.mvtitle = movie_name;
      DBMS_OUTPUT.put_line('The director of the movie is:');
      DBMS_OUTPUT.put_line(director_name);

      IF (awards>=4) THEN
            DBMS_OUTPUT.put_line('It is an exceptional movie.');
      ELSE
            DBMS_OUTPUT.put_line('It is not an exceptional movie.');
      END IF;
END;
```

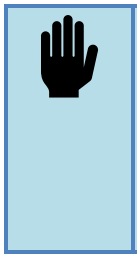
3.6 Executing in circles – for loop

There are two kinds of PL/SQL FOR loops: the numeric FOR loop and the cursor FOR loop. The numeric FOR loop is the traditional and familiar *counted* loop. The number of iterations is specified in the loop's range scheme. The general syntax of the numeric FOR loop is given below.

FOR loop_index_variable IN [REVERSE] lowest_value .. highest_value
    LOOP
        executable statements;
    END LOOP;

REVERSE is an optional keyword, used to generate descending values.

The following example demonstrates how a FOR loop can be used in a PL/SQL program to generate a sequence of numbers between 1 & 10.



```
…
num INT;
BEGIN
    FOR num IN 1 .. 10
    LOOP
        DBMS_OUTPUT.PUT_LINE('The current value of num is:' || num);
    END LOOP;
END;
```

FOR loops can be more effectively used with *cursors*. We'll discuss use for FOR loops to fetch data from a cursor later.

3.7 Executing in circles – simple (unconditional) loop

The structure of the simple loop is the most basic of all the loop constructs. It consists of the LOOP keyword, the body of executable code, and the END LOOP keywords, as shown below:

```
LOOP
        executable statements;
END LOOP;
```

Be very careful when you use simple loops. Make sure they always have a way to stop. Top force a simple loop to stop processing, execute an EXIT or EXIT WHEN statement within the body of the loop. The following example demonstrates how EXIT WHEN is used within a simple loop.

```
…
num INT;
BEGIN
    LOOP
    EXIT WHEN num > 10;
        DBMS_OUTPUT.PUT_LINE('The current value of num is:' || num);
        num := num +1;
    END LOOP;
END;
```

3.8 Executing in circles – while loop

The WHILE loop is a conditional loop that continues to execute as long as the boolean condition defined in the loop evaluates to TRUE. Because the WHILE loop execution depends on a condition and is not fixed, use a WHILE loop if you don't know ahead of time the number of iterations a loop must execute.

The general syntax of the WHILE loop is:

```
WHILE condition
LOOP
        executable statements;
END LOOP;
```

The following simple example demonstrates how you can use a WHILE loop to achive the same result as in the previous activity.

```
…
num INT;
BEGIN
     num :=1;
     WHILE (num<=10)
     LOOP
          DBMS_OUTPUT.PUT_LINE('The current value of num is:' || num);
          num := num +1;
     END LOOP;
END;
```

3.9 Defining a cursor and fetching data from a table

A cursor is another type of variable in PL/SQL. Usually when you think of a variable, a single value comes to mind. A cursor is a variable that points to a row of data from the results of a query. In a multiple-row result set, you need a way to scroll through each line to analyse the data. A cursor is used for just that. When you work with a cursor, there are a few things that you should do in a specific order.

Declaring a cursor
Similar to any other variable, a cursor is declared in the DECLARE section. The general syntax of a cursor declaration is given below:

```
CURSOR cursor_name AS
          sql_query;
```

Opening a cursor
Now that you have defined your cursor, you have to open it before using it to fetch data. OPEN command is used to open a cursor. General syntax of a cursor OPEN command is given below:

```
OPEN cursor_name;
```

You open the cursor in the body of the PL/SQL block.

Fetching data from a cursor
FETCH command executes a SQL statement associated with a cursor, and populates the cursor with data. The general syntax of the FETCH command is given below.

```
LOOP
     FETCH cursor_name INTO variable;
     EXIT WHEN cursor_name%NOTFOUND;
     statements to do anything with fetched data;
END LOOP;
```

Closing a cursor
When you have finished using a cursor in a block, you should close the cursor, using CLOSE command. General syntax of a cursor close statement is given below:

CLOSE cursor_name;

So, you have all the required procedures to play around with a 'cursor'. The following example demonstrates how a cursor can be used to fetch data from a multiple-row SQL query.

```
CREATE OR REPLACE PROCEDURE show_movie_title_list
                            (director_name IN director.dirname%type)
AS

CURSOR mv_cursor IS
       SELECT m.mvtitle -- there is no INTO clause1
           FROM movie m JOIN director d ON m.dirnumb = d.dirnumb
       WHERE lower(d.dirname) = lower(director_name);

l_movie_name movie.mvtitle%type;

BEGIN

DBMS_OUTPUT.put_line('The movies directed by ' || director_name);
OPEN mv_cursor;

LOOP
    FETCH mv_cursor INTO l_movie_name;
    EXIT WHEN mv_cursor%NOTFOUND;

    DBMS_OUTPUT.put_line(l_movie_name);
END LOOP;

CLOSE mv_cursor;
END;
```

A more flexible version.
In the following version, a slight modification is made to the way how data is fetched from the cursor.
A local variable is defined using the cursor itself. So, this new variable can hold one whole row fetched from the cursor. It will contain all the column values within that row. After the row is fetched, you can extract each of the column value from that variable.

```
CREATE OR REPLACE PROCEDURE show_movie_title_list
                            (director_name IN director.dirname%type)
AS

CURSOR mv_cursor IS
       SELECT * -- there is no INTO clause1
           FROM movie m JOIN director d ON m.dirnumb = d.dirnumb
       WHERE lower(d.dirname) = lower(director_name);

l_movie mv_cursor%ROWTYPE;

BEGIN

DBMS_OUTPUT.put_line('The movies directed by ' || director_name);
```

```
OPEN mv_cursor;

LOOP
    FETCH mv_cursor INTO l_movie;
    EXIT WHEN mv_cursor%NOTFOUND;

    DBMS_OUTPUT.put_line(l_movie.mvtitle);
END LOOP;

CLOSE mv_cursor;
END;
```

You can ask why we should go into such a great lengths when you can achieve the above result, just using a simple SQL statement. That's just a simple example to demonstrate how we use cursors in PL/SQL.

The next example shows how we can use a PL/SQL block to achieve some thing that's impossible to obtain from a simple SQL statement.

For each of the movie in the database where director is still alive, show the title, director's name, and a list of stars played!

```
CREATE OR REPLACE PROCEDURE show_movie_title_and_stars
                            (director_name IN director.dirname%type)
AS
        CURSOR mv_cursor IS SELECT m.mvnumb, m.mvtitle
            FROM movie m JOIN director d ON m.dirnumb = d.dirnumb
            WHERE lower(d.dirname) = lower(director_name);

        l_movie mv_cursor%ROWTYPE;

        CURSOR star_cursor IS SELECT *
            FROM movstar ms JOIN star s ON ms.starnumb = s.starnumb
            WHERE ms.mvnumb = l_movie.mvnumb;
        l_star star_cursor%ROWTYPE;

BEGIN
        OPEN mv_cursor;
        LOOP
            FETCH mv_cursor INTO l_movie;
            EXIT WHEN mv_cursor%NOTFOUND;

            -- DBMS_OUTPUT.put_line('The title of the movie is:');
            DBMS_OUTPUT.put_line(l_movie.mvtitle);
            DBMS_OUTPUT.put_line('Stars');

            OPEN star_cursor;
            LOOP
                FETCH star_cursor INTO l_star;
                EXIT WHEN star_cursor%NOTFOUND;
                      DBMS_OUTPUT.put_line('    ' || l_star.starname);

            END LOOP;
            CLOSE star_cursor;
```

```
        END LOOP;
        CLOSE mv_cursor;

EXCEPTION

      WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('The query did not return a result set');

END;
```

This is a sort of actions that you can't do with just SQL.

3.10 Using Update Cursors

During the last lab activity, we declared a SELECT cursor and fetch tuples using the declared cursor. However, it merely retrieves tuples, and do not allow you to make changes to the relations.

Very often, you require modifying the fetched tuples as a part of the whole process. For example, you retrieve the movie tuple for the movie titled, '2001', and update its mvtitle as '2001:The Space Oddysey'. PL/SQL provides a convenient syntax for doing this. This method consists of two parts: the FOR UPDATE clause in the cursor declaration and WHERE CURRENT OF clause in an UPDATE or DELETE statement.

FOR UPDATE clause
The FOR UPDATE clause is part of a SELECT statement. It is legal as the last clause of the statement, after the ORDER BY clause (it exists). The syntax is:

SELECT ... FROM ... FOR UPDATE [OF column_names]

Where column_names is a reference to the columns on which update query is performed. (This is optional). For example, the following declarative section defines two cursors that are both legal forms of the FOR UPDATE syntax.

```
DECLARE
        CURSOR c_movie IS
                SELECT *
                        FROM movie
                        WHERE mvtitle = '2001'
                        FOR UPDATE OF mvtitle;

        CURSOR c_member IS
                SELECT *
                        FROM member
                        WHERE mmbst = 'VIC'
                        FOR UPDATE;
```

WHERE CURRENT OF clause

If the cursor is declared with the FOR UPDATE clause, the WHERE CURRENT OF clause can be used in an UPDATE or DELETE statement. The syntax for this clause is:

WHERE CURRENT OF cursor_name

Where cursor_name is the name of the cursor that has been declared with the FOR UPDATE clause.

For example, the following code segments will double the bonus rentals for each member.

```
CREATE OR REPLACE PROCEDURE updateMemberBonus
AS

    CURSOR member_data IS
        SELECT bonus
            FROM member
            FOR UPDATE OF bonus;

    old_bonus INTEGER;
    new_bonus INTEGER;

BEGIN
    OPEN member_data;
    LOOP
        FETCH member_data INTO old_bonus;
        EXIT WHEN member_data%NOTFOUND;

        new_bonus := old_bonus * 2;

        UPDATE member
            SET bonus = new_bonus
            WHERE CURRENT OF member_data;
    END LOOP;
    CLOSE member_data;
END;
```

Exercise:

- Write a PL/SQL program to change the 'mmbcty' of the members who live in 'Hudson' to 'Ann Arbor'.