PL/SQL

Date: 25/5/15

1. **PL/SQL Introduction:**
➔ Select……into clause
➔ Variable attribute(%type,%type)
➔ Conditional control statements
➔ Bind variables
2. **CURSORS:**
➔ Implicit cursors
➔ Explicit cursors
➔ Explicit cursor life cycle
➔ Cursor for loop
➔ Parameterized cursors
➔ Update, Delete, Statements are used in cursors(without using where current of, for update clauses)
➔ Implicit cursor attributes.
3. **EXCEPTIONS:**
➔ Predefined Exceptions
➔ User defined exceptions
➔ Unnamed exceptions
➔ Exception propagation
➔ Error trapping functions(sqlcode, sqlerrm)
➔ Raise_application_error()
4. **SUB PROGRAMS:**
   STORED PROCEDURES
➔ Parameter modes(in, out, in out)
➔ No copy
➔ Autonomous transactions
➔ Authid current_user
   STORED FUNCTIONS
➔ DML operations used in functions
➔ SELECT…..into clause used in functions
➔ Cursors used in functions
5. **TRIGGERS:**
➔ Row level triggers
➔ Row level trigger applications
➔ Statement level triggers
➔ Trigger execution order
➔ Compound triggers(11g)
➔ Follows clause(11g)
➔ Mutating error
➔ System level triggers(or) DDL triggers

6. **PACKAGES:**
→ Global variables
→ Serially_reusable pragma
→ Overloading procedures
→ Forward declaration

7. **TYPES USED IN PACKAGES:**
→ PL/SQL record
→ Index by table(or) PL/SQL table(or) Associative Array
→ Nested table
→ V array

8. **BULK BIND:**
→ Bulk collect clause
→ For all statements
→ Indices of clause(10g)
→ sql%bulk_rowcount()
→ Bulk exceptions handling through sql%bulk_exceptions
→ Error_code, error_index.

9. **REF CURSOR:**
→ Strong REF cursor
→ Weak REF cursor
→ sys_ref cursor
→ Passing sys_ref cursor as parameter to the stored procedure.

10. **LOCAL SUBPROGRAMS:**
→ Local Procedures
→ Local Functions
→ Passing ref cursor as parameter to the local sub programs
→ Passing types as parameter to the local sub programs.

11. **UTL_FILE PACKAGE**

12. **SQL* LOADER:**
→ Flat files, control files, bad files, discard files, log files.

13. **LARGE OBJECT(LOBS):**
→ Internal Large Objects(clob, blob)
→ External Large Objects(bfile)

14. **Avoiding mutating error using compound Trigger**

15. **Where "current of", "for update of" clauses are used in cursors.**

16. **Member Procedures, Member Functions**

17. **Dynamic SQL**

18. **PREDEFINED PACKAGES:**
→ dbms_flashback
→ dbms_job
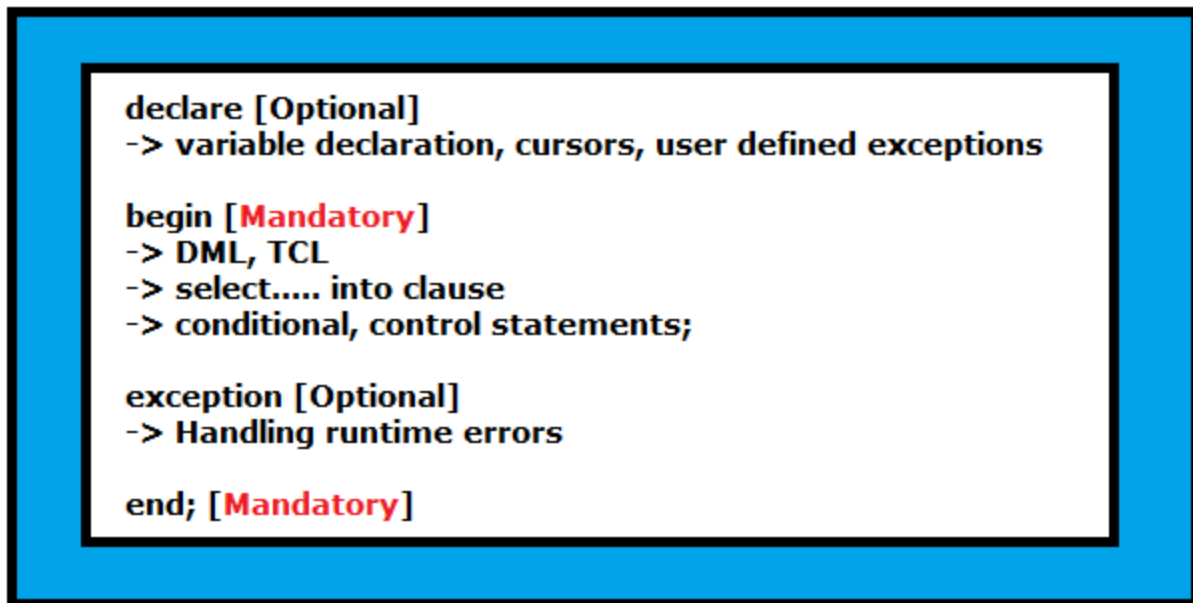→ dbms_pipe

19. **11G Features**

PL/SQL:

PL/SQL is a Procedural Language Extension for SQL. Oracle 6.0 introduced PL/SQL.

1. Oracle 6.0----→ PL/SQL 1.0
2. Oracle 7.0----→ PL/SQL 2.0
3. Oracle 8.0----→ PL/SQL 8.0
4. -------------------------------
5. Oracle 11.2----→ PL/SQL 11.2

Basically, PL/SQL is an Block Structured Programming Language. When we are submitting PL/SQL blocks into the Oracle server then all Procedural statements are executed within PL/SQL engine and also all SQL statements are separately executed by using SQL engine.
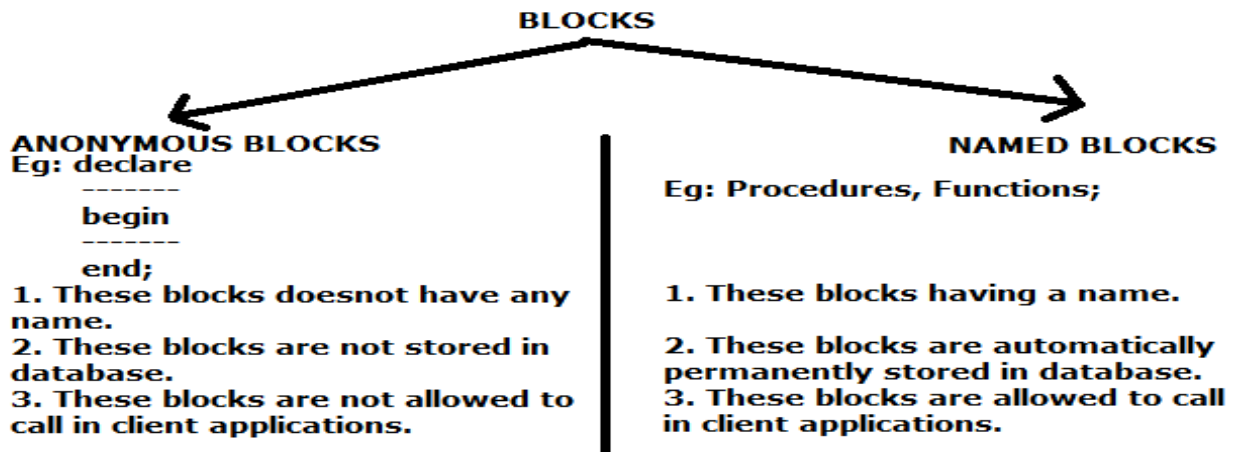
BLOCK STRUCTURE

```
declare [Optional]
-> variable declaration, cursors, user defined exceptions

begin [Mandatory]
-> DML, TCL
-> select..... into clause
-> conditional, control statements;

exception [Optional]
-> Handling runtime errors

end; [Mandatory]
```

Date: 26/5/15

PL/SQL having two types of blocks.
1. Anonymous blocks
2. Named blocks

3

BLOCKS

ANONYMOUS BLOCKS
Eg: declare
----
begin
----
end;
1. These blocks doesnot have any name.
2. These blocks are not stored in database.
3. These blocks are not allowed to call in client applications.

NAMED BLOCKS
Eg: Procedures, Functions;

1. These blocks having a name.
2. These blocks are automatically permanently stored in database.
3. These blocks are allowed to call in client applications.

Declaring a Variable:
Syntax: variablename datatype(size)
Eg:
sql> declare
     a number(10);
     b varchar2(10);
Storing a value into variable:
Using assignment operator(:=) we can store a value in variable.
Syntax: variablename:=value;
Eg:
Sql> declare
     a number(10);
     begin
     a:=50;
     end;
     /

Display a message (or) variable name:
Syntax: dbms_output.put_line('message');
(OR)
Syntax: dbms_output.put_line(variablename);
dbms_out---→ PACKAGE NAME
putput_line--→ PROCEDURE NAME

Eg: sql> set serveroutput on;
Sql> begin
dbms_output.put_line('welcome');
end;
/
OUTPUT: welcome
Eg: sql> declare
   a number(10);   (Or)   a number(10):=50; (it is also possible)
   begin

4

```
a:=50;
dbms_output.put_line(a);
end;
/
```
SELECT..... into clause:

Select.....into clause is used to retrieve data from table and store it into PL/SQL variable.
Select......into clause always returns "Single record (or) Single value" at a time.
Syntax:
select columnname1, columnname2……. into variablename1, variablename2,…..
from tablename where condition;
(where condition must return a single record)
Select…. into clause is used in executable section of the PL/SQL block.
Q) Write PL/SQL program for user entered empno then display name of the employee and his salary from emp table.
Ans: sql> declare
```
        v_ename varchar2(20);
        v_sal number(10);
        begin
                select ename, sal into v_ename, v_sal from emp
                where empno= &employeeno;
                dbms_output.put_line(v_ename||' `||v_sal);
          end;
          /
```
OUTPUT: Enter value for employee no: 7369
SMITH   800

Date: 27/5/15

NOTE: In PL/SQL when a variable contains "not null" (or) "constant" clause then we must assign the value when we are declaring the variable in declare section of the PL/SQL block.
Syntax:
Variablename datatype(size) not null:= value;
Syntax:
variablename constant datatype(size):= value;
Eg:
```
Sql> declare
    a number(10) not null:=50;
    b constant number(10) :=9;
   begin
        dbms_output.put_line(a);
        dbms_output.put_line(b);
   end;
   /
```
Output: 50   9

NOTE: We can also use "default" clause in place of "assignment operator" when we are assigning the value in declare section of the PL/SQL block.

Eg:
```
Sql> declare
        a number default 50;
    begin
    `        dbms_output.put_line(a);
    end;
    /
```
Output: 50

Q) Write a PL/SQL program maximum salary from emp table and store it into PL/SQL variable and display max salary.

ANS: sql> set serveroutput on;
```
Sql> declare
        v_sal number(10);
    begin
        select max(sal) into v_sal from emp;
        dbms_output.put_line('maximum salary'||v_sal);
    end;
    /
```
Output: maximum salary: 5000

Eg:
```
Sql> declare
    a number(10);
    b number(10);
    c number(10);
    begin
    a:= 90;
    b:= 30;
    c:= greates(a,b)     c:=max(a,b)[error: group function cannot used in PL/SQL expre]
    dbms_output.put_line(c);
    end;
/
```
OUTPUT: 90

NOTE: In PL/SQL expressions we are not allowed to use "Group Functions", "Decode conversion Function". But we are allowed to use Number Functions, Character Functions, Decode Functions, Date Conversion Functions in PL/SQL expressions.

Eg 1: sql> declare
```
        a varchar2(10);
      begin
        a:= upper('dinesh');
        dbms_output.put_line(a);
```

```
            end;
            /
```
Output: DINESH

Eg 2: sql> declare
```
            a date;
            begin
             a:= next_date('1 ug-15')+1;
             dbms_output.put_line(a);
            end;
            /
```
Output: 13-AUG-15

VARIABLE ATTRIBUTES(Anchor Notations):

Variable attributes are used in place of data types in variable declaration. Whenever we are using variable attributes Oracle server automatically allocates memory for the variables based on the corresponding column data type in a table.

PL/SQL having two types of variable attributes.

1. Column level Attributes
2. Row level Attributes

1. **Column Level Attributes:** In this methos we are defining attributes for individual columns. Column Level attributes are represented by using **"%type"**.

   Syntax: variablename tablename. Columnname %type;

   Whenever we are using "column level attributes" oracle server automatically allocates memory for the variables as same as corresponding column data type in a table.

   Eg: sql> declare
```
            v_ename  emp.ename%type;
            v_sal emp.sal %type;
            v_hiredate emp.hiredate %type;
         begin
            select ename, sal, hiredate into v_ename, v_sal, v_hiredate
            from emp where empno= &no;
            dbms_output.put_line(v_ename||'`||v_sal||'`||v_hiredate);
         end;
         /
```
Output: Enter value for no: 7902
FORD   3000   03-DEC-81

2. **Row Level Attributes:** In this methods a single variable can represent all different data types in a row within a table. This variable is also called as "**Record Type Variable**". It is also same as structures in "C" language. Row level attributes are represented by **"%rowtype".**

   Syntax: variablename tablename %rowtype;

7

Eg: sql> declare
     i emp%rowtype;
   begin
    select ename, sal, hiredate into i.ename, i.sal, i.hiredate
    from emp where empno=&no;
    dbms_output.put_line(i.ename||' `||i.sal||' `||i.hiredate);
   end;
  /

Output: Enter value for no: 7902
FORD   3000   03-DEC-81

Example:
sql> declare
     i emp%rowtype;
  begin
    select * into i from emp
    where empno=&no;
    dbms_output.put_line(i.ename||' `||i.sal||' `||i.hiredate||' `||i.deptno);
  end;
  /

Output: Enter value for no: 7902
FORD   3000   03-DEC-15

	empno	ename	job	MGR	hiredate	sal	comm	deptno
i	7902	FORD	Analyst	7839	03-DEC-81	3000	----	20

Date: 29/5/15

PL/SQL Data types and Variables:

1. It supports all SQL databases(scalar data types) + Boolean Data types.
2. Composite Data types
3. Ref Objects
4. Large Objects(lobs) -> clob, blob, bfile
5. Bind Variables (or) Non PL/SQL variables

BIND Variables:

It is an Session variable(when ever necessary we can create) created at "HOST" environment (Sql plus DOS prompt) that's why these variables are also called as "Host Variables". These variables are used in SQL, PL/SQL that's why these variables are also called as "Non PL/SQL variables". We can also use the variables in PL/SQL to execute when sub programs having OUT, INOUT parameters.

Step 1: Creating a BIND variable

Syntax: sql>variable variablename datatype;
Step 2: using BIND variable
Syntax: sql>:bindvariablename;
Step 3: display value from BIND variable.
Syntax: sql> print bindvariablename;
Eg:
sql> variable g number;
sql> declare
       a number(10):= 900;
  begin
       :g := a/2;
       end;
       /
PL/SQL procedure successfully completed.
Sql> print g; (or) sql> print :g;
Output: G

      450

Conditional Statements:
IF
1. **if**
2. **if-else**
3. **elsif**

1. **if:**
   Syntax: (olden day syntax style)
   if condition then
   stmts;
   end if;

2. **if-else:**
   Syntax:
   if condition then
       stmts;
   else
       stmts;
   end if;

3. **elsif:** To check more number of conditions then we are using "elsif".
   Syntax:
   if condition1 then
     stmts;
   elsif condition2 then
     stmts;

9

```
elsif condition3 then
   stmts;
else
   stmts;
end if;
```

Example:
```
Sql> set serveroutput on;
Sql> declare
   v_deptno dept.deptno%type;
     begin
          select deptno into v_deptno from dept
          where deptno = &deptno;
          if v_deptno=10
          dbms_output.put_line('ten');
          elsif v_deptno=20
          dbms_output.put_line('twenty');
           elsif v_deptno=30
          dbms_output.put_line('thirty');
          else
          dbms_output.put_line('others');
          end if;
      end;
    /
```
Output: Enter value for deptno: 20
         one
sql> /
Output: Enter value for deptno: 40
         others
Output: Enter value for deptno: 90
Error: ORA-01403: no data found.

NOTE 1: When a PL/SQL block contains "Select into" clause and also if requested data not available in a table, then Oracle server returns an error "ORA-01403: no data found."
NOTE 2: When a PL/SQL block contains pure DML statements and also through these statements if requested data not available in a table then Oracle server does not return any error message. To handle these type of blocks we are using "Implicit Cursor Attributes".
Eg: sql> begin
         delete from emp where ename='welcome';
       end;
        /
PL/SQL procedure successfully completed.

NOTE 3: In PL/SQL blocks whenever "Select into" clause try to return multiple records at a time (or) try to return multiple values in a columns at a time then Oracle server returns an error "ORA-1422: Exact fetch returns more than requested number of rows".
Eg:
sql> declare
   i emp%rowtype;
   begin
      select * into i from emp where deptno=10;
      dbms_output.put_line(i.ename||' '||i.sal||' '||i.job);
  end;
  /
Error: ORA-1422: Exact fetch returns more than requested number of rows.

Date: 30/5/15

2. CASE STATEMENT: Oracle 8.0, introduced case statement and also Oracle 8i introduced case conditional statement. This statement is also called as "Searched Case".
Syntax:
case variablename
when value1 then
stmts;
when value2 then
stmts;
…………
else stmt n;
end case;
Eg:
sql> declare
v_deptno number(10);
begin
select deptno into v_deptno from emp
where deptno= &deptno;
case v_deptno
when 10 then
dbms_output.put_line('ten');
when 20 then
dbms_output.put_line('twenty');
else
dbms_output.put_line('others');
end case;
end;
/
Output: Enter value for deptno=10
ten

Case Conditional Statement (OR) Searched Case(Oracle 8i):
Syntax:
case
when condition1 then
stmts;
when condition2 then
stmts;
else
stmts;
end case;

Example:
case
when v_deptno=10 then
dbms_output.put_line('ten');
when v_deptno between 20 and 30 then
dbms_output.put_line('twenty and thirty');
else
dbms_output.put_line('others');
end case;
end;
/
CONTROL STATEMENTS:
1. Simple Loop
2. While Loop
3. For Loop

1. **SIMPLE LOOP:** This loop is also called as "Infinite loop". Here body of the loop statements are executed repeatedly.
   Syntax:
   loop
   stmts;
   end loop;

   Example:
   Sql> begin
      loop
      dbms_output.put_line('welcome');
      end loop;
      end;
   If we want to exit from "Infinite loop" then we are using Oracle provided predefined methods.

Method 1: Default Method
Syntax:
exit when true condition;
Example:
Sql> declare
   n number(10):= 1;
   begin
       loop
       dbms_output.put_line(n);
       exit when n>=10;
       n:= n+1;
       end loop;
   end;
/

Method 2: (Using "IF")
Syntax:
if true condition then
exit;
end if;
Example:
Sql> declare
   n number(10) := 1;
   begin
       loop
       dbms_output.put_line(n);
       if n>=10 then
       exit;
       end if;
       n:= n+1;
       end loop;
   end;
   /

Method 3: (While Loop)
Here body of the loop statements are executed repeatedly until condition is false. In "While Loop" whenever condition is true then only loop body is executed.
Syntax:
while condition
loop
   stmts;
end loop;

Example:
```
Sql> declare
   n number(10) := 1;
   begin
        while n<=10
        loop
        dbms_output.put_line(n);
        n:= n+1;
        end loop;
   end;
   /
```
FOR LOOP:
Syntax:
```
for indexname in lowerbound..upperbound
loop
   stmts;
end loop;
```
Example:
```
Sql> declare
   n number(10);
   begin
        for    1..10
        loop
        dbms_output.put_line(n);
        end loop;
   end;
   /
```
Example:
```
Sql> declare
   n number(10);
   begin
        for n in reverse 1..10
        loop
        dbms_output.put_line(n);
        end loop;
   end;
   /
```

Example:
```
Sql> begin
   for i in 1..10
   loop
   dbms_output.put_line(i);
   end loop;
```

14

end;
/
NOTE: In "for loops" index variable internally behaves like a "Integer" variable that's why in this case not require to declare variable in declare section.
Example:
Sql> create table test(sno number(10));
Sql> begin
   for i in 1..10
   loop
   insert into test values(i);
   end loop
   end;
   /
Sql> select * from test;

CURSOR

Cursor is an Private SQL memory area which is used to process multiple records and also this is an "Record by Record" process.
All database systems having two types of cursors:
1. Implicit cursors
2. Explicit cursors.

Date: 1/6/15

1. **IMPLICIT CURSORS:** For SQL statements returns single record is called "Implicit Cursor". Implicit Cursor memory area is also called as "Context Area".
Whether PL/SQL block contains "Select….into" clause (or) pure DML statements then Oracle server internally automatically creates an memory area. This memory area is also called as "Implicit Cursor" (or) "SQL area" (or) "Context Area". This memory area returns single record when a PL/SQL block contains "Select…into" clause. This memory area also returns multiple records when a PL/SQL block contains pure DML statements. But these multiple records are processes at a a time by "SQL Engine".
Example: Example of Implicit Cursor.
Sql> declare
   v_ename varchar2(10);
    v_sal number(10);
    begin
       select ename, sal into v_ename, v_sal from emp where empno= &empno;
       dbms_output.put_line(v_ename||' '||v_sal);
    end;
    /
Output:
Enter value for empno: 7902
FORD 3000
EXPLICIT CURSOR: (Static Cursor)

For SQL statements returns multiple records is called "Explicit cursors". And also this is an "Record by Record" process.

Explicit cursor memory area is also called as "Active set area".

Explicit Cursor Life Cycle:

1. Declare
2. Open
3. Fetch
4. Close

1. **DECLARE:** In "Declare" section of the PL/SQL block we are defining the cursor using following.

   Syntax: cursor cursorname is select * from tablename where condition;[group by, having..etc….]

   Example:

   Sql> declare
       cursor c1 is select * from emp where sal>2000;

2. **OPEN:** In all databases whenever we are opening the cursor then only database servers retrieve data from table into cursor memory area. Because in all database systems when we are opening the cursors then only cursor "Select" statements are executed.

   Syntax: open cursorname;

   This statement is used in "Executable Section" of the PL/SQL block.

   NOTE: Whenever we are opening the cursor "implicit" cursor pointer always points to the **"FIRST RECORD"** in the cursor.

3. **FETCH:** (Fetching data from cursor)

   Using fetch statement we are fetching data from cursor into PL/SQL variables.

   Syntax: fetch cursorname into variablename1, variablename2,….;

4. **CLOSE:** Whenever we are closing the cursor all the resources allocated from cursor memory area is automatically released.

   Syntax: close cursorname;

   Example:

   Sql> declare
       cursor c1 is select ename, sal from emp;
       v_ename varchar2(20);
       v_sal number(10);
       begin
       open c1;
       fetch c1 into v_ename, v_sal;
       dbms_output.put_line(v_ename||' `||v_sal);
       fetch c1 into v_ename, v_sal;
       dbms_output.put_line('My Second Employee name is:'||v_ename);
       fetch c1 into v_ename, v_sal;
       dbms_output.put_line(v_ename||'high salary');

```
        close c1;
     end;
     /
```
Output:
```
SMITH   3000
MY SECOND EMPLOYEE NAME IS: ALLEN
WARD    HIGH SALARY
```

Date: 2/6/15

Explicit Cursor Attributes:
Every explicit cursor having "4" attributes. These are
1. %notfound
2. %found
3. %isopen
4. %rowcount

When we are using these attributes in PL/SQL block then we must specify cursor name along with these attributes.

Syntax: cursorname%attributename;

Except "%rowcount" all other cursor attributes returns "**Boolean Value**" either "True (or) False". Whereas "%rowcount" attribute always returns "**Number**" datatype.

1. **%NOTFOUND:** This attribute always returns Boolean value either true (or) false. When "**Fetch**" statement does not return any row then "%notfound" attribute returns true. If Fetch statements returns atleast one row then "%notfound" returns false.

   Syntax: cursorname%notfound;

   Q) Write PL/SQL cursor program to display all employee name, salary from emp table using %notfound attribute?

   ANS: sql> declare
```
            cursor c1 is select ename, sal from emp;
            v_ename varchar2(20);
            v_sal number(10);
            begin
            open c1;
            loop
            fetch c1 into v_ename, v_sal;
            exit when c1.%notfound;
            dbms_output.put_line(v_ename||' `||v_sal);
            end loop;
            close c1;
            end;
            /
```
   Q) Write PL/SQL program to display first "5" highest salary employees from emp table using "%rowcount" attribute?
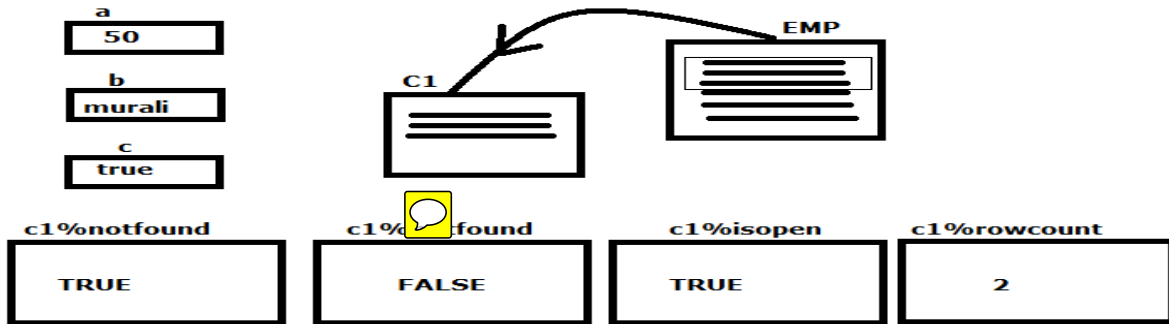
ANS: sql> declare

    cursor c1 is select ename, sal from emp;

    v_ename varchar2(20);

    v_sal number(10);

    begin

    open c1;

    loop

    fetch c1 into v_ename, v_sal;

    dbms_output.put_line(v_ename||' '||v_sal);

    exit when c1%rowcount>=5;

    end loop;

    close c1;

    end;

    /

Q) Write PL/SQL program to display even number of records from emp table using "%rowcount" attributes?

ANS: sql> declare

    cursor c1 is select ename, sal from emp;

    v_ename varchar2(20);

    v_sal number(10);

    begin

    open c1;

    loop

    fetch c1 into v_ename, v_sal;

    if mod(c1%rowcount,2)=0 then

    dbms_output.put_line(v_ename||' '||v_sal);

    end if;

    close c1;

    end;

    /

In Oracle, whenever we are creating a cursor then automatically "4" memory locations are allocated along with cursor memory area. These memory locations behaves like a variables. These variables also stores only one value at a time. These variables are identified through "Explicit Cursor Attributes".

Sql> declare

  a number(10);

  b varchar2(10);

  c boolean;

  begin

  a:= 50;

  b:= 'murali';

  c:= 'true'   end;

c1%notfound	c1%isfound	c1%isopen	c1%rowcount
TRUE	FALSE	TRUE	2

%ROWCOUNT: This attribute always returns number datatype i.e., it returns "**Number of Records Number"** fetched from the cursor.

Syntax: cursorname%rowcount;

Example:

Sql> declare
   cursor c1 is select ename, sal from emp;
   v_ename varchar2(10);
   v_sal number(10);
   begin
   open c1;
   fetch c1 into v_ename, v_sal;
   dbms_output.put_line(v_ename||' `||v_sal);
   fetch c1 into v_ename, v_sal;
   dbms_output.put_line(v_ename||' `||v_sal);
dbms_output.put_line('Number of Records Number fetched from cursor is:'||' `||c1%rowcount);
close c1;
end;
/

Output:

SMITH 1000
ALLEN 1600
Number of Records Number fetched from cursor is: 2

Date: 3/6/15

Q) Write a PL/SQL cursor program to display "5" record from emp table using "%rowcount" attribute.

ANS: sql> declare
      cursor c1 is select ename, sal from emp;
      v_ename varchar2(10);
      v_sal number(10);
      begin
      open c1;
      loop
      fetch c1 into v_ename, v_sal;

19

```
        exit when c1%notfound;
        if c1%rowcount=5 then
        dbms_output.put_line(v_ename||' `||v_sal);
        end if;
        end loop;
        close c1;
        end;
        /
Output: MARTIN 1250
```

NOTE: Using cursor we can also transfer data from Oracle table into another Oracle table, arrays, operating system files.

Q) Write a PL/SQL cursor program to transfer ename, sal who are getting more than 2000 salary emp table into another table?
ANS: sql> create table target(sno number(10), name varchar2(20), sal number(10));
```
sql> declare
    cursor c1 is select * from emp where sal>2000;
    i emp%rowtype;
    n number(10);
    begin
    open c1;
    loop
    fetch c1 into i;
    exit when c1%notfound;
    n:= c1%rowcount;
    insert into target value(i.name, i.sal);
    end loop;
    close c1;
    end;
    /
Output: select * from target;
```

Q) Write a PL/SQL cursor program to display total salary from emp table without using sum functions?
ANS: sql> declare
```
        cursor c1 select sal from emp;
        v_sal number(10);
        n number(10):=0;
        begin
        open c1;
        loop
        fetch c1 into v_sal;
        exit when c1%notfound;
```

```
        n:=n+v_sal;
        end loop;
        dbms_output.put_line('Total salary is:'||' '||n);
        close c1;
        end;
        /
```
Output: Total salary is: 29225.
NOTE: When a resource table have NULL value column and also when we are performing summation based on that column then we must use "NVL()" function.
Eg: n:= n+(nvl(v_sal,0));

%FOUND: This attribute also returns Boolean value either "true" or "false". It returns true when "fetch" statement returns atleast one row. It returns false when fetch statement does not return a row.
Syntax: cursorname%found;
Eg: sql> declare
```
        cursor c1 is select * from emp where ename='&ename';
        i emp%rowtype;
        begin
        open c1;
        fetch c1 into i;
        if c1%found then
        dbms_output.put_line(i.v_ename||' '||i.v_sal);
        else
        dbms_output.put_line(' your employee does not exist');
        end if;
        close c1;
        end;
        /
```
Output: Enter value for ename= SMITH
        SMITH 3000
Enter value for ename= abc
Your employee does not exist;

Q) Write a PL/SQL cursor program which display all employees and their salaries from emp table using %found attributes.
ANS: sql> declare
```
        cursor c1 is select ename, sal from emp;
        i emp%rowtype;
        begin
        open c1;
        fetch c1 into I;
        while(c1%found)
        loop
```

```
        dbms_output.put_line(i.ename||' `||i.sal);
        fetch c1 into i;
        end loop;
        close c1;
        end;
        /
```

Output:

Date:4/6/15

4. %ISOPEN": This attribute also returns Boolean value either true (or) false. This attribute returns true when cursor is already open otherwise it returns false.

Syntax: cursorname%isopen;

Eg: sql> declare
```
        cursor c1 is select * from emp;
        i emp%rowtype;
        begin
        if not c1%isopen then
        open c1;
        end if;
        loop
        fetch c1 into i;
        exit when c1%notfound;
        dbms_output.put_line(i.ename||' `||i.sal);
        end loop;
        close c1;
        end;
        /
```

Implicit Cursor Attribute Name	Return Values	
%found	TRUE	Fetch statement returns at least on row
	FALSE	Fetch statement dosenot return any row
%notfound	TRUE	Fetch statement does not return any row
	FALSE	Fetch statement returns at least one row
%isopen	TRUE	When cursor is already open
	FALSE	When cursor is not open
%rowcount	NUMBER	Returns number of records number fetched from the cursor

ELIMINATING EXPLICIT CURSOR LIFE CYCLE (OR) CURSOR "FOR" LOOPS:
(shortcut method of cursors)

Using cursor "For" loops we are eliminating explicit cursor life cycle. Here we are no need to use Open, Fetch, Close statements explicitly. Whenever we are using cursor "For" loops internally Oracle server only Open the cursor, Fetch data from the cursor and Close the cursor automatically.

Syntax:
for indexvariablename in cursorname
loop
   statements;
end loop;

➔ Data transferred from cursor to index variable

NOTE: In cursor "For" loops index variable internally behaves like a record type variable (%rowtype) cursor "For" loops are used in "Executable section" of the PL/SQL block cursor "For" loops are also called as "Shortcut method of the cursor.

Q) Write a PL/SQL cursor program to display all employee names and their salaries from emp table using cursor "For" loops?

ANS: sql> declare
          cursor c1 is select * from emp;
          begin
          for i in c1
          loop
          dbms_output.put_line(i.ename||' `|||i.sal);
          end loop;
          end;
          /

Output:

NOTE: We can also eliminate "Declare section" of the cursor using cursor "For" loops. In this case we are specifying cursor select statement in place of cursor name within cursor "For" loops.

Syntax:
for indexvariablename in (select statement)
loop
   statements;
end loop;

Eg: sql> begin
          for i in (select * from emp);
          loop
          dbms_output.put_line(i.ename||' `||i.sal);
          end loop;
          end;
          /

Q) Write a cursor program to display "5" record from emp table using cursor "For" loops?

23

ANS: sql> declare

　　　cursor c1 is select * from emp;

　　　begin

　　　for i in c1

　　　loop

　　　if c1%rowcount=5 then

　　　dbms_output.put_line(i.ename||' `||i.sal);

　　　end if;

　　　end loop;

　　　end;

　　　/

Output: MARTIN　　　1250

Q) Write a PL/SQL cursor program which displays total salary from emp table without using sum function with using cursor "For" loops?

ANS: sql> declare

　　　cursor c1 is select * from emp;

　　　n number(10):=0;

　　　begin

　　　for i in c1

　　　loop

　　　n:= n+i.sal;

　　　end loop;

　　　dbms_output.put_line('Total Salary is:'||n);

　　　end;

　　　/

Output: Total Salary is: 29250

Example:

Sql> declare

　　cursor c1 is select * from emp;

　　begin

　　for i in c1

　　loop

　　if i.sal>2000 then

　　dbms_output.put_line(i.ename||' `||i.sal);

　　else if

　　dbms_output.put_line(i.ename||' `||'Low Salary');

　　end if;

　　end loop;

　　end;

　　/

Output:

PARAMETERIZED CURSOR: In Oracle, we can also pass parameters to the cursor. These type of cursors are also called as "Parameterized Cursors". In "Parameterized Cursors" always we are defining for parameters in cursor declaration and also pass actual parameters when we are opening the cursor.

Syntax: cursor cursorname(parametername datatype)

   is select * from tablename where columnname= parametername;

parameter name -→ Formal Parameter

Syntax: open cursorname(actual parameters);

NOTE: In Oracle, when we are defining Formal parameters in cursors, procedures, functions then we are not allowed to use data type "size" in formal parameter declaration.

Eg: sql> declare
    cursor c1(p_deptno number) is select * from emp where deptno= p_deptno;
    i emp%rowtype;
    begin
    open c1(10);
    loop
    fetch c1 into i;
    exit when c1%notfound;
    dbms_output.put_line(i.ename||' `||i.sal||' `||i.deptno);
    end loop;
    close c1;
    end;
    /

Output: CLARK   3550   10
         KING   6200   10
         MILLER   3100   10

Date: 5/6/15

Q) Write a PL/SQL cursor program using parameterized cursor for passing job as a parameter from emp table then display the employees working as CLERK's and ANALYST's and also display output statically by following format?
Employees working as CLERKS
SMITH
ADAMS
JAMES
MILLER
Employees working as ANALYSTS
SCOTT
FORD
ANS: sql> declare
        cursor c1(p_job varchar2) is select * from emp where job= p_job;

```
i emp%rowtype;
begin
open c1('CLERK')
dbms_output.put_line('Employees working as Clerks');
loop
fetch c1 in i
exit when c1%notfound;
dbms_output.put_line(i.ename);
end loop;
close c1;
open c1('ANALYST');
dbms_output.put_line('Employees working as Analysts');
loop
fetch c1 in i;
exit when c1%notfound;
dbms_output.put_line(i.ename);
end loop;
close c1;
end;
/
```

NOTE: Before we are reopening the cursor we must close the cursor properly otherwise Oracle server returns an error. "ORA-06511: Cursor is already open".

NOTE: When we are not opening the cursor but we are trying to perform operations on the cursor then Oracle server returns an error "ORA-1001: Invalid Cursor".

Example: Converting to Parameterized cursor Shortcut method of Cursor.

```
Sql> declare
    cursor c1(p_deptno number) is select * from emp where deptno= p_deptno;
    begin
    for i in c1(10)
    loop
    dbms_output.put_line(i.ename||' `||i.sal||' `||i.deptno);
    end loop;
    end;
    /
```

Output:
```
    CLARK 2450 10
    KING 5000 10
    MILLER 1300 10
```

Applying same technique to another example:

```
Sql> declare
    cursor c1(p_job varchar2) is select * from emp where job=p_job;
```

26

```
begin
dbms_output.put_line('Employees working as Clerks');
for i in c1('CLERK')
loop
dbms_output.put_line(i.ename);
end loop;
dbms_output.put_line('Employees working as Analysts');
for i in c1('ANALYST')
loop
dbms_output.put_line(i.ename);
end loop;
end;  /
```

Output:
Employees working as Clerks
SMITH
ADAMS
JAMES
MILLER
Employees working as Analysts
SCOTT
FORD

NOTE: In all database systems when ever we are defining multiple cursors and also when we are passing data from one cursor into another cursor then receiving cursor must be a Parameterized cursor. Generally using parameterized cursors we can generate "Maste-Detailed Report"(Detailed means Child table)

Example:

Q) Write a PL/SQL cursor program to retrieve all deptno's from dept table into static cursor and also pass these deptno's from static cursor into another parameterized cursor which returns employee details from emp table absed on passed department numbers?

ANS: sql> declare
```
        cursor c1 is select deptno from dept;
        cursor c2(p_deptno number) is select * from emp where deptno= p_deptno;
        begin
        for i in c1
        loop
        dbms_output.put_line('My Department Number is:'||' '||i.deptno);
        for j in c2(i.deptno)
        loop
        dbms_output.put_line(j.ename||' '||j.sal||' '||j.deptno);
        end loop;
        end loop;
        end;
        /
```

Output:
my department number is: 10
        CLARK 2450 10
        KING 5000 10
        MILLER 1300 10
my department number is: 20
        SMITH 800 20
        JONES 2975 20
        SCOTT 3000 20
        ADAMS 1100 20
        FORD 3000 20
my department number is: 30
        ALLEN 1600 30
        WARD 1250 30
        MARTIN 1250 30
        BLAKE 2850 30
        TURNER 1500 30
        JAMES 950 30
my department number is: 40

In Parameterized cursor we can also pass default values by using "default" or ":=" operators.

Syntax:
parametername datatype [default (or) :=] actual value;

Example: sql> declare
        cursor c1(p_deptno number default 30) is select * from emp where deptno= p_deptno;
           OR
        cursor c1(p_deptno number:=30) is select * from emp where deptno= p_deptno;
        begin
        for i in c1( )
        loop
        dbms_output.put_line(i.ename||' `||i.sal);
        end loop;
        end;
        /

Date: 6/6/15

NOTE: In Oracle cursors, we can also use functions group functions, expressions in cursor select statement. In this case we must create alias name for those functions (or) expressions in cursor select statement and also we must create cursor record type variable(%rowtypw) in declare section of the cursor.

Syntax: variablename cursorname%rowtype;

Q) Write a PL/SQL cursor program which display total salary from emp table using "sum()"?

ANS: sql> declare

        cursor c1 is select sum(sal) a from emp;

        i c1%rowtype;

        declare

        open c1;

        fetch c1 into i;

        dbms_output.put_line('total salary is:'||' ' ||i.a);

        close c1;

        end;

        /

Output: Total salary is: 40425

Q) Write a PL/SQL cursor program using parameterized cursor for passing deptno as parameter that display total number of employees, total salary, minimum salary, maximum salary of that deptno from emp table?

ANS: sql> declare

        cursor c1(p_deptno number) is select count(*) a, sum(sal) b, min(sal) c, max(sal)

        d from emp where deptno=p_deptno;

        i c1%rowtype;

        begin

        open c1(&deptno);

        fetch c1 into i;

        dbms_output.put_line('Department number is:'||' '||i.deptno);

        dbms_output.put_line('Number of employees:'||' '||i.a);

        dbms_output.put_line('Total Salary is:'||' '||i.b);

        dbms_output.put_line('Minimum Salary is:'||' '||i.c);

        dbms_output.put_line('Maximum Salary is:'||' '||i.d);

        close c1;

        end;

        /

UPDATE, DELETE statements used in Explicit cursor(Without using **where current of, for update** clauses):

Q) Write a PL/SQL cursor program to modify salaries of the employees in emp table based on following conditions:

1. If job='CLERK' then increment sal-> 100

2. If job='SALESMAN' then decrement sal-> 200

ANS: sql> declare

        cursor c1 is select * from emp;

        i emp%rowtype;

        begin

        open c1;

        loop;

        fetch c1 into i;

        exit when c1%notfound;

```
if job='CLERK' then
update emp set sal=sal+100 where empno=i.empno;
elsif job='SALESMAN' then
update emp set sal=sal-200 where empno=i.empno;
end if;
end loop;
close c1;
end;
/
```
Output: select * from emp;

Date: 8/6/15

IMPLICIT CURSOR ATTRIBUTES:

When a PL/SQL block contains "Select…. into" clause (or) pure DML statements then Oracle server internally automatically allocates a memory area this memory area is also called as "SQL area"(or)"Context area" (or)"Implicit area". This memory area returns a single record when PL/SQL block contains "Select…. into" clause. This memory area also returns multiple records when PL/SQL block contains pure DML statements. But these multiple records does not controlled by user explicitly i.e., these all records processed at a time by SQL engine.
Along with this memory area Oracle server automatically created "4" memory locations. These memory location behaves like a variables. The variables are identified through "Implicit Cursor Attributes". These Implicit Cursor Attributes are :
1. SQL%NOTFOUND
2. SQL%FOUND
3. SQL%ISOPEN
4. SQL%ROWCOUNT

In all databases always "SQL%isopen" returns false. Whereas "SQL%notfound", "SQL%found" attributes returns Boolean value either true(or) false and also SQL%rowcount attribute always returns number datatype.
Example: sql> declare
```
delete from emp where ename= 'welcome';
if sql%found then
dbms_output.put_line('u r record deleted');
end if
dbms_output.put_line('u r record does not exist');
end if;
end;
/
```
Output: u r record does not exist
Example: sql> begin

30

```
update emp set sal= sal+100 where job='CLERK';
dbms_output.put_line('Affected number of clerks are:'||' '||sql%rowcount);
end;
/
```
Output: Affected number of clerks are: 4

<mark>EXCEPTIONS</mark>

Exception is an error occurred during runtime whenever exception is occurred use an appropriate exception name in exception handles under exception section within PL/SQL block.

Oracle have 3 types of "Exceptions".
1. Predefined exceptions
2. User defined exceptions
3. Unnamed exceptions

1. **Predefined exceptions:** Oracle provided 20 predefined exception names for regularly "Occurred runtime errors". Whenever run time errors occurred use in appropriate predefined exception name in exception handled under exception section of the PL/SQL block.

Syntax: when predefinedexceptionname1 then
       Stmts;
     when predefinedexceptionname2 then
       Stmts;

   ----
     when others then
      stmts;

PREDEFINED EXCEPTION NAMES:
1. no_data_found
2. too_many_rows
3. zero_divide
4. invalid_cursor
5. cursor_already_open
6. dup_val_on_index
7. **invalid_number**
8. **value_error**

1. **no_data_found:** When a PL/SQL block contains "select….into" clause and also if requested data not available in a table. Then Oracle server returns an error "ORA-1403: no data found". To handle this error Oracle provided "no_data_found" exception name.

Example: sql> declare
      v_ename varchar2(20);

```
v_sal number(10);
begin
select ename, sal into v_ename, v_sal from emp where empno= &no;
dbms_output.put_line(v_sal);
exception
when no_data_found then
dbms_output.put_line('u r employee does not exist');
end;
/
```
Output: enter value for no: 7902

    WARD 3000

Enter value for no: 1111

U r employee does not exist

2. **too_many_rows:** In PL/SQL blocks whenever "select…..into" clause try to return multiple records from a table (or) try to return multiple values at a time from a single column then Oracle server returns an error "ORA-1422: exact fetch returns more than requested number of rows". For handling this error Oracle provided "too_many_rows" predefined exception name.

Example: sql> declare
```
v_sal number(10);
begin
select sal into v_sal from emp;
dbms_output.put_line(v_sal);
exception
when too_many_rows then
dbms_output.put_line('not to return multiple rows');
end;
/
```
Output: not to return multiple rows.

3. **Zero_divide:** In PL/SQL when we are trying to perform division with zero then oracle server returns an error "ORA-1476: division is equal to zero". For handling this error we are using "Zero_divide" exception name.

Example: sql> begin
```
dbms_output.put_line(5/0);
exception
when zero_divide then
dbms_output.put_line('not to perform division with zero');
end;
/
```
Ouptut: not to perform division with zero.

Date: 9/6/15

6. Dup_val_on_index: In Oracle, when we try to insert duplicate values into "Primary key" or "unique key" then Oracle server returns an error "ORA-0001: Unique constraint violated". For handling this error oracle provided "dup_val_on_index" exception name.

Eg: sql> begin
　　insert into emp (empno)values(7369);
　　exception
　　when dup_val_on_index then
　　dbms_output.put_line('not to insert duplicates');
　　end;
　　/

Output: not to insert duplicates.

4. Invalid_cursor: In Oracle, when we are not "Open" the cursor, but we try to perform operations on the cursor then Oracle server returns an error "ORA-1001: Invalid cursor". For handling this error Oracle provided "invalid_cursor" exception name.

Example: sql> declare
　　　　cursor c1 is select * from emp;
　　　　i emp%rowtype;
　　　　begin
　　　　loop;
　　　　fetch c1 into i;
　　　　exit when c1%notfound;
　　　　dbms_output.put_line(i.ename||' '||i.sal);
　　　　end loop;
　　　　close c1;
　　　　exception
　　　　when invalid_cursor then
　　　　dbms_output.put_line('first we must open the cursor');
　　　　end;
　　　　/

Output: first we must open the cursor.

5. Cursor_already_open: In Oracle, before we are reopening the cursor then we must close the cursor properly otherwise Oracle server returns an error "ORA-6511: cursor already open". For handling this error we are using "cursor_already_open" exception name.

Example: sql> declare
　　　　cursor c1 is select * from emp;
　　　　i emp%rowtype;
　　　　begin
　　　　open c1;
　　　　loop;
　　　　fetch c1 into i;
　　　　exit when c1%notfound;

```
        dbms_output.put_line(i.ename||' '||i.sal);
        end loop;
        open c1;
        exception
        when cursor_already_open then
        dbms_output.put_line('first close the cursor to reopen the cursor');
        end;
        /
```
Output: first close the cursor to reopen the cursor.

7. invalid_number, value_error: When a PL/SQL block contains SQL, Procedural statements try to convert string type to number type (or) try to convert date string into date type the Oracle server returns two types of errors. These are invalid number , value error.

INVALID_NUMBER: When a PL/SQL block contains SQL statements and also those statements try to convert string type to number type (or) date string into date type then Oracle server returns an error "ORA-1722: Invalid number" for handling this error Oracle provided "invalid_number" exception name.

Example: sql> begin
```
        insert into emp(empno, ename, sal) values(1,'murali','abc');
        exception
        when invalid_number then
        dbms_output.put_line('insert proper data only');
        end;
        /
```
Output: insert proper data only

Example: sql> declare
```
        v_deptno varchar2(4):= '&deptno';
        v_dname varchar2(10):= '&dname';
        v_loc varchar2(10):= '&loc';
        begin
        insert into dept values(v_deptno, v_dname, v_loc);
        exception
        when invalid_number then
        dbms_output.put_line('Enter proper data only');
        end;
        /
```
Output: Enter value for deptno= x
        Enter value for dname = y
        Enter value for loc = z
Enter proper data only

VALUE_ERROR: When a PL/SQL block contains procedural statements and also those statements try to convert string type to number type then Oracle server returns an error

34

"ORA-6502: numeric or value error: character to number conversion error". For handling this error then we are using "value_error" exception name.

Example: sql> declare
        z number(10);
        begin
        z:= '&x'+'&y';
        dbms_output.put_line(z);
        exception
        when value_error then
        dbms_output.put_line('enter proper data only');
        end;
        /

Output: Enter value for x: a
        Enter value for y: b
Enter proper data only

NOTE: In Oracle, when we are try to store more data than the "varchar2" data type size specified at variable declaration then also Oracle server returns an error "ORA-6502: number (or) value error: character string buffer too small". For handling this error also then we are using "value_error" exception name.

Example: sql> declare
        z varchar2(3);
        begin
        z:= 'abcd';
        dbms_output.put_line(z);
        exception
        when value_error then
        dbms_output.put_line('Invalid string length');
        end;
        /

Output: Invalid string length

Date: 10/6/15

NOTE: When we try to store more data than the data type size in number data type also then Oracle server returns an error "ORA-6502: numeric (or) value error: number precision too large" for handling this error also we are using "value_error" exception name.

Example: sql> declare
        a number(3);
        begin
        a:= 99999;
        dbms_output.put_line(a);
        exception
        when value_error then

35

```
            dbms_output.put_line('not to store more data');
            end;
            /
Output: not to store more data.
--------------------------------------------------------------------------------------------
```

Example: sql> declare
```
            z varchar2(3):= 'abcd';
            begin
            dbms_output.put_line(z);
            exception
            when value_error then
            dbms_output.put_line('Invalid string length');
            end;
```
Output: **ERROR:** ORA-6502: numeric (or) value error: character string buffer too small.

Exception Propagation:

In PL/SQL exceptions also occurred in declare section, executable section, exception section.

When exceptions were occurred in executable section those exceptions are handled either in Inner blocks (or) in Outer blocks where as when exceptions are occurred in declare section, exception section then those exceptions are handled in Outer blocks only this is called "PL/SQL exception propagation".

Example: sql> begin
```
            declare
            z varchar2(3):= 'abcd';
            begin
            dbms_output.put_line(z);
            exception
            when value_error then
            dbms_output.put_line('Invalid string length');
            end;
            exception
            when value_error then
            dbms_output.put_line('Invalid string length handled using Outer blocks only');
            end;
            /
```
Output: Invalid string length handled using Outer blocks only.

USER DEFINED EXCEPTIONS: In Oracle, we can also create our own exception name and also raise explicitly whenever necessary. These type of exceptions are also called as "User Defined Exceptions".

In all databases, if we want to return messages based on client business rules then we are using user defined exceptions.

Handling User Defined Exceptions:

Step 1: Declare
Step 2: Raise
Step 3: Handling Exception

Step 1: Declare: In declare section of the PL/SQL block we can create our own exception name by using exception predefined type.
Syntax: userdefinedname exception;
Example: sql> declare
          a exception;

Step 2: Raise: We must raise under defined exceptions explicitly by using "RAISE" statement.
Syntax: raise userdefinedexceptionname;
Raise statement is used in either in executable section (or) in exception section of the PL/SQL block.

Step 3: Handling Exception: We can also handle user defined exceptions same like a predefined exceptions using exception handler under exception section within PL/SQL block.
Syntax: when userdefinedexceptionname 1 then
      stmts;
    when userdefinedexceptionname 2 then
      stmts;

   ----
    when others then
      stmts;

Q) Write PL/SQL program raise a user defined exception on Wednesday?
ANS: sql> declare
      a exception;
      begin
      if to_char(sysdate, 'DY')= 'WED' then
      raise a;
      end if;
      exception
      when a then
      dbms_output.put_line('my exception raised on Wednesday');
      end;
      /
Output: my exception raised on Wednesday

Example: sql> declare
      a exception;
      v_sal number(10);
      begin

```
            select sal into v_sal from emp where empno= 7369;
            if v_sal>2000 then
            raise a;
            else
            update emp set sal= sal+100 where empno = 7369;
            end if;
            exception
            when a then
            dbms_output.put_line('salary already high');
            end;
            /
```
Output: Salary already high

Date: 11/6/15

Q) Write a PL/SQL program using following table whenever user entered empno is more than 10 or less than 1(<1) then raise a user defined exception and also whenever user entered empno is available in a table then display name if the employee based on user entered number.

sql> create table test as select rownum empno, ename from emp where rownum<=10;

sql> select * from emp;

ANS: sql> declare
```
            a exception;
            v_ename varchar2(10);
            v_empno number(10);
            begin
            v_empno= &empno;
            if v_empno>10 or v_empno<1 then
            raise a;
            else
            select ename into v_ename from test where empno=v_empno;
            dbms_output.put_line(v_ename);
            end if;
            exception
            when a then
            dbms_output.put_line(' u r empno is out of range ');
            end;
```

TESTING EXCEPTION PROPAGATION USING USER DEFINED EXCEPTION:

Exception Raised in Executable section:

When exception is raised in executable section those exceptions are handled either in Inner block (or) in Outer block.

Method 1: **Handled Using Inner block**

Example: sql> declare
```
            a exception;
```

```
begin
raise a;
exception
when a then
dbms_output.put_line('Handled using Inner block');
end;
/
```
Output: Handled using Inner block

Method 2: Handled using Outer block

Example: sql> declare
```
        a exception
        begin
                begin
                raise a;
                end;
        exception
        when a then
        dbms_output.put_line('Handled using Outer block');
        end;
        /
```
Output: Handled using Outer block

Exception Raised in Exception section:

In PL/SQL when exceptions raised in exception section those exceptions are handled in outer blocks only.

Example: sql> declare
```
        z1 exception;
        z2 exception;
        begin
                begin
                raise z1;
                exception
                when z1 then
                dbms_output.put_line('Z1 handled');
                raise z2;
        end;
        exception
        when z2 then
        dbms_output.put_line('Z2 handled');
        end;
        /
```
Output: Z1 handled
        Z2 handled

Example: sql> declare
```
        cursor c1 is select * from emp where deptno=&deptno;
```

```
i emp%rowtype;
a exception;
begin
open c1;
fetch c1 into i;
if c1%rowcount=0 then
raise a;
else
while(c1%found)
loop
dbms_output.put_line(i.ename||' `||i.sal||' `||i.hiredate);
fetch c1 into i;
end loop;
end if;
close c1;
exception
when a then
dbms_output.put_line(' u r requested deptno is not available in emp table');
end;
/
```

Output:

NOTE: In Oracle, we can also raise predefined exceptions explicitly by using "RAISE" statement.

Syntax: raise predefinedexceptionname;

Example: sql> declare

```
cursor c1 is select * from emp where deptno=&deptno;
i emp%rowtype;
begin
open c1;
fetch c1 into i;
if c1%rowcount=0 then
raise no_data_found;
else
while(c1%found)
loop
dbms_output.put_line(i.ename||' `||i.sal||' `||i.hiredate);
fetch c1 into i;
end loop;
end if;
close c1;
exception
when no_data_found then
dbms_output.put_line(' u r requested deptno is no available in emp table');
end;
```

/
Output: Enter value for deptno: 90
u r requested deptno is not available in emp table

Date: 12/6/15

<mark>**Unnamed Exception:**</mark> If we want to handle other than Oracle, 20 predefined exception name errors then we are using "Unnamed Exceptions". In this method we are creating our own exception name and associate that exception name with appropriate error number by using "EXCEPTION_INTIT()" function.
Syntax: pragma exception_init(userdefinedexceptionname, errornumber);
This function is used in declare section of the PL/SQL block.
NOTE: Here "Pragma" is an "Complier Directive" i.e., whenever we are using this pragma Oracle server internally associates error number with exception name before compilation process.
Example: sql> begin
　　　　insert into emp(empno, ename) values(null, 'abc');
　　　　end;
Error: ORA-01400: cannot insert NULL into EMPNO
Solution: sql> declare
　　　　a exception;
　　　　pragma exception_init(a, -1400);
　　　　begin
　　　　insert into emp(empno,ename) values(null,'abc');
　　　　exception
　　　　when a then
　　　　dbms_output.put_line('Duplicate values not allowed');
　　　　end;
　　　　/
Output: Duplicate values not allowed.
Error: ORA-2292
Example: sql> declare
　　　　a exception;
　　　　pragma exception_init(a,-2292);
　　　　begin
　　　　delete from dept where deptno=10;
　　　　exception
　　　　when a then
　　　　dbms_output.put_line('not to delete master records');
　　　　end;
　　　　/
Output: not to delete master records
Q) Write a PL/SQL program to handle -2291 error number by using exception_init function from emp, dept table?

41

ANS: sql> declare
    a exception
    pragma exception_init(a, -2291);
    begin
    insert into emp(empno,deptno) values(1,50);
    exception
    when a then
    dbms_output.put_line('not to insert other than primary key values');
    end;
    /
Output: not to insert other than primary key values

RAISE_EXCEPTION_ERROR:
raise_exception_error is a predefined exception procedure is available in "dbms_standard" package(sql> desc dbms_standard)
If you want to display user defined exception messages in more descriptive from then only we are using this procedure. i.e., if you want to display user defined exception messages as same as oracle error displayed format then we must use raise_exception_error procedure. This procedure accepts two parameters.
Syntax: raise_exception_error(error number, message);
Error number → -20000 to -20999
Message → upto 512 characters
This procedure is used in either executable section (or) in exception section of the PL/SQL block.
Example: sql> declare
    a exception
    begin
    if to_char(sysdate,'DY')='FRI' then
    raise a;
    end if;
    exception
    raise_exception_error(-20234, ' My exception raised on Friday');
    end;
    /
Output: my exception raised on Friday
NOTE: raise_exception_error is an exception procedure that's why when condition is true it raise a message and also stop the execution and also control does not goes to remaining program. Whereas put_line is normal procedure whenever condition is true it display message and also control goes to remaining program.
Example: Difference between dbms_output.put_line raise_exception_error():
Sql> begin
    if to_char(sysdate,'DY')= 'FRI' then
    dbms_output.put_line(' my exception raised on Friday');
    end if;

```
dbms_output.put_line('tomorrow there is no class');
end;
```
Output: my exception raised on Friday

Tomorrow there is no class

Sql> begin
```
    if to_char(sysdate, 'DY')= 'FRI' then
    raise_exception_error(-20324, 'my exception raised on Friday');
    end if;
    dbms_output.put_line('tomorrow there is no class');
    end;
    /
```
Output: ORA-20324: my exception raised on Friday

Date: 15/6/15

NOTE: Generally, raise_application_error procedure used in triggers because when an exception procedure. It stops execution when condition is true and also it stops invalid data entry based on condition. Whereas dbms_output.put_line procedure does not stops invalid data entry based on the condition because this is a normal procedure.

Error Trapping Function: Oracle having predefined error trapping functions. These are

1. Sqlcode

2. Sqlerrm

These two predefined functions are used in when "**others then**" clause are also used in our own exception handler.

These two error trapping functions error numbers, error number with error messages.

In Oracle, if we want to know which run time error is occurred for a particular PL/SQL block at execution time then we must use "SQLCODE()" function.

Eg: declare
```
    i emp%rowtype;
    begin
    select * into i from emp where deptno=&deptno;
    dbms_output.put_line(i.ename||' '||i.sal);
    exception
    when others then
    dbms_output.put_line(sqlcode);
    end;
```
Output: Enter value for deptno=10

                -1422

Enter value for deptno= 'a'

                -1722

NOTE: Using SQL code we can also handle unnamed exception.

Example: sql> begin

```
delete from emp where deptno=10;
exception
when others then
if sqlcode=-2292 then
dbms_output.put_line('not to delete master record');
end if;
end;
/
```

NOTE: In Oracle, we are not allowed to use sqlcode, sqlerrm functions directly in DML statements. If we want to use these functions in DML then first we must declare variables in declare section of PL/SQL block and then assign values into variables and then only those variables are used in DML statements.

Q) Write a PL/SQL program which stores error number, error number with error message of a PL/SQL block into another table?
ANS: sql> create table target(errno number(10), errmsg varchar2(2000));

```
Sql> declare
    v_errno number(10);
    v_errm varchar2(2000);
    v_sal number(10);
    begin
    select sal into v_sal from emp;
    exception
    when others then
    v_errno := sqlcode;
    v_errm := sqlerrm;
    insert into target values(v_errno, v_errm);
    end;
    /
Sql> select * from target;
```

SQLCODE return values	Meaning
0	No Errors
1	User Defined Exceptions
Negative(-ve)	Oracle Errors
100	No Data Found

Generally, sqlcode() returns numbers where as sqlerrm() returns error number with error messages.

Example: sql> declare
```
        a exception
        begin
        raise a;
        exception
```

44

```
        when a then
        dbms_output.put_line(sqlcode);
        dbms_output.put_line(sqlerrm);
        end;
        /
```
Output: 1
        User Defined Exception

NOTE: In Oracle, we can also pass sqlcode, sqlcode return values into sqlerrm function. In Oracle, if we want to view sqlcode return value meanings then we must pass sqlcode into sqlerrm function within executable section of the PL/SQL block.

Example: sql> begin
```
        dbms_output.put_line(sqlerrm(sqlcode));
        dbms_output.put_line(sqlerrm(100));
        dbms_output.put_line(sqlerrm(1));
        dbms_output.put_line(sqlerrm(-1722));
        end;
        /
```
Output:
ORA-0000: normal, successful completion
ORA-01403: no data found
User Defined Exception
ORA-01722: Invalid Number

SUB PROGRAMS

Sub programs are name PL/SQL block which is used to solve particular task all database systems having two types of sub programs.
1. Procedures → May or may not return value
2. Functions → Must return a value

Date: 16/6/15

PROCEDURE: Procedure is an named PL/SQL block which is used to solve particular task and also procedure may or may not return values. In Oracle, whenever we are using create or replace keyword in front of the procedure then those procedures are automatically, permanently stored in database. That's why these procedures are also called as "Stored Procedures".

Generally, procedures are used to improve performance of the application because in all database procedures internally having one time compilation. By default one time compilation program units improves performance of the application.

```
┌─────────────────────────────────────┐
│ Using a Editor write a procedure to  │
│ solve peritcular                     │
└─────────────────────────────────────┘
            │
            ▼
┌─────────────────────────────────────┐
│ Using sqlplus tool load procedure    │
│ into Oracle database                 │
└─────────────────────────────────────┘
            │
            ▼
        ┌──────────┐
        │ Compiled │
        └──────────┘
            │
            ▼
        ┌──────────┐
        │ Parsed   │
        └──────────┘
            │
            ▼
        ┌──────────┐
        │ Execute  │
        └──────────┘
```
show errors
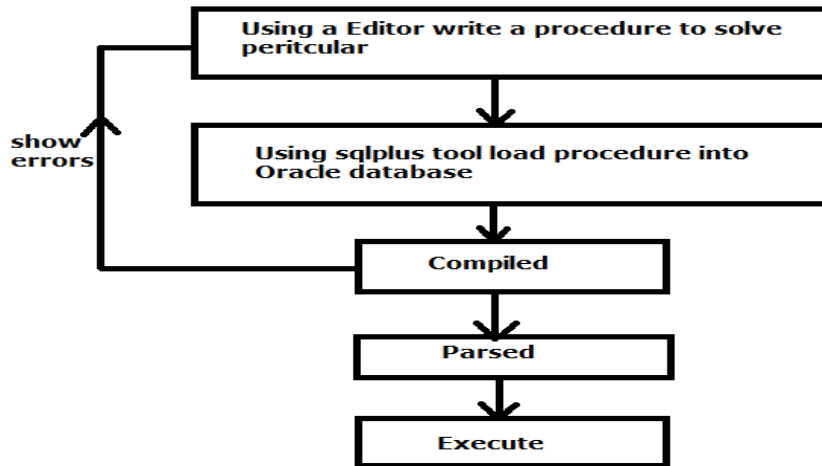
Every Procedure having two parts
1. Procedure Specification
2. Procedure Body.
In Procedure specification we are specifying name of the procedure and type of the parameters where as a Procedure body we are solving actual task.

Syntax:
create or replace procedure procedurename(formal parameters)
is/as
variable declarations, cursor, user defined exceptions;
begin

[exception]

     ------
end[procedurename];

Formal Parameters:
Syntax: parametername[mode] datatype;
Different kind of modes: IN, OUT, IN OUT

Executing a Procedures:
Method 1: Using Bind Variable
Syntax: Sql> exec procedurename(actual parameters)
Method 2: Using Anonymous block
Syntax: sql> begin
          procedurename(actual parameters);
          end;
          /
Method 3: Using Call Statement
Syntax: sql> call procedurename(actual parameters);

46

To View Errors:

Syntax: sql> show errors;

Q) Write a PL/SQL stored procedure program for passing empno as a parameter that display name of the employee and his salary from emp table?

ANS: sql> create or replace procedure p1(p_deptno number)

```
    is
    v_ename varchar2(10);
    v_sal number(10);
    begin
    select ename, sal into v_ename, v_sal from emp where empno= p_empno;
    dbms_output.put_line(v_ename||' '||v_sal);
    end;
    /
```

Output: exec (7566);

```
    JONES 3960
```

Execution:

Method 1: sql> exec p1(7902);

```
    FORD 4000
```

Method 2: sql> begin

```
        p1(7902);
        end;
        /
```

FORD 4000

Method 3: sql> call p1(7902);

```
        FORD 4000
```

In Oracle, all stored Procedures information stored under "user_procedure", "user_source" data dictionaries.

If we want to view code of the procedure then we are using "user_source" data dictionary.

Eg: sql> desc user_sourse;

Sql> select text from user_source where name='P1';

Q) Write a PL/SQL stored Procedure program for passing deptno as a parameter then display employee details from emp table based on passed deptno?

ANS: sql> create or replace procedure p1(v_deptno number)

```
    is
    cursor c1 is select * from emp where deptno=v_deptno;
    i emp%rowtype;
    begin
    open c1;
    loop
    fetch c1 into I;
    exit when c1%notfound;
```

47

```
dbms_output.put_line(i.ename||' `||i.sal||' `||i.deptno);
end loop;
close c1;
end;
/
```
Output: sql> exec p1(10);

```
CLARK       3750  10
KING        6400  10
MILLER      3700  10
```

Date: 17/6/15

Procedure parameters:
Parameter is a name which is used to pass values into procedure and also return values from the procedure. Every procedure having two types of parameters.
1. Formal Parameters
2. Actual Parameters

1. **Formal Parameters:** Formal Parameters must be specified in procedure specification. Formal Parameters specifies name of the parameter, type of the parameter, mode of the parameter.
Syntax: sql> parametername [mode] datatype;
NOTE: In Oracle, when we are defining formal parameters in cursors, procedures, functions then we are not allowed to use data type size in formal parameter declaration.
MODE: Mode specifies propose of the parameter Oracle procedures having three types of modes.
1. IN
2. OUT
3. IN OUT

1. **IN mode:** In Oracle, by default parameter mode is "IN" mode. IN mode is used to pass values into procedure body. IN mode behaves like a constant in procedure body. In this parameter we cannot assign new value in procedure body. This parameter behaves like a read only value.
Syntax: parametername in datatype;
Q) Write a PL/SQL stored procedure program to insert a record into dept table by using "IN" parameters?
ANS: sql> create or replace procedure p1(p_deptno in number, p_dname in varchar2, p_loc in varchar2)

```
is
begin
insert into dept values(p_deptno,p_dname,p_loc);
dbms_output.put_line(' u r record inserted through procedures');
end;
```

48

    /

Output: sql> exec p1(1,'a','b');

    U r record inserted through procedures.

2. **OUT mode:** OUT mode is used to return values from the procedure body. OUT parameter behaves like a "uninitialized" variable in procedure body. Here explicitly we must specify OUT keyword.

Syntax: parametername out datatype;

Eg: sql> create or replace procedure p1(a in number, b our number)

    is

    begin

    b:= a*a;

    end;

    /

In Oracle, when a sub program having OUT, IN OUT parameters then those type of sub programs are executed using following '2' methods.

Method 1: Using Bind variable

Method 2: Using Anonymous block

Method 1: Using Bind Variable

Eg: Sql> variable z number;

Sql> exec p1(a, :z);

Sql> print z;

    Z

    8

Method 2: Using Anonymous block

Eg: sql> declare

    x number(10);

    begin

    p1(8,x);

    dbms_output.put_line(x);

    end;

    /

Output: 64

Q) Write a PL/SQL stored procedure program for passing empname as IN parameter then return salary of the employee using OUT parameter from emp table?

ANS: sql> create or replace procedure p1(p_ename in varchar2, p_sal out number)

    is

    begin

    select sal into p_sal from emp where ename=p_ename;

    end;

    /

Output:

Method 1: sql> variable a number;

    Sql> exec p1('SMITH', :a);

    Sql> print a;

    A

    2400

Method 2: sql> declare

    b number(10);

    begin

    p1('SMITH', b);

    dbms_output.put_line(b);

    end;

    /

Output: 2400

Date: 18/6/15

Q) Write a PL/SQL stored procedure for passing deptno as IN parameter then return dname, loc using OUT parameter from dept table?

ANS: sql> create or replace procedure p1(p_deptno in number, p_dname out varchar2, p_loc out varchar2)

as

begin

select dname, loc into p_dname, p_loc from dept where deptno=p_deptno;

end;

/

Output:

Execution

Using Bind variables:

Sql> variable x varchar2(10);

Sql> variable y varchar2(10);

Sql> exec p1(10, :x, :y);

Sql> print x y;

Output:

    A

ACCOUNTING

    B

NEW YORK

Execution

Using Anonymous block

50

```
Sql> declare
    x varchar2(10);
    y varchar2(10);
    begin
    p1(30,x,y);
    dbms_output.put_line(x||' `||y);
    end;
    /
```
Output: SALES CHICAGO

Q) Write a PL/SQL stored procedure program for passing deptno as a IN parameter then return number of employees number in that deptno by using OUT parameter from emp table?
ANS: sql> create or replace procedure p1(p_deptno in number, p_count out number)
```
    as
    begin
    select count(*) into p_count from emp where deptno=p_deptno;
    end;
    /
```
Output:
Execution: Using Bind variable
Sql> variable a number;
Sql> exec p1(10,:a);
Sql> print a;
```
    A
----------
    3
```

PASS BY VALUE, PASS BY REFERENCE:
Whenever we are using Modular programming all languages supports two types of Pass parameters mechanisms.
1. Pass By Value
2. Pass By Reference
These two passing parameter mechanisms specifies whenever we are modifying format parameters then actual parameters are effected (or) not. In Pass By Value method actual values does not change in main program.
Because here internally copy of the values are passed into calling program.
If you want to change actual parameters automatically based on formal parameters modifications then we must use "Pass By Reference".
Oracle also supports these two passing parameter mechanisms when we are using some program parameters. In Oracle, by default all in parameters uses "Pass By Reference" method internally where as by default all OUT parameters internally uses "Pass By Value" method. That's why whenever we are returning values using OUT parameter internally copy of the values of created.

Whenever we are returning large amount of data using OUT parameter then Oracle server internally create copy of the values. This process automatically degrades the performance of the procedures.

To overcome this problem for improve performance of the procedure Oracle 9i introduced "nocopy" compiler hint along with OUT parameter.

Syntax: parametername out nocopy datatype;

Eg: sql> create or replace procedure p1(p_deptno in number, p_count out nocopy number)
```
is
begin
select count(*) into p_count from emp where deptno=p_deptno;
end;
/
```

3. **IN OUT:** This mode is used to passing values into procedure and also return values from the procedure. This parameter behaves like a constant initialized variable in a procedure body. Here also explicitly we must specify "IN OUT" keyword also explicitly we must specify "IN OUT" keyword.

Syntax: parametername in out datatype;

Date: 19/6/15

EG: sql> create or replace procedure p1(a in out number)
```
is
begin
a := a*a;
end;
/
```
Execution:
Method 1: Using Bind variable
Sql> variable x number;
Sql> exec :x:=8;
Sql> exec p1(:x);
Sql> print x;
Method 2: Using anonymous block
Sql> declare
```
x number(10):=&x;
begin
p1(x);
dbms_output.put_line(x);
end;
/
```
Output:
Enter value for x= 4
16

Q) Write a PL/SQL stored procedure for passing empno as parameter then return salary for the emp by using IN OUT parameter from emp table?

ANS: sql> create or replace procedure p1(p_x in out number)

```
as
begin
select sal into p_x from emp where empno=p_x;
end;
/
```

Execution:

Method 1: Using Bind variable

Sql> variable a number;

Sql> exec :a:=7902;

Sql> exec p1(:a);

Sql> print a;

Output:

```
    A
------------
    4000
```

Method 2: Using Anonymous block

Sql> declare

```
b number(10):= &b;
begin
p1(:b);
dbms_output.put_line(b);
end;
/
```

IN parameter Execution methods:

In Oracle, procedure IN parameter having three types of execution methods.

1. Positional Notations
2. Named Notations(=>)
3. Mixed Notations

EG: sql> create or replace procedure p1(p_deptno in number, p_dname in varchar2, p_loc in varchar2)

as

begin

insert into dept values(p_deptno,p_dname,p_loc);

end;

/

Execution:

Using Positional Notation:

Sql> exec p1(5,'x','y');

Sql> exec p1(p_dname=>'p', p_loc=>'q', p_deptno=> 8);

Mixed Notations:

It is the combination of Positional, Named notations. Generally after Positional these can be all Named notations but after Named there cannot be Positional.

EG: sql> exec p1(4,p_loc=> 'q'. p_dname=> 'p');

NOTE: In Oracle, we can also pass default values using procedure IN parameters. In this case we must use "Default" or ":=" operator.

Syntax: parametername in datatype default[:=] actualname;

EG: sql> create or replace procedure p1(p_deptno in number, p_dname in varchar2, p_loc in varchar2 default 'hyd')

```
    is
    begin
    insert into dept values(p_deptno, p_dname, p_loc);
    end;
    /
```
Sql> exec p1(1, 'a');
Sql> select * from dept;

AUTONOMOUS TRANSACTIONS:

Anonymous transactions are independent transaction used in anonymous blocks, procedure, functions, triggers.

"Autonomous procedures" are independent procedures used in main transaction these procedure used in main transaction these procedure transactions are never effected from main transaction COMMIT, ROLLBACK.

Generally autonomous transactions are defined in child procedures. If you want to make a procedure autonomous then we are using "Autonomous_transaction pragma", commit.

Syntax:

pragma autonomous_transaction;

This pragma is used in declare section of the procedure.

Syntax:

create or replace procedure procedurename(formal parameter)

is/as

pragma autonomous_transactions;

begin

Commit;

[Exception]

----

end[procedurename];

Date: 20/6/15

Using Autonomous Transactions:

EG: sql> create table test(name varchar2(10));

Sql> create or replace procedure p1

```
            is
            pragma autonomous_transaction;
            begin
            insert into test values('INDIA');
            commit;
            end;
            /
  Sql> begin
            insert into test values('hyd'):
            insert into test values('mumbai');
            p1;
            rollback;
            end;
            /
  Sql> select * from test;
```
Output: NAME

             INDIA

```
  Sql> delete from test;
```

Without Using Autonomous Transactions:
```
  Sql> create or replace procedure p1
            is
            begin
            insert into test values('INDIA');
            commit;
            end;
            /
  Sql> begin
            insert into test values('hyd');
            insert into test values('mumbai');
            p1;
            rollback;
            end;
            /
  Sql> select * from test;
```
Output: NAME

          ----
                  hyd
                  Mumbai
                  INDIA

In Oracle, when a procedure having commit and also when we are calling this procedure in a PL/SQL block then this procedure commit not only saves procedure transactions but also saves all the above procedure transactions.

To overcome this problem Oracle 8.1.6 introduced autonomous transactions. When we are using this transactions in child procedures then autonomous procedure transactions are never effected in main transactions.

Autonomous Transactions in Anonymous block:

SESSION 1	SESSION 2

Main Transaction
sql> create table test(sno number (10));
sql> insert into test values(1);
sql> insert into test values(2);
sql> select * from test;
SNO
----- 	
1
2
Child Transaction(autonomous transaction)
sql> declare
pragma autonomou_transaction;
begin
for i in 3..10
loop
insert into test values(i);
end loop;
commit;
end;
/
sql> select * from test;
SNO
---- 	
1
2
3
4
5
6
7
8
9
10
sql> rollback;
sql> select * from test;
SNO
---- 	
3
4
5
6
7
8
9
10
	sql> select * from test;
	No rows selected
	
	sql> select * from test;
	SNO
	---- 	
	3
	4
	5
	6
	7
	8
	9
	10

Handled, Unhandled Exceptions in Procedures:

Whenever we are calling Inner Procedure into Outer Procedure then we must handle Inner Procedure Exceptions within Inner Procedure. Otherwise Oracle server automatically calls Outer Procedure default exception handler.

EG: INNER PROCEDURE(Individual Procedure)

Sql> create or replace procedure p1(a in number, b in number)

```
    is
    begin
    dbms_output.put_line(a/b);
    exception
    when zero_divide then
    dbms_output.put_line('b cannot be zero');
    end;
```

OUTER PROCEDURE

Sql> create or replace procedure p1

```
    is
    begin
    p1(5,0);
    exception
```

```
    when others then
    dbms_output.put_line('any error');
    end;
    /
```
Ouptput: exec p2;
    b cannot be zero.

We can also drop a procedure by using
⇨ drop procedure procedurename;

FUNCTIONS

Function is an Named PL/SQL block which is used to solve particular task and also function must return a value.
Function also having two(2) types.
1. Function Specification
2. Function Body
In Function Specification we are specifying name of the function and type of the parameters.
Whereas in Function Body we are solving actual task.
Syntax:
```
create or replace function functionname(formal parameter)
return datatype
is as
variable declarations, cursors, user defined exception;
begin
    -----
    -----
return expression;
[exception]
    -----
    -----
end [function name];
```

Execution:
Method 1: (using SELECT statement)
When a function having all IN parameters or when a function does not have parameters then those functions are allowed to execute using "SELECT" statement.
Syntax:
select functionname(actual parameters) from dual;
Method 2: (using anonymous block)
Syntax:
```
Sql> begin
    varname := functionname(actual parameter);
    end;
```

57

/

Date: 22/6/15

EG: sql> create or replace function f1(a varchar2)
     return varchar2
     is
     begin
     return a;
     end;
Execution:
Method 1: using SELECT statement
Sql> select f1('hi') from dual;
     hi
Method 2: using Anonymous block
Sql> declare
     z varchar2;
     begin
     z:= f1('welcome');
     dbms_output.put_line(z);
     end;
     /
Welcome

Q) Write a PL/SQL stored function for passing number as a parameter then return a message either even or odd based on the passed number
ANS: sql> create or replace function f1(a number)
     return varchar2
     is
     begin
     if mod(a,2)=0 then
           return even;
     else
           return odd;
     end if;
     end;

Execution:
Method 1: using SELECT statement
Sql> select f1(5) from dual;
     Odd
Method 2: using anonymous block
Sql> declare
     z varchar2(10);

```
    begin
    z := f1(8);
    dbms_output.put_line(z);
    end;
    /
even
```

Method 3: using BIND variable
```
Sql> variable x varchar2(10);
Sql> begin
    :x := f1(10);
    end;
sql> print x;
even
```

Method 4: using "Packaged Procedure"
```
Sql> exec dbms_output.put_line(f1(3));
    odd
```

Method 5:
```
Sql> begin
    dbms_output.put_line(f1(7));
    end;
    /
odd
```

NOTE: In Oracle, we can also use "User Defined Functions" in DML statements.
EG: sql> create table test(msg varchar2(10));
```
Sql> insert into test values(f1(8));
    1 row created
Sql> select * from test;
    MSG
  ------------
    Even
```
NOTE: In Oracle, we can also use predefined functions in User Defined Functions and also "CALL" these User defined functions with same table (or) in different table using "SELECT".
EG: Q) Write a PL/SQL stored function which returns max(sal) from emp table?
ANS: sql> create or replace function f1
```
    return number
    is
    v_sal number
    begin
    select max(sal) into v_sal from emp;
    return v_sal;
```

59

end;

/

Execution

Sql> select f1 from dual;

6400

Sql> select ename, sal, f1 from emp;

Sql> select ename, sal, f1-1 from emp;

Q) Write a PL/SQL stored procedure for passing ename as parameter then return job of the employee from emp table?

ANS: sql> create or replace function f1(p_ename varchar2)

return varchar2

is

v_job varchar2(10);

begin

select job into v_job from emp where ename= p_ename;

return v_job;

end;

/

Execution:

Sql> select f1('SMITH') from dual;

CLERK

Q) Write a PL/SQL stored function for passing empno as a parameter then return tax of the employee from emp table by using following conditions?

Conditions : 1. If annsal> 10000 then tax→ 10% of annsal

2. If annsal>15000 then tax→ 20% of annsal

3. If annsal>20000 then tax→ 30% of annsal

4. else tax→ 0%

ANS: sql> create or replace function f1(p_ename varchar2)

return number

is

v_sal number(10);

v_tax number(10);

annsal number(10);

begin

select sal into v_sal from emp where ename= p_ename;

annsal= v_sal*12;

if annsal>10000 and annsal<=15000 then

v_tax := annsal*0.1;

elsif annsal>15000 and annsal<=20000 then

v_tax := annsal*0.2;

elsif annsal>20000 then

v_tax := annsal*0.3;

```
else
v_tax := 0;
end if;
return v_tax;
end;
/
```

Execution:

Sql> select f1(7566) from dual;

```
    f(7566)
  -------------
    13950
```

Date: 23/6/15

Q) Write a PL/SQL stored function for passing empno, date as parameters then return number of years that employee is working based on passed date from emp table?

ANS: sql> create or replace function f1(p_empno number, p_date date)

```
    return number
    is
    x number(10);
    begin
    select months_between(p_date, hiredate)/12 into x from emp where empno=p_empno;
    return round(x);
    end;
    /
```

Execution:

Method 1: Using Dual table

Sql> select f1(7902, sysdate) from dual;

```
    34
```

Method 2: Using Emp table

Sql> select empno, ename, hiredate, f1(empno,sysdate) ||' '|| 'years' "EXPR" from emp where empno=7566;

Output: EMPNO ENAME   HIREDATE   EXPR

```
      ------------------------------------------
        7566  JONES   02-APR-81    34 years
```

NOTE: Prior to Oracle11g, we are not allowed to use "Named notations", "Mixed notations" in function executions where as in Oracle 11g we are allowed to used named, mixed notations when a subprogram is executed using SELECT statement.

IN ORACLE 11G:

Eg: sql> select empno, ename, hiredate, f1(p_empno=>empno, p_date=> sysdate)||' '|| 'years' "EXPR" from emp where empno=7566;

DML statements:

In Oracle, we can also use DML statements in user defined functions but we are not allowed to execute those functions using SELECT statements but we are allowed to execute those functions by using anonymous blocks.

Q) Write a PL/SQL stored function for passing empno as a parameter from emp table that delete that employee record in emp table and also automatically display delete number of records number from emp table?

ANS: sql> create or replace function f1(p_empno number)

```
    return number
    is
    v_count number(10);
    begin
    delete from emp where empno= p_empno;
    v_count:= sql%rowcount;
    return v_count;
    end;
    /
```

Execution:

Method 1: using SELECT statement

Sql> select f1(1) from dual;

Error: cannot perform a DML operation inside a query.

Method 2: Using Anonymous block

Sql> declare

```
    a number(10);
    begin
    a:= f1(1);
    dbms_output.put_line(a);
    end;
    /
```

Output: 0

In Oracle, when a function contains DML statement those functions are not allowed to execute by using SELECT statement.

To overcome this problem Oracle 8i introduced "Autonomous Transactions" within user defined functions. These only we are allowed to execute these functions by using "SELECT" statement.

Syntax:

create or replace function functionname(formal parameter)

return datatype

is/as

pragma autonomous_transaction;

begin

```
    -----
    -----
```

return expression;

commit;

[exeception]

    -----
end [functionname];

EG: sql> create or replace function f1(p_empno number)
    return number
    is
    v_count number(10);
    pragma autonomous_transaction;
    begin
    delete from emp where empno=p_empno;
    v_count= sql%rowcount;
    commit;
    end;
    /

OUT mode: We can also use OUT parameter within user defined functions but these functions are not allowed to execute by using SELECT statement. Using OUT parameters we can return more than one value from functions.

Q) Write a PL/SQL stored functions for passing deptno as a IN parameter and return dname, loc by using OUT parameters from emp table?

ANS: sql> create or replace function f1(p_deptno in number, p_dname out varchar2, p_loc out varchar2)
return varchar2
is
begin
select dname, loc into p_dname, p_loc from emp where deptno=p_deptno;
return p_dname;
end;
/

Execution: (using BIND variable)
Sql> variable a varchar2(10);
Sql> variable b varchar2(10);
Sql> variable c varchar2(10);
Sql> begin
    :a := f1(10, :b, :c);
    end;
    /

Sql> print b c;
```
        B
------------------
ACCOUNTING
        C
------------------
NEW YORK
```

Date: 24/6/15

Cursors used in Functions:
In Oracle, we can also develop our own aggregate functions same like a predefined aggregate functions when these user defined aggregate function returns multiple values then we are using "Cursors" within user defined functions. These functions also used in "**group by**" clause same like a predefined aggregate functions.

Q) Write a PL/SQL stored function for passing deptno as a parameter then return employees in that deptno from emp table and also use this function in SELECT statement.
ANS: sql> create or replace function f1(p_deptno number)
```
    return varchar2
    is
    cursor c1 is select ename from emp where deptno=p_deptno;
    x varchar2(200);
    begin
    for i in c1
    loop
    x:=x||' `||i.ename;
    end loop;
    return x;
    end;
    /
```
Execution:
Sql> select deptno, f1(deptno) from emp group by deptno;
Output: DEPTNO   ENAME
```
        --------------------------------------
        10     KING CLARK MILLER
        20     SMITH JONES SCOTT ADAMS FORD
        30     ALLEN WARD MARTIN BLAKE TURNER JAMES
```
NOTE: Oracle 10g, introduced "**wm_concat()**" predefined aggregate function which returns multiple values. This function accepts all datatypes columns along with ","(comma) separate string. This function also used in "group by" clause same like a predefined aggregate function.

EG 1: sql> select deptno, wm_concat(ename) from emp group by deptno;
EG 2: sql> select job, wm_concat(ename) from emp group by job;

In Oracle, all stored functions information stored under "user_procedures", "user_source" data dictionaries. If we want to view code of the functions then we are using "user_source" data dictionary.

EG: sql> desc user_source;

Sql> select text from user_source where ename='F1';

We can drop a function by using

⇨ **drop function functionname;**

PACKAGE

Package is an database object which encapsulates procedures, functions, cursors, global variables, constants, types into single unit.

Generally, packages does not accepts parameters an also cannot be invoked directly. Generally, packages are used to improves performance of the application because whenever we are calling a packaged sub program first time then automatically total packages loaded into memory area whenever user calling sub sequent sub program calls then oracle server calling those sub programs from memory area not from disk This process automatically reduces disk I/O(input/output) operations.

That's why packages also improves performance of the applications.

Packages also having 2 parts.

1. Package Specification
2. Package Body

In Oracle, by default package specification objects are public, where as package body objects are private. In package specification we are declaring variables, constants, cursors, types, procedures, functions. Whereas in package body we are implementing procedures, functions.

Syntax:

create or replace package packagename

IS/AS

➔ Global variables declarations, constant declarations;

➔ Cursors declaration;

➔ Types declarations;

➔ Procedure declarations;

➔ Function declaration;

end;

Package Body:

Syntax:

Sql> create or replace package body packagename

IS/AS

➔ Procedure implementation;

➔ Function implementation;

end;

Calling Package Procedures:
Method 1:
Sql> exec packagename.procedurename(actual parameters);

Method 2: (using anonymous block)
Syntax:
Sql> begin
    packagename. procedurename(actual parameters);
    end;
    /
Calling Package Functions:
Method 1: (using SELECT statement)
Sql> select packagename. functionname(actual parameters) from dual;
Method 2: (using anonymous block)
Syntax:
Sql> begin
    varname:=packagename. functioname(actual parameter);
    end;
    /

Date: 25/6/15

EG: sql> create or replace package pj1;
    is
    procedure p1;
    procedure p2;
    end;
    /
This is "Package Specification".

Sql> create or replace package body pj1
    is
    procedure p1
    begin
    dbms_output.put_line('First Procedure');
    end p1;
    procedure p2
    begin
    dbms_output.put_line('Second Procedure');
    end p2;
    end;
    /
This is "Package Body".
Execution:

Sql> exec pj1.p1;
First Procedure
Sql> exec pj1.p2;
Second Procedure

Global Variables:
In Oracle, we are defining Global Variables in package specification only.
EG: sql> create or replace package pj2
    is
    g number(10):=1000;
    procedure p1;
    function f1(a number) return number;
    end;
    /
Sql> create or replace package body pj2
    is
    procedure p1
    is
    x number(10);
    begin
    x:=g/2;
    dbms_output.put_line(x);
    end p1;
    function f1(a number) return number
    is
    begin
    return a*g;
    end f1;
    end;
    /
Execution:
Sql> exec pj2.p1;
    500
Sql> select pj2.f1(4) from dual;
    4000

Serially_reusable pragma:
In Oracle, if you want to maintain state of the global variable or state of the globalised cursor then we must use "serially_reusable" pragma in packages.
Syntax:
pragma serially_resuable;

State of the Global Variable:

Eg: sql> create or replace package pj3
    is
    g number(10):=5;
    pragma serially_reusable;
    end;
    /
Sql> begin
    pj3.g:=90;
    end;
    /
Sql> begin
    dbms_output.put_line(pj3.g);
    end;
    /
Output: 5

State of the Cursor:
Eg: sql> create or replace package pj4
    is
    cursor c1 is select * from emp;
    pragma serially_reusable;
    end;
    /
Sql> begin
    open pj4.c1;
    end;
    /
/*Here cursor is not closed*/
Sql> begin
    open pj4.c1;
    end;
    /
PL/SQL procedure successfully completed
/*cursor is reusable in other blocks*/

Overloading Procedures:
Overloading refers to same name can be used for different purpose. In Oracle, we can also implement overloading procedures by using packages. Overloading procedures having same name with different types or different number of parameters.
EG: sql> create or replace package pj5
    is
    procedure p1(a number, b number);
    procedure p1(x number, y number);
    end;

68

/
Sql> create or replace package body pj5
    is
    procedure p1(a number, b number)
    is
    c number(10);
    begin
    c:= a+b;
    dbms_output.put_line(c);
    end p1;
    procedure p1(x number, y number)
    is
    z number(10);
    begin
    z:=x-y;
    dbms_output.put_line(z);
    end p1;
    end;
    /
Output: sql> exec pj5.p1(4,5);
Error: too many declarations of "P1" match this call

NOTE: In Oracle, when overloading procedures having same number of parameters and also having same types then we are allowed to execute those overloading procedures using "Named notations" only.
Sql> exec (a=>5, b=>4);
    9
Sql> exec(x=> 8, y=> 3);
    5

Forward Declaration:
Declaring a procedures in package body is called "Forward Declaration". Generally, before we are calling private procedures into public procedures first we must implement private procedures in package body before calling otherwise use a forward declaration in package body.
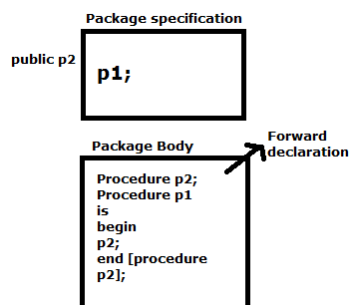
EG: sql> create or replace package pj6
    is

69

```
    procedure p1;
    end;
    /
Sql> create or replace package body pj6
    is
    procedure p2;    /*Forward Declaration*/
    procedure p1
    is
    begin
    p2;
    end p1;
    procedure p2
    is
    begin
    dbms_output.put_line('Private proc');
    end p2;
    end;
    /
```

Execution:
Sql> exec pj6.p1;
    Private proc

Date: 26/6/15

In Oracle, we can also create our own user defined data types by using "type" keyword.
Generally, user defined data types are created in two step process that is first we are creating user defined types from appropriate syntax and then only we are creating a variable from that type. PL/SQL having following user defined types. These are:
1. PL/SQL record
2. Index by table (or) PL/SQL table (or) Associative Array
3. Nested Table
4. Varray
5. Ref Cursor

1. Index by Table (or) PL/SQL table (or) Associative Array
Index by table is a user defined type which is used to store number of data items in a single image. Index by table is an Unbound table.
Basically, Index by table having "Key-Value pairs" i.e., here value field stores actual data where as key field stores Indexes. These Indexes are either integers (or) characters. And also these Indexes are positive, negative numbers. Here key field behaves like a primary key that's why key field does not accepts duplicate values.

Generally Index by tables are used to improves performance of the application because by default these tables are stored in RAM memory and That's why those tables are also called as Memory Tables.

For improve performance of the application Oracle provides special data type "Binary_integer" for key field within "Index by table"

This is an User defined type so we are creating in two step process. That is first we are creating type then only we are creating variable of that type.

Syntax:
1. Type typename is table of datatype(size)
    index by binary_integer;
2. Variablename typename;

EG: sql> declare
    type t1 is table of number(10)
    index by binary_integer;
    v_t t1;
    begin
    v_t(1):=10;
    v_t(2):=20;
    v_t(3):=30;
    v_t(4):=40;
    v_t(5):=50;
    dbms_output.put_line(v_t(1));
    dbms_output.put_line(v_t.first);
    dbms_output.put_line(v_t.last);
    dbms_output.put_line(v_t.prior(3));
    dbms_output.put_line(v_t.next(3));
    dbms_output.put_line(v_t.count);
    v_t.delete(1,3);
    dbms_output.put_line(v_t.count);
    v_t.delete;
    dbms_output.put_line(v_t.count);
    end;
    /
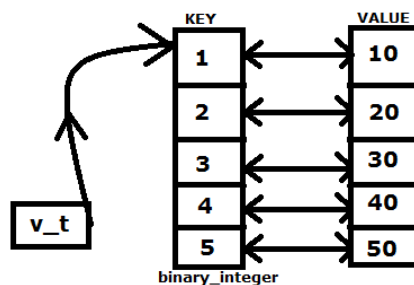
Q) Write a PL/SQL program which transfer all employee names from emp table and storing into index by table and also display content from index by table?

ANS: sql> declare
    type t1 is table of varchar2(10)
    index by binary_integer;
    v_t t1;
    cursor c1 is select ename from emp;
    n number (10):=1;
    begin
    open c1;
    loop
    fetch c1 into v_t(n);
    exit when c1%notfound;
    n:=n+1;
    end loop;
    close c1;
    for I in v_t.first..v_t.last
    loop
    dbms_output.put_line(v_t(i));
    end loop;
    end;
    /

Date: 27/6/15

When a resource table having large amount of data and also when we are trying to transfer date by using "Cursors" then those type of applications degrades performance because cursor internally uses record by record process to overcome this problem if we want to improve performance of the application Oracle 8i introduced "BULK COLLECT" clause.

When we are using "**Bulk Collect**" clause Oracle server selects column data at a time and store that data into "Collections".

Syntax:
select * bulk collect into collectionvarname from tablename where condition;

sql> declare
    type t1 is table of varchar2(10)
    index by binary_integer;
    v_t t1;
    begin
    select ename bulk collect into v_t from emp;
    for I in v_t.first..v_t.last
    loop
    dbms_output.put_line(v_t(i));
    end loop;
    end;
    /

72

Q) Write a PL/SQL program which stores next 10 dates into index by table and also display content from index by table?

ANS: sql> declare

    type t1 is table of date

    index by binary_integer;

    v_t t1;

    begin

    for i in 1..10

    loop

    v_t(i):= sysdate+i;

    end loop;

    for i in v_t.first..v_t.last

    loop

    dbms_output.put_line(v_t(i));

    end loop;

    end;

    /

Q) Write a PL/SQL program which transfer all employees joining dates from emp table into index by table and also display content from index by table?

ANS: sql> declare

    type t1 is table of date

    index by binary_integer;

    v_t t1;

    begin

    select hiredate bulk collect into v_t from emp;

    for i in v_t.first..v_t.last

    loop

    dbms_output.put_line(v_t(i));

    end loop;

    end;

    /
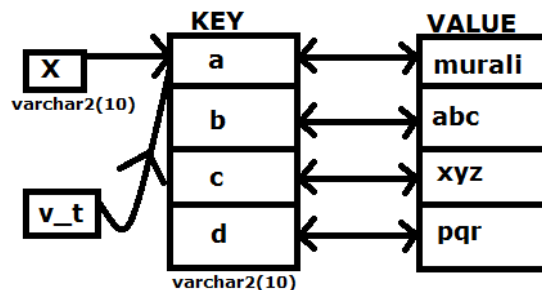
EG: sql> declare

    type t1 is table of varchar2(10)

    index by binary_integer;

    v_t t1;

    x varchar2(10);

```
begin
v_t('a'):='murali';
v_t('b'):='abc';
v_t('c'):='xyz';
v_t('d'):='pqr';
x:='a';
loop
dbms_output.put_line(v_t(x));
x:=v_t.next(x);
end;
/
```

Using RECORD type data type(%rowtype) in value field:
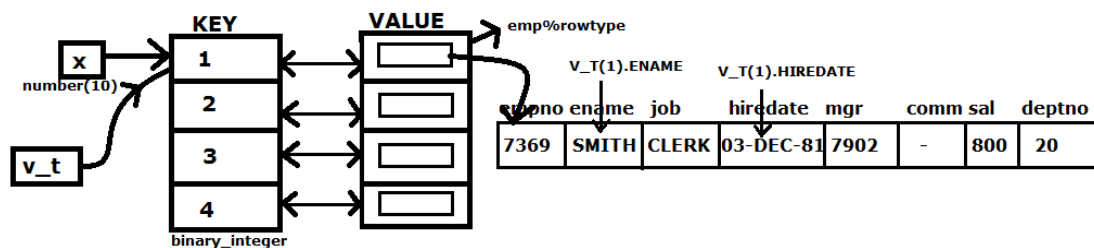

```
Sql> declare
    type t1 is table of emp%rowtype
    index by binary_integer;
    v_t t1;
    x number(10);
    begin
    select * bulk collect into v_t from emp;
    x:=1;
    loop
    dbms_output.put_line(v_t(x).ename||' '||v_t(x).sal||' '||v_t(x).hiredate);
    x:=v_t.next(x);
    exit when x is null;
    end loop;
    end;
    /
```

EG: sql> declare
    type t1 is table of emp%rowtype
    index by binary_integer;
    v_t t1;
    x number(10);
    begin

```
select * bulk collect into v_t from emp;
for i in v_t.first..v_t.last
loop
dbms_output.put_line(v_t(i).ename||' '||v_t(i).sal||' '||v_t(i).job);
end loop;
end;
/
```

Date: 29/6/15

RETURN RESULT SET:
If we want to return large amount of data from Oracle database into client application then we are using following two methods:
1. Using Index by table
2. Using Ref Cursor
If we want to develop these type of applications first we must develop database server applications by using Procedure "OUT" parameters (or) Functions.
Using Index by Table:
Q) Write a PL/SQL database server application using functions which returns large amount of data from emp table into client applications.
ANS: sql> create or replace package pj1

```
            is
            type t1 is table of emp%rowtype
            index by integer_binary;
            function f1 return t1;
            end;
            /
Sql> create or replace package body pj1
    is
    function f1 return t1
    is
    v_t t1;
    begin
    select * bulk collect into v_t from emp;
    return v_t;
    end f1;
    end;
    /
```
Execution by using PL/SQL client:
```
Sql> declare
    x pj1.t1;
    begin
    x:=pj1.f1;
    for i in x.first..x.last
```

```
loop
dbms_output.put_line(x(i).ename||' '||x(i).sal||' '||x(i).deptno);
end loop;
end;
/
```

EXISTS collection method used in Index by tables:
Exists collection method used in "Index by table", "Nested table", "Varray". Exists collection method always returns "Boolean" value either true (or) false.
If we want to test weather requested data is available (or) not available in collection is then we must use exist collection method.
Syntax:
collectionvarname.exists(indexname);

EG: sql> declare
```
    type t1 is table of number(10)
    index by binary_integer;
    v_t t1;
    z boolean;
    begin
    v_t(1):= 10;
    v_t(2):= 20;
    v_t(3):= 30;
    v_t(4):= 40;
    v_t(5):= 50;
    z:=v_t.exists(3);
    if z= true then
    dbms_output.put_line('u r requested index "3" exists with having an element'
||' '||v_t(3));
    else
    dbms_output.put_li8ne('u r requested index does not exists');
    end if;
    for i in v_t.first..v_t.last
    loop
    dbms_output.put_line(v_t(i));
    end loop;
    end;
    /
```

EG: sql> declare
```
    type t1 is table of varchar2(10)
    index by binary_integer;
    v_t t1;
    begin
```

```
select ename bulk collect into v_t from emp;
v_t .delete(3);
for i in v_t.first..v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
end;
/
```
Output: SMITH
ALLEN
ORA-1403: no data found

Whenever Index by table or nested table having gaps and also when we are try to display returns an error "ORA-1403: no data found" to overcome this problem we must use exists collection method.

EG: sql> declare
```
    type t1 is table of varchar2(10)
    index by binary_integer;
    v_t t1;
    begin
    select ename bulk collect into v_t from emp;
    v_t.delete(3);
    for i in v_t.first..v_t.last
    loop
    if v_t.exists(i) then
    dbms_output.put_line(v_t(i));
    end if;
    end loop;
    end;
    /
```

<mark>Nested Table, Varray:</mark>

Oracle 8.0, introduced nested table, varray's. These are also user defined types which is used to store number of data items in a single unit. Before we are using Nested table, Varray's then we must initialize those collections by using "Constructor". Here Constructor name is also same as type name.

Date: 30/6/15

Nested Table:

Oracle 8.0 introduced nested table. Nested table also user defined type which is used to store number of data items in a single unit. Nested table also Unbounded table. In Nested table by default indexes are start with "1" and also these indexes are consecutive and also these indexes are integers only and does not have negative numbers generally we are not allowed to store index by table permanently into oracle database and also we cannot add (or) remove

indexes to overcome these problems oracle introduced extension of the indes by table called Nested table which stores permanently into Oracle database by using SQL and also we can add (or) remove indexes by using "Extend". "Trim" collection methods. This is an user defined type so we are creating "Type" then only we are creating "Variable" as that type.

Syntax 1:

type typename is table of datatype(size);

Syntax 2:

variablename typename:= typename(); /*typename()-> constructor name*/

Nested table having "Exists". "Extend", "Trim", "First", "Last", "Prior", "Next", "Count", "delete(index)", "delete(index1, index n)", "delete" collection methods.

EG: sql> declare
```
    type t1 is table of number(10)
    index by binary_integer;
    v_t t1;
    begin
    v_t(500):=80;
    dbms_output.put_line(v_t(500));
    end;
    /
```
Output: 80

Index by tables are basically sparse(no need to allocate memory) i.e., we are not allocated memory explicitly upto those indexes.

Internally Oracle server only reserved the memory where as Nested tables are basically not "Sparse" i.e., we are reserved the memory explicitly by using "Extend" collection method.

EG: sql> declare
```
    type t1 is table at number(10);
    v_t t1:=t1();
    begin
    v_t(500):= 800;
    dbms_output.put_line(v_t(500));
    end;
    /
```
Error: subscript beyond count.

Solution:

Sql> declare
```
    type t1 is table of number(10);
    v_t t1:= t1();
    begin
    v_t.extend(500);
    v_t(500):= 80;
```

```
    dbms_output.put_line(v_t(500));
    end;
    /
```
Output: 80

EG: sql> declare
```
    type t1 is table of number(10);
    v_t t1:= t1();
    begin
    v_t.extend(5);
    v_t(1):= 10;
    v_t(2):= 20;
    v_t(3):= 30;
    dbms_output.put_line(v_t(1));
    end;
    /
```
Output: 10

NOTE: Without using "Extend" collection method also we can store data directly into nested tables. In this case we must specify actual data within "Constructor" itself.

EG: sql> declare
```
    type t1 is table of number(10);
    v_t t1:= t1(10,20,30,40,50);
    begin
    dbms_output.put_line(v_t.first);
    dbms_output.put_line(v_t.last);
    for i in v_t.first..v_t.last
    loop
    dbms_output.put_line(v_t(i));
    end loop;
    end;
    /
```
Output: 1
             5
             10
             20
             30
             40
             50

EG: sql> declare
```
    type t1 is table of number(10);
    v_t t1;
    v_t2 t1:=t1();
    begin
    if v_t1 is null then
```

79

```
    dbms_output.put_line('v_t1 is null');
    else
    dbms_output.put_line('v_t1 is not null');
    end if;
    if v_t2 is null then
    dbms_output.put_line('v_t2 is null');
    else
    dbms_output.put_line('v_t2 is not null');
    end if;
    end;
    /
```
Output: v_t1 is null
      v_t2 is not null

Explanation:
1. declare
   type t1 is table of number(10);
   v_t1 t1;   /* v_t1 -> null*/
2. declare
   type t1 is table of number(10);
   v_t2 t1:= t1();   /*v_t2 -> |- |- |- |- | */
3. declare
   type t1 is table of number(10);
   v_t t1:= t1();
   begin
   v_t.extend;
   v_t(1) := 10;
   dbms_output.put_line(v_t(1));
   end;
   /   /* v_t -> |10|-|-|-|-| */
Sql> declare
   type t1 is table of varchar2(10);
   v_t t1:= t1('a', 'b', 'c', 'd');
   begin
   dbms_output.put_line(v_t.first);   -> 1
   dbms_output.put_line(v_t.last);   -> 4
   dbms_output.put_line(v_t.prior(3)); -> b
   dbms_output.put_line(v_t.next(3)); -> d
   dbms_output.put_line(v_t.count);   -> 4
   v_t. extend;
   v_t(5):= 'e';
   v_t. extend(2);
   v_t(6) := 'f';
   v_t(7) := 'g';

```
v_t. extend(3,2);
v_t. trim;
dbms_output.put_line(v_t.count);
for i in v_t.first..v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
v_t.delete;
dbms_output.put_line(v_t.count);
end;
/
```

Date: 2/7/15

Difference between Trim, Delete:
Trim→ Delete Indexes last indexes onwards
Delete(Index)→ Delete any element in any position within Nested table

EG: sql> declare
```
    type t1 is table of number(10);
    v_t t1:= t1(10,20,30,40,50);
    begin
    dbms_output.put_line(v_t.count);
    v_t.trim;
    dbms_output.put_line(v_t.count);
    v_t.delete(2);
    for i in v_t.first..v_t.last
    loop
    if v_t.exists(i) then
    dbms_output.put_line(v_t(i));
    end if;
    end loop;
    end;
    /
```

Q) Write a PL/SQL program to retrieve all employee names from emp table and store it into nested table and also display content from nested table?
ANS: sql> declare
```
    type t1 is table of varchar2(10);
    v_t t1:= t1();
    cursor c1 is select ename from emp;
    n number(10):= 1;
    begin
```

81

```
for i in c1
loop
v_t.extend;
v_t(n) := i.ename;
n:= n+1;
end loop;
for i in v_t.first..v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
end;
/
```

Oracle 8i introduced "**Bulk Collect**" clause which is used to transfer data from table into collections. When we are transferring data into Nested table by using "Bulk Collect" clause then we are not allowed to use "**Extend**" collection method. Because internally Bulk Collect clause only reserve memory in Nested table.

EG: sql> declare
```
    type t1 is table of varchar2(10);
    v_t t1:= t1();
    begin
    select ename bulk collect into v_t from emp;
    for i in v_t.first..v_t.last
    loop
    dbms_output.put_line(v_t(i));
    end loop;
    end;
    /
```

<mark>VARRAY:</mark>

Oracle 8.0 introduced Varray. This is a user defined type which is used to store number of data items in a single unit. Basically it is an bounded table. Using Varray's we can store upto 2GB data. In Varray's also always Indexes start with "1" and also these Indexes are numbers and also these Indexes are consecutive. Before we are storing actual data into Varray's also then we must initialize by using Constructor. Here Constructor name is also same as type name. This is an user defined type so we creating in two step process.

Syntax:
1. type typename is varray(maxsize) of datatype(size);
2. variablename typename:= typename();   /* constructor name */

EG: sql> declare
```
    type t1 is varray(10) of number(10);
    v_t t1:= t1(10,20,30,40,50,60,70);
    begin
```

82

```
        dbms_output.put_line(v_t.limit);
        dbms_output.put_line(v_t.count);
        dbms_output.put_line(v_t.first);
        dbms_output.put_line(v_t.last);
        dbms_output.put_line(v_t.prior(3));
        dbms_output.put_line(v_t.next(3));
        v_t.extend(3,4);
        v_t.trim;
        dbms_output.put_line(v_t.count);
        for i in v_t.first..v_t.last
        loop
        dbms_output.put_line(v_t(i));
        end loop;
        v_t.delete;
        dbms_output.put_line(v_t.count);
        end;
        /
```

NOTE: In Varray's it cannot delete particular elements (or) range of elements by using "Delete" collection method. But we can delete all elements by using "Delete" collection method. That's why varray's does not have any gaps.

Q) Write a PL/SQL program to transfer first 10 employee names from emp table and store it into "Varray" and also display content from Varray.

ANS: sql> declare

```
        type t1 is varray(10) of varchar2(10);
        v_t t1:= t1();
        begin
        select ename bulk collect into v_t from emp where rownum<=10;
        for i in v_t.first..v_t.last
        loop
        dbms_output.put_line(v_t(i));
        end loop;
        end;
        /
```

Difference between Index by table, Nested table, Varray:

INDEX BY TABLE	NESTED TABLE	VARRAY
1) Index by tables are unbound tables having key value paris.	1) Nested tables are unbound tables	1) Varrays are bounded table and also stored upto 2GB data.
2) Index by tables are not allowed to store in Oracle database.	2) Nested tables are allowed to store in database permanently by using SQL	2)Varray also allowed to store in database permanently by using SQL
3) We cannot add (or) remove Indexes.	3) We can add (or) remove indexes by using Extend, Trim Collection methods.	3) We can add (or) remove indexes by using Extend, Trim collection methods.
4) Here Indexes are either integers (or) characters and also these indexes are +ve, -ve numbers.	4) By default Indexes are start with "1" and also there is no -ve numbers.	4) By default indexes are start with "1" and also does not have -ve numbers.
5) Index by tables having exists, first, last, prior, next, count, delet(index), delect(index1, index n), delect collection methods	5) Nested tables having existis, first, last, prior, next, count, delet(index), delete (index1, index n), delect collection methods.	5)Varray having exists, limit, extend, trim, first, last, prior, next, count, delete collection methods.

BULK BIND

Oracle 8i introduced Bulk bind process. In this process we are using Bulk updates, Bulk inserts, Bulk deletes by using "forall" statements through collections. In this Bulk bind process we can process all data at a time in a collection by using SQL engine.

Date: 3/7/15

When ever we are submitting PL/SQL block into oracle server then all SQL statements are executed within SQL Engine and also all procedural statements are executed within PL/SQL engine. These type of execution methods are also called as "**Context Switching Execution**" methods whenever PL/SQL block having more number of SQL, Procedural statements then these type of "Context Switching Execution" methods degrades performance of the application. To overcome this problem for improves performance of the application the Oracle 8i introduced Bulk Bind process. This process we are reducing number of context switching through collections.

Without Using Bulk Bind(performance penality for many context switching)
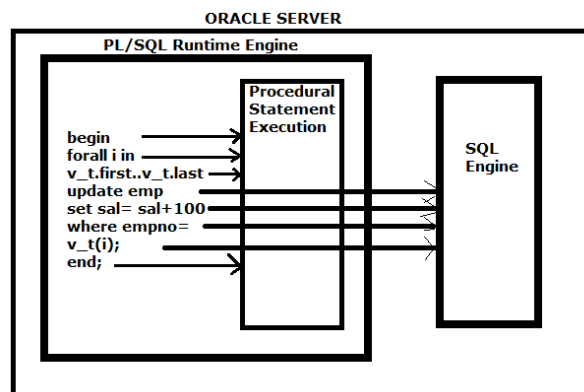
ORACLE SERVER
PL/SQL Runtime Engine
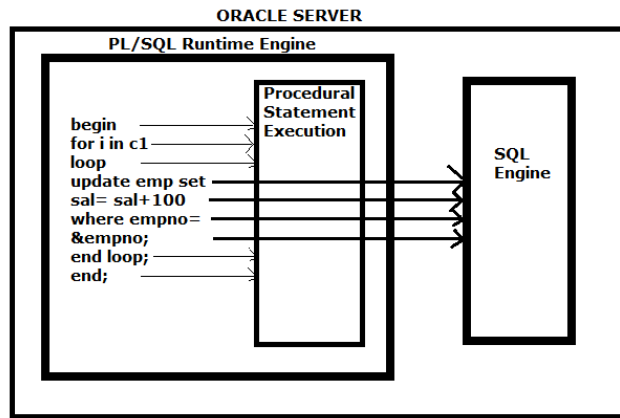
In Bulk Bind process we are using "forall" statements.

Syntax:

forall indexvarname in collectionvarname.first..collectionvarname.last

DML statement where columnname= collectionvarname(indexvarname);

Using Bulk Bind(Improves performance because of much less overhead)

In Bulk Bind process, we are processing all data in a collection at a time by using SQL engine through "forall" statements. Before we are using this process we must "fetch" the data from resource into "Collections" by using "Bulk Collect" clause. That's why "Bulk Bind" is an two stop process.

Step 1: Fetching data from resource into collection by using "Bulk Collect" clause.

Step 2: Process all data in a collection at a time by using SQL engine through "forall" statements (actual bulk bind).

Step 1: Fetching Data from resource into Collection by using Bulk Collect clause.

Before we are using forall statements we must fetching data from resource into collection by using "Bulk Collect" clause always selects column data at a time and store it into "Collections".

In Oracle, Bulk Collect clause is used in

1. SELECT….into clause
2. cursor…….. fetch …….. statement
3. DML ……. Returning….. into clause

1. Bulk Collect clause used in SELECT…… into clause:

85

Syntax:

select * bulk collect into collectionvarname from tablename where condition;

EG: sql> declare

    type t1 is table of emp%rowtype

    index by binary_integer;

    v_t t1;

    begin

    select * bulk collect into v_t from emp;

    for i in v_t.first..v_t.last

    loop

    dbms_output.put_line(v_t(i).ename);

    end loop;

    end;

    /

2. Bulk Collect clause used in cursor......fetch statement:

Syntax:

fetch cursorname bulk collect into collectionvarname[limit anynumber];

"**Limit**" clause is an optional clause which is used to reduce number of values within PGA memory area. Because in PL/SQL collections are executed within PGA memory area.

EG: sql> declare

    type t1 is table of varchar2(10)

    index by binary_integer;

    v_t t1;

    cursor c1 is select ename from emp;

    begin

    open c1;

    fetch c1 bulk collect into v_t limit 5;

    close c1;

    for i in v_t.first..v_t.last

    loop

    dbms_output.put_line(v_t(i));

    end loop;

    end;

    /
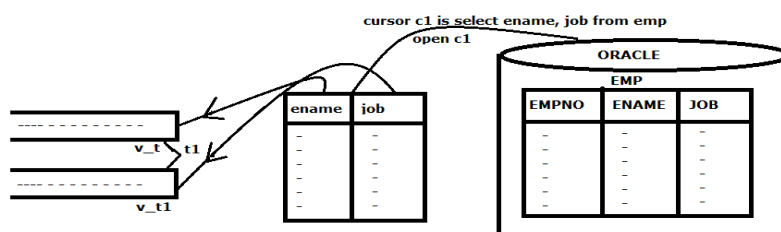
EG: sql> declare

    type t1 is table of varchar2(10)

    index by binary_integer;

```
v_t t1;
v_t1 t1;
begin
open c1;
fetch c1 bulk collect into v_t, v_t1;
close c1;
for i in v_t.first..v_t.last
loop
dbms_output.put_line(v_t(i)||' '||v_t1(i));
end loop;
end;
/
```

Date: 4/7/15

Calculate Elapsed time in PL/SQL blocks:

In PL/SQL if we want to calculate elapsed time then we are using "get_time" function from dbms_utility package. This function always returns number data type.

Syntax:

Variablename:= dbms_utility.get_time;
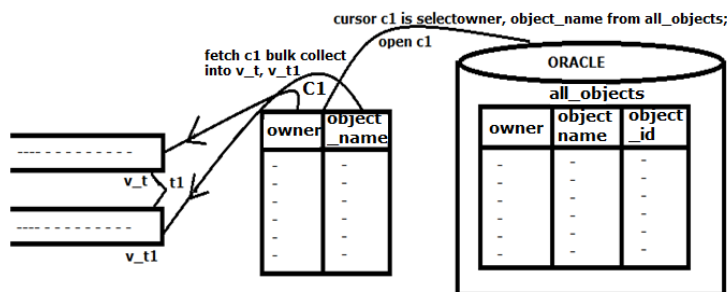
EG: sql> declare

```
type t1 is table of all_objects.object_name%type
index by binary_integer;
v_t t1;
v_t1 t1;
cursor c1 is select owner, object_name from all_objects;
x1 number(10);
x2 number(10);
begin
x1:= dbms_utility.get_time;
for i in c1
loop
null;
end loop;
x2:= dbms_utility.get_time;
dbms_output.put_line('Elapsed time for normal fetch'||' '||(x2-x1)||' '||'hsecs');
```

87

```
    x1:= dbms_utility.get_time;
    open c1;
    fetch c1 bulk collect into v_t, v_t1;
    close c1;
    x2:= dbms_utility.get_time;
    dbms_output.put_line('Elapsed time for bulk fetch'||' '||(x2-x1)||' '||'hsecs');
    end;
    /
```
Output: Elapsed time for normal fetch 101 hsecs
        Elapsed time for bulk fetch 64 hsecs.

Returning into claused are used in dml statements only. These clauses returns transactional data from DML statements and store it into variables.
EG: sql> variable a varchar2(10);
    Sql> update emp set sal= sal+100 where ename= 'KING' returning job into :a;
    1 row updated
    Sql> print a;
Output: A

    PRESIDENT

These **returning into** clauses always returns single record transactions at a time if we want to store multiple records transactions then Oracle 8i onwards we can also use **Bulk Collect** clause. In these **returning into** clause by using collections.

```
EG: sql> declare
    type t1 is table of number(10)
    index by binary_integer;
    v_t t1;
    begin
    update emp set sal= sal+100 where job= 'CLERK' returning sal bulk collect into v_t;
    dbms_output.put_line('Affected number of clerks'||' '|| sql%rowcount);
    for i in v_t.first..v_t.last
    loop
    dbms_output.put_line(v_t(i));
    end loop;
    end;
    /
Output: Affected number of clerks: 4
    3100
    3800
    3650
    4700
```

STEP 2: Process all data in a Collection at a time by using SQL Engine through "forall" statements: (Actual Bulk Bind)

Syntax:
forall index varname in collection var.first..collectionvar.last DML statements where columnname= collection var(index varname);

EG: sql> declare
    type t1 is varray(10) of number(10);
    v_t t1:= t1(10,20,30,40,50);
    begin
    forall i in v_t.first..v_t.last
    update emp set sal= sal+100 where deptno= v_t(i);
    end;
    /

ORACLE SERVER

PL/SQL Runtime Engine

EMP
emp

```
begin
forall i in v..........
update......
else empno= v_t
(i);
end;
```

Procedural
statement
Execution

SQL
Engine

bulk collect   forall
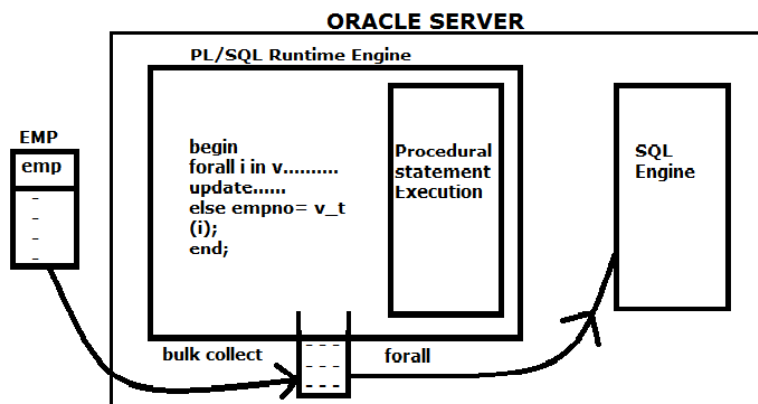
Q) Write a PL/SQL program to retrieve all employee numbers from emp table and store it into index by table using bulk collect clause and also these employee number based modify salaries in emp table by using bulk bind process(using forall statements)?
ANS: sql> declare
    type t1 is table of number(10)
    index by binary_integer;
    v_t t1;
    begin
    select empno bulk collect into v_t from em;
    forall i in v_t.first..v_t.last
    update empno= v_t(i);
    end;
    /

Date: 6/7/15

EG: sql> declare
    type t1 is table of number(10)
    index by binary_integer;
    v_t t1;
    begin
    select ename bulk collect into v_t from emp;
    v_t delete(3);
    forall i in v_t.first..v_t.last
    update emp set sal= sal+100 where empno=v_t(i);
    end;
    /

Error: element at index(3) does not exist.

Whenever index by table (or) nested table having "**Gaps**" then we are not allowed to use "Bulk Bind" process (forall statements). To overcome this problem then we are using "Varray" in Bulk Bind process. Because "Varray" does not have any gaps.
But Varrays stores upto 2GB data. To overcome all these problems Oracle 10G introduced "Indices of" clause within "Bulk Bind" process. This clause is used in "forall" statements.

Syntax:
forall indexvarname in indices of collectionvarname
DML statement where columnname= collectionvarname(indexvarname);
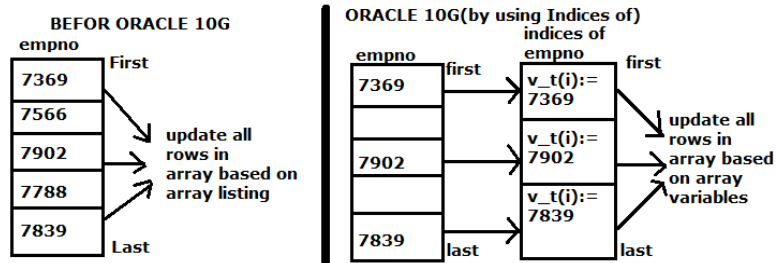
Solution: (indices of → 10G)
Sql> declare
    type t1 is table of number(10)
    index by binary_integer;
    v_t t1;
    begin
    select empno bulk collect into v_t from emp;
    v_t. delete(3);
    forall i in dices of v_t
    update emp set sal= sal+100 where empno= v_t(i);
    end;
    /

Sql%bulk_rowcount:

This attribute also returns "number" data type. If you want to count affected number of rows in each process after Bulk bind(forall statement) then we must use "sql%bulk_rowcount" attribute.

Syntax:

Sql%bulk_rowcount(indexvarname);

EG: sql> declare
    type t1 is varray(10) of number(10);
    v_t t1:=(10,20,30,40,50);
    begin
    forall i in v_t.first..v_t.last
    update emp set sal= sal+100 where deptno= v_t(i);
    for i in v_t.first..v_t.last
    loop
    dbms_output.put_line('affected   number   of   rows   in   deptno'||'   '||v_t(i)||'
'||'Is'||sql%bulk_rowcount(i));
    end;
    /
Output: affected number of rows in deptno 10 is 3

Bulk Inserts:

Q) Write PL/SQL program which transfer all employee names from emp table and store it into another table by using bulk bind process?

ANS: sql> create table target(name varchar2(10));
Sql> declare
    type t1 is table of varchar2(10)
    index by binary_integer;
    v_t t1;
    begin
    select ename bulk collect into v_t from emp;
    v_t(3):= 'murali';
    v_t(4):= 'abc';
    forall i in v_t.first..v_t.last
    insert into target values(v_t(i));
    end;
    /

FORALL and DML errors (or) BULK EXCEPTIONS:

FORALL statements executes multiple DML statements whenever an exception occurs in one of those DML statements then the default behavior is :

91

1. That statement is rolled back and "forall" stops.
2. All previous statements cannot be rolled back.

To overcome these problem Oracle provided a special mechanism for handling these type of exceptions. These are called "Bulk Exceptions".

EG: sql> declare
    type t1 is varray(10) of number(10);
    v_t t1:= t1(10,20,30,40,50);
    begin
    v_t(3):= null;
    v_t(4):= null;
    forall i in v_t.first..v_t.last
    insert into target values(v_t(i));
    end;
    /

Error: ORA-01400: cannot insert NULL values into SNO.

If we want to handle "Bulk Exceptions" then we must we "save exceptions" clause within "forall" statements.

Syntax:
forall indexvarname in collectionvar.first..collectionvar.last save exception
DML statement where columnname= collectionvarname(indexvarname);

"**Save Exceptions**" clause tell to Oracle save exceptions information in "sql%bulk_exceptions" collection and continue processing of all DML statements sql%bulk_exceptions is a pseudo collection which having only one collection method "**COUNT**".

Date: 7/7/15

SQL%BULK_EXCEPTION:
Sql%bulk_exceptions is a pseudo collection which having only one collection method count.
For each exception raised in "Bulk Bind" process then Oracle server internally populates Sql%bulk_exceptions pseudo collection. This pseudo collection having only one collection method count.
Sql%bulk_exceptions pseudo collection having two fields these are:
1. ERROR_INDEX
2. ERROR_CODE
Error_index field stores index number in collection where the error is occurred.
Error_code stores Oracle error numbers where error is occurred in collections.

SQL%BULK_EXCEPTIONS

error_index	error_code
1	1400
2	1400
3	2299
4	1403

Steps to handle Bulk Exceptions in PL/SQL block:

Step 1: In Exception section of the PL/SQL block count number of exceptions in a collection through Sql%bulk_exceptions pseudo collections.
Syntax: varname:= Sql%bulk_exceptions.count;

Step 2: Before we are catching errors then we must use "Save Exceptions" clause in forall statements.
Syntax:
forall indexvarname in collection var.first.. collection var.last save exceptions
DML statement where columnname= collectionvarname(index varname);
EG: sql> declare
    type t1 is varray(10) of number(10);
    v_t t1:= t1(10,20,30,40,50);
    z number(10);
    begin
    v_t(2):= null;
    v_t(3):= null;
    forall i in v_t.first..v_t.last save exceptions /* allows processing forall statements even
    insert into target values(v_t(i));   after error occurred also*/
    exception
    when others then
    z:= sql%bulk_exceptions.count;
    dbms_output.put_line(z);
    end;
    /
 Output: 2 --→ errors are occurred.
Sql> select * from target;
    SNO

    10
    40
    50

NOTE: If we want to display error_index, error_code values then we must iterate sql%bulk_exceptions pseudo collection with in a loop.
EG: sql> declare

```
type t1 is varra(10) of number(10);
v_t t1:= t1(10,20,30,40,50);
z number(10);
begin
v_t(2):= null;
v_t(3):= null;
forall i in v_t.first..v_t.last save exceptions
insert into target values(v_t(i));
exception
when others then
z:= sql%bulk_exceptions.count;
dbms_output.put_line(z);
for i in 1..z
loop
dbms_output.put_line(Sql%bulk_exceptions(i).error_index);
dbms_output.put_line(Sql%bulk_exceptions(i).error_code);
end loop;
end;
/
```

Output:

2	/*Exception count*/
2	/*Error_Index*/
1400	/*Error_code*/
3	/*Error_Index*/
1400	/*Error_code*/

BULK DELETES:

```
Sql> declare
type t1 is varray(10) of number(10);
v_t t1:= t1(10,20,30,40,50,60);
begin
forall i in v_t.first..v_t.last
delete from emp where deptno= v_t(i);
end;
/
```

NOTE: "**forall**" statements is the most important performance enhancement in PL/SQL through the "forall" statement we can executes Bulk DML's. In this process whenever error is occurred handled those errors by using sql%bulk_exceptions pseudo collection.

PL/SQL Record: This is an user defined type which is used to represent different data types into single unit it is also same as structures in 'C' language.

This is an User defined type so we are creating a two step process i.e., first we are creating type then only we care creating a variable from that type.

Syntax:

1. type typename is record(attribute 1 datatype(size),……..);
2. variablename typename;

94

EG: sql> declare
  type t1 is record(a1 number(10), a2 varchar2(10), a3 number(10));
  v_t t1;
  begin
  v_t.a1:= 101;
  v_t.a2:= 'murali';
  v_t.a3:= 3000;
  dbms_output.put_line(v_t.a1||' '||v_t.a2||' '||v_t.a3);
  end;
  /
Output: 101 murali 3000

PL/SQL record used in package:
Sql> create or replace package pj1
  is
  type t1 is record(a1 number(10), a2 varchar2(10), a3 number(10));
  procedure p1;
  end;
  /
Sql> create or replace package body pj1
  is
  procedure p1
  is
  v_t t1;
  begin
  select empno, ename, sal into v_t from emp where ename= 'SMITH';
  dbms_output.put_line(v_t.a1||' '||v_t.a2||' '||v_t.a3);
  end p1;
  end;
  /
Sql> exec pj1.p1;
  7369 SMITH   3800

REF Cursors (or) Cursor Variables (or) Dynamic Cursors:
Oracle 7.2 introduced REF Cursors.
REF Cursors are user defined types which is used to process multiple records and also this is an record by record process.
Generally in static cursors we can execute only one SELECT statement for a single active set area. Where as if we want to execute number of SELECT statement dynamically for a single active set area. Then we must use REF Cursors are also called as Dynamic Cursor.
Generally, we are not allowed to pass static cursor as parameter to sub programs as we are allow to pass REF Cursor as parameter to the sub programs because basically REF Cursor is a user defined type.

95

In Oracle we can also pass an user defined types as parameter to the sub programs.

Date: 8/7/15

Generally, cursors does not return multiple records into client applications.
Whereas Ref cursors returns multiple records from database server into9 client applications.
All database systems having two types of Ref cursors.
1. Strong Ref cursors
2. Weak Ref cursors
Strong Ref cursors is a Ref cursors which having return type. Whereas Weak Ref cursors is a Ref cursors also an user defined type so we are creating in two step process i.e., first we are creating type then only we are creating variable from that type. That's why Ref cursors is also called as Cursor Variable.

Syntax:
1. type typename is ref cursor return recordtypedatatype;
   variablename typename;
   variablename-------> strong ref cursor variable

2. type typename is ref cursor;
   variablename.typename;
   variablename------→ weak ref cursor variable
In Ref Cursors we can specify SELECT statement through "open……for" clause.

Syntax:
open refcursorname for select * from tablename where condition;

This clause is used in executable session of the PL/SQL block.

EG: sql> declare
    type t1 is ref cursor;
    v_t t1;
    i emp%rowtype;
    begin
    open v_t for select * from emp where sal>2000;
    loop
    fetch v_t into i;
    exit when v_t%notfound;
    dbms_output.put_line(i.ename||' `||i.sal);
    end loop;
    close v_t;
    end;
    /

Q) Write a PL/SQL program by using ref cursor whenever user entered deptno "10" then display 10th department details from emp table whenever user entered deptno 20 then display 20th dept details from dept table?

ANS: sql> declare

```
    type t1 is ref cursor;
    v_t t1;
    i emp%rowtype;
    j dept%rowtype;
    v_deptno number(10):= &deptno;
    begin
    if v_deptno=10 then
    open v_t for select * from emp where deptno= v_deptno;
    loop
    fetch v_t into i;
    exit when v_t%notfound;
    dbms_output.put_line(i.ename||' `||i.sal||' `||i.deptno);
    end loop;
    elsif v_deptno=20 then
    open v_t for select * from dept where v_deptno=20;
    loop
    for v_t into j;
    exit when v_t%notfound;
    dbms_output.put_line(j.deptno||' `||j.dname||' `||j.loc);
    end loop;
    end if;
    close v_t;
    end;
    /
```

Output: Enter value for deptno= 10

```
    CLARK        3000  10
    KING         7600  10
    MILLER       5300  10
```

Sql> /

Enter value for deptno= 20

```
    20     RESEARCH   DALLAS
```

SYS_REFCURSOR:

Oracle 9i introduced "sys_refcursor" predefined type in place of "Weak Refcursor".

Syntax:

Refcursorvariablename sys_refcursor;

EG: sql> declare

```
    v_t sys_refcursor;
    i emp%rowtype;
```

```
begin
open v_t for select * from emp;
loop
fetch v_t into i;
exit when v_t%notfound;
dbms_output.put_line(i.ename||' '||i.sal);
end loop;
close v_t;
end;
/
```

Passing sys_refcursor as IN parameter to the Stored Procedure:

Q) Write a PL/SQL stored procedure for passing sys_refcursor as IN parameter which display all employee names and their salaries from emp table?

ANS: sql> create or replace procedure p1(v_t in sys_refcursor)

```
        is
        i emp%rowtype;
        begin
        loop
        fetch v_t into i;
        exit when v_t%notfound;
        dbms_output.put_line(i.ename||' '||i.sal);
        end loop;
        close v_t;
        end;
        /
```

NOTE: In all databases whenever we are passing refcursor as IN parameter to the stored procedure then we are not allowed to use "open……for" statement. Because "open……for" return values from the procedures.

Execution:

```
Sql> declare
    v_t sys_refcursor;
    open v_t for select * from emp;
    p1(v_t);
    close v_t;
    end;
    /
```

Q) Write a PL/SQL stored procedure for passing refcursor as OUT parameter which returns employee details from emp table?

ANS: sql> create or replace procedure p1(v_t out sys_refcursor);

```
    is
```

98

```
begin
open v_t for select * from emp;
end;
/
```

Execution: (using PL/SQL client)
```
Sql> declare
v_t sys_refcursor;
i emp%rowtype;
begin
p1(v_t);
loop
fetch v_t into i;
exit when v_t%notfound;
dbms_output.put_line(i.ename||' `||i.sal);
end loop;
close v_t;
end;
/
```

Date: 9/7/15

Q) Write a PL/SQL stored function which returns employee details from emp table by using sys_refcursors?

ANS: sql> create or replace function f1
```
return sys_refcursor
is
v_t sys_refcursor;
begin
open v_t for select * from emp;
return v_t;
end;
/
```
Execution: select f1 from dual;

LOCAL SUB PROGRAMS:
Local Sub Programs are name PL/SQL blocks which is used to solve particular task. These sub programs does not have "create or replace" keywords. And also these sub programs does not store permanently into database. Oracle having 2 types of Local Sub Programs.
1. Local Procedures
2. Local Functions.
These Local Sub Programs are used in Anonymous blocks, Stored Procedures. Local Sub Programs must be defined in "Bottom of Declare Session" in anonymous block, stored procedure and also "**CALL**" these sub programs in immediate "Executable Session".

99

Syntax:

Declare

➔ Variable declarations, Constant declarations;

➔ Types declarations;

➔ Cursor declarations;

➔ Procedure procedurename(formal parameters)

   Is/As

   Begin

   [Exception]

   ----

   end [procedurename];

➔ Function functionname(formal parameters) return datatype

   Is/As

   Begin

   Return expression;

   [Exception}

   End [Function name];

   Begin

   Procedurename(actual parameters);

   Varname:= functionname(actual parameter);

   End;

EG: sql> declare

   procedure p1

   is

   begin

   dbms_output.put_line('Local Procedure');

   end p1;

   begin

   p1;

   end;

   /

Output: Local Procedure

EG: sql> create or replace procedure p2

   is

   procedure p1

   is

   begin

   dbms_output.put_line('Local Procedure');

   end p1;

```
    begin
    p1;
    end;
    /
```

Execution: sql> exec p2;
    Local Procedure

EG: (Reusing Local Procedures by "calling" in anonymous blocks)
Sql> declare

```
    type t1 is refcursor return emp%rowtype;
    v_t t1;
    procedure p1(p_t in t1)
    is
    i emp%rowtype;
    begin
    loop
    fetch p_t into i;
    exit when p_t%notfound;
    dbms_output.put_line(i.ename||' '`||i.sal);
    end loop;
    end p1;
    begin
    open v_t for select * from emp where rownum<=10;
    p1(v_t);
    close v_t;
    open v_t for select * from emp where ename like 'M%';
    p1(v_t);
    close v_t;
    open v_t for select * from emp where job= 'CLERK';
    p1(v_t);
    close v_t;
    end;
    /
```
EG: (Using Stored Procedure)
Sql> create or replace procedure p2

```
    is
    type t1 is recursor return emp%rowtype;
    v_t t1;
    procedure p1(p_t in t1)
    is
    i emp%rowtype;
    begin
    loop
    fetch p_t into i;
```

```
        exit when p_t%notfound;
        dbms_output.put_line(i.ename||' '||i.sal);
        end loop;
        end p1;
        begin
        open v_t for select * from emp where rownum<=10;
        p1(v_t);
        close v_t;
        open v_t for select * from emp where ename like 'M%';
        p1(v_t);
        close v_t;
        open v_t for select * from emp where job= 'CLERK';
        p1(v_t);
        close v_t;
        end p2;
        /
```

Output: sql> exec p2;
        Local Procedure

EG: (Using Packages and Procedure and Refcursors)

```
sql> create or replace package pj1
        is
        type t1 is refcursor return emp%rowtype;
        type t1 is refcursor return dept%rowtype;
        procedure p1(v_t1 out t1);
        procedure p2(v_t2 out t2);
        end;
        /
```

```
Sql> create or replace package body pj1
        is
        procedure p1(v_t1 out t1)
        is
        begin
        open v_t1 for select * from emp;
        end p1;
        procedure p2(v_t2 out t2)
        is
        begin
        open v_t2 for select * from dept;
        end p2;
        end;
        /
```

Exception: (Using Bind Variable)

Sql> variable a refcursor;

Sql> variable b refcursor;
Sql> exec pj1.p1(:a);
Sql> exec pj1.p2(:b);
Sql> print a b;
NOTE: In packages we are not allowed to create "REFCURSOR" variables.
EG: sql> create or replace package pj2

    is
    type t1 is refcursor;
    v_t t1;
    end;
Warning: package created with compilation errors.
ERROR: PLS-00094: cursor variable cannot be declared as part of a package.

Date: 10/7/15

Passing Index by tables as IN parameter to the Local Procedure:
EG: sql> declare

    type t1 is table of emp%rowtype
    index by binary_integer;
    v_t t1;
    procedure p1(p_t in t1)
    is
    begin
    for i in v_t.first..v_t.last
    loop
    dbms_output.put_line(v_t(i).ename);
    end loop;
    end p1;
    begin
    select * bulk collect into v_t from emp;
    p1(v_t);
    end;
    /

Passing Index by table as OUT parameter to the Local Procedure:
Sql> declare

    type t1 is table of emp%rowtype
    index by binary_integer;
    v_t t1;
    procedure p1(p_t out t1)
    is
    begin
    select * bulk collect into p_t from emp;
    end p1;

```
begin
p1(v_t);
for i in v_t.first..v_t.last
loop
dbms_output.put_line(v_t(i).ename);
end loop;
end;
/
```

Using Index by table as Return type from Local Funtion:

```
Sql> declare
    type t1 is table of emp%rowtype
    index by binary_integer;
    procedure p1(p_t in t1)
    is
    begin
    for i in p_t.first..p_t.last
    loop
    dbms_output.put_line(p_t(i).ename);
    end loop;
    end p1;
    function f1 return t1
    is
    v_t t1;
    begin
    select * bulk collect into v_t from emp;
    return v_t;
    end f1;
    begin
    p1(f1);
    end;
    /
```

DBMS_UTILITY package:

DBMS_UTILITY package internally having "Index by Table" and also this package having two procedures. These are:

1. Comma_to_table
2. Table_to_comma

1. Comma_to_tabel: This Procedure is used to convert "," separated strings into "Index by Table" values. These Procedure accepts "3" parameters.

Syntax:

dbms_utility.comma_to_table(string variablename, binary_integer variablename, Index by table variablename);

2. Table_to_comma: It is used to convert Index by table values into ","(comma) separated strings.

Syntax:

dbms_utility.table_to_comma(Index by table variablename, binary_integer variablename, String variablename);

Before we are using these Procedures we must create "Index by table" variable by using "uncl_array" type from "dbms_utility"package in declare section of the PL/SQL block.

Syntax:

Index by table variablename dbms_utility.uncl_array;

Q) Write a PL/SQL program which converts comma separated string into Index by table values and also display content from Index by table?

ANS: sql> declare

```
    v_t dbms_utility.uncl_array;
    x binary_integer;
    str varchar2(100);
    begin
    str:= 'a,b,c,d,e,f';
    dbms_utility.comma_to_table(str, x, v_t);
    for i in v_t.fist..v_t.last
    loop
    dbms_output.put_line(v_t(i));
    end loop;
    end;
    /
```

Q) Write a PL/SQL program to retrieve all department names from dept table and display into comma separated string by using dbms_utility package?

ANS: sql> declare

```
    v_t dbms_utility.uncl_array;
    str varchar2(100);
    begin
    select dname bulk collect into v_t from dept;
    dbms_utility.table_to_comma(v_t, x, str);
    dbms_output.put_line(str);
    end;
    /
```

Date: 11/7/15

UTL_FILE package:

Oracle 7.3 introduced "UTL_FILE" package. This package is used to write data into an Operating System file and also read data from an Operating System fie. If you want to write data into file then we using "PUTF" procedure and also if we want to read data from file then we are using "get_line" procedure from "UTL_FILE" package.

In Oracle, before we are using "UTL_FILE" package "LOB's" then we must create alias directory related to physical directory based on following syntax.

Syntax:
create or replace directory directoryname as 'pathe';

Before we are creating alias directory then data base administrator must give create and directory system privilege given to the user.

Syntax:
grant create an directory to username1, username2,……;

Sql> conn sys as sysdba;
    Password: sys
Sql> grant create any directory to scott;
Sql> conn scott/tiger;
Sql> create or replace directory XYZ as 'C:\';
       Directory Created

Before we are performing READ, WRITE operations in operating system file then we must give READ, WRITE object privilege on alias directory by using following syntax.

Syntax:
grant read, write on directory directoryname to username1, username2,…..;
sql> conn sys as sysdba;
    password: sys
sql> grant read, write on directory XYZ to scott;
sql> conn scott/tiger;

Writing DATA into OS file:
Step 1: Before we are opening the file we must create a file pointer variable by using "file_type" from "UTL_FILE" package in declare section of the PL/SQL block.
Syntax:
filepointer variablename utl_file.file_type;

Step 2: Before we are writing data into an external file then we must open the file by using "**FOPEN**" function from "UTL_FILE" package. This function is used in "Executable section" of the PL/SQL block. This function accepts "3" parameters and returns "file_type".
Syntax:
Filepointervariablename:= utl_file.fopen('alias directoryname', 'filename', 'mode');
MODE-----> 3 types(w→Write, r→Read, a→Append)

Step 3: If we want to write data into file then we are using "**PUTF**" procedure from "UTL_FILE" package.
Syntax:

utl_file.putf(filepointervariablename, 'content');

Step 4: After writing data into file then we must close the file by using "**FCLOSE**" procedure from "UTL_FILE" package.
Syntax:
Utl_file.fclose(filepointer variablename);
EG: sql> declare
    fp utl_file.file_type;
    begin
    fp:= utl_file.fopen('XYZ', 'file1.txt', 'w');
    utl_file.putf(fp, 'abcdefg');
    utl_file.fclose(fp);
    end;
    /

Q) Write a PL/SQL program to retrieve all employee names from emp table and store it into an Operating System file by using "UTL_FILE" package.
ANS: sql> declare
    cursor c1 is select * from emp;
    fp utl_file.file_type;
    begin
    fp:= utl_file.fopen('XYZ', 'file1.txt', 'w');
    for i in c1
    loop
    utl_file.putf(fp, i.ename);
    end loop;
    utl_file.fclose(fp);
    end;
    /

NOTE: Whenever we are writing data into an external file by using "UTL_FILE" package "PUTF" procedure then automatically table column data stored in horizontal manner within a file. To overcome this problem if we want to store our own format then we must use "%s" access specifier within second parameter of the "PUTF" procedure. And also if we want to store every data item in next line then we are using "/n" along with "%s" access specifier. This parameter must be specified within ' '(single quotes).
EG: sql> declare
    fp utl_file.file_type;
    cursor c1 is select * from emp;
    begin
    fp:= utl_file.fopen('XYZ', 'file2.txt', 'w');
    for i in c1
    loop
    utl_file.putf(fp, '%s/n', i.ename);

```
    end loop;
    utl_fie.fclose(fp);
    end;
    /
```

Date: 13/7/15

Read data from Flat Files:
If we want to read data from an Operating System file then we must use "get_line" procedure from "utl_file" package.
Syntax:
utl_file.get_line(filepointer variablename, buffer variablename);
Before we are using this procedure then we must use "**Read mode (r)**" in "**fopen**" function from "utl_file" package.

Q) Write a PL/SQL program by using utl_file package read data from file1.txt and display that data?
ANS: sql> declare
```
    fp utl_file.file_type;
    x varchar2(200);
    begin
    fp:= utl_file.fopen('XYZ', 'file1.txt', 'r');
    utl_fie.get_line(fp,x);
    dbms_output.put_line(x);
    utl_file.fclose(fp);
    end;
    /
```
Output: abcdefg

Q) Write a PL/SQL program to retrieve number of data items from 'file2.txt' and display that data by using utl_file package?
ANS: sql> declare
```
    fp utl_file.file_type;
    x varchar2(200);
    begin
    fp:= utl_file.fopen('XYZ', 'file2.txt', 'r');
    loop
    utl_file.get_line(fp,x);
    dbms_output.put_line(x);
    end loop;
    utl_file.fclose(fp);
    exception
    when no_data_found then
    null;
```

```
end;
/
```

NOTE: When we are reading multiple data items from an external file by using utl_file package then Oracle server returns and error "**ORA-1403: no data found**" whenever control reached end of file to overcome this problem we must use "**no_data_found**" exception name.

SQL*LOADER (or) SQL LOADER:

Sql Loader is an utility program which is used to transfer data from "Flat Files" into Oracle database. Sql Loader is also called as "**BULK LOADER**".

SQL Loader always executes "**Control files**" this file extension is "**.ctl**". Based on the type of flat file we are creating a control file into SQL Loader tool. Then only SQL Loader transfer this flat file data and store it into Oracle database.

During this process SQL Loader tool automatically creates an "**LOG file**" as same name as "Control file". This "**LOG file**" stores all other files information and also these files stores Loaded, Rejected number of records "numbers". And also this file stores Oracle error numbers, error messages. This file also stores "Elapsed time".
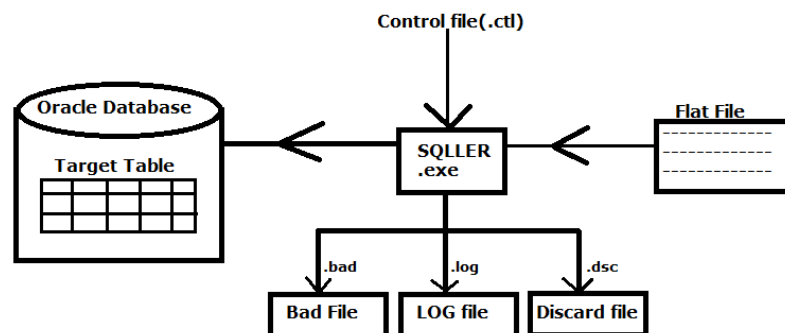
Whenever we are transferring data from flat file into Oracle database based on some reasons some records are not loaded into database those records are also called as "**Rejected records**" (or) "**Discard records**". Those rejected (or) discarded records also stored in "Bad Files", "Discard Files". Bad files stores rejected records based on data type mismatch, business roles violation. Discard file also stores rejected records based on when condition within "Control file".

FLAT FILES:

Flat file is an structured file which contains number of records. All database systems having two types of Flat files.
1. Variable Length Record Flat file.
2. Fixed Length Record Flat file.

1. Variable Length Record Flat file:

A Flat file which having "Delimiter" is called "Variable length record Flat file".
EG: 101, abc, 2000
      102, kkk, 9000
      103, pqr, 7000

2. Fixed Length Record Flat File:

A Flat File which does not have "Delimiter" is called "Fixed Length Record Flat File".
EG: 101 abc 2000
    102 kkk 9000
    103 pqr 7000

Date: 14/7/15

Creating a Control File for variable length record Flat file:

Always control file execution start with "Load data" clause. After load data clause we must specify path of the flat file by using "INFILE" clause.
Syntax:
infile "path of the flat file"

NOTE: We can also use flat file data within "control file" itself. In this case we must use "*****" in place of path of the flat file within "INFILE" clause and also we must specify begin data clause in the above of the flat file data.
EG: load data
    infile *

    ----
    begindata
    101,abc,2000
    102,xyz,5000
After specifying path of the flat file then we are loading data into Oracle database by using "into table tablename" clause.
Before "into table tablename" clause we are using "insert/append/replace/truncate" clauses. If your target table is an empty table then we are using "insert" clause. By default clause is "INSERT" clause.
Based on the flat files data after "into table tablename" clause we are specifying following clauses. These are
1. Fields terminated by "Delimiter name"
2. Optionally enclosed by "Delimiter name"
3. Trailing "Nullcols".
After specifying these clauses then we must specify target table columns within paranthesis.

CONTROL FILE(.CTL):
Syntax:
load data
infile 'path of flat file'
badfile 'path of flat file'
discardfile 'path of discard file'
insert/append/replace/truncate

110

into table tablename
fields terminated by 'delimiter name'
optionally enclosed by
'delimiter name'
trailing nullcols
(col1,col2,col3,......)

Invoking SQLLOADER:
Start→Run→cmd→D:\sqlldr userid= scott/tiger(press "ENTER")
                    Control= path of the control file.
Step 1: Creating text file file1.txt
101,abc,3000
102,xyz,4000
103,pqr,9000
104,mmm,5000
Save this "note pad" file as "**file1.txt**".

Step 2: Creating table in Oracle database.
Sql> create table target(empno number(10), ename varchar2(10), sal number(10));

Step 3: Creating Control file(.ctl) by opening new note pad and write following code
load data
infile 'C:\ file.txt'
insert
into table target
fields terminated by ','
(empno, ename, sal)
Now save the file as "save as" → "murali.ctl" and "type"→ "all types".

Step 4: Execution process
Goto Command prompt and to the directory where "**sql loader"** is stored.
C:\ sqlldr userid= scott/tiger
      Control= C:\ murali.ctl
Now goto Oracle database and check to know the data is dumped into target file.
Sql> select * from target;
During this process "SQL LOADER" automatically creates a "LOG File" as same name as "Control file". This "LOG file" stores all other files information Loaded, Rejected number of records "Numbers". And also "LOG file" stores Oracle error numbers, error message.
            (OR)
Inserting data directly through Control File .
load data
infile *
insert
into table target

111

field terminated by ','
(empno, ename, sal)
begindata
101,abc,2000
102,xyz,5000
Now execute the "Control files" as previous using command prompt.
After that check the target file to know whether the data is transferred.
Sql> select * from target;

CONSTANT, FILLER clauses used in Control File(.ctl):
CONSTANT: If we want to store default values into Oracle database by using SQL Loader then we must use "CONSTANT" clause.
Syntax:
Columnname constant 'default value'
➔ /* default value means 'NUMBERS also in single quotes (' ')

If flat file having less number of fields and also if target table require more number of fields then only we are using "CONSTANT" clause.
FILLER: If you want to skip columns from flat files then we must use "FILLER" clause.
Syntax:
columnname filler (or) anyname filler

When flat file having more number of columns and also target table requires less number of columns then only we are using "FILLER" clause.

Step 1: creating flat file file1.txt
101, abc
102, xyz
103,pqr

Step 2: Creating target table in database
Sql> create table target(empno number(10), ename varchar2(10), loc varchar2(10));

Step 3: Creating Control file (.ctl) by using Constant and Filler
load data
infile 'C:\ file.txt'
insert
into table target
fields terminated by ','
(empno, ename filler, loc constant 'hyd')
Here second column ename is skipped by using "FILLER" clause and 'loc' column is filled with "hyd" by usind "CONSTANT" clause.

BAD file:

This file extension is .bad. Bad files are automatically created as same name as "flat file name". We can also create bad file explicitly by specifying "bad file" clause within "control file".

Syntax:

badfile 'path of badfile'

Bad files stores rejected records based on following reasons:

1. Data type mismatch
2. Business rule violation

1. Data type mismatch

EG: create a flat file 'file1.txt'

101,abc

102,xyz

103,pqr

104,kkk

➔ Creating a Target file

Sql> create table target(empno number(10), ename varchar2(10));

➔ Creating a Control file(.ctl)

load data

infile 'C:\ file1.txt'

insert

into table target

fields terminated by ','

(empno, ename)

➔ Execution of 'text1.txt' file

C:\ sqlldr userid= scott/tiger

Date: 15/7/15

➔ Bad file file1.bad

'102',abc

'103',pqr

2. Based on Business Rule Violations:

EG:

Step 1: Creating a flat file "file1.txt"

101,abc,9000

102,xyz,2000
103,pqr,8000
104,kkk,3000

Step 2: Creating a table in Oracle database.
Sql> create table targer(empno number(10), ename varchar2(10), sal number(10) check(sal>5000));

Step 3: Creating a Control file "murali.ctl"
load data
infile 'C:\ file1.txt'
insert into table target
fields terminated by ','
(empno, ename, sal)

Step 4: Execute through command prompt

Step 5: sql> select * from target;
Bad file:
102, xyz, 2000
104, kkk, 3000

NOTE: Whenever flat file having null values in last fields of a column then SQLLDR rejected those null value records. And also those null value records are automatically stores in "Bad file". If we want to store those null value records in Oracle database then we must use "Trailing nullcols" clause within control file.
EG:
➔ Creating text file file1.txt
101,abc,9000
102,xyz
103,pqr
104,kkk,3000
➔ Creating target file in Oracle database
Sql> create table target(empno number(10), ename varchar2(10), sal number(10));
➔ Creating control file "murali.ctl"
load data
infile 'C:\ file1.txt'
insert into table target
fields terminated by ','
trailing nullcols
(empno, ename, sal)
➔ Executing through command prompt
➔ Sql> select * from target;

RECNUM:

This clause is used in control file only. This clause automatically assigns numbers to loaded, rejected number of records.

Syntax:

Columnname recnum

EG:

➔ Creating flat file file1.txt

101,abc

'102',xyz

'103',pqr

104,kkk

➔ Creating table in Oracle database

Sql> create table target(empno number(10), ename varchar2(10), rno number(10));

➔ Creating a Control file(.ctl) 'murali.ctl'

load data

infile 'C:\ file1.txt'

insert into table target

fields terminated by ','

(empno, ename, rno recnum)

➔ Sql> select * from target;

EMPNO	ENAME	RNO
101	abc	1
104	kkk	4

Bad file:

'102',xyz

'103',pqr

DISCARD FILE(.DSC):

Discard file also stores rejected records based on when condition within "Control fiel" (.ctl). This when condition must be specified in after "into table tablename" clause within "control file".

Syntax:

when condition

Discard file extension is (.dsc). Generally Bad files are created automatically but discard files are not created automatically in this case we must specify discard file within control file by using "discardfile" clause.

Syntax:

discardfile 'path of the discard file'

EG:

➔ Creating a flat file file1.txt

101,abc,10

102,xyz,20

103,pqr,10
104,kkk,30
➔ Creating a table in Oracle database
Sql> create table target(empno number(10), ename varchar2(10), deptno number(10));
➔ Creating a control file 'muarli.ctl'
load data
infile 'C:\file1.txt'
discardfile 'C:\file1.dsc'
insert into table target
when deptno='10'
fields terminated by ','
(empno, ename, deptno)
➔ Executing through command prompt
➔ Sql> select * from target;

EMPNO	ENAME	DEPTNO
101	abc	10
103	pqr	10

Discard file:
102,xyz,20
104,kkk,30

NOTE: Always "WHEN" condition values must be specified within single quotes (' ').
NOTE: In "WHEN" clause we are not allowed to use other than "=", "<>" Relational operator.
NOTE: In "WHEN" clause we are not allowed to use logical operator "OR" but we are not allowed to use logical operator "AND".

Functions used in Control Files:
We can also use Oracle predefined, user defined functions in "Control files".
In this case we must specify functions functionality within double quotes(" ") and also we must use Colon operator(:) infront of the column name within functions functionality.

Syntax:
columnname "functionname(:columnname)"

EG:
➔ Creating a flat file file1.txt
101,abc,m
102,xyz,f
103,pqr,m
104,kkk,f
➔ Creating table in Oracle database.
Sql> create table target(empno number(10), ename varchar2(10), gender varchar2(10));
➔ Creating a control file 'murali.ctl'

116

load data
infile 'C:\ file1.txt'
insert into table target
fields terminated by ','
(empno, ename, gender decode(:gender, 'm', 'male', 'f', 'female')")

➔ Sql> select * from target;

EMPNO	ENAME	GENDER
101	abc	male
102	xyz	female
103	pqr	male
104	kkk	female

Date: 16/7/15

Dates used in Control File:
Method 1: using Data type
Method 2: using to_date()

Method 1: Using Data Type
Syntax:
colname date"flatfile dateformat"
EG:
➔ Creating a flat file file1.txt
101,abc,120705
102,xyz,240307
103,pqr,190804

➔ Creating a table in Oracle database
Sql> create table target(empno number(10), ename varchar2(10), col3 date);

➔ Creating a Control file 'murali.ctl'
load data
infile 'C:\ file1.txt'
insert into table target
fields terminated by ','
(empno, ename, col3 date "DDMMYY")

➔ Sql> select * from target

EMPNO	ENAME	COL3
101	abc	12-JUN-05
102	xyz	24-MAR-07

      103   pqr   19-AUG-04

Method 2: Using to_date()
➔ Creating a control file 'murali.ctl'
load data
infile 'C:\ file1.txt'
insert into table target
fields terminated by ','
(empno, ename, col3 "to_date(:col3, 'DDMMYY')")

Sequences used in Control files:
Syntax:
colname "sequencename.nextval"
EG:
➔ Creating a flat file file1.txt
101
102
103
104
105
➔ Creating a table in Oracle database
Sql> create sequence s1;
Sql> create table target(sno number(10));
➔ Creating a Control file 'muarli.ctl'
load data
infile 'C:\ file1.txt'
insert into table target
fields terminated by ','
(sno "s1.nextval")
➔ Sql> select * from target;
SNO

1
2
3
4
5

Creating a Control file for fixed length Record Flat file:
A flat file which does not have delimiters is called Fixed length record flat file. When resource
has fixed length record flat file then we must use "Position" clause within control file. In this

"position" clause we must specify starting, ending position of the every fields by using ":" operator.

Syntax:

position (startingpoint:endingpoint)

Along with "position" clause we are using SQLLDR data types. Then are:

1. Integer external
2. Decimal external
3. Char

Syntax:

Colname partition(startingpoint:endingpoint) sqlldrdatatype

NOTE: Whenever we are using functions (or) expression within control file then we are not allowed to use SQLLDR data types. In place of this one we must use functions functionality.

Syntax:

Colname partition(startingpoint:endingpoint) "functioname(:colname)"

EG:

➔ Creating a flat file file1.txt

101abc2000

102xyz5000

103pqr8000

➔ Creating table in Oracle database

Sql> create table target(empno number(10), ename varchar2(10), sal number(10));

➔ Creating a control file 'muarli.ctl'

load data

infile 'C:\ file1.txt'

insert into table target

(empno position(01:03) integer external, ename position(04:06) char, sal position(07:10) integer external)

➔ Sql> select * from target;

EMPNO   ENAME   SAL

----

101   abc   2000

102   xyz   5000

103   pqr   8000

Question by MURALI SIR

➔ Sql> create sequence s1;

➔ Sql> create table target(sno number(10), value number(10), time date, col1 number(10), col2 varchar2(10), col3 date);

➔ Creating a Control file

load data

infile *

insert into table target

(sno "s1.nextval", value constant '50', time "to_char(sysdate, 'HH24:MI:SS')", col1 position(01:06) ":col1/100", col2 position(07:13) upper(:col2)", col3 position(14:19) "to_date(:col3, 'DDMMYY')")

begindata

100000aaaaaaa060805

200000bbbbbbb170903

➔ Sql> select * from target;

SNO	VALUE	TIME	COL1	COL2	COL3
1	50	18:11:12	100000	aaaaaaa	06-AUG-05
2	50	18:11:59	200000	bbbbbbb	17-SEP-03

Date: 17/7/15

NOTE: In Oracle, infront of the "sysdate", "user" functions we are not allowed to use "**:**" (colon) operator and also not allowed to use "**:old**", "**:new**" qualifiers.

Using Sql Loader we can also transfer number of flat files data into single Oracle table by using number of "**INFILE**" clauses within Control file.

Using Sql Loader we can also transfer single flat file data into number of Oracle table then we must use number of "**into table tablename**" clauses. But when resource having different databases data (or) combination of flat file, databases data then Sql Loader cannot transfer this data into Oracle table. To overcome this problem now a days organization uses data ware housing tools which transferred data into target database.

TRIGGERS

Trigger is also same as "Stored Procedure" and also it automatically invoked DML operations performed on "**Tabular View**".

Oracle having 2 types of triggers.
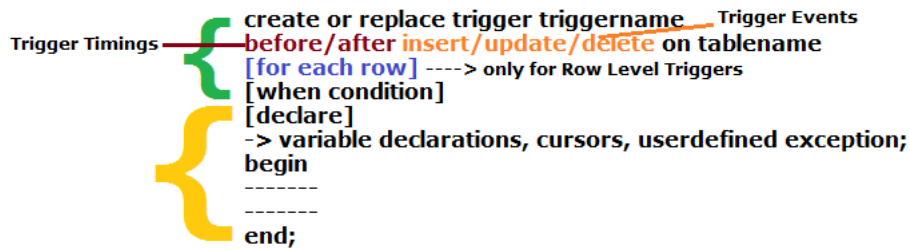
1. Statement Level Trigger.
2. Row Level Trigger.

In Statement Level Triggers, Trigger body is executed only "**ONCE**" per DML Statement.

Whereas in Row Level Triggers, Trigger body is executed per each for DML Statement.

Triggers also having two parts.

1. Trigger Specification.
2. Trigger Body.

Syntax: /*[......] means optional*/

```
                    { create or replace trigger triggername___ Trigger Events
Trigger Timings ____ { before/after insert/update/delete on tablename
                    { [for each row] ----> only for Row Level Triggers
                    { [when condition]
                    { [declare]
                    { -> variable declarations, cursors, userdefined exception;
                    { begin
                    { -------
                    { -------
                    { end;
```

Difference between Statement Level Triggers and Row Level Triggers:
Statement Level Triggers:
EG:
Sql> create or replace trigger tr1
    after update emp
    begin
    dbms_output.put_line('Tomorrow Holiday');
    end;
    /
Testing:
Sql> update emp set sal=sal+100 where deptno=10;
Tomorrow Holiday
3 rows updated.
➔ In Statement Level Trigger number of rows effected (or) not effected also it will fires and print message only once.
Sql> update emp set sal=sal+100 where deptno=80;
Tomorrow Holiday
0 rows updated
Sql> drop trigger tr1;

Row Level Trigger:
Sql> create or replace trigger tr1
    after update emp
    for each row
    begin
    dbms_output.put_line('Tomorrow Holiday');
    end;
    /
Testing:
Sql> update emp set sal= sal+100 where deptno=10;
Tomorrow Holiday
Tomorrow Holiday
Tomorrow Holiday
3 rows updated.
sql> update emp set sal=sal+100 where deptno=90;
0 rows updated
Row Level Triggers:

In Row Level Triggers trigger body is executed for each row for DML statement. That is why we are using "**for each row**" clause in trigger specification. And also DML transaction values are automatically stores in 2 **rollback segment qualifiers**. These are

1. :new
2. :old

These rollback segment qualifiers is also called as Record type variables. These qualifiers are used in either trigger specification (or) in trigger body.

Syntax:

:old.columnname

Syntax:

:new.columnname

NOTE: Whenever we are using these qualifiers in trigger specification then we are not allowed to use ":" infront of the qualifier name.

Syntax:

old.columnname

Syntax:

new.columnname

INSERTION:

EMP

EMPNO	ENAME	SAL
1	ABC	2000
2	XYZ	3000

Sql> insert into emp values (3, 'murali', 5000);

:NEW		EMPNO	ENAME	SAL
		3	MURALI	5000
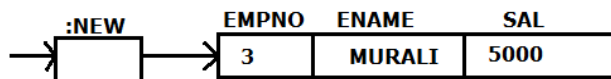
Ename ----> :new.ename

Sal ----> :new.sal

UPDATION:

Sql> update emp set sal=8000 where empno=1;

:OLD

EMPNO	ENAME	SAL	:old.sal
1	ABC	2000	

:NEW

EMPNO	ENAME	SAL	:new.sal
1	ABC	8000	
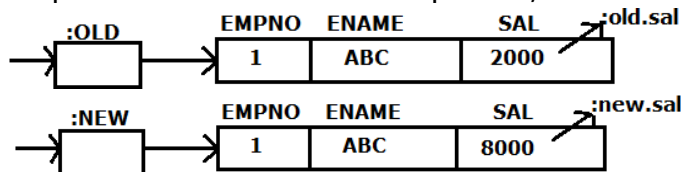
DELETION:

Sql> delete from emp where empno=1;

:OLD

EMPNO	ENAME	SAL
1	ABC	2000

:old.empno   :old.ename

	INSERT	UPDATE	DELETE
:NEW	YES	YES	NO
:OLD	NO	YES	YES
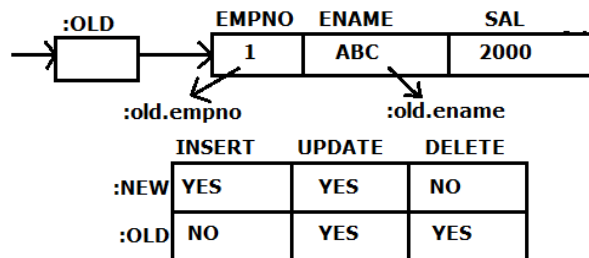
Q) Write a PL/SQL row level trigger on employee table whenever user inserting data then salary should be more than 5000?

ANS: sql> create or replace trigger tr1

```
before insert on emp
for each row
begin
if :new.sal<5000 then
raise_application_error(-20143, 'Salary should be less than 5000');
end if;
end;
/
```

Testing:

Sql> insert into emp(empno, sal) values(1,3000);

ERROR: ORA-20143: Salary should not be less than 5000.

Sql> insert into emp(empno, sal) values(1,6000);

1 row inserted.

Date: 20/7/15

Q) Write PL/SQL Row Level Trigger on the following table not allowed to insert duplicate data into that table?

ANS: sql> create table test(sno number(10));

Sql> insert into test values(&sno);

10 10 10 20 20 30 30

Sql> select * from test;

SNO

10

10

10

20

20

30

30

Sql> create or replace trigger tr1

```
before insert on test
for each row
cursor c1 is select * from test;
begin
for in c1
loop
if i.sno=:new.sno then
raise_application_error(-20143, 'We cannot insert duplicate values');
end if;
```

```
    end loop;
    end;
    /
```

Execution:
Sql> insert into test values(10);
ERROR: ORA-20143: We cannot insert duplicate values
Sql> insert into test values(1);
    1 row inserted.

Q) Write a PL/SQL Row Level Trigger on emp table not allowed to insert duplicate data into empno column without using cursors?
ANS: sql> create or replace trigger tr1
```
    before insert on emp
    for each row
    v_count number(10);
    begin
    select count(*) into v_count from test where sno=:new.sno;
    if v_count>=1 then
    raise_application_error(-20143, 'Cannot insert duplicate values');
    elsif v_count=0 then
    dbms_output.put_line('Value is Inserted');
    end if;
    end;
    /
```
Execution:
Sql> insert into test values(10);
ERROR: ORA-20143: Cannot insert duplicate values.
Sql> insert into test values(1);
Value is Inserted
1 row inserted

Q) Write a PL/SQL Row Level Trigger on emp table whenever user inserting data if job is 'ANALYST' then automatically set comm of the analyst to 'NULL' in emp table?
ANS: sql> create or replace trigger tr1
```
    before insert on emp
    for each row
    when(new.JOB='ANALYST')
    begin
    if :new.comm is not null then
    :new.comm:=null;
    end if;
    end;
    /
```

Testing:
Sql> insert into emp(empno, ename, job, comm)values(1,'murali','ANALYST',3000);
1 row inserted.
Sql> select * from emp;

EMPNO	ENAME	JOB	COMM
1	MURALI	ANALYST	--

Date: 21/7/15

Q) Write a PL/SQL Row Level Trigger on emp table whenever user modifying salaried then automatically display old salary, new salary, salary difference for the transaction?

ANS: sql> create or replace trigger tr1
    after update on emp
    for each row
    declare
    x number(10);
    begin
    x:= :new.sal-:old.sal;
    dbms_output.put_line('Old Salary is:'||' '||:old.sal);
    dbms_output.put_line('New Salary is:'||' '||:new.sal);
    dbms_output.put_line('Difference Salary is:'||' '||x);
    end;
    /

Testing:
Sql> update emp set sal=sal+100 where ename='SMITH';
Output:
Old Salary is: 800
New Salary is: 900
Difference Salary is: 100

Q) Write a PL/SQL Row Level Trigger on dept table whenever user modifying deptno's in dept table then automatically those modifications are effected in emp table?

ANS: sql> create or replace trigger tr2
    after update on dept
    begin
    update dept set deptno=:new.deptno where depnto=:new.deptno;
    end;
    /

Testing:
Sql> update dept set deptno=1 where deptno=10;
Sql> select * from dept;
Sql> select * from emp;

Q) Write a PL/SQL Row Level Trigger on emp table whenever user deleting data those deleted records are automatically stored in another tables?

ANS: sql> create table test(empno number(10), ename varchar(10), sal number(10));

sql> create or replace trigger tr1

    after delete on emp

    for each row

    begin

    insert into test values(:old.empno, :old.ename, :old.sal);

    end;

    /

Testing:

Sql> delete from emp where deptno=10;

Sql> select * from test;

NOTE: In Oracle, we can also change peritcular column values by using "OF" clauses. These clauses are used in trigger specification only. Insert, Delete events does not contain "OF" clauses.

Syntax:

update of columnname;

AUDITING OF COLUMNS:

In all databases, when we are modifying data in a particular tbale column then those transaction values are automatically stored in another table is called Auditing a Column.

Q) Write a PL/SQL Row Level Trigger on emp table whenever user modifying values in salary column those transaction details are automatically stored in another table?

ANS: sql> create table test(oldempno number(10), oldename varchar2(10), oldsalary number(10), newsalary number(10), col5 date, username varchar2(10));

Sql> create or replace trigger tr1

    after update **of** sal on emp

    begin

    insert into test values(:old.empno, :old.ename, :old.sal, :new.sal, sysdate, user);

    end;

    /

Testing:

Sql> update emp set sal=sal+100 where deptno=10;

3 rows updated

Sql> select * from test;

NOTE: In Oracle, we are not allowed to use old, new qualifiers in front of the sysdate, user functions.

AUTO INCREMENT:

126

In all databases, if we want to generate primary key values automatically then we are using auto increment concept. In Oracle, we are implementing auto increment concept by using "**Row level Triggers, sequences**". That is, we are creating a Sequence in "SQL" and use this sequence in "PL/SQL" row level trigger.

EG: sql> create table test(sno number(10) primary key, name varchar2(10));
Sql> create sequence s1
    start with 1
    increment by 1;
sql> create or replace trigger th1
    before insert on test
    for each row
    begin
    select s1.nextval into :new.sno from dual;
    end;
    /
Testing:
Sql> insert into test(name) values('&name');
murali
dinesh
naresh
siva
sql> select * from test;
SNO   NAME
----
1   murali
2   dinesh
3   naresh
4   siva

Here SNO values are generated automatically by using "**AUTO INCREMENT**".

NOTE: Oracle 11g, introduced variable assignment concept when we are using "Sequences" in PL/SQL block i.e., we are not allowed to use "DUAL" table, "Select….into" clause.
Syntax:
Sql> begin
    Variablename:= sequencename.nextval;
    end;

For previous example [only for 11g, 12c]
Sql> create or replace trigger tr1
    before insert on test
    for each row

```
begin
:new.sno:= s1.nextval;
end;
/
```

Generating Alpha Numeric data as Primary key in "AUTO INCREMENT" concept:
EG: sql> create table test(sno varchar2(10) primary key, name varchar2(10));
Sql> create sequence s1;
Sql> create or replace trigger tr1
```
    before insert on test
    begin
    select 'ABC'||lpad(s1.nextval, 10, '0') into :new.sno from dual;
    end;
    /
```
Testing:
Sql> insert into test(name) values('&name');
```
    Murali
    Abc
    Xyz
    Kkk
```
Sql> select * from test;
```
    SNO            NAME
---------------------------------
ABC0000000001 Murali
ABC0000000002 Abc
ABC0000000003 Xyz
ABC0000000004 Kkk
```

Date: 22/7/15

Oracle trigger having two timing points. These are **BEFORE, AFTER**
Whenever we are using "**Before**" timing trigger code is executed first then only DML transaction values are executed. Whereas in "**After**" timing DML statements are executed first then only trigger code is executed. Whenever we are specifying **BEFORE** timing DML statements are first effected in trigger then only those values are stored in database. That's why in Oracle if we want to assign values into ":new" qualifiers then we must use before timing. Otherwise Oracle server returns an error cannot change ":new" values for this trigger type.

Q) Write a PL/SQL row level trigger on emp table whenever user inserting data into ename column then automatically inserted value converted into upper case?
ANS: sql> create or replace trigger tr1
```
    before insert on emp
    for each row
```

```
begin
:new.ename:=upper(:new.ename);
end;
/
```

Testing:

Sql> insert into emp(empno, ename) values(1, 'murali');

```
   EMPNO              ENAME
-------------------------------------------
    1                  MURALI
```

STATEMENT LEVEL TRIGGER:

In statement level triggers, trigger body is executed only once per DML statements. Statement level triggers does not have old, new qualifiers. Statement level triggers does not contain "for each row" clause. Generally, if we want to define time component based on conditions then we are using statement level triggers.

EG:

Q) Write a PL/SQL statement level trigger on emp table not to perform DML operations on Saturday and Sunday ?

ANS: sql> create or replace trigger tr1

```
    before insert or update or delete on emp
    begin
    if to_char(sysdate, 'DY') in ('SAT', 'SUN') then
    raise_application_error(-20143, 'We cannot perform DML operations on Saturday and
Sunday');
    end if;
    end;
    /
```

Testing:

/*First change system date to Saturday or Sunday and then perform testing*/

Sql> delete from emp where deptno=10;

Error: ORA-20143: We cannot perform DML operations on Saturday and Sunday

NOTE: We are not allowed to use "when" clause in "Statement Level Trigger".
NOTE: We can also convert statement level trigger into row level triggers and row level trigger into statement level triggers without having qualifiers(old,new).

Converting above Statement Level Trigger to Row Level Trigger.

Sql> create or replace trigger tr1

```
    before insert or update or delete on emp
    for each row
    when (to_char(sysdate, 'DY') in ('SAT', 'SUN'))
    begin
    raise_application_error(-20143, 'We cannot perform DML operations on Saturday and
Sunday');
```

```
end;
/
```

NOTE: In all databases, statement level triggers performance is very high compared to row level triggers. Because in statement level triggers trigger body is executed only once per DML statements.

Date: 23/7/15

Q) Write a PL/SQL Statement Level Trigger on emp table not to perform DML Operations in last date of the month?
ANS: sql> create or replace trigger tr1
```
        before insert or delete or update on emp
        begin
        if sysdate:= last_day(sysdate) then
        raise_application_error(-20143, 'Cannot perform DML's on last day of the month');
        end if;
        end;
        /
```
Testing:
Sql> delete from emp where deptno=10;
ORA-20143: Cannot perform DML's on last day of the month

Trigger Events:
In Oracle, triggers if we want to define multiple conditions on different tables within trigger body then we are using trigger body events. These are Inserting, Updating, Deleting clauses. These clauses are also called as Trigger Predicate clauses. These clauses are used in trigger body only. These clauses are used in either Statement Level triggers (or) Row Level triggers.
Syntax:
```
if inserting then
    stmts;
elsif updating then
    stmts;
elsif deleting then
    stmts;
end if;
```

Q) Write a PL/SQL statement level trigger on emp table not to perform DML operation in any days by using triggering events?
ANS: sql> create or replace trigger tr2
```
        before insert or delete or update on emp
        begin
        if inserting then
        raise_application_error(-20143, 'We cannot perform Inserting');
```

130

elsif updating then
raise_application_error(-20143, 'We cannot perform Updating');
elsif deleting then
raise_application_error(-20111, 'We cannot perform Deleting');
end if;
end;
/

Testing:
Sql> delete from emp where deptno=10;
ORA-20111: We cannot perform Deleting
EG: sql> create or replace trigger tr1
before insert or update or delete on emp
declare
z varchar2(20);
begin
if inserting then
z:= 'rows Inserted';
elsif updating then
z:= 'rows Updated';
elsif deleting then
z:= 'rows Deleted';
end if;
insert into test values(z);
end;
/
Sql> create table test(msg varchar2(20));
Testing:
Sql> insert into emp(empno) values(1);
Sql> insert into emp(empno) values(2);
Sql> update emp set sal= sal+100 where deptno=10;
Sql> delete from emp where empno in(1,2);
Sql> select * from test;
MSG

Rows inserted
Rows inserted
Rows updated
Rows deleted

Q) Write a PL/SQL Row Level Trigger on emp table whenever user inserting data those records are inserted in another table and also whenever user modify data on emp table and then those transactional details are stored in another table and also whenever user deleting data then those deleted records are stored in another table?
ANS: sql> create table t1(empno number(10), name varchar2(10), salary number(10));

131

Sql> create table t2(empno number(10), name varchar2(10), salary number(10));
Sql> create table t3(empno number(10), name varchar2(10), salary number(10));
Sql> create or replace trigger tk2

```
    after insert or update or delete on emp
    for each row
    begin
    if inserting then
    insert into t1 values(:new.empno, :new.ename, :new.sal);
    elsif updating then
    insert into t2 values(:old.empno, :old.ename, :new.sal);
    elsif deleting then
    insert into t3 values(:old.empno, :old.ename, :old.sal);
    end if;
    end;
    /
```

Testing:
Sql> delete from emp where deptno=10;
Sql> select * from t3;

```
    EMPNO        ENAME       SALARY
------------------------------------------------
```

TRIGGER EXECUTION ORDER:
1. Before Statement Level
2. Before Row Level
3. After Row Level
4. After Statement Level

EG: sql> create table test(sno number(10));
Sql> create or replace trigger tz1

```
    after insert on test
    for each row
    begin
    dbms_output.put_line('After Row Level');
    end;
    /
```
Sql> create or replace trigger tz2

```
    after insert on test
    begin
    dbms_output.put_line('After Statement Level');
    end;
    /
```
Sql> create or replace trigger tz3

```
    before on test
    for each row
```

```
       begin
       dbms_output.put_line('before Row Level');
       end;
       /
Sql> create or replace trigger tz1
       before insert on test
       begin
       dbms_output.put_line('Before Statement Level');
       end;
       /
```

Testing:

```
Sql> insert into test values(10);
       before statement level
       before row level
       after row level
       after statement level
```

Date: 24/7/15

COMPUND TRIGGER:

Oracle 11g, introduced "Compound Trigger". Compound Triggers allows different blocks within a trigger to be executed at different timing points. Compound Trigger also having global declare section same like a packages.

Syntax:

```
create or replace trigger triggername
for insert or update or delete on tablename
compound trigger
➔ Variable declarations, constant declarations;
➔ Types declarations;
➔ Local Sub program implementations;
before statement is
begin
------
------
end[before statement];
before each row is
begin
------
------
end[before each row];
after each row is
begin
------
------
```

end[after each row];
after statement is
begin

------
end[after statement];
end;

Q) Write a PL/SQL trigger which display default trigger execution order?
ANS: sql> create table test(sno number(10));
Sql> create or replace trigger tr1
    for insert on test
    compound trigger
    before statement is
    begin
    dbms_output.put_line('Before Statement Level');
    end[before statement];
    before each row is
    begin
    dbms_output.put_line('Before Row Level');
    end[before row level];
    after each row is
    begin
    dbms_output.put_line('After Row Level');
    end[after each row];
    after statement is
    begin
    dbms_output.put_line('After Statement Level');
    end[after statement level];
    end;

Testing:
Sql> insert into test values(10);
before statement level
before row level
after row level
after statement level

"FOLLOWS" clause: (Guarantee Execution Order)
Oracle 11g, introduced "FOLLOWS" clause in triggers. Generally, whenever we are using same level of triggers on same table then we cannot control execution order of the triggers. To overcome this problem Oracle 1g introduced "FOLLOWS" clause which is used to control

execution order of the triggers explicitly when we are using same level of triggers on same table. This clause is used in Trigger Specification only.

Syntax:
create or replace trigger triggername
before/after insert/update/delete on tablename
[for each row]
[when (condition)]
follows another triggername1, another triggername2,…..
[declare]

----
begin

[execption]

----
end;
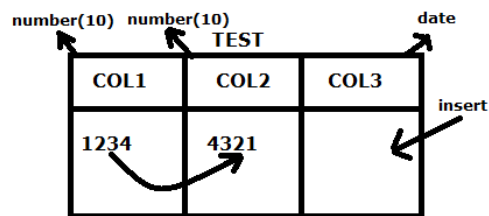
EG: sql> create table test(col1 number(10), col2 number(10), col3 date);
Sql> create sequence s1 start with 1234;
Sql> create or replace trigger tr1
    before insert on test
    for each row
    begin
    select s1.nextval into :new.col1 from dual;
/* :new.col1= s1.nextval; (for 11g) */
    dbms_output.put_line('Trigger1 fired');
    end;
    /
Sql> create or replace trigger tr2
    before insert on test
    for each row
    begin
    select reverse(to_char(:new.col1)) into :new.col2 from dual;
    dbms_output.put_line('Trigger2 fired');
    end;
    /

Testing:

Sql> insert into test(col3) values(sysdate);
    Trigger2 fired
    Trigger1 fired
Sql> select * from test;

Output:

COL1	COL2	COL3
1234	------	24-JUL-15

Solution: "FOLLOWS" clause using in Oracle 11g

Sql> create or replace trigger tr2
    before insert on test
    for each row
    follows tr1
    begin
    select reverse(to_char(:new.col1)) into :new.col2 from dual;
    dbms_output.put_line('Trigger2 fired');
    end;
    /

Testing:

Sql> insert into test(col3) values(sysdate);
Trigger1 fired
Trigger2 fired
Sql> select * from test;

Output:

COL1	COL2	COL3
1234	4321	24-JUL-15

NOTE: In Oracle, if we want to provide guarantee execution order when we are using same level of triggers on same table then we must use "FOLLOWS" clause.

<u>CALLING PROCEDURE IN TRIGGER:</u>

In Oracle, we can also "CALL" procedure into trigger by using "CALL" statement.

Syntax:

create or replace trigger triggername
before/replace insert/update/delete on tablename
call procedurename

Date: 25/7/15

EG: sql> create table test(totsal number(20));

136

Sql> create or replace procedure p1
    is
    v_sal number(20);
    begin
    delete from test;
    select sum(sal) into v_sal from emp;
    insert into test values(v_sal);
    end;
    /
Sql> create or replace trigger tg1
    after insert or update or delete on emp
    call p1
    /
Testing:
Sql> update emp set sal= sal+100;
Sql> select * from test;
    TOTSAL

    32125

Q) Write a PL/SQL trigger on emp table whenever user deleting data on emp table then automatically display remaining number of records number in bottom of the delete statement?
ANS: sql> create or replace trigger th1
    after delete on emp
    default
    v_count number(10);
    begin
    select count(*) into v_count from emp;
    dbms_output.put_line(v_count);
    end;
    /
Testing:
Sql> delete from emp where empno=1;
16
1 row deleted
In case Row Level Trigger
Sql> create or replace trigger tr1
    after delete on emp
    for each row
    declare
    v_count number(10);
    begin
    select count(*) into v_count from emp;
    dbms_output.put_line(v_count);

137

end;
    /
Trigger is created
➔ During Runtime
Sql> delete from emp where empno=1;
Error: ORA-04091: Table scott.emp is mutating.

MUTATING ERROR:

When a Row Level Trigger based on a table then trigger body cannot read data from same table and also we cannot perform DML operations on same table. If you are trying this then Oracle server return an error "ORA-04091: table is mutating". This error is called "Mutating Error". And also this table is called Mutating Table and also this trigger is called Mutating Trigger.

Mutating is an "Run time" error occurred in Row Level Triggers. Mutating Error does not occurred in "Statement Level Trigger".

In Oracle, whenever we are using DML transactions those transaction values are automatically committed into database when we are using Statement Level Triggers. When we are trying to read this committed data by using trigger body then database servers does not return any errors. That's why statement level trigger does not return any Runtime error.

Whenever we are using Row Level Trigger then DML transaction values are not committed automatically on database. Using trigger body when we try to read not committed data by using same table then database server returns mutating errors. That's why Row Level Triggers only returns mutating errors.

In Oracle, for avoiding mutating errors then we are using autonomous transactions in triggers. In this case we must use "automomous_transaction" pragma in declare section of the trigger and also we must use commit in trigger body. Autonomous transactions automatically avoids rotating errors also these transactions returns previous results.

EG: create or replace trigger tr1
    after delte on emp
    for each row
    declare
    v_count number(10);
    pragma autonomous_transactions;
    begin
    select count(*) into v_count from emp;
    commit;
    dbms_output.put_line(v_count);
    end;
    /
Trgger created.
Sql> delete from emp where empno= 7788;
14
1 row deleted

In Oracle, autonomous transactions returns previous results. To overcome this problem for avoiding Mutating error purpose Oracle provided solution using following steps through packaged global variable because by defaults packages global variable statement is committed.

In Oracle 11g, onwards we can also avoid mutating errors by using triggers.
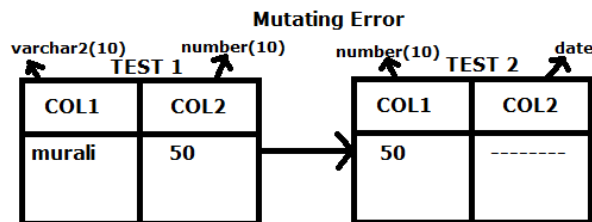
Steps to avoid Mutating Errors in Oracle:

Step 1: Create packaged global variable
Step 2: Create after row level trigger
Step 3: Create after statement level trigger

Date: 26/7/15

Mutating Error

Sql> create table test1(col1 varchar2(20), col2 number(10));
Sql> create table test2(col1 number(10), col2 date);
Sql> create or replace trigger tr1
    after insert on test1
    for each row
    declare
    id number(10);
    begin
    select col2 into id from test1 where col2= :new.col2;
    insert into test2 values(id, sysdate);
    end;
    /
Trigger created
Testing:
Sql> insert into test2 values('murali', 50);
Error: ORA-04091: Table is mutating

Solution:
Step 1: creating Packaged global variable
Sql> create or replace package pn1
    is
    id number(10);
    end;
    /
Step 2: creating after row level trigger
Sql> create or replace trigger tr1

after insert on test1

for each row

begin

pn1.id:= :new.col2;

end;

/

Step 3: create after statement level trigger

Sql> create or replace trigger tr2

after insert on test1

begin

insert into test2.values(pn1.id, sysdate);

end;

/

Testing:

Sql> insert into test1 values('murali', 50);

1 row inserted

Sql> select* from test1;

Sql> select * from test2;

NOTE: In Oracle 11g, onwards we can also avoid mutating error by using compound trigger. Because compound trigger having global declarations section same like packages.

Q) Write a PL/SQL Compound trigger to avoid mutating errors.

ANS: sql> create or replace trigger tr1

for insert on test1

compound trigger

id number(10);

after each row is

begin

id:= :new.col2;

end after each row;

after statement is

begin

insert into test2 values(id, sysdate);

end after statement;

end;

/

"WHEN" clause used in Trigger:

In Oracle, we can also define logical conditions in trigger specification by using "WHEN" condition but "WHEN" clause is used in Row Level Triggers only. Whenever we are using "WHEN" clause then we are not allowed to use ":" infront of the qualifier names in "WHEN" clause logical condition.

Syntax:

when (condition)

Always "WHEN" clause condition must be an SQL expression and also "WHEN" clause condition returns Boolean values either "true" or "false". Whenever condition is "true" then only trigger body is executed.

EG: sql> create or replace trigger tr1
    after insert on emp
    when (new.empno=1)
    begin
    dbms_output.put_line('When empno=1 then only my trigger body is executed');
    end;
    /

Testing:
Sql> insert into emp(empno) values(1);
When empno=1 then only my trigger body is executed
1 row inserted
Sql> insert into emp(empno) values(2);
1 row inserted

SYSTEM TRIGGER (or) DDL Triggers:
In all databases we can also write a trigger based on DDL events. These type of triggers are also called as System Trigger (or) DDL Triggers. These Triggers are created by database Administrators.
In Oracle, we are defining system triggers in two levels. These are database level, schema level.

Syntax:
create or replace trigger triggername
before/after create/alter/drop/truncate/rename on database (or) username.schema
declare

------
begin

------
end;

EG: sql> conn sys as sysdba
    Password:sys
Sql> create or replace trigger tr1
    after create on database
    begin
    dbms_output.put_line('Somebody has created some database object');
    end;
    /

Testing:
Sql> create table Sunday(sno number(10));

Somebody has created some database object

Oracle having predefined trigger event attribute functions which is used to control DDL events on schema level. These functions are used by database administrator in system level triggers.

Q) Write a PL/SQL system level trigger on scott schema not to drop emp table?
ANS: sql> conn scott/tiger;
Sql> create or replace trigger tr1
    before drop on scott.schema
    begin
    if ora_dict_obj_name= 'EMP' and ora_dict_obj_name= 'TABLE' then
    raise_application_error(-20143, 'we cannot drop emp table');
    end if;
    end;
    /
Testing:
Sql> drop table emp;
Error: ORA-20143: we cannot drop emp table.

ENABLE (or) DISABLE triggers:
In Oracle, we can also enable (or) disable single trigger (or) enable (or) disable all triggers in a table by using "ALTER" command.

ENABLE/DISABLE a single trigger:
Syntax:
alter trigger triggername enable/disable;

ENABLE/DISABLE all triggers in a table:
Syntax:
alter table tablename enable/disable all triggers;

EG: sql> alter table emp disable all triggers;

➔ All triggers information stored under "user_triggers" data dictionary.
Sql> desc user_triggers;

➔ We can also drop a trigger by using "drop trigger triggername";

WHERE CURRENT OF, FOR UPDATE clauses used in cursor:
Q) Write a PL/SQL program to modify salaries of the employees in emp table based on following conditions:
1. if job= 'CLERK' then increment sal-----> 100
2. if job= 'MANAGER' then increment sal -----> 200
ANS: sql> declare

142

```
cursor c1 is select * from emp;
i emp%rowtype;
begin
open c1;
loop
fetch c1 into i
exit when c1%notfound;
if i.job= 'CLERK' then
update emp set sal= sal+100 where empno= i.empno;
else if i.job= 'MANAGER' then
update emp set sal= sal-200 where empno= i.empno;
end if;
end loop;
close c1;
end;
/
```

Date: 27/7/15

In all databases, whenever we are performing DML operations then automatically database servers uses default locks. If you want to perform locks before update (or) before then we must use locking mechanism explicitly by using cursors. If you want to establishes locks then we are using "FOR UPDATE" clause in cursor "SELECT" statement.

Syntax:
cursor cursorname is select * from tablename where condition for condition;
Whenever we are opening the cursor then only oracle server internally uses exclusive locks based on the "FOR UPDATE" clause.

WHERE CURRENT OF:
WHERE CURRENT OF clause is used in update, delete statements within cursor. WHERE CURRENT OF clause uniquely identifying a record because WHERE CURRENT OF clause internally using "ROW ID".
NOTE: In all database whenever we are using WHERE CURRENT OF clause then we must use FOR UPDATE clause within cursor SELECT statement.
Syntax:
update tablename set columnname= new value WHERE CURRENT OF cursorname ;
Syntax:
Delete from tablename WHERE CURRENT OF cursorname;

NOTE: WHERE CURRENT OF clause is used to update (or) delete lastly fetched row from the cursor. Whenever resource table does not have primary key then we are updating and deleting records from those tables by using cursors then we must use "WHERE CURRENT OF" clause.

Whenever after processing data we are releasing locks by using "COMMIT" in PL/SQL blocks.

Q) Write PL/SQL cursor program to modify salaries of the employees in the following table?
-> increment sal in each record by 1000
ANS: sql> create table test(ename varchar2(20), sal number(10));
Sql> insert into test values('&ename', &sal);
Sql> select * from test;

```
     ENAME      SAL
  -------------------------------
     a          1000
     b          2000
     a          3000
     b          4000
     c          5000
```

Without using WHERE CURRENT OF clause.
sql> declare
    cursor c1 is select * from test;
    begin
    for i in c1
    loop
    update test set sal= sal+1000 where ename= i.ename;
    end loop;
    end;
    /
Output:
Sql> select * from test;

```
     ENAME      SAL
  -------------------------------
     a          3000
     b          4000
     a          5000
     b          6000
     c          6000
```

Sql> rollback;

With Using "WHERE CURRENT OF" clause.
Sql> declare
    cursor c1 is select * from test for update;
    begin
    for i in c1
    loop
    update test set sal= sal+100 where current of c1;
    end loop;

```
    commit;
    end;
    /
```

Testing:

Sql> select * from test;

ENAME	SAL
a	2000
b	3000
a	4000
b	5000
c	6000

This is correct result.

Q) Write a PL/SQL cursor program to increment salary of the 5th record from emp table?

ANS:
```
sql> declare
    cursor c1 is select * from emp for update;
    begin
    for i in c1
    loop
    if c1%rowcount=5 then
    update emp set sal= sal*0.5 where current of c1;
    end if;
    end loop;
    commit;
    end;
    /
```

Output:

Sql> select * from emp;

EG: sql> create table test(a number(10), b number(10), c number(10));

Sql> insert into test(a, b) values(&a, &b);

A	B	C
5	4	
5	4	
5	4	
5	4	

Q) Write a PL/SQL cursor program to update the column "C" from the following table based on following condition?

For 1st record update c=a+b

145

For 2nd record update c=a-b
For 3rd record update c=a*b
For 4th record update c=a/b

ANS: sql> declare

```
cursor c1 is select * from test for update;
begin
for i in c1
loop
if c1%rowcount=1 then
update test set c=a+b where current of c1;
elsif c1%rowcount=2 then
update test set c=a-b where current of c1;
elsif c1%rowcount=3 then
update test set c=a*b where current of c1;
elsif c1%rowcount=4 then
update test set c=a/b where current of c1;
end if;
end loop;
commit;
end;
/
```

Sql> select * from test;

A	B	C
5	4	9
5	4	1
5	4	20
5	4	1

Q) Write a PL/SQL program by using cursor locking mechanism all employees having KING as their manager raise 5% of the salary increase in emp?

ANS: sql> declare

```
cursor c1(p_empno) is select sal from emp where mgr= p_mgr for update;
v_mgr number(10);
begin
select empno into v_mgr from emp where ename= 'KING';
for i in1(v_mgr)
loop
update emp set sal=sal*0.5 where current  of c1;
end loop;
commit;
end;
/
```

Date: 28/7/15

LOBS(Large Objects):

Oracle 8.0 introduced LOBS. These are predefined data types which is used to store large amount of data, images into database.

In Oracle, if we want to store more than 2000 bytes of alpha numeric data then we are using "varchar2" data type. If we want to store more than 4000 bytes of alpha numeric data then we are using "LONG" data type. Because varchar2 data type stores upto 4000 bytes of alpha numeric data.

Syntax:

Columnname long

LONG data type stores upto 2GB data. But there can be only one LONG column for a table. And also we cannot create primary key on LONG data type column. To overcome this problems Oracle 8.0 introduced "CLOB" datatype.

EG:

Sql> create table test(col1 long, col2 long);

ERROR: a table may contain only one column of type LONG.

Sql> create table test(col long primary key);

ERROR: key column cannot be of LONG data type.

If we want to store binary data then we are using RAW data type. It stores upto 2000 bytes of binary data.

Syntax:

Columnname raw(size)

If we want to store more than 2000 bytes of binary data then we are using LONG RAW data type. It stores upto 2GB data. But there can be only 1 long raw column for a table. To overcome this problem Oracle 8.0 introduced "BLOB" data type.

Syntax:

Columnname long raw

EG: sql> create table test(col1 raw(200));

Sql> insert into test values('1010101');

Sql> select * from test;

Sql> drop table test;

Sql> create table test(col1 long raw);

Sql> desc test;

All database systems having 2 types of large objects.

1. Internal Large Objects.
2. External Large Objects.

1. Internal Large Objects:

Internal Large Objects are stored within database. Oracle having 2 types of Internal Large Objects.

1. Character Large Object(CLOB)
2. Binary Large Object(BLOB)

Syntax:

Columnname clob
Syntax:
Columnname blob

2. External Large Objects:

External Large Objects are stored in outside of the database. i.e., these objects are stored in Operating System files. This is an "BFILE" data type.
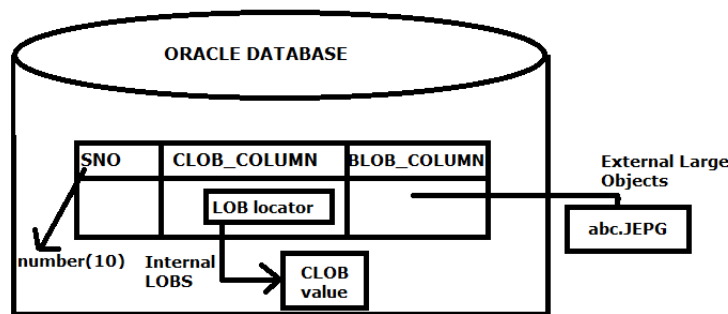**Syntax:**
Columnname bfile

Difference between LONG and LOBS:

LONG	LOBS
1. Can store upto 2GB data	1. Can store upto 4GB data
2. A table contain only one LONG column	2. A table contain more than one LOB columns.
3. Subquery's cannot select a LONG datatype.	3. Subquery's can select LOB column.

LOB Locator:

The data in LOB column not stored directly in database. Same like a other data type column in the row. Instead of this one it stores a pointer that pointer points to where the actual data stored in elsewhere in the database. This pointer is also called as LOB locator. In case of external LOBS pointer points to the data within operating system file.

EMPTY_CLOBS AND EMPTY_BLOBS functions:

These functions are used in SQL DML. These functions are used to initialize LOB locator into empty locator in Insert, Update statement.
NOTE: Before we are writing data into LOBS by using "**dbms_lob**" package (or) by using OCI(Oracle Call Interface) then we initialize LOB locator into empty locator by using "empty_clob" and "empty_clob" functions.

Difference between EMPTY, NULL:

EG: sql> create table test(col1 number(10), col2 clob);
Sql> insert into test values(1, 'abc');

Sql> insert into test values(2, empty_clob);
Sql> insert into test values(3, null);
Sql> select * from test;

```
    COL1        COL2
  --------------------------
    1            abc
    2
    3
```

Sql> select * from test where col2 is not null;

```
    COL1        COL2
  ---------------------------
    1            abc
    2
    3
```

Sql> select * from test where col2 is null;

```
    COL1        COL2
  ---------------------------
    3
```

Date: 29/7/15

Storing Large amount of data in CLOB column (or) Storing an image into BLOB column by using "dbms_lob" package:
Before we are storing data into LOBS we must create alias directories related to physical directory by using following syntax when we are using "dbms_lob" package.
Syntax:
Create or replace directory directoryname as 'path';
Directory name ----> Capital letters.
EG: sql> conn sys as sysdba;
    Enter password: sys
Sql> grant create any directory to scott;
Sql> conn scott/tiger;
Sql> create or replace directory XYZ as 'C:\';

Step 1: Create a table in database by using LOB columns
EG: sql> create table test(sno number(10), col2 clob);

Step 2: Develop a PL/SQL block which stores data into LOB variables by using CLOB, BLOB, BFILE data types.
Step A: In declare section of the PL/SQL block we must define LOB variables by using CLOB, BLOB, BFILE data types.
Step B: After declaring the variables in executable section of the PL/SQL block we must initialize "**LOB locator**" into empty by using "**empty_clob()**", "**empty_blob()**" functions.

And also by using "**returning into**" clause we must store LOB locator value into appropriate LOB locator variable by using following syntax.

Syntax:

Insert into tablename(clobcolumnname) values (empty_clob()) returning clobcolname into clobvariablename;

Before we are using "dbms_lob" package we must store "BFILE" pointer in "BFILE" variable by using "BFILE" name function. This function accepts two parameters. These are alias directory name, filename.

Syntax:

Bfilevariablename:= bfilename('aliasdirectoryname', 'filename');

Step C: Before we are loading data into LOB column just we must open the BFILE pointer by using "**fileopen**" procedure from "dbms_lob" package.

Syntax:

dbms_lob.fileopen(bfilevariablename);

Step D: If we want to load data into LOB column by using "dbms_lob" package then we must use "**loadfromfile**" procedure from "dbms_lob" package. This procedure accepts 3 parameters.

Syntax:

Dbms_lob.loadfromfile(lobvariablename, bfilevariablename, lengthofbfile);

NOTE: If we want to return "length of the BFILE" then we are using "**GETLENGTH()**" from "dbms_lob" package.

Syntax:

dbms_lob.getlength(bfilevarname);

Step E: After loading data into LOB column then we must close the file by using "fileclose" procedure from "dbms_lob" package.

Syntax:

dbms_lob.fileclose(bfilevarname);

EG: sql> declare
    v_clob clob;
    v_bfile bfile;
    begin
    insert into test values(1, empty_clob()) returning col2 into v_clob;
    v_bfile:= bfilename('XYZ', 'file1.txt');
    dbms_lob.fileopen(v_bfile);
    dbms_lob.loadfromfile(v_clob, v_bfiel, dbms_lob.getlength(v_bfile));
    dbms_lob.fileclose(v_bfile);
    end;
    /

Testing:

Sql> select * from test;
    COL1   COL2
  ----

1   welcome

NOTE: In Oracle, we can also store large amount of data or images into Oracle database by using "BFILE" datatype. But this BFILE data cannot be displayed in database.
EG: sql> create table test(col1 bfile);
Sql> insert into test values(bfilename('XYZ', 'file1.txt'));
Sql> select * from test;
Error: column or attribute type cannot be displayed by SQL*Plus.

DYNAMIC SQL:

Oracle 7.1, introduced Dynamic SQL. In Dynamic SQL, SQL statements are executed at runtime.
Generally, in PL/SQL we are not allowed to use DDL, DCL statements. If we want to use those statements in PL/SQL then we are using Dynamic SQL constructor. Generally, in Dynamic SQL, SQL statements must be specified within single quotes. And also those statements are executed by using "**execute immediate**" clause. This clause must be used in Executable Section of the PL/SQL block.
Syntax:
begin
execute immediate 'sql statement';
end;
/
EG: sql> begin
execute immediate 'create table w1(sno number(10));
end;
/

Q) Write a PL/SQL program to create a "ROLE"?
ANS: sql> conn sys as sysdba;
    Enter Password: sys
Sql> begin
    execute immediate 'create role r1';
    end;
    /

Date: 30/7/15

Q) Write a PL/SQL Stored Procedure for passing table as a parameter then drop table by using Dynamic SQL?
ANS: sql> create or replace procedure p1(p_table varchar2)
    is
    begin
    execute immediate 'drop table'||p_table;
    end;

151

/

Execution:
Sql> exec p1('t1');

Passing Value into Dynamic SQL statement:

If we want to pass values into Dynamic SQL statement then we are using "**place holders**" in place of actual values in Oracle place holders are represented by using ":" and also in these place holders we are passing values through "**using**" clause.

Q) Write a Dynamic SQL program to using a record into dept table?
ANS: sql> declare
    v_deptno number(10):= &deptno;
    v_dname varchar2(10):= '&dname';
    v_loc varchar2(10):= '&loc';
    begin
    execute immediate 'insert into dept values(:1, :'2', :'3')' using v_deptno, v_dname, v_loc;
    end;
    /
Enter value for deptno: 1
Enter value for dname: a
Enter value for loc   : b
Sql> select * from dept;

Retrieving values from Dynamic SQL statements:

If we want to retrieve data from database by using Dynamic SQL statement then we are using "into" clause.

Q) Write a Dynamic SQL program which retrieves maximum salary from emp table and also display that salary?
ANS: sql> declare
    v_sal number(10);
    begin
    execute immediate 'select max(sal) from emp' into v_sal;
    dbms_output.put_line(v_sal);
    end;
    /

Output: 5000

Q) Write a Dynamic SQL program which retrieves all employee names from emp table and store it into index by table and also display content from index by table through bulk collect clause?
ANS: sql> declare

```
type t1 is table of varchar2(10)
index by binary_integer;
v_t t1;
begin
execute immediate 'select ename from emp' bulk collect into v_t;
for i in v_t.first..v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
end;
/
```

NOTE: Whenever we are using "into", "using" clauses in a Single Dynamic SQL statement then **into** clause always precedes than the **using** clause.

Q) Write a Dynamic SQL program for user entered deptno then display deptname, location from dept table by using into, using clauses?
ANS: sql> declare

```
    v_deptno number(10):= &deptno;
    v_dname varchar2(10);
    v_loc varchar2(10);
    begin
    execute immediate 'select dname, loc from dept where deptno:=1 into v_dname, v_loc
using v_dept';
    dbms_output.put_line(v_dname||' '||v_loc);
    end;
    /
```

Output:
Enter value for deptno: 10
ACCOUNTING   NEWYORK

MEMBER PROCEDURE, MEMBER FUNCTIONS:

Oracle 8.0, introduced object technology in Oracle object type is an User defined type. Which contains number data and also member subprograms these member subprograms access member data.

Object Type:
Object is an User defined type which is used to represent different data types in a Single unit.
Syntax:
Create or replace type typename as object(attribute1 datatype(size),…….);

After creating object type we must instantiating that object in PL/SQL block by using following **syntax:**

Instantiating an Object:
Syntax:
Declare
Variablename objectname;

Assign values into Object type:
Syntax:
Variablename:= objectname(value1, value2,…..);

Sql> create or replace type student as object stno number(10), sname varchar2(10));
/
Instantiating an Object:
Sql> declare
    st1 student;
    begin
    st1:= student(10, 'murali');
    dbms_output.put_line('student id is:'||' '||st1.sno);
    dbms_output.put_line('student name is:'||' '||st1.name);
    end;
    /
Output:
Student id is : 101
Student name is: murali
If we want to define member subprograms then we must use type body within object type.
Object type having two parts.
1. Object Specification
2. Object Body

In Object specification we must declare member subprograms. In object body we are implementing those member subprograms. Once we are implementing members subprograms then only we are calling those subprograms through object instance.
Syntax:
Object Specification:
Create or replace type typename as object(attributename datatype(size), member procedure declaration, member function declaration);
Object Body:
Syntax:
Create or replace type body typename
as
member procedure implementation;
member function declaration;
end;
/

Sql> create or replace type circle as object(r number, member procedure p1, member function f1 return number);
/
Sql> create or replace type body circle
    as
    member procedure p1
    is
    begin
    dbms_output.put_line('proc');
    end p1;
    member function f1 return number
    is
    begin
    return 2*3.14*r;
    end f1;
    end;
    /
Execution: (Instantiating an Object)
Sql> declare
    Obj1 circle;
    begin
    obj1:= circle(4);
    dbms_output.put_line(obj1.f1);
    end;
    /
Output: 25.12

12C Features: (2013 june 29)

1. Invisible Columns:
Oracle 12C, introduced invisible columns either at the time of table creation or after table creation using alter command.
Syntax:
Columnname datatype(size) invisible
Eg: sql> create table test(sno number(10), name varchar2(10));
    Sql> insert into test values(50);
    Error: Insufficient values inserted.
    Sql> alter table test modify name invisible;
    Sql> insert into test values(50);
    1 row inserted
    Sql> select * from test;
    SNO   NAME

    50

2. Identity Columns:

Oracle 12C, introduced identity columns to generate primary key values automatically. Prior to Oracle 12C if we want to generate primary key values automatically then we are using Sequences, Trigger. Oracle 12C provides auto increment concept without using triggers by using following two methods.

1. With using Sequences
2. Without using Sequences.

With Using Sequences:

If we want to generate primary key value automatically then we are using "**default**" clause along with sequence pseudo columns within primary key.

Syntax:

columnname datatype(size)

default sequencename. nextval.primarykey

EG: sql> create table test(sno number(10))

    default s1.nextval primary key, name varchar2(10);

sql> insert into test(name) values('&name');

murali

abc

sql> select * from test;

NAME

Murali

Abc

Without using Sequences: (Inbuilt sequences)

Oracle 12C, also provides generating primary key values automatically by using "Identity columns" in this case we are using generated as "Identity" clause. Whenever we are using this clause Oracle server internally automatically create a sequence. That's why in this method we are not allowed to create sequence explicitly.

Syntax:

columnname.datatype(size) generated as identity primary key

EG: sql> create table test(sno number(10) generated as identity primary key, name varchar2(10));

Sql> insert into test(name) values('&name');

a

b

sql> select * from test;

SNO   NAME

----

1   a

2   b

3. Truncate table cascade:

Prior to Oracle 12C, when we are truncating master table if child table records are exists within child table then oracle server returns an error.

Error: unique/ primary keys in table references by Enabled Foreign keys.

If child table contains within "ON DELETE" cascade (or) without "ON DELETE" cascade also. To overcome this problem Oracle 12C introduced "truncate table cascade" clause which truncates master table records automatically if child table records exists also. But in this case child table must contain "ON DELETE cascade" clause.

Syntax:

truncate table tablename cascade;

EG: sql> create table mas(sno number(10) primary key);

Sql> insert into mas values(&sno);

Sql> select * from mas;

SNO

1

2

Sql> create table child(sno number(10) references mas(sno) on delete cascade);

Sql> insert into child values(&sno);

Sql> select * from child;

SNO

1

1

2

2

1

2

1

Sql> truncate table mas cascade;

Sql> select * from mas;

Sql> select * from child;

4. Session Sequence:

Oracle 12C, introduced Session Specific sequence by using "**SESSION**" keyword.

Syntax:

Create sequence sequencename session;

Session 1:

sql> create sequence s1 session;

Sql> select s1.nextval from dual;

Session 2:

Sql> select s1.nextval from dual;
SNO

1

Date: 3/8/15

5. Top-N query Extension:
Oracle 12C, introduced fetch first, fetch next, offset clauses for returning toppest, highest or lowest values in a table.
Prior to Oracle 12C, we can achieve Top-N analysis by using Inline views, Rownum.
Syntax:
select * from tablename order by columnname asc/desc offset n rows/fetch first/next n rows only;

Q) Write a query to display first 5 highest salary employees from emp table by using fetch first values?
ANS: sql> select * from emp order by sal desc fetch first 5 rows only;

Q) Write a query to display except first 3 rows then display next onwards 5 rows only from emp table?
ANS: sql> select * from emp order by sal desc offset 3 rows fetch first 5 rows only;

WITH clause introduced in PL/SQL functions:
Oracle 12C, introduced "WITH" clause functions and those functions are immediately calling in **SELECT** statement. These used defined functions are "Local functions".
Prior to Oracle 12C, Oracle 9i introduced "WITH" clauses in Top-N analysis. Those "WITH" clauses temporary tables are used in SELECT statement.
Syntax:
with anyname as
(select columns from tablename)
select col1, col2 from anyname where condition;
EG:
Q) Write a query to display first 5 highest salary employees from emp table by using WITH clauses?
ANS: sql> select temp as
      (select ename, sal from emp order by sal desc)
      Select ename, sal from temp where rownum<=5;
Output:
ENAME   SAL

KING   5400
SCOTT   3200
FORD   3200

158

JONES   2375
BLAKE   2250

In Oracle 12C
Syntax:
with
function functionname(formal parameters)
return datatype
is/as
-----
begin
-----
return expr;
------
end[function name]
select functionname(actual parameters) from dual;
EG:
Sql> with function f1(a number)
     return number
     is
     begin
     return a*a;
     end
     select f1(4) from dual;
Output: 16

ACCESSESIBLE BY clause:
Oracle 12C, introduced "Accessesible By" clauses within "Stored sub programs" which provide
more security for the accessing sub programs. Accessible by clause used in sub program
specification only.
Syntax:
create or replace procedure procedurename(formal parameters) accessesible by(another
procedure name)
is/as
begin

[exeception]
----
end[procedurename];
Sql> create or replace procedure p1
     is/as
     begin
     p2;
     dbms_output.put_line('first proc');

```
          end;
          /
Sql> create or replace procedure p2
     accessesible by(p1)
     is
     begin
     dbms_output.put_line('second proc');
     end;
     /
```

Oracle 11G features:

1. Oracle 11G introduced read-only tables by using "alter" command. In these tables we cannot perform DML operations.

Syntax:
 alter table tablename read only;

Syntax:
alter table tablename read write;

2. Simple_integer:

Oracle 11G, introduced "Simple_integer" data type in PL/SQL. Simple_integer data type internally having "not null" clause. That's why we must assign the value at the time of variable declaration in declare section of the PL/SQL block.

Syntax:
variablename simple_integer:= value;

EG: sql> declare
```
     a simple_integer:= 50;
     begin
     dbms_output.put_line(a);
     end;
     /
```
Output: 50

NOTE: Simple_integer data type performance is very high compare to "pls_integer" data type.

3. Pivot():

Oracle 11G, introduced pivot() function which is used to display aggregate values in tabular format and also rows are converted into columns. Prior to Oracle 11G if we want to achieve pivoting reports then we are using "decode()" within "group by" clause.

But Pivot() function performance is very high compare to decode().

Syntax:
select * from
(select col1, col2,…. From tablename)
Pivot(aggregate functionname(colname) for columnname in(value1, value2…..));

EG:

160

Sql> select * from
(select job, deptno, sal from emp pivot(sum(sal) for deptno in (10 as deptno10, 20 as deptno20, 30 as deptno30));

Output:

JOB	DEPTNO10	DEPTNO20	DEPTNO30
CLERK	2100	3100	1500
SALESMAN			
PRESIDENT	5400		
MANAGER	2050	2375	2250
ANALYST		6400	

4. Virtual Columns:
Oracle 11G, introduced "Virtual columns" which is used to store "Stored Expressions" directly in database by using "generated always as" clause.

5. Continue statement:
Oracle 11G, introduced "**Continue**" statement within PL/SQL loops. It is also same as "C" language Continue statement.
Syntax:
Continue
EG:
Sql> begin
    for i in 1..10
    loop
    if i =5 then
    continue;
    end if;
    dbms_output.put_line(i);
    end loop;
    end;
    /
Whenever we are using "Continue" statement automatically control sends to the beginning of the loop.

6. Oracle 11G, introduced "Compound Triggers".
7. Oracle 11G, introduced "FOLLOWS" clause.
8. Variable Assignment:
Oracle 11G, introduced "Variable" assignment concept when we are using sequences in PL/SQL block. In this case we are not allowed to use Dual table.
Syntax:

```
begin
varname:= sequencename.nextval;
end;
/
```

9. Enable (or) Disable clause:

Oracle 11G, introduced Enable (or) Disable clauses in trigger specification itself.

Syntax:
```
create or replace trigger triggername
befor/after insert/update/delete on tablename
[for each row]
[when condition]
[enable/disable]
[declare]
begin
-------
-------
end;
/
```

10. Oracle 11G, introduced "**Named**", "**Mixed**" notations when a subprogram is executed by using "SELECT" statement.

Predefined Packages:

DBMS_SQL executes dynamically constructed SQL statements and PL/SQL blocks of code.
DBMS_PIPE communicates between different Oracle sessions through a pipe in the RDBMS shared memory.
DBMS_JOB submits and manages regularly scheduled jobs for execution inside the database.
DBMS_LOB accesses and manipulates Oracle8's large objects (LOBs) from within PL/SQL programs.

Murali sir mail id: "**mvmsrinivas@gmail.com**"

Document prepared by: L. Dinesh kumar(8341084198)

END OF PL/SQL NOTES ☺
ALL THE BEST