LAB BOOK 5

COMP41090 SQL PROGRAMMING

ANDREW DOYLE
STUDENT NUMBER: 12252388
MSc COMPUTER SCIENCE (CONVERSION)

23rd April 2013

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

This introduction serves sets out the format of the author`s submission for this Lab Book. For each question, the SQL code is provided in this document accompanied by explanatory notes which serve to demonstrate and reinforce knowledge of the topics covered. The notes may prove useful to the author as a reference point for future Lab Book or Project submissions.

In each section screenshots are also provided which provide evidence of the SQL code being implemented in the database together with the actual output to the screen. Additionally, separate .sql files are included with the submission (one for each question), which the Examiner may wish to utilize to test the code.

QUESTION 1

Figure 1.1 – Simple Loop

SQL USED:

```
BEGIN
    FOR i in 1..100
    LOOP
        dbms_output.put_line(i);

    END LOOP;
END;
/
```

Explanatory Notes

In the line **FOR i IN 1..100**, **i** represents the variable that will be used to output the number (**i** does not need to be separately declared).**IN 1..100** defines the range of values (you could also have a tedious comma separated list of values).

Within the loop **I** is output to the screen using **DBMS_OUTPUT.PUT_LINE**.

Figure 1.2 – Simple Loop (Remaining output)

QUESTION 2

SQL USED:

```
BEGIN

<<outer>>
FOR i IN 1 .. 10 LOOP
      <<inner>>
   FOR j IN 1 .. 10 LOOP
     dbms_output.put_line(I || . ||
     j);
   END LOOP inner;
END LOOP outer;

END;
/
```

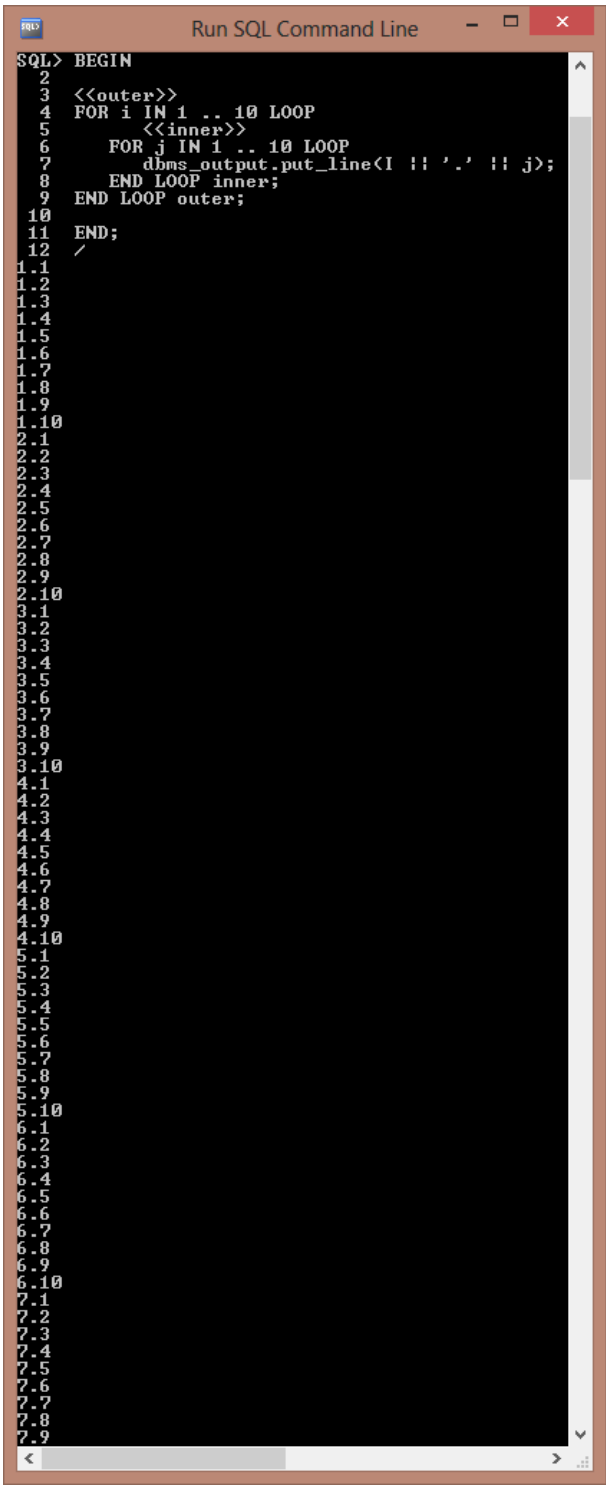
Explanatory Notes

The nested loop is labelled using the triangle brackets <<label>>. The author has used the alias <<outer>> to label the outer loop – it could be any label you wish.

The outer loop loops from 1 to 10 for **i**. The inner loop loops from 1 to 10 for **j**. In order to output in the required format i.e. 1.1,1.2,1.3 etc., within the **DBMS_OUTPUT** statement, **I** is joined (using the || characters) to **j**. A '.' Is also added in the join.

Both the inner and outer loops are then ended.

Figure 2.1 – Nested Loop



QUESTION 3

SQL USED:

```
DECLARE encountered NUMBER;

BEGIN
<<outer>>
FOR i IN 1 .. 30 LOOP
    encountered := 0;
    <<inner>>
        FOR j In 2..i-1 LOOP
            IF MOD(i,j) = 0 THEN
            dbms_output.put_line(i);
        encountered := 1;
        END IF;
        EXIT WHEN encountered =1;
        END LOOP inner;
    END LOOP outer;

END;
/
```

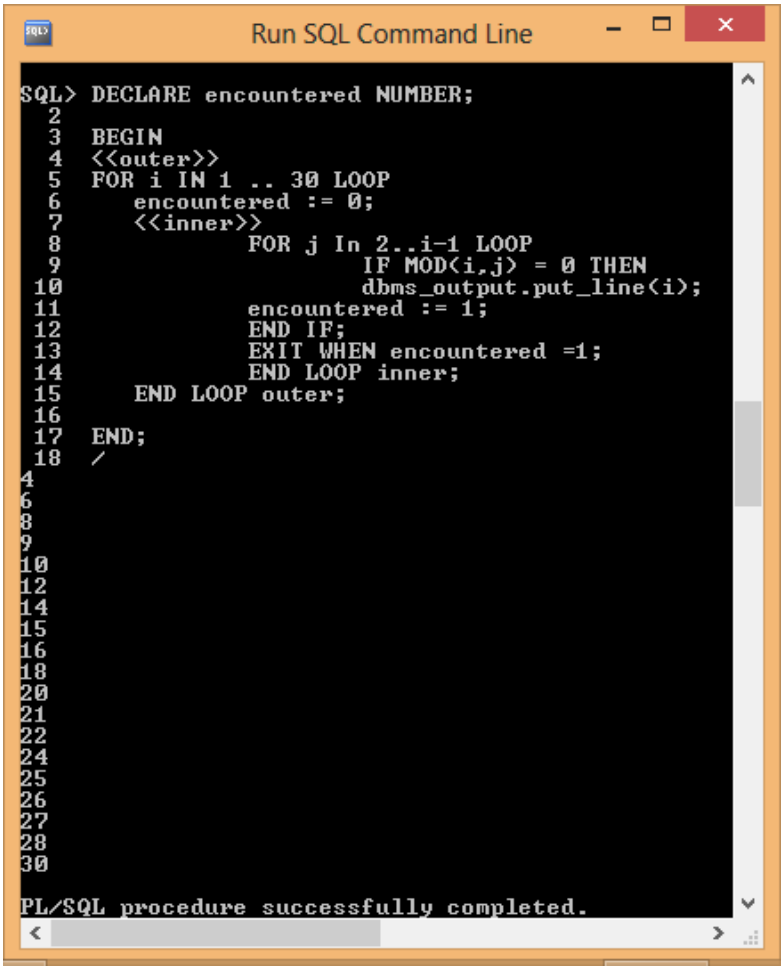
Explanatory Notes

Encountered is declared which will be used as an exit condition later in the SQL code. The outer loop loops from 1 to 30 for **i**. **Encountered** is set to zero.

In the inner loop, **j** loops from 2 to **i**-1; so in the first iteration **j** is 2 and **I** is 1. An **if-statement** tests of the **modulus** of I and j is zero (**IF MOD (I,j) = 0**, and if this is true then **I** is **DMBS_OUTPUT** to the screen.

In the third iteration, for example, **j** is 4 and **I** is 3, the modules of j and i is not zero so **I** is not output to the screen.

Figure 3.1 – Prime Number Loop



QUESTION 4

SQL USED:

```
CREATE SEQUENCE s2
    START WITH 310
    INCREMENT BY 1;

CREATE OR REPLACE TRIGGER
q4_trigger
BEFORE INSERT ON employee
FOR EACH ROW
    BEGIN
    :NEW.EMP_ID:=S2.NEXTVAL;
    END;
/

BEGIN
INSERT INTO EMPLOYEE (FNAME,
LNAME, MANAGER_EMP_ID)
VALUES('Panda', 'Bear', 304);
END;
/
```

Explanatory Notes

A **sequence** is used in this question to generate values for the **emp_id** primary key. The sequence is named **s2** and is set to start with 310 (emp_id 310). You could also set a min and max value but it is not necessary for this exercise.

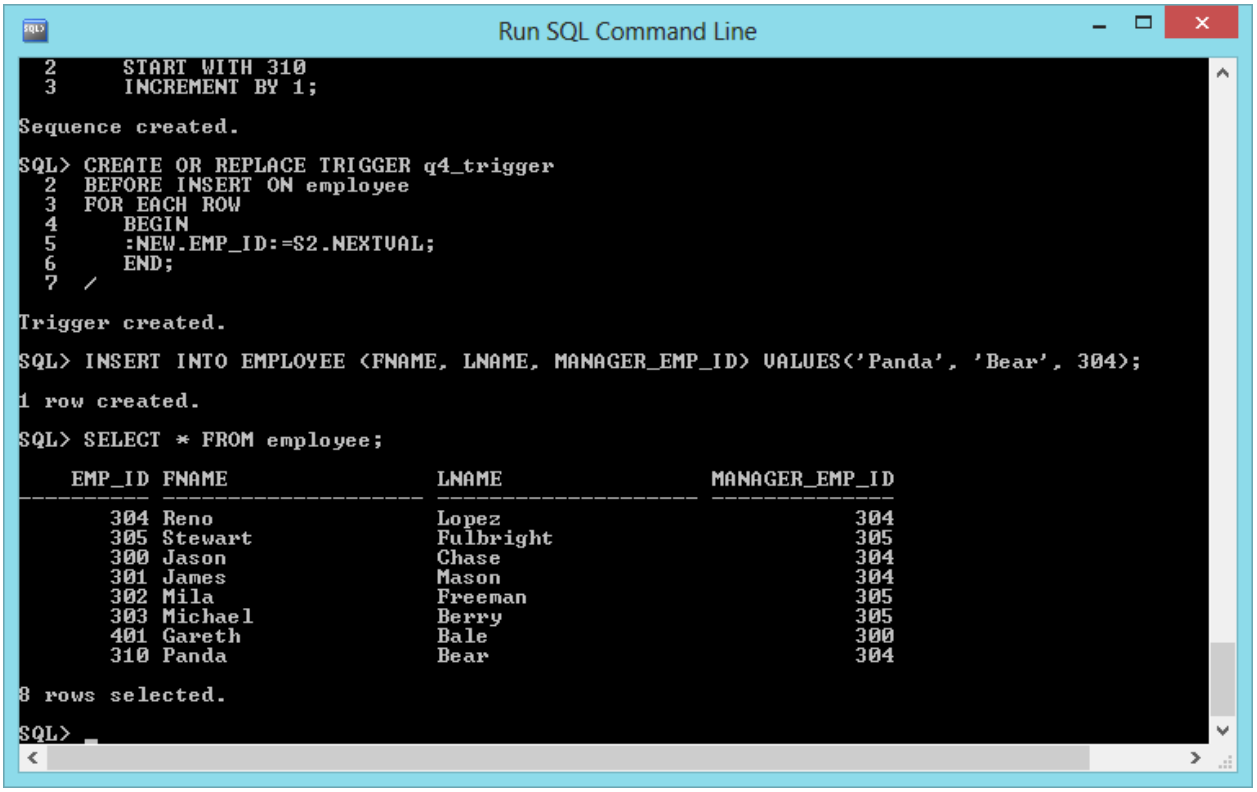
Finally, in the sequence, the value is incremented by 1, so the next incremented value after 310 would be 311, 312 etc.

A trigger is then created, named **q4_trigger**, with a **BEFORE INSERT** statement (**ON** the employee table) which specifies the new value of the **emp_id** (using **:NEW.EMP_ID** syntax) is equal to the next value from the **s2** sequence (using **S2.NEXTVAL** syntax). **FOR EACH ROW** was previously specified to ensure the action could be executed multiple times.

To test that the trigger works, an employee with an **FNAME** of 'Panda' , **LNAME** of 'Bear' and **MANAGER_EMP_ID** of 304 is inserted into the employee table. Selecting all data from the table as shown in Figure 4.2 demonstrates that the trigger has been implemented successfully.

Figure 4.1 - Table before Sequence

Figure 4.2 - Sequence Implementation



QUESTION 5

SQL USED:

```
ALTER TABLE employee ADD salary NUMBER(10);
DESCRIBE employee;
SELECT * FROM employee;

UPDATE employee SET salary='10000' WHERE
emp_id=304;
UPDATE employee SET salary='20000' WHERE
emp_id=305;
UPDATE employee SET salary='30000' WHERE
emp_id=300;
UPDATE employee SET salary='40000' WHERE
emp_id=301;
UPDATE employee SET salary='50000' WHERE
emp_id=302;
UPDATE employee SET salary='60000' WHERE
emp_id=303;
UPDATE employee SET salary='90000' WHERE
emp_id=401;
UPDATE employee SET salary='110000' WHERE
emp_id=310;

CREATE OR REPLACE TRIGGER q5trigger
    AFTER INSERT OR UPDATE ON employee
    FOR EACH ROW
BEGIN
    IF :NEW.salary > 100000 THEN
    dbms_output.put_line(:NEW.emp_id || ' '
|| :NEW.salary);
    END IF;
END;
/
INSERT INTO EMPLOYEE (FNAME, LNAME,
MANAGER_EMP_ID, SALARY) VALUES('Pandaaa',
'Bearaa', 304,160000);
```

Explanatory Notes

The aim of this question is to create a trigger which checks if an employee`s salary is **set to be** more than 100,000, and if it is, the salary value and employee ID are output to the screen.

Because there was no salary field initially, it was added and values were inserted.

The trigger is created with an **AFTER INSERT OR UPDATE** clause. In other words, after an insert or an update is carried out on the employee table, the actions in the trigger body are carried out.

The body contains an **if-statement** which checks if **the** new row inserted contains a salary field with a value greater than 100,000, and if so, **THEN** the employee id and salary values are **DBMS_OUTPUT** to the screen.

A new employee is inserted into the table with a salary value greater than 100,000. As shown in Figure 5.3, this demonstrates that the trigger has been successfully implemented.

Figure 5.1 - Adding Salary attribute to employee table

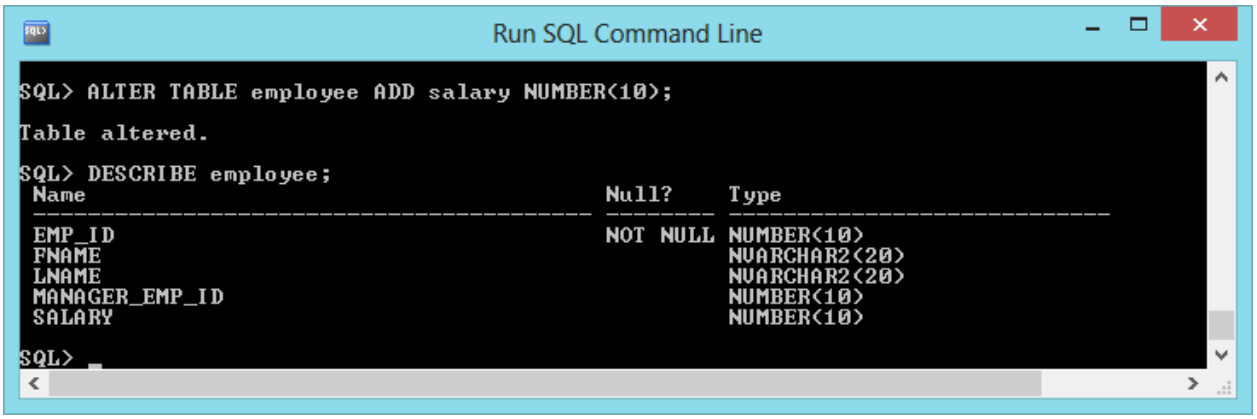


Figure 5.2 – Adding Values to Salary Column

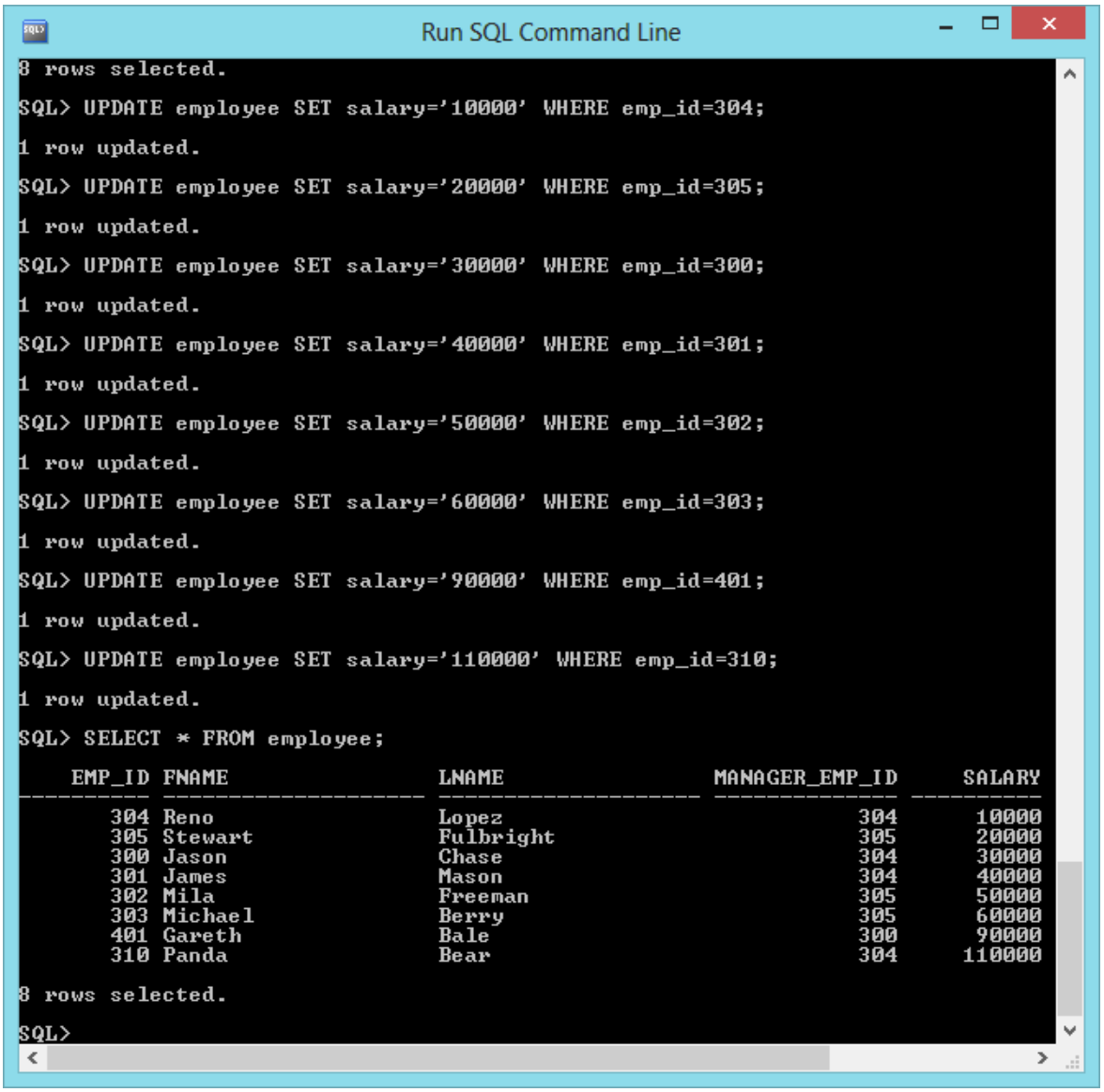
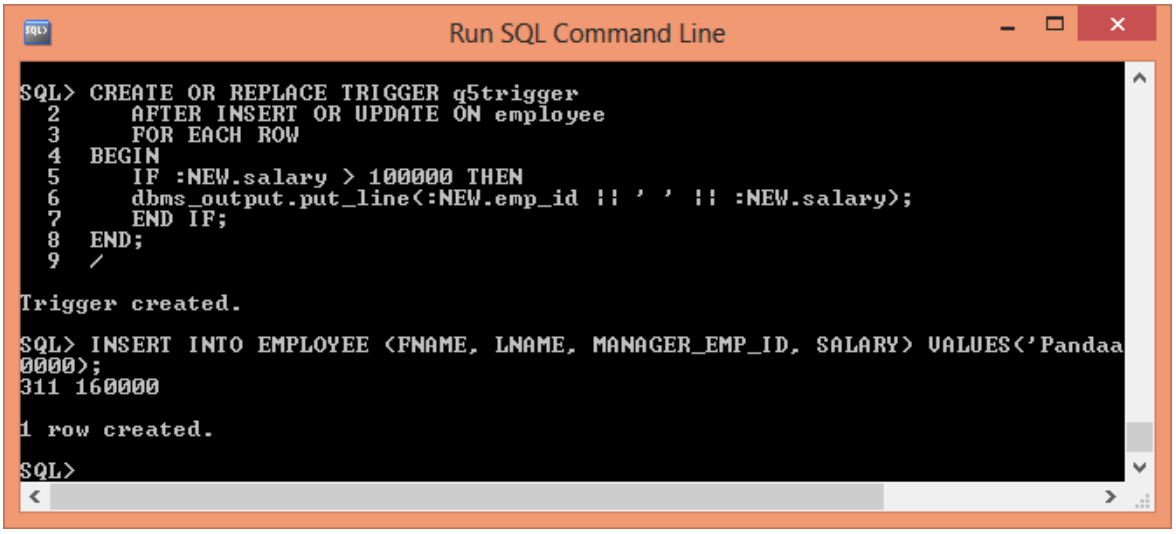


Figure 5.3 – Implementing and testing the trigger



QUESTION 6

SQL USED:

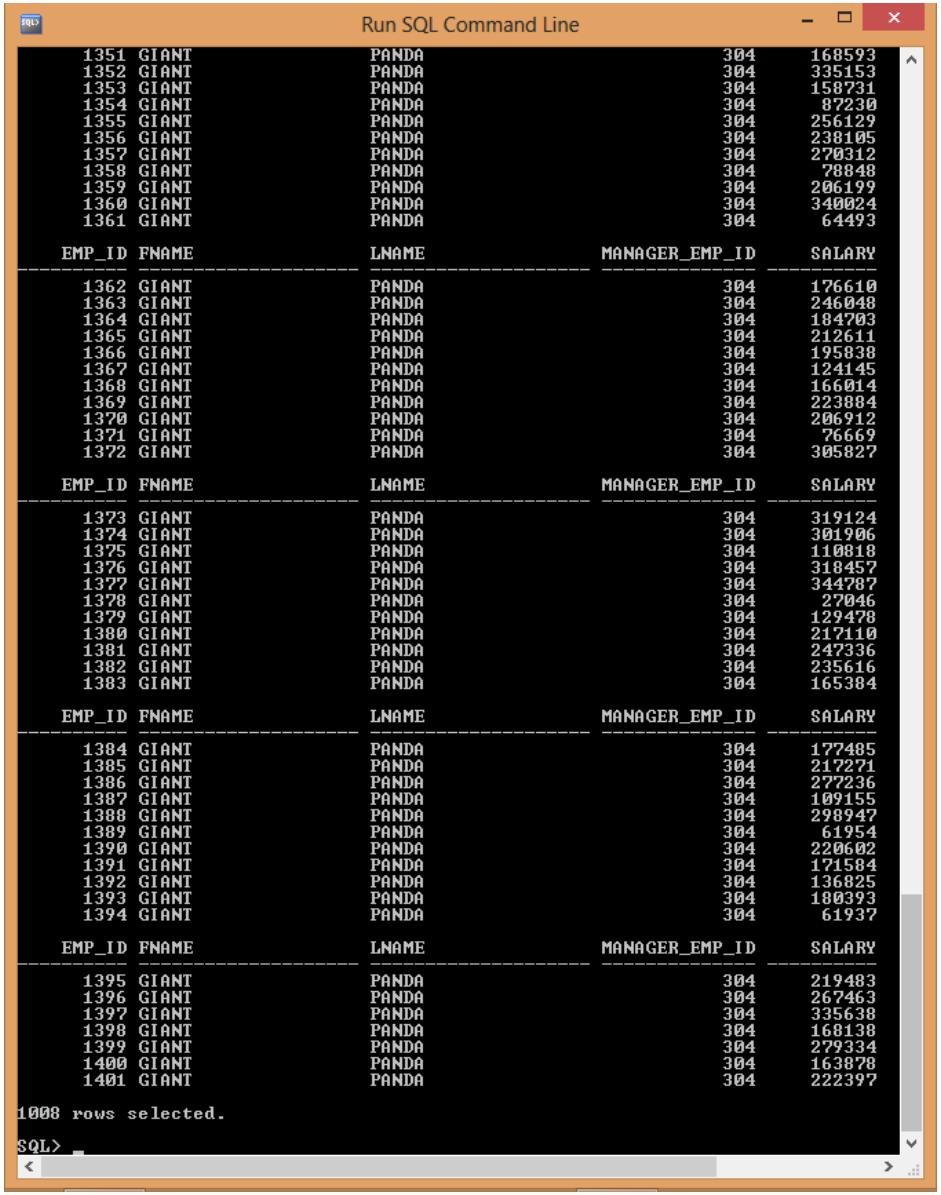
```
DECLARE
    rand_salary NUMBER;
BEGIN
FOR i IN 1..1000 LOOP
    SELECT DBMS_RANDOM.VALUE(12000,350000) INTO
rand_salary FROM DUAL;
    INSERT INTO EMPLOYEE (FNAME, LNAME,
MANAGER_EMP_ID, SALARY) VALUES('GIANT', 'PANDA',
304, rand_salary);
END LOOP;
END;
/
```

Explanatory Notes

A loop is carried out from 1 to 1000. Within the loop **SELECT DBMS_RANDOM.VALUE** is used to insert random values into the temporary variable **rand_salary** (which was declared earlier). The first figure represents the lowest number and the second figure is one number higher than the highest number to be inserted (i.e. 349999 in this scenario).

Then, an insert is carried out, with **rand_salary** being the distinguishing input. The **emp_id** is automatically generated by the trigger and sequence from

Figure 6.1 - Employee Table after 1000 employees inserted.



QUESTION 7

SQL USED

```
-- Create Procedure
CREATE OR REPLACE PROCEDURE q7_salary_over_ninety_k
IS
BEGIN
    UPDATE employee SET salary = '90000';
END q7_salary_over_ninety_k;
/


--Declare the Exception

DECLARE
q7_exception EXCEPTION;
--Begin the cursor
BEGIN
DECLARE
CURSOR lab_5_q7 IS  -- Declaration of cursor name
-- Below, the columns are selected from the employee table
SELECT emp_id, fname, sname, manager_emp_id, salary FROM employee;
-- Below, a variable is declared which represents every column in a
row
q7_cursor lab_5_q7 %ROWTYPE;

BEGIN -- sub-block begins
OPEN lab_5_q7;
FETCH lab_5_q7 INTO q7_cursor;
WHILE  lab_5_q7 %FOUND LOOP
    IF q7_cursor.salary < 20000 THEN RAISE_APPLICATION_ERROR
    (-123456,'You have employees with salaries < €20,000, you should
pay them more than that!');
            END IF;
IF q7_cursor.salary >90000 THEN RAISE q7_exception;
END IF;
FETCH lab_5_q7 INTO q7_cursor;

END LOOP;
CLOSE lab_5_q7;
    END; -- sub-block ends

EXCEPTION
    WHEN q7_exception THEN
BEGIN
    q7_salary_over_ninety_k;
RAISE; -- reraise the current exception
END;

/
```

Figure 7.1 – Creating Procedure

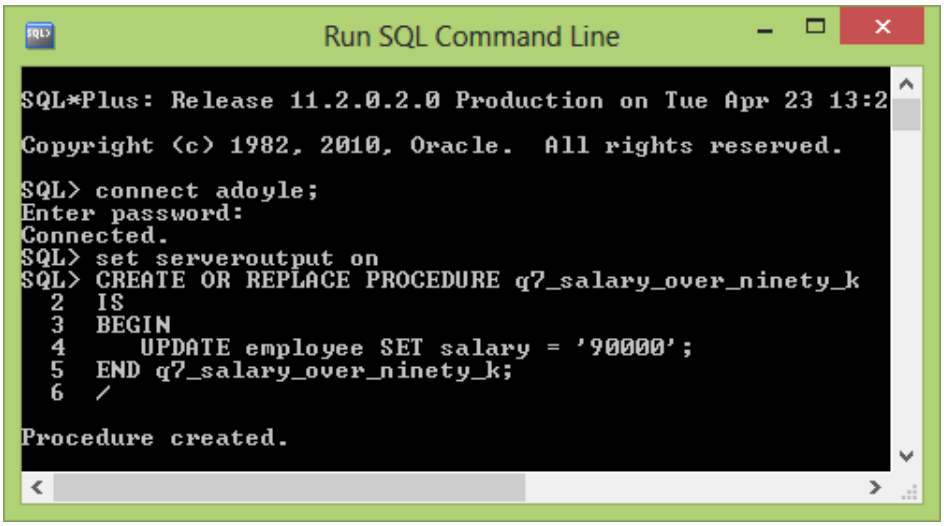
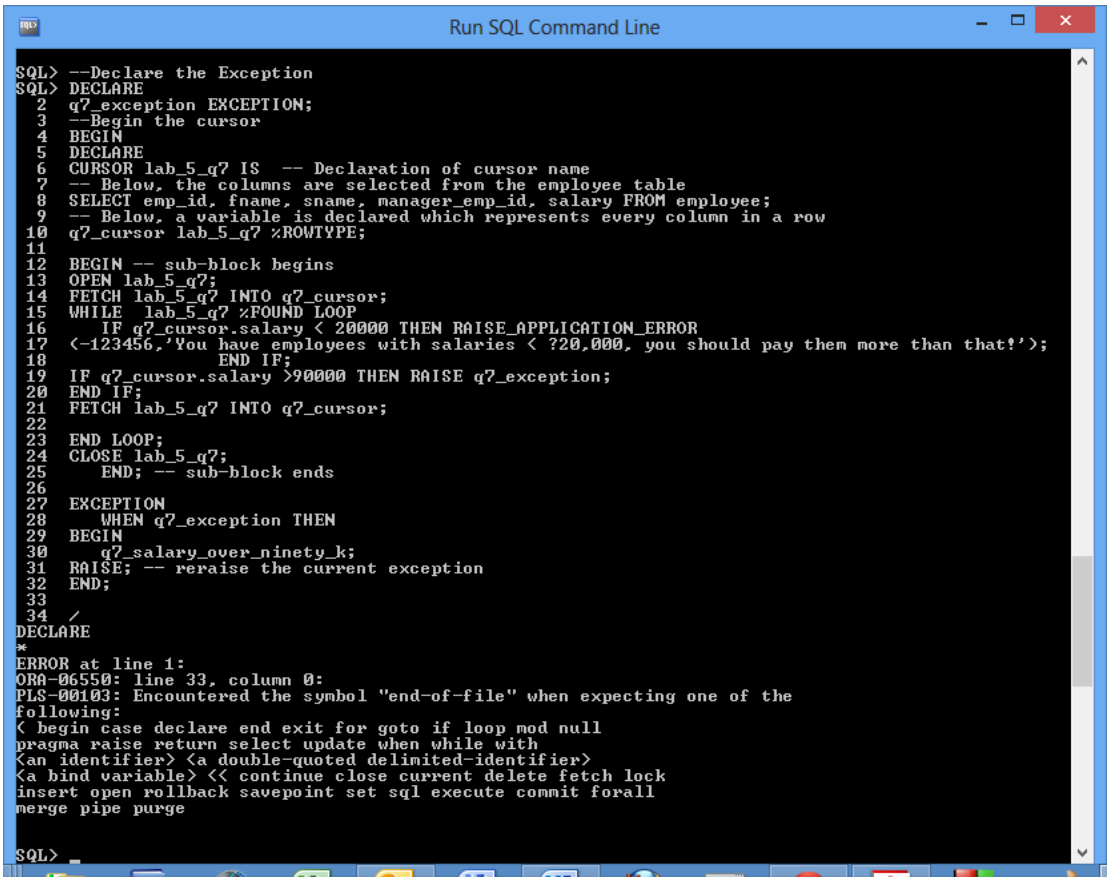


Figure 7.2 – Implementation of Exception and Cursor



[Read here for more information on %ROWTYPE]

Explanatory Notes

A procedure is created to specify what happens (update to 90000) when a salary is found to be over 90000. An exception is then declared. A cursor is then created and all columns are selected from the table. %ROWTYPE is used when creating the variable **q7_cursor**; this represents every column in a row.

Then, the cursor is opened and fetched into the **q7_cursor**. A loop is carried out and if a salary is found to be less than 20,000, an application error is raised together with a custom output message. Another if-statement tests if any salary fields are over 90,000, and if so, the exception **q7_exception** is raised.

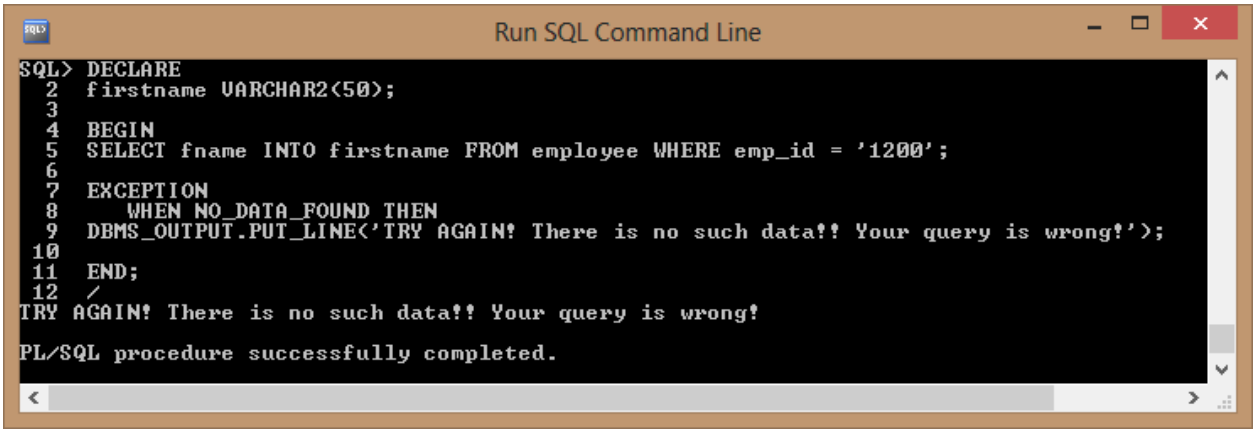
In the exception declaration, the procedure is called to update the salary fields and is re-raised.

QUESTION 8

SQL USED

```
DECLARE

firstname VARCHAR2(50);

BEGIN
SELECT fname INTO firstname FROM employee WHERE
emp_id = 1200;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE(TRY AGAIN! There is no such
data!! Your query is wrong!);

END;
/
```

Figure 8.1 – Implementation of Exception Catch



Explanatory Notes

A variable is declared, **firstname**. A **SELECT INTO** statement takes the **fname** attribute from the employee table where the employee id is 1200. Such an employee ID does not exist.

An exception is created to the **NO_DATA_FOUND** error message, so that when no data is found, the custom error message is displayed in lieu of the pre-defined error; as shown in Figure 8.1.