National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Fall 2021, Lab Manual – 08

Course Code: CL-2005	Course : Database Systems Lab
Instructor(s) :	Muhammad Nadeem, Amin Sadiq, Erum, Fizza, Mafaza, Ali Fatmi

Contents:

1. PL/SQL
2. Block Structure
3. Variable & types
4. Conditional Logic
5. Loops
6. Views
7. Procedures
8. Functions
9. Cursors

PL/SQL

PL/SQL is a combination of SQL along with the procedural features of programming languages. PL/SQL is a completely portable, high-performance transaction-processing language.it provides a built-in, interpreted and OS independent programming environment. It is tightly integrated with SQL and offers extensive error checking, numerous data types, and variety of programming structures. It also supports structured programming through functions and procedures, object-oriented programming, supports the development of web applications and server pages.

Block Structure

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling>
END;
```

Note: add the following command on the top of the script **"set serveroutput on"**

```
set serveroutput on

DECLARE
   Sec_Name varchar2(20) := 'Sec-A';
   Course_Name varchar2(20) := 'Database Systems Lab';
BEGIN
    dbms_output.put_line('This is : '|| Sec_Name || ' and the course
   is ' || Course_Name);
END;
```

Delimiter	Description	Delimiter	Description
+, -, *, /	Addition, subtraction/ negation, multiplication, division	:	Host variable indicator
%	Attribute indicator	,	Item separator
'	Character string delimiter	"	Quoted identifier delimiter
.	Component selector	=	Relational operator
(,)	Expression or list delimiter	@	Remote access indicator
||	Concatenation operator	;	Statement terminator
**	Exponentiation operator	:=	Assignment operator
<<, >>	Label delimiter (begin and end)	=>	Association operator
/*, */	Multi-line comment delimiter (begin and end)	<, >, <=, >=	Relational operators
--	Single-line comment indicator	<>, '=, ~=, ^=	Different versions of NOT EQUAL
..	Range operator		

Variable & types

```
set serveroutput on
DECLARE
   a integer := 10;
   b integer := 20;
   c integer;
   f real;
BEGIN
   c := a + b;
   dbms_output.put_line('Value of c: ' || c);
   f := 70.0/3.0;
   dbms_output.put_line('Value of f: ' || f);
END;
```

```
set serveroutput on
DECLARE
   a integer := 10;
   b integer := 20;
   c integer;
   f real;
BEGIN
   c := a + b;
   dbms_output.put_line('Value of c: ' || c);
   f := 70.0/3.0;
   dbms_output.put_line('Value of f: ' || f);
END;
```

```
DECLARE
   -- Global variables
   num1 number := 95;
   num2 number := 85;
BEGIN
   dbms_output.put_line('Outer Variable num1: ' || num1);
   dbms_output.put_line('Outer Variable num2: ' || num2);
   DECLARE
      -- Local variables
      num1 number := 195;
      num2 number := 185;
   BEGIN
      dbms_output.put_line('Inner Variable num1: ' || num1);
      dbms_output.put_line('Inner Variable num2: ' || num2);
   END;
END;
```

```
DECLARE
   e_id employees.EMPLOYEE_ID%type;
   e_name  employees.FIRST_NAME%type;
   e_lname employees.LAST_NAME%type;
   d_name  DEPARTMENTS.DEPARTMENT_NAME%type;
BEGIN
   SELECT EMPLOYEE_ID,FIRST_NAME,LAST_NAME,DEPARTMENT_NAME
   INTO e_id, e_name, e_lname, d_name
   FROM employees inner join  DEPARTMENTS
   on  employees.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID and
EMPLOYEE_ID =100 ;
    dbms_output.put_line('EMPLOYEE ID: ' ||e_id);
    dbms_output.put_line('EMPLOYEE First Name: ' ||e_name);
    dbms_output.put_line('EMPLOYEE Last Name: ' ||e_lname);
    dbms_output.put_line('DEPARTMENT Name: ' ||d_name);
END;
```

Conditional Logic

1. IF - THEN statement
2. IF-THEN-ELSE statement
3. IF-THEN-ELSIF statement
4. Case statement
5. Searched CASE statement
6. Nested IF-THEN-ELSE

```
DECLARE
   e_id employees.EMPLOYEE_ID%type := 100;
   e_sal  employees.SALARY%type;
BEGIN
   SELECT salary INTO e_sal FROM employees WHERE EMPLOYEE_ID = e_id;
   IF (e_sal >=5000)
   THEN
   UPDATE employees SET salary = e_sal+1000 WHERE EMPLOYEE_ID= e_id;
      dbms_output.put_line ('Salary updated');
   END IF;
END;
```

```
Declare
  n_count number;
  e_id employees.EMPLOYEE_ID%type := 1100;
Begin
  Select count(1) into n_count from employees   Where EMPLOYEE_ID = e_id;
  if n_count > 0 then
     dbms_output.put_line('record already exists.');
  else
      INSERT INTO employees

(employee_id,first_name,last_name,email,phone_number,hire_date,job_id,salary,commission
_pct,manager_id,department_id)
       VALUES (e_id,'Bruce','Austin','DAUSTIN7','590.423.4569','25-JUN-
05','IT_PROG',6000,0.2,100,60);
      dbms_output.put_line('record inserted with Employee ID: ' ||e_id);
  end if;
End;
```

```
DECLARE
   e_id employees.EMPLOYEE_ID%type := 100;
   e_sal  employees.SALARY%type;
BEGIN
   SELECT salary INTO e_sal FROM employees WHERE EMPLOYEE_ID = e_id;
   IF (e_sal <=25000)   THEN
   UPDATE employees SET salary = e_sal+100 WHERE EMPLOYEE_ID= e_id;
      dbms_output.put_line ('Salary updated:' ||e_sal);
   ELSIF  (e_sal >=20000) THEN
   UPDATE employees SET salary = e_sal+200 WHERE EMPLOYEE_ID= e_id;
      dbms_output.put_line ('Salary updated:'||e_sal);
  ELSIF  (e_sal <=15000) THEN
   UPDATE employees SET salary = e_sal+300 WHERE EMPLOYEE_ID= e_id;
      dbms_output.put_line ('Salary updated:'||e_sal);
  ELSE
  UPDATE employees SET salary = e_sal+400 WHERE EMPLOYEE_ID= e_id;
      dbms_output.put_line ('Salary updated:'||e_sal);
   END IF;
END;
```

```
DECLARE
   e_id employees.EMPLOYEE_ID%type := 100;
   e_sal  employees.SALARY%type;
   e_did  employees.DEPARTMENT_ID%type;
BEGIN
   SELECT salary,DEPARTMENT_ID INTO e_sal,e_did FROM employees WHERE EMPLOYEE_ID =
e_id;
   CASE e_did
   when  80 then
   UPDATE employees SET salary = e_sal+100 WHERE EMPLOYEE_ID= e_id;
   dbms_output.put_line ('Salary updated:' ||e_sal);
   when  50 then
   UPDATE employees SET salary = e_sal+200 WHERE EMPLOYEE_ID= e_id;
      dbms_output.put_line ('Salary updated:'||e_sal);
   when  40 then
   UPDATE employees SET salary = e_sal+300 WHERE EMPLOYEE_ID= e_id;
      dbms_output.put_line ('Salary updated:'||e_sal);
   ELSE
   dbms_output.put_line('No such Record');
   END CASE;
END;
```

```
DECLARE
   e_id employees.EMPLOYEE_ID%type := 100;
   e_sal  employees.SALARY%type;
   e_did  employees.DEPARTMENT_ID%type;
BEGIN
   SELECT salary,DEPARTMENT_ID INTO e_sal,e_did FROM employees WHERE EMPLOYEE_ID =
e_id;
   CASE
   when  e_did = 80 then
   UPDATE employees SET salary = e_sal+100 WHERE EMPLOYEE_ID= e_id;
   dbms_output.put_line ('Salary updated:' ||e_sal);
   when  e_did = 50 then
   UPDATE employees SET salary = e_sal+200 WHERE EMPLOYEE_ID= e_id;
      dbms_output.put_line ('Salary updated:'||e_sal);
   when  e_did = 40 then
   UPDATE employees SET salary = e_sal+300 WHERE EMPLOYEE_ID= e_id;
      dbms_output.put_line ('Salary updated:'||e_sal);
   ELSE
   dbms_output.put_line('No such Record');
   END CASE;
END;
```

```
DECLARE
   e_id employees.EMPLOYEE_ID%type := 100;
   e_sal  employees.SALARY%type;
   e_did  employees.DEPARTMENT_ID%type;
   e_com  employees.COMMISSION_PCT%type;
BEGIN
  SELECT salary,DEPARTMENT_ID,COMMISSION_PCT INTO e_sal,e_did,e_com FROM
employees WHERE EMPLOYEE_ID = e_id;
   IF (e_did=90)   THEN
     IF (e_sal >=20000 AND e_sal <=250000 )   THEN
      UPDATE employees SET salary = (e_sal+00)*(1+e_com) WHERE EMPLOYEE_ID=
e_id;
      dbms_output.put_line ('Salary updated:' ||e_sal);
     ELSIF (e_sal >=15000 AND e_sal <=20000 )   THEN
      UPDATE employees SET salary = (e_sal+20)*(1+e_com) WHERE EMPLOYEE_ID=
e_id;
      dbms_output.put_line ('Salary updated:' ||(e_sal+100)*(1+e_com));
        END IF;
    END IF;

    IF (e_did=40)   THEN
     IF (e_sal >=10000 AND e_sal <=15000 )   THEN
      UPDATE employees SET salary = (e_sal+00)*(1+e_com) WHERE EMPLOYEE_ID=
e_id;
      dbms_output.put_line ('Salary updated:' ||e_sal);
     ELSIF (e_sal >=5000 AND e_sal <=10000 )   THEN
      UPDATE employees SET salary = (e_sal+20)*(1+e_com) WHERE EMPLOYEE_ID=
e_id;
      dbms_output.put_line ('Salary updated:' ||(e_sal+100)*(1+e_com));
     END IF;
   END IF;
END;
```

Loops

```
SET SERVEROUTPUT ON;
DECLARE
  BEGIN
   FOR c IN (SELECT EMPLOYEE_ID, FIRST_NAME, SALARY  FROM employees
              WHERE DEPARTMENT_ID = 90)
   LOOP
      DBMS_OUTPUT.PUT_LINE (
         'Salary for the employee ' || c.FIRST_NAME || ' is: ' || c.SALARY);
   END LOOP;
END;
```

Views

View is a virtual table that does not physically exist. Rather, it is created by a query joining one or more tables. A view contains no data itself. A view is simply any SELECT query that has been given a name and saved in the database. For this reason, a view is sometimes called a named query or a stored query.

Benefits of using Views

- Commonality of code being used. Since a view is based on one common set of SQL this means that when it is called it's less likely to require parsing.
- Views have long been used to hide the tables that actually contain the data you are querying. Also, views can be used to restrict the columns that a given user has access to.

Types of views:

1. **Updateable Views:**

The data dictionary views ALL_UPDATABLE_COLUMNS, DBA_UPDATABLE_COLUMNS, and USER_UPDATABLE_COLUMNS indicate which view columns are updatable. View does not hold any data so the impact of the DML operation will be direct on master/base table.

2. **Read-Only Views:**

A view is *read-only* if it is *not* delete-able, updatable, or insert-able. A view can be read-only if it is a view that does not comply with at least one of the rules for delete-able views.

3. **Materialized Views**

Materialized views are schema objects that can be used to summarize, pre compute, replicate, and distribute data. E.g. to construct a data warehouse. A materialized view provides indirect access to table data by storing the results of a query in a separate schema object. Unlike an ordinary view, which does not take up any storage space or contain any data. A materialized view can be stored in the same database as its base table(s) or in a different database. Materialized views stored in the same database as their base tables can improve query performance through query rewrites. Query rewrites are particularly useful in a data warehouse environment. A materialized view log is a schema object that records changes to a master table's data so that a materialized view defined on the master table can be refreshed incrementally.

```
CREATE or REPLACE VIEW EMP_Det AS
  SELECT DISTINCT EMPLOYEES.EMPLOYEE_ID, EMPLOYEES.FIRST_NAME,
EMPLOYEES.EMAIL,DEPARTMENTS.DEPARTMENT_NAME FROM EMPLOYEES INNER JOIN DEPARTMENTS
  ON EMPLOYEES.EMPLOYEE_ID = DEPARTMENTS.DEPARTMENT_ID
  WHERE EMPLOYEES.DEPARTMENT_ID = 80;
    select * from emp_det;
  select * from employees;
  update emp_det set FIRST_NAME='Ali' where EMPLOYEE_ID=170;
  delete from emp_det where EMPLOYEE_ID=170;
```

```
create or replace view x as
select * from employees /* your query */
with read only;

select * from x;

update x set salary = 100 where employee_id =100;
```

```
  CREATE MATERIALIZED VIEW MAT_EMP_Det
AS
     SELECT DISTINCT EMPLOYEES.EMPLOYEE_ID, EMPLOYEES.FIRST_NAME,
EMPLOYEES.EMAIL,DEPARTMENTS.DEPARTMENT_NAME FROM EMPLOYEES INNER JOIN DEPARTMENTS
  ON EMPLOYEES.EMPLOYEE_ID = DEPARTMENTS.DEPARTMENT_ID
  WHERE EMPLOYEES.DEPARTMENT_ID = 80;

    update emp_det set FIRST_NAME='Fatmi' where EMPLOYEE_ID=150;
    select * from employees where EMPLOYEE_ID=150;
    select * from MAT_EMP_Det where EMPLOYEE_ID=150;
```

Functions

A stored function (also called a user function or user-defined function) is a set of PL/SQL statements you can call by name. Stored functions are very similar to procedures, except that a function returns a value to the environment in which it is called. User functions can be used as part of a SQL expression.

1. Scalar Value functions
2. Inline table valued functions
3. Multi statement table valued functions

```
CREATE or replace  FUNCTION CalculateSAL(DEPT_ID in NUmber)
RETURN NUMBER
IS
Total_Salary NUmber;
BEGIN
Select sum(Salary) into Total_Salary from employees where DEPARTMENT_ID= 80;
RETURN(Total_Salary);
END;
 select CalculateSAL(80) from dual;
```

```
CREATE or replace  FUNCTION CalculateTOTALSAL
RETURN NUMBER
IS
Total_Salary NUmber;
BEGIN
Select sum(Salary) into Total_Salary from employees;
RETURN(Total_Salary);
END;
select CalculateTOTALSAL from dual;
```

```
CREATE or replace  TYPE EMP_OBJ_TYPE as OBJECT (
  EMPLOYEE_ID NUMBER(6,0),
  FIRST_NAME VARCHAR(30),
  LAST_NAME VARCHAR(30),
  DEPARTMENT_ID NUMBER(4,0)
);

CREATE TYPE EMP_TBL_TYPE as TABLE OF EMP_OBJ_TYPE;

CREATE OR REPLACE FUNCTION GETALL
RETURN EMP_TBL_TYPE
IS
 EMPLOYEE_ID NUMBER(6,0);
  FIRST_NAME VARCHAR(30);
  LAST_NAME VARCHAR(30);
  DEPARTMENT_ID NUMBER(4,0);
  -- NESTED TABLE VARIALE DECLARATION AND INITIALIZATION

    EMP_DETAILS EMP_TBL_TYPE := EMP_TBL_TYPE();
BEGIN
-- EXTENDING THE NESTED TABLE
EMP_DETAILS.EXTEND();
---- GET THE REQUIRED DATA INTO VARIABLES
SELECT EMPLOYEE_ID,FIRST_NAME, LAST_NAME,DEPARTMENT_ID INTO
EMPLOYEE_ID,FIRST_NAME,LAST_NAME,DEPARTMENT_ID FROM EMPLOYEES where EMPLOYEE_ID=100;
-- USING A OBJECT CONSTRUCTOR, TO INSERT THE DATA INTO THE NESTED TABLE
EMP_DETAILS(1) := EMP_OBJ_TYPE(EMPLOYEE_ID,FIRST_NAME,LAST_NAME,DEPARTMENT_ID);
RETURN EMP_DETAILS;
END;
/
SELECT * FROM TABLE(GETALL);
```

```
CREATE OR REPLACE FUNCTION GETALL1
RETURN EMP_TBL_TYPE
IS
 EMPLOYEE_ID NUMBER(6,0);
  FIRST_NAME VARCHAR(30);
  LAST_NAME VARCHAR(30);
  DEPARTMENT_ID NUMBER(4,0);
  -- NESTED TABLE VARIALE DECLARATION AND INITIALIZATION

    EMP_DETAILS EMP_TBL_TYPE := EMP_TBL_TYPE();
BEGIN
-- EXTENDING THE NESTED TABLE
EMP_DETAILS.EXTEND();
---- GET THE REQUIRED DATA INTO VARIABLES
SELECT EMP_OBJ_TYPE( EMPLOYEE_ID,FIRST_NAME, LAST_NAME,DEPARTMENT_ID) bulk collect
INTO EMP_DETAILS FROM EMPLOYEES;
-- USING A OBJECT CONSTRUCTOR, TO INSERT THE DATA INTO THE NESTED TABLE
RETURN EMP_DETAILS;
END;
/
SELECT * FROM TABLE(GETALL1);
```

Stored Procedures

A stored procedure is a PL/SQL block which performs a specific task or a set of tasks. A procedure Has a name, contains SQL queries and is able to receive parameters and return results. A procedure Is similar to functions (or methods) in programming languages.

Benefits of stored procedure

Reusability: Create a procedure once and use it any number of times at any number of places. You just need to call it and your task is done.

Easy maintenance: If instead of using a procedure, you repeat the SQL everywhere and if there is a change in logic, then you need to update it at all the places. With stored procedure, the change needs to be done at only one place.

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE PROCEDURE Insert_Data(STREET_ADDRESS IN VARCHAR,POSTAL_CODE IN VARCHAR
Default 'NULL', CITY VARCHAR, STATE_PROVINCE VARCHAR,COUNTRY_ID CHAR)
IS
  Total_record  INT;
  LOCATION_ID  Number;
BEGIN
    SELECT count(LOCATION_ID) into LOCATION_ID from LOCATIONS;
  LOCATION_ID :=LOCATION_ID+1;
  Total_record :=LOCATION_ID;
  INSERT INTO LOCATIONS(LOCATION_ID,STREET_ADDRESS,POSTAL_CODE,CITY,STATE_PROVINCE)
VALUES(LOCATION_ID,STREET_ADDRESS,POSTAL_CODE,CITY,STATE_PROVINCE);

     dbms_output.put_line('NEW RECORD INSERTED WITH ID : ' || LOCATION_ID);
       dbms_output.put_line('TOTAL NO OF RECORDS : ' || Total_record);
END;
exec Insert_Data('DHA','1234','KARACHI','SINDH','PK');
```

Cursors

A cursor is a pointer that points to a result of a query. PL/SQL has two types of cursors: implicit cursors and explicit cursors.

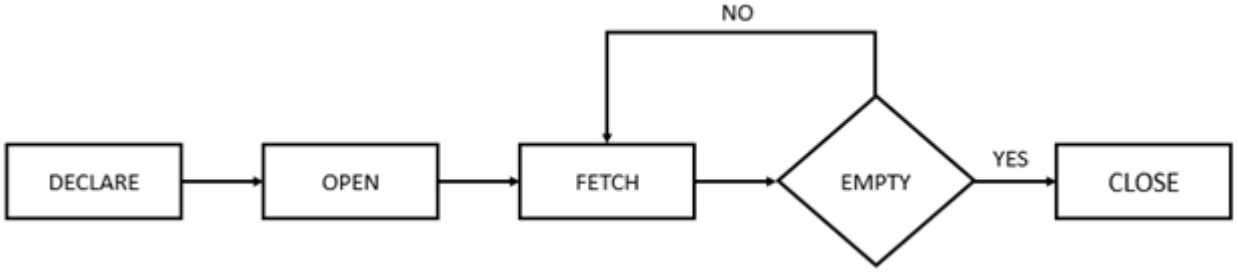
Implicit cursors

Whenever Oracle executes an SQL statement such as SELECT INTO, INSERT, UPDATE, and DELETE, it automatically creates an implicit cursor Oracle internally manages the whole execution cycle of implicit cursors and reveals only the cursor's information and statuses such as SQL%ROWCOUNT, SQL%ISOPEN, SQL%FOUND, and SQL%NOTFOUND.

Explicit cursors

An explicit cursor is an SELECT statement declared explicitly in the declaration section of the current block or a package specification. For an explicit cursor, you have control over its execution cycle from OPEN, FETCH, and CLOSE.

The following illustration shows the execution cycle of an explicit cursor:



```
SET SERVEROUTPUT ON;
DECLARE
  CURSOR Cursor_EMP IS
  SELECT * FROM employees ORDER BY salary DESC;
   -- record
   row_emp Cursor_EMP%ROWTYPE;
BEGIN
  OPEN Cursor_EMP;
 -- LOOP
    FETCH  Cursor_EMP  INTO row_emp;
    --EXIT WHEN Cursor_EMP%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( 'EMPLOYEE id: ' ||row_emp.EMPLOYEE_ID || ' EMPLOYEE NAME: ' ||
row_emp.FIRST_NAME || ' EMPLOYEE CONTACT: ' || row_emp.PHONE_NUMBER || '.');
 -- END LOOP;
  CLOSE Cursor_EMP;
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
  CURSOR Cursor_EMP IS
  SELECT * FROM employees ORDER BY salary DESC;
   -- record
   row_emp Cursor_EMP%ROWTYPE;
BEGIN
  OPEN Cursor_EMP;
  LOOP
    FETCH  Cursor_EMP  INTO row_emp;
    EXIT WHEN Cursor_EMP%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( 'EMPLOYEE id: ' ||row_emp.EMPLOYEE_ID || ' EMPLOYEE NAME: ' ||
row_emp.FIRST_NAME || ' EMPLOYEE CONTACT: ' || row_emp.PHONE_NUMBER || '.');
  END LOOP;
  CLOSE Cursor_EMP;
END;
```