Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Advanced PLSQL

Lesson – Dynamic SQL

Lesson Objectives

On completion of this lesson on Dynamic SQL, you will be able to:

- State the need for Dynamic SQL
- Understand the ways of implementing Dynamic SQL
- Understand the use of the various Options of the EXECUTE IMMEDIATE command

What is Dynamic SQL

Predictable Jobs

- Most PL/SQL programs do a specific, predictable job. For example, a stored procedure might accept an employee number and increase his salary by updating the sal column in the emp table. In this case, the full text of the UPDATE statement is known at compile time. Such statements do not change from execution to execution. So, they are called *static* SQL statements.

Dynamic Jobs

- However, some programs must build and process a variety of SQL statements at run time. For example, a stored procedure which accepts a table-name as a parameter and deletes all the row from that table.
- Such statements can, and probably will, change from execution to execution. So, they are called *dynamic* SQL statements.

What is Dynamic SQL ….contd

Dynamic Statements

- Dynamic SQL statements are stored in character strings built by your program at run time. Such strings must contain the text of a valid SQL statement or PL/SQL block. They can also contain placeholders for bind arguments. A *placeholder* is an undeclared identifier, so it's name, to which you must prefix a colon, does not matter.
- For example, PL/SQL makes no distinction between the following strings:
  - 'DELETE FROM emp WHERE sal > :my_sal AND comm < :my_comm'
  - 'DELETE FROM emp WHERE sal > :s AND comm < :c'.

Why do we need Dynamic SQL

You need dynamic SQL in the following situations:

- You want to execute a SQL data definition statement (such as CREATE), a data control statement (such as GRANT), or a session control statement (such as ALTER SESSION). **In PL/SQL, such statements cannot be executed statically.**
- You want more flexibility. For example, you might want to **defer your choice of schema objects until run time.** Or, you might want your program to build **different search conditions for the WHERE clause** of a SELECT statement. A more complex program might choose from various SQL operations, clauses, etc.
- You want better performance as compared to DBMS_SQL, something easier to use(Native Dynamic SQL). This is specifically an advantage of Native Dynamic SQL(using EXECUTE IMMEDIATE command)

Execute Immediate

- To process native dynamic SQL statements, you use the **EXECUTE IMMEDIATE** statement.

```
begin
        dbms_output.put_line('hello');
        create table mydtable(empno number, ename char(10));
end;
/
```

this will give an error

- The dynamic string can contain any SQL statement (*without* the terminator) or any PL/SQL block(with the terminator). The string can also contain placeholders for bind arguments.

```
begin
        dbms_output.put_line('hello');
        execute immediate 'create table mydtable1(empno number, ename char(10))';
end;
/
```

this will work and create the table.

USING clause of Execute Immediate

To provide values to the place-holders, we use the **USING clause of EXECUTE IMMEDIATE** statement :

```
declare
        empid number(4) := 101;
        ebon number(7):=12000;
begin
        execute immediate 'create table mybonus(id number, amt number)';
        execute immediate 'insert into mybonus values (:1, :2)' using empid, ebon;
end;
/

declare
        empid number(4) := 101;
        ebon number(7):=12000;
        sqlt varchar2(100):= 'insert into mybonus1 values (:1, :2)';
begin
        execute immediate 'create table mybonus1(id number, amt number)';
        execute immediate sqlt using empid, ebon;
end;
/
```

USING clause of Execute Immediate ….contd

- To provide values to the place-holders, we use the **USING clause of EXECUTE IMMEDIATE** statement :

```
declare
        sqlt varchar2(100):= 'insert into mybonus11 values (:1, :2)';
begin
        execute immediate 'create table mybonus11 (id number, amt number)';
        execute immediate sqlt using &empid, &ebon;
end;
/

declare
        sqlt varchar2(100):= 'insert into mybonus11 values (:1, :2)';
begin
        execute immediate 'create table mybonus11 (id number, amt number)';
        execute immediate sqlt using 101, 12000;
end;
/
```

USING clause of Execute Immediate ….contd

- To provide values to the place-holders, we use the **USING clause of EXECUTE IMMEDIATE** statement :

```
declare
        vdeptno number(2):=10;
        sqlt varchar2(100);
begin
        execute immediate 'select * from emp';
        --execute immediate 'select * from emp where deptno=vdeptno';
        --the above statement will give an error as inside the sql string vdeptno has no meaning and you have to
        --use pdeptno to treat it as placeholder, followed by the  using clause. In this case, it will search for a
        --column named vdeptno in the table and since such a column does not exist, it will give error  'vdeptno
        --is invalid identifier'
        execute immediate 'select * from emp where deptno=:pdeptno' using vdeptno;
        sqlt:='select * from emp where deptno =:b';
        execute immediate sqlt using  vdeptno;
end;
/
```

INTO clause to retrieve values

- To retrieve values into variables, we use the **INTO clause of EXECUTE IMMEDIATE** statement :

```
declare
        eid number:=7900;
        edata emp%rowtype;
begin
        execute immediate 'select * from emp where empno=:1' into edata using eid;
        dbms_output.put_line('employee id : '||edata.empno||' sal is : '||edata.sal);

        eid:=7902;
        execute immediate 'select * from emp where empno=:1' into edata using eid;
        dbms_output.put_line('employee id : '||edata.empno||' sal is : '||edata.sal);
exception
        when no_data_found then
                dbms_output.put_line('no such employee ');
end;
/
```

PLSQL block using EXECUTE IMMEDIATE

- Use **of EXECUTE IMMEDIATE** to generate a PLSQL block dynamically :

```
create or replace procedure addone(pempno number, psal number) is
begin
        insert into mybonus1 values(pempno, psal);
end;
/

declare
        plsql_block varchar2(500);
begin
        plsql_block := 'begin addone(105, 15000); end;';
        execute immediate plsql_block;
end;
/
```

PLSQL block using EXECUTE IMMEDIATE ….contd

- Use **of EXECUTE IMMEDIATE** to generate a PLSQL block dynamically :

```
declare
        plsql_block varchar2(500);
begin
        plsql_block := 'begin addone(:a, :b); end;';
        execute immediate plsql_block using 7905, 18000;
end;
/
```

RETURNING INTO clause to retrieve values through DMLs

- To retrieve values into variables from DMLs, we use the **RETURNING INTO clause of EXECUTE IMMEDIATE** statement :

```
declare
        sql_stmt varchar2(100):='update emp set sal = sal+2000 where id = :1 returning sal into :2';
        emp_id number:=&eno;
        bon number;
begin
        execute immediate sql_stmt using emp_id returning into bon;
        dbms_output.put_line(bon);
end;
/
```

The ‘Returning into’ clause returns the new updated salary into the variable bon.

RETURNING INTO clause to retrieve values through DMLs …contd

- To retrieve values into variables from DMLs, we use the **RETURNING INTO clause of EXECUTE IMMEDIATE** statement :

```
declare
        emp_id number:=&eno;
        emp_id1 number;
        bon number;
begin
        execute immediate 'delete from emp where id=:1 returning empno, sal into :2,:3' using emp_id
        returning into emp_id1, bon;
        dbms_output.put_line('deleted employee is :'||emp_id1||' and his salary is : '||bon);
end;
/
```

Stored Procedure generating a Dynamic SQL statement

- Stored Procedure to accept the name of a database table (such as 'emp') and an optional where-clause condition (such as 'sal > 2000'). if you omit the condition, the procedure deletes all rows from the table. Otherwise, the procedure deletes only those rows that meet the condition :

```
create procedure delete_rows( table_name in varchar2,  condition in varchar2 default null) as
        where_clause varchar2(100) := ' where ' || condition;
begin
        if condition is null then
                where_clause := null;
        end if;
        execute immediate 'delete from ' || table_name || where_clause;
end;
/

exec delete_rows('salgrade','grade>=4');
```

USING clause with IN, OUT and IN OUT modes

- With the USING clause, you need not specify a parameter mode for input bind arguments because the default mode is IN. However, OUT and IN OUT modes can be specified as per situation.

```
declare
        sql_stmt varchar2(200);
        my_empno number:= 909;
        my_ename varchar2(30);
        my_comm number;
        my_sal number := 3250;
        mynewsal number;
begin
        sql_stmt := 'update emp set sal = :1 where empno = :2 returning ename, comm, sal  into :3, :4,:5';
        /* bind returned values through USING clause. */
        execute immediate sql_stmt using my_sal, my_empno, out my_ename, out my_comm, out mynewsal;
        dbms_output.put_line(my_ename||my_comm||mynewsal);
end;
/
```

USING clause with IN, OUT and IN OUT modes ……contd

- When appropriate, you must specify the OUT or IN OUT mode for bind arguments passed as parameters. For example, suppose you want to call the following standalone procedure:

```
create sequence deptno_seq start with 51 increment by 1

CREATE or replace PROCEDURE create_dept(deptno IN OUT NUMBER, dname IN VARCHAR2, loc IN VARCHAR2)  AS
BEGIN
If deptno<=50 then
        SELECT deptno_seq.NEXTVAL INTO deptno FROM dual;
End if;
        INSERT INTO dept VALUES (deptno, dname, loc);
END;
/
```

USING clause with IN, OUT and IN OUT modes ……contd

- To call the procedure from a dynamic PL/SQL block, you must specify the IN OUT mode(only IN or only OUT will give an error) for the bind argument associated with formal parameter deptno, as follows:

```
declare
        plsql_block VARCHAR2(500);
        new_deptno NUMBER(2);
        new_dname VARCHAR2(14) := 'ADVERTISING';
        new_loc VARCHAR2(13) := 'NEW YORK';
begin
        plsql_block := 'BEGIN create_dept(:a, :b, :c); END;';
        EXECUTE IMMEDIATE plsql_block USING IN OUT new_deptno, new_dname, new_loc;
        dbms_output.put_line(new_deptno);
end;
/
```

Using Cursor Variables with Dynamic SQL

- To associate a cursor-variable with a query using OPEN….FOR:

```
declare
        type empcurtyp is ref cursor; -- define weak ref cursor type
        emp_cv empcurtyp; -- declare cursor variable
        my_ename varchar2(15);
        my_sal number := 1000;
begin
        open emp_cv for 'select ename, sal from emp where sal > :s' using my_sal;
        -- open cursor variable
        loop
                fetch emp_cv into my_ename, my_sal; -- fetch next row
                exit when emp_cv%notfound; -- exit loop when last row is fetched
                dbms_output.put_line('Name : '||my_ename||' Salary : '||my_sal);
        end loop;
        close emp_cv; -- close cursor variable
end;
/
```

Using Cursor Variables with Dynamic SQL ……contd

➢ To associate a query accepted as string-parameter with a cursor-variable using OPEN….FOR:

```
Create or replace procedure myproc(pquery varchar2, psal emp.sal%type) is
        emp_cv sys_refcursor; -- declare cursor variable
        my_ename emp.ename%type;
        my_sal emp.sal%type;
begin
        open emp_cv for pquery using psal;
        loop
                fetch emp_cv into my_ename, my_sal; -- fetch next row
                exit when emp_cv%notfound; -- exit loop when last row is fetched
                dbms_output.put_line('Name : '||my_ename||' Salary : '||my_sal);
        end loop;
        close emp_cv; -- close cursor variable
end;
/

exec myproc('select ename, sal from emp where sal>:x',1000);
```

Using OBJECT-TYPE and VARRAY with Dynamic SQL

- To create an Object-type and a Varray Object-type :

```
create type person as object (name varchar2(25), age number);
/

Create or replace type myhobby is varray(10) of varchar2(25);
/
```

- Using dynamic SQL, you can write a package of procedures that uses these types, as follows:

```
create package teams as
        procedure create_table (tab_name varchar2);
        procedure insert_row (tab_name varchar2, p person, h myhobby);
        procedure print_table (tab_name varchar2);
end;
/
```

Using OBJECT-TYPE and VARRAY with Dynamic SQL …contd

```
create or replace  package body teams as
   procedure create_table (tab_name varchar2) is
   begin
            execute immediate 'create table ' || tab_name ||'(pers person, hobbs myhobby)';
   end;
   procedure insert_row (tab_name varchar2, p person, h myhobby) is
   begin
            execute immediate 'insert into ' || tab_name || ' values (:1, :2)' using p, h;
   end;
   procedure print_table (tab_name varchar2) is
            type refcurtyp is ref cursor;
            cv refcurtyp;
            p person;
            h myhobby;
   begin
            open cv for 'select pers, hobbs from ' || tab_name;
```

Using OBJECT-TYPE and VARRAY with Dynamic SQL …contd

```
        loop
                fetch cv into p, h;
                exit when cv%notfound;
                        Dbms_output.put_line('Customer Age :'|| p.age||' Customer Name :' || p.name);
                For y in 1..h.count
                Loop Dbms_output.put_line('Hobbies '||y||' '||h(y)); End loop;
        end loop;
        close cv;
  end; end;
/
```

- Grant the below mentioned privilege or else you wont be able to create a table thru native dynamic sql through a stored or packaged procedure.

```
Grant create any table to scott;
```

- Execute the packaged-procedures as follows :

```
execute teams.create_table('a');
execute teams.insert_row('a' ,person ('Hemant', 28), myhobby('cricket', 'hockey', 'footbal'));
execute teams.print_table('a');
```

Using BULK FETCH and BULK COLLECT with Dynamic SQL

```
DECLARE
        TYPE EmpCurTyp IS REF CURSOR;
        TYPE NumList IS TABLE OF NUMBER;
        TYPE NameList IS TABLE OF VARCHAR2(15);
        emp_cv EmpCurTyp;
        empnos NumList;
        sals NumList;
        enames NameList;
        a number;
        ctr number:=1;
 BEGIN
        OPEN emp_cv FOR 'SELECT empno, ename FROM emp';
        --following statement is implementation of BULK…FETCH
        FETCH emp_cv BULK COLLECT INTO empnos, enames;
        CLOSE emp_cv;
```

Using BULK FETCH and BULK COLLECT with Dynamic SQL … contd

```
        --following statement is implementation of BULK…COLLECT
        EXECUTE IMMEDIATE 'SELECT sal FROM emp'  BULK COLLECT INTO sals;
        a:=empnos.count;
        While ctr<=a loop
                Dbms_output.put_line(empnos(ctr)||' '||enames(ctr)||' '||sals(ctr));
                ctr:=ctr+1;
        End loop;
END;
/
```

Using RETURNING BULK COLLECT with Dynamic SQL

```
DECLARE
        TYPE NameList IS TABLE OF VARCHAR2(15);
        enames NameList;
        comm_amt NUMBER := 500;
        sql_stmt VARCHAR(200);
        a number;
        ctr number:=1;
BEGIN
        sql_stmt := 'UPDATE emp SET comm = :1 where deptno=30 RETURNING ename INTO :2';
        EXECUTE IMMEDIATE sql_stmt USING comm_amt RETURNING BULK COLLECT INTO enames;
        a:=enames.count;
        While ctr<=a loop
                Dbms_output.put_line(enames(ctr));
                ctr:=ctr+1;
        End loop;
END;
/
```

Using BULK BIND and BULK COLLECT with Dynamic SQL

```
DECLARE
        TYPE NumList IS TABLE OF NUMBER;
        TYPE NameList IS TABLE OF VARCHAR2(15);
        empnos NumList; enames NameList;
        a number;  ctr number:=1;
BEGIN
        empnos := NumList(7900,7902, 7369,7844,7876,7788,7782);
        --Implementation of BULK FOR ALL using the USING clause
        FORALL i IN 1..5
        EXECUTE IMMEDIATE 'UPDATE emp SET sal = sal * 1.1 WHERE empno = :1 RETURNING ename INTO :2'
        USING empnos(i) RETURNING BULK COLLECT INTO enames;
        a:=enames.count;
        While ctr<=a loop
                Dbms_output.put_line(enames(ctr));
                ctr:=ctr+1;
  End loop;
END;
/
```

Use of Concatenation for Object/Column names

- Object names/column names in the dynamic sql string have to be appended using the concatenation operator.

```
declare
        mytabx varchar2(10):='mytabx';
begin
        execute immediate 'create  table '||mytabx||'(x number, y char)';
End;
/
```

- Using clause cannot be used to replace object names/column names.

```
declare
        mytabx varchar2(10):='mytablex';
begin
        execute immediate 'create  table :1(x number, y char)' using trim(mytabx);
end;
/
```

1.16: Example

Use of Dynamic SQL to Disable Triggers

- Stored Procedure which will disable the triggers on the table-name passed as parameter to it :

```
CREATE OR REPLACE PROCEDURE DISABLE_TRIGGERS(TABLE_NAME VARCHAR2)  is
        A  NUMBER(4);
        B NUMBER(4);
BEGIN
        SELECT COUNT(*) INTO A FROM USER_TABLES WHERE TABLE_NAME
        =upper(DISABLE_TRIGGERS.TABLE_NAME);

        IF A=0 THEN
                DBMS_OUTPUT.PUT_LINE('NO SUCH TABLE EXISTS IN DATA DICTIONARY');
        ELSE
                SELECT COUNT(*) INTO B FROM USER_TRIGGERS WHERE  TABLE_NAME =
                upper(DISABLE_TRIGGERS. TABLE_NAME);
                DBMS_OUTPUT.PUT_LINE('TOTAL NO OF TRIGGERS ON GIVEN TABLE IS:'||B);
If b>0 then
EXECUTE IMMEDIATE 'ALTER TABLE '|| DISABLE_TRIGGERS. TABLE_NAME ||' DISABLE ALL
TRIGGERS';
```

Use of Dynamic SQL to Disable Triggers ……contd

```
        If b>0 then
                EXECUTE IMMEDIATE 'ALTER TABLE '|| DISABLE_TRIGGERS. TABLE_NAME ||'
                DISABLE ALL TRIGGERS';
        Else
                DBMS_OUTPUT.PUT_LINE('There are no triggers to disable');
        End if;
  END IF;
END;
/
```

Use of DBMS_SQL for Dynamic SQL

- DBMS_SQL is an Oracle-supplied Package which can be used to implement Dynamic SQL using the various procedures and functions of the said package

- Oracle8*i* introduces native dynamic SQL, an alternative to DBMS_SQL. Using native dynamic SQL, you can place dynamic SQL statements directly into PL/SQL blocks. In most situations, native dynamic SQL can replace DBMS_SQL. Native dynamic SQL is easier to use and performs better than DBMS_SQL.

- Implementation of Dynamic SQL using DBMS_SQL is lengthier and more complicated as compared to native dynamic SQL

- For better performance and something easier to use, EXECUTE IMMEDIATE(Native Dynamic SQL) is preferred over DBMS_SQL

Procedures and Functions of DBMS_SQL for Dynamic SQL

OPEN_CURSOR

To process a SQL statement, you must have an open cursor. When you call the OPEN_CURSOR function, you receive a cursor ID number for the data structure representing a valid cursor maintained by Oracle. These cursors are used only by the DBMS_SQL package.

PARSE

Every SQL statement must be parsed by calling the PARSE procedure. Parsing the statement checks the statement's syntax and associates it with the cursor in your program.

You can parse any DML or DDL statement.

DDL statements are run on the parse, which performs the implied commit.

DEFINE_COLUMN

The columns of the row being selected in a SELECT statement are identified by their relative positions as they appear in the select list, from left to right. For a query, you must call one of the define procedures (DEFINE_COLUMN, DEFINE_COLUMN_LONG, or DEFINE_ARRAY) to specify the variables that are to receive the SELECT values, much the way an INTO clause does for a static query.

Procedures and Functions of DBMS_SQL for Dynamic SQL … contd

- **BIND_VARIABLE**

  Many DML statements require that data in your program be input to Oracle. When you define a SQL statement that contains input data to be supplied at runtime, you must use placeholders in the SQL statement to mark where data must be supplied.

  For each placeholder in the SQL statement, you must call one of the bind procedures, BIND_VARIABLE or BIND_ARRAY, to supply the value of a variable in your program (or the values of an array) to the placeholder.

- **EXECUTE**

Call the EXECUTE function to run your SQL statement.

- **FETCH_ROWS**

The FETCH_ROWS function is used to fetch the current row.

- **COLUMN_VALUE**

Call COLUMN_VALUE to read the values of the columns of the fetched row into variables.

- **CLOSE_CURSOR**

When you no longer need a cursor for a session, close the cursor by calling CLOSE_CURSOR.

Use of DBMS_SQL to delete rows

- Procedure that deletes all of the employees from the EMP table whose salaries are greater than the salary passed as parameter using DBMS_SQL :

```
CREATE OR REPLACE PROCEDURE demo(salary IN NUMBER) AS
        cursor_name INTEGER;
        rows_processed INTEGER;
BEGIN
        cursor_name := dbms_sql.open_cursor;
        dbms_output.put_line(cursor_name);
        DBMS_SQL.PARSE(cursor_name, 'DELETE FROM emp WHERE sal > :x', dbms_sql.native);
        DBMS_SQL.BIND_VARIABLE(cursor_name, ':x', salary);
        rows_processed := dbms_sql.execute(cursor_name);
        DBMS_SQL.close_cursor(cursor_name);
        Dbms_output.put_line(rows_processed);
EXCEPTION
WHEN OTHERS THEN
```

Use of DBMS_SQL to delete rows ….contd

```
EXCEPTION
                    WHEN OTHERS THEN
                              DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
/
 exec demo(4000);
```

- The 3rd parameter of DBMS_SQL.PARSE, that is, ‘dbms_sql.native’, which specifies the language flag. The “NATIVE” setting means that behavior of working with Dynamic SQL using DBMS_SQL should be according to the current version of the Oracle Database.
- language_flag: Determines how Oracle handles the SQL statement. The following options are recognized:
  - V6 (or 0) specifies version 6 behavior.
  - NATIVE (or 1) specifies normal behavior for the database to which the program is connected.
  - V7 (or 2) specifies Oracle database version 7 behavior.

Use of DBMS_SQL to execute simple SQL commands

```
CREATE OR REPLACE PROCEDURE exec(STRING IN varchar2) AS
        cursor_name INTEGER;
        ret INTEGER;
BEGIN
        cursor_name := DBMS_SQL.OPEN_CURSOR;
        dbms_output.put_line(cursor_name);
        DBMS_SQL.PARSE(cursor_name, string, DBMS_SQL.native);
        ret := DBMS_SQL.EXECUTE(cursor_name);
        dbms_output.put_line(ret);
        DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
/
```

Use of DBMS_SQL to execute simple SQL commands …..contd

- Simple SQL statements can be dynamically generated at runtime by the calling the EXEC procedure. The SQL statement can be a DDL statement or a DML without binds.

- After creating the EXEC procedure, you could make the following calls:

```
exec exec('delete from emp');

exec exec('Drop table employee');

exec exec('delete from dept');

exec exec('Drop table dept');
```

Use of Procedures and Functions of DBMS_SQL

➢Procedure to copy rows from EMPSOURCE table to EMPTARGET table :

```
Create table empsource as select empno id, ename name, hiredate birthdate from emp;
Create table emptarget as select empno id, ename name, hiredate birthdate from emp where 1=0;

CREATE OR REPLACE PROCEDURE copy(source IN VARCHAR2, destination IN VARCHAR2) IS
        id_var NUMBER;
        name_var VARCHAR2(30);
        birthdate_var DATE;
        source_cursor INTEGER;
        destination_cursor INTEGER;
        ignore INTEGER;
        a number;
```

Use of Procedures and Functions of DBMS_SQL ….cont

```
BEGIN
        -- Prepare a cursor to select from the source table:
        source_cursor := dbms_sql.open_cursor;
        DBMS_SQL.PARSE(source_cursor, 'SELECT id, name, birthdate FROM ' || source,
        DBMS_SQL.native);
        DBMS_SQL.DEFINE_COLUMN(source_cursor, 1, id_var);
        DBMS_SQL.DEFINE_COLUMN(source_cursor, 2, name_var, 30);
        DBMS_SQL.DEFINE_COLUMN(source_cursor, 3, birthdate_var);
        ignore := DBMS_SQL.EXECUTE(source_cursor);
        dbms_output.put_line('row processed '|| ignore);
                -- returns 0
        -- Prepare a cursor to insert into the destination table:
        destination_cursor := DBMS_SQL.OPEN_CURSOR;
        DBMS_SQL.PARSE(destination_cursor, 'INSERT INTO ' || destination || ' VALUES (:id_bind,
        :name_bind, :birthdate_bind)', DBMS_SQL.native);
```

Use of Procedures and Functions of DBMS_SQL ….cont

```
-- Fetch a row from the source table and insert it into the destination table:
LOOP
        A:= DBMS_SQL.FETCH_ROWS(source_cursor);
        IF a>0 THEN
                dbms_output.put_line(a);
                        --prints 1 for each fetch
                Dbms_output.put_line('hi'||id_var||name_var||birthdate_var);
                        -- get column values of the row
                DBMS_SQL.COLUMN_VALUE(source_cursor, 1, id_var);
                DBMS_SQL.COLUMN_VALUE(source_cursor, 2, name_var);
                DBMS_SQL.COLUMN_VALUE(source_cursor, 3, birthdate_var);
                Dbms_output.put_line('hi'||id_var||name_var||birthdate_var);
```

Use of Procedures and Functions of DBMS_SQL ….cont

```
                /* Bind the row into the cursor that inserts into the destination table.*/
                /* You could alter this example by inserting an if condition before the bind, if required*/
                DBMS_SQL.BIND_VARIABLE(destination_cursor, ':id_bind', id_var);
                DBMS_SQL.BIND_VARIABLE(destination_cursor, ':name_bind', name_var);
                DBMS_SQL.BIND_VARIABLE(destination_cursor, ':birthdate_bind', birthdate_var);
                ignore := DBMS_SQL.EXECUTE(destination_cursor);
        ELSE
                -- No more rows to copy:
                EXIT;
        END IF;
        END LOOP;
-- Commit and close all cursors:
COMMIT;
DBMS_SQL.CLOSE_CURSOR(source_cursor);
DBMS_SQL.CLOSE_CURSOR(destination_cursor);
```

Use of Procedures and Functions of DBMS_SQL ….cont

```
  EXCEPTION
        WHEN OTHERS THEN
              IF DBMS_SQL.IS_OPEN(source_cursor) THEN
                    DBMS_SQL.CLOSE_CURSOR(source_cursor);
              END IF;
              IF DBMS_SQL.IS_OPEN(destination_cursor) THEN
                    DBMS_SQL.CLOSE_CURSOR(destination_cursor);
              END IF;
              RAISE;
END;
/

exec copy('empsource', 'emptarget');
```

BULK Array Binds with DMLs using DBMS_SQL

In a DELETE statement you could bind in an array in the WHERE clause and have the statement be run for each element in the array:

```
declare
        stmt varchar2(200);
        dept_no_array dbms_sql.Number_Table;
        c number;
        dummy number;
begin
        dept_no_array(1) := 10;
        dept_no_array(2) := 20;
        dept_no_array(3) := 30;
        dept_no_array(4) := 40;
        dept_no_array(5) := 50;
        dept_no_array(6) := 60;
```

BULK Array Binds with DMLs using DBMS_SQL ….cont

```
stmt := 'delete from emp where deptno = :dept_array';
c := dbms_sql.open_cursor;
dbms_output.put_line('cursor ' ||c);
dbms_sql.parse(c, stmt, dbms_sql.native);
dbms_sql.bind_array(c, ':dept_array', dept_no_array, 1, 4);
dummy := dbms_sql.execute(c);
dbms_output.put_line('dummy ' || dummy);
dbms_sql.close_cursor(c);
```

BULK Array Binds with DMLs using DBMS_SQL ….cont

```
exception
        when others then
                if dbms_sql.is_open(c) then
                        dbms_sql.close_cursor(c);
                end if;
                raise;
end;
/
```

- In the example above, only elements 1 through 4(optionally) are used as specified by the bind_array call. Each element of the array potentially deletes a large number of employees from the database.

BULK Array Binds with DMLs using DBMS_SQL ….cont

➢Following is an example of a bulk INSERT statement :

create table newemp as select empno, ename from emp where 1=0;

```
declare
        stmt varchar2(200);
        empno_array dbms_sql.Number_Table;
        empname_array dbms_sql.Varchar2_Table;
        c number;
        dummy number;
begin
        for i in 0..9 loop
                empno_array(i) := 1000 + i;
                empname_array(i) := 'Emp'||i;
        end loop;
        stmt := 'insert into newemp values(:num_array, :name_array)';
```

BULK Array Binds with DMLs using DBMS_SQL ….cont

```
        c := dbms_sql.open_cursor;
        dbms_output.put_line('cursor ' ||c);
        dbms_sql.parse(c, stmt, dbms_sql.native);
        dbms_sql.bind_array(c, ':num_array', empno_array);
        dbms_sql.bind_array(c, ':name_array', empname_array);
        dummy := dbms_sql.execute(c);
        dbms_output.put_line('dummy ' || dummy);
        dbms_sql.close_cursor(c);
exception
        when others then
                if dbms_sql.is_open(c) then dbms_sql.close_cursor(c); end if;
                raise;
end;
/
```

When the execute takes place, all 10 of the employees are inserted into the newemp table.

BULK Array Binds with DMLs using DBMS_SQL ….cont

➢Following is an example of an bulk UPDATE statement.

```
declare
        stmt varchar2(200);
        emp_no_array dbms_sql.Number_Table;
        emp_name_array dbms_sql.Varchar2_Table;
        c number;
        dummy number;
begin
        for i in 0..9 loop
                emp_no_array(i) := 1000 + i;
                emp_name_array(I) := 'newemp'||i;
        end loop;
        stmt := 'update newemp set ename = :name_array where empno = :num_array‘;
        c := dbms_sql.open_cursor;
        dbms_output.put_line('cursor ' || c);
```

BULK Array Binds with DMLs using DBMS_SQL ….cont

```
        dbms_sql.parse(c, stmt, dbms_sql.native);
        dbms_sql.bind_array(c, ':num_array', emp_no_array);
        dbms_sql.bind_array(c, ':name_array', emp_name_array);
        dummy := dbms_sql.execute(c);
        dbms_output.put_line('dummy ' || dummy);
        dbms_sql.close_cursor(c);
exception
  when others then
        if dbms_sql.is_open(c) then dbms_sql.close_cursor(c); end if;
        raise;
end;
/
```

The two collections are always stepped in unison. If the WHERE clause returns more than one row, then all those employees get the name the emp_name_array happens to be pointing to at that time.

Collection Types provided by DBMS_SQL

- Collection types which can be used as data-types to delcare collection variables in PLSQL :

```
type Number_Table IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
type Varchar2_Table IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
type Date_Table IS TABLE OF DATE INDEX BY BINARY_INTEGER;
type Blob_Table IS TABLE OF BLOB INDEX BY BINARY_INTEGER;
type Clob_Table IS TABLE OF CLOB INDEX BY BINARY_INTEGER;
type Bfile_Table IS TABLE OF BFILE INDEX BY BINARY_INTEGER;
type Urowid_Table IS TABLE OF UROWID INDEX BY BINARY_INTEGER;
```