ORACL 10g - PL/SQL

INTRODUCTION

CHARACTERSTICS

1. **Highly structured, readable and accessible language.**
2. **Standard and Protable language.**
3. **Embedded language.**
4. **Improved execution authority.**

10g FEATURES

1. ***Optimized compiler***

.

To change the optimizer settings for the entire database, set the database parameter PLSQL_OPTIMIZE_LEVEL. Valid settings are as follows

0 - No optimization

1 - Moderate optimization

2 - Aggressive optimization

These settings are also modifiable for the current session.

SQL> alter session set plsql_optimze_level=2;

Oracle retains optimizer settings on a module-by-module basis. When you recompile a particular module with nondefault settings, the settings will stick allowing you to recompile later on using REUSE SETTINGS.

SQL> Alter procedure proc compile plsql_optimize_level=1;

SQL> Alter procedure proc compile reuse settings;

2. ***Compile-time warnings.***

Starting with oracle database 10g release 1 you can enable additional

compile-time warnings to help make your programs more robust. The compiler can detect potential runtime problems with your code, such as identifying lines of code that will never be run. This process, also known as *lint checking*.

To enable these warnings fo the entire database, set the database parameter PLSQL_WARNINGS. These settings are also modifiable for the current session.

SQL> alter session set plsql_warnings = 'enable:all';

The above can be achieved using the built-in package DBMS_WARNING.

3 Conditional compilation.

Conditional compilation allows the compiler to allow to compile selected parts of a program based on conditions you provide with the $IF directive.

4 Support for non-sequential collections in FORALL.

5 Improved datatype support.

6 Backtrace an exception to its line number.

When handling an error, how can you find the line number on which the error was originally raised?

In earlier release, the only way to do this was allow you exception to go unhandled and then view the full error trace stack.

Now you can call DBMS_UTILITY.FORMAT_ERROR_BACKTRACE function to obtain that stack and manipulate it programmatically within your program.

7 Set operators for nested tables.

8 Support for regular expressions.

Oracle database 10g supports the use of regular expressions inside PL/SQL

code via four new built-in functions.

- ✓ **REGEXP_LIKE**
- ✓ **REGEXP_INSTR**
- ✓ **REGEXP_SUBSTR**
- ✓ **REGEXP_REPLACE**

9 Programmer-defined quoting mechanism.

Starting with oracle database 10g release 1, you can define your own quoting mechanism for string literals in both SQL and PL/SQL.

Use the characters q'(q followed by a single quote) to note the programmer-defined deliemeter for you string literal.

Ex:

```
DECLARE
   v varchar(10) := 'computer';
BEGIN
   dbms_output.put_line(q'*v  = *' || v);
   dbms_output.put_line(q'$v  = $' || v);
END;
```

Output:

```
v = computer
v = computer
```

10 Many new built-in packages.

DBMS_SCHEDULER

Represents a major update to DBMS_JOB. DBMS_SCHEDULER provides much improved functionality for scheduling and executing jobs defined via stored procedures.

DBMS_CRYPTO

Offers the ability to encrypt and decrypt common oracle datatype, including RAWs, BLOBs, and CLOBs. It also provides globalization support for encrypting data across different charactersets.

DBMS_MONITOR

Provides an API to control additional tracing and statistics gathering of sessions.

DBMS_WARNING

Provides an API into the PL/SQL compiler warnings module, allowing you to read and change settings that control which warnings are suppressed, displayed, or treated as errors.

STANDARD PACKAGE

Oracle has defined in this special package. Oracle defines quite a few identifiers in this package, including built-in exceptions, functions and subtypes.
You can reference the built-in form by prefixing it with STANDARD.

The basic unit in any PL/SQL program is block. All PL/SQL programs are composed of blocks which can occur sequentially or nested.

BLOCK STRUCTURE

```
Declare
-- declarative section
Begin
-- executable section
Exception
-- exception section
End;
```

In the above declarative and exceptiona sections are optional.

BLOCK TYPES

1. **Anonymous blocks**
2. **Named blocks**
   - ✓ **Labeled blocks**
   - ✓ **Subprograms**
   - ✓ **Triggers**

ANONYMOUS BLOCKS

Anonymous blocks implies basic block structure.

Ex:

```
BEGIN
   Dbms_output.put_line('My  first program'):
END;
```

LABELED BLOCKS

Labeled blocks are anonymous blocks with a label which gives a name to the block.

Ex:

```
<<my_bloock>>
BEGIN
   Dbms_output.put_line('My  first program'):
END;
```

SUBPROGRAMS

Subprograms are procedures and functions. They can be stored in the database as stand-alone objects, as part of package or as methods of an object type.

TRIGGERS

Triggers consists of a PL/SQL block that is associated with an event that occur in the database.

NESTED BLOCKS

A block can be nested within the executable or exception section of an outer block.

IDENTIFIERS

Identifiers are used to name PL/SQL objects, such as variables, cursors, types and subprograms. Identifiers consists of a letter, optionally followed by any sequence of characters, including letters, numbers, dollar signs, underscores, and pound signs only. The maximum length for an identifier is 30 characters.

QUOTED IDENTIFIERS

If you want to make an identifier case sensitive, include characters such as spaces or use a reserved word, you can enclose the identifier in double quotation marks.

Ex:

```
DECLARE
    "a" number := 5;
    "A" number := 6;
BEGIN
    dbms_output.put_line('a  = ' || a);
    dbms_output.put_line('A  = ' || A);
END;
```

Output:

```
a = 6
A = 6
```

COMMENTS

Comments improve readability and make your program more understandable. They are ignored by the PL/SQL compiler. There are two types of comments available.

1. **Single line comments**
2. **Multiline comments**

SINGLE LINE COMMENTS

A single-line comment can start any point on a line with two dashes and continues until the end of the line.

Ex:

```
BEGIN
    Dbms_output.put_line('hello');          -- sample program
END;
```

MULTILINE COMMENTS

Multiline comments start with the /* delimiter and ends with */ delimiter.

Ex:

```
BEGIN
    Dbms_output.put_line('hello');          /* sample program */
END;
```

VARIABLE DECLERATIONS

Variables can be declared in declarative section of the block;

Ex:

```
DECLARE
    a number;
    b number := 5;
    c number default 6;
```

CONSTANT DECLERATIONS

To declare a constant, you include the CONSTANT keyword, and you must supply a default value.

Ex:

```
DECLARE
    b constant number := 5;
```

```
c constant number default 6;
```

NOT NULL CLAUSE

You can also specify that the variable must be not null.

Ex:

```
DECLARE
    b constant number not null:= 5;
    c number not null default 6;
```

ANCHORED DECLERATIONS

PL/SQL offers two kinds of achoring.

1. **Scalar anchoring**
2. **Record anchoring**

SCALAR ANCHORING

Use the %TYPE attribute to define your variable based on table's column of some other PL/SQL scalar variable.

Ex:

```
DECLARE
    dno dept.deptno%type;
    Subtype t_number is number;
    a t_number;
    Subtype t_sno is student.sno%type;
    V_sno t_sno;
```

RECORD ANCHORING

Use the %ROWTYPE attribute to define your record structure based on a table.

Ex:

```
`DECLARE
 V_dept dept%rowtype;
```

BENEFITS OF ANCHORED DECLARATIONS

1. Synchronization with database columns.
2. Normalization of local variables.

PROGRAMMER-DEFINED TYPES

With the SUBTYPE statement, PL/SQL allows you to define your own subtypes or aliases of predefined datatypes, sometimes referred to as abstract datatypes.
There are two kinds of subtypes.

1. Constrained
2. Unconstrained

CONSTRAINED SUBTYPE

A subtype that restricts or constrains the values normally allowd by the datatype itself.

Ex:

```
Subtype positive is binary_integer range 1..2147483647;
```

In the above declaration a variable that is declared as positive can store only ingeger greater than zero even though binary_integer ranges from -2147483647..+2147483647.

UNCONSTRAINED SUBTYPE

A subtype that does not restrict the values of the original datatype in variables declared with the subtype.

Ex:

```
Subtype float is number;
```

DATATYPE CONVERSIONS

PL/SQL can handle conversions between different families among the datatypes.

Conversion can be done in two ways.

1. **Explicit conversion**
2. **Implicit conversion**

EXPLICIT CONVERSION

This can be done using the built-in functions available.

IMPLICIT CONVERSION

PL/SQL will automatically convert between datatype families when possible.

Ex:

```
DECLARE
   a varchar(10);
BEGIN
   select deptno into a from dept where dname='ACCOUNTING';
END;
```

In the above variable a is char type and deptno is number type even though, oracle will automatically converts the numeric data into char type assigns to the variable.

PL/SQL can automatically convert between

1. **Characters and numbers**
2. **Characters and dates**

VARIABLE SCOPE AND VISIBILITY

The scope of a variable is the portion of the program in which the variable can be accessed. For PL/SQL variables, this is from the variable declaration until the end of the block. When a variable goes out of scope, the PL/SQL engine will free the memory used to store the variable.

The visibility of a variable is the portion of the program where the variable can be accessed without having to qualify the reference. The visibility is always within

the scope. If it is out of scope, it is not visible.

Ex1:

```
DECLARE
     a number;       -- scope of a
BEGIN
--------
   DECLARE
      b number;      -- scope of b
   BEGIN
   -----
   END;
------
END;
```

Ex2:

```
DECLARE
    a number;
    b number;
BEGIN
    -- a , b available here
    DECLARE
       b char(10);
    BEGIN
       -- a and char type b is available here
    END;
    -----
END;
```

Ex3:

```
<<my_block>>
DECLARE
    a number;
    b number;
BEGIN
     -- a , b available here
     DECLARE
         b char(10);
     BEGIN
         -- a and char type b is available here
```

```
            -- number type b is available using <<my_block>>.b
        END;
        ------
    END;
```

PL/SQL CONTROL STRUCTURES

PL/SQL has a variety of control structures that allow you to control the behaviour of the block as it runs. These structures include conditional statements and loops.

1. If-then-else
2. Case
   - ✓ Case with no else
   - ✓ Labeled case
   - ✓ Searched case
3. Simple loop
4. While loop
5. For loop
6. Goto and Labels

IF-THEN-ELSE

Syntax:

```
If <condition1> then
   Sequence of statements;
Elsif <condition1> then
      Sequence of statements;
……
Else
    Sequence of statements;
End if;
```

Ex:

```
DECLARE
    dno number(2);
BEGIN
```

```
        select deptno into dno from dept where dname = 'ACCOUNTING';
        if dno = 10 then
          dbms_output.put_line('Location  is NEW YORK');
        elsif dno = 20 then
            dbms_output.put_line('Location  is DALLAS');
        elsif dno = 30 then
            dbms_output.put_line('Location  is CHICAGO');
        else
            dbms_output.put_line('Location  is BOSTON');
        end if;
    END;
```

Output:

```
Location is NEW YORK
```

CASE

Syntax:

```
Case test-variable
    When value1 then sequence of statements;
    When value2 then sequence of statements;
    ……
    When valuen then sequence of statements;
    Else sequence of statements;
End case;
```

Ex:

```
    DECLARE
       dno number(2);
    BEGIN
        select deptno into dno from dept where dname = 'ACCOUNTING';
        case dno
            when 10 then
                dbms_output.put_line('Location  is NEW YORK');
            when 20 then
```

```
                    dbms_output.put_line('Location is DALLAS');
                when 30 then
                    dbms_output.put_line('Location is CHICAGO');
                else
                    dbms_output.put_line('Location is BOSTON');
            end case;
        END;
```

Output:

```
        Location is NEW YORK
```

CASE WITHOUT ELSE

Syntax:

```
    Case test-variable
        When value1 then sequence of statements;
        When value2 then sequence of statements;
        ……
        When valuen then sequence of statements;
    End case;
```

Ex:

```
        DECLARE
            dno number(2);
        BEGIN
            select deptno into dno from dept where dname = 'ACCOUNTING';
            case dno
                when 10 then
                    dbms_output.put_line('Location is NEW YORK');
                when 20 then
                    dbms_output.put_line('Location is DALLAS');
                when 30 then
                    dbms_output.put_line('Location is CHICAGO');
                when 40 then
```

```
                    dbms_output.put_line('Location is BOSTON');
                end case;
        END;
```

Output:

```
        Location is NEW YORK
```

LABELED CASE

Syntax:

```
    <<label>>
    Case test-variable
        When value1 then sequence of statements;
        When value2 then sequence of statements;
        ……
        When valuen then sequence of statements;
    End case;
```

Ex:

```
        DECLARE
            dno number(2);
        BEGIN
            select deptno into dno from dept where dname = 'ACCOUNTING';
            <<my_case>>
            case dno
                when 10 then
                    dbms_output.put_line('Location is NEW YORK');
                when 20 then
                    dbms_output.put_line('Location is DALLAS');
                when 30 then
                    dbms_output.put_line('Location is CHICAGO');
                when 40 then
                    dbms_output.put_line('Location is BOSTON');
             end case my_case;
        END;
```

Output:

```
Location is NEW YORK
```

SEARCHED CASE

Syntax:

```
Case
    When <condition1> then sequence of statements;
    When <condition2> then sequence of statements;
    ……
    When <conditionn> then sequence of statements;
End case;
```

Ex:

```
DECLARE
    dno number(2);
BEGIN
    select deptno into dno from dept where dname = 'ACCOUNTING';
    case dno
        when dno = 10 then
            dbms_output.put_line('Location  is NEW YORK');
        when dno = 20 then
            dbms_output.put_line('Location  is DALLAS');
        when dno = 30 then
            dbms_output.put_line('Location  is CHICAGO');
        when dno = 40 then
            dbms_output.put_line('Location  is BOSTON');
    end case;
END;
```

Output:

```
Location is NEW YORK
```

SIMPLE LOOP

Syntax:

```
Loop
Sequence of statements;
Exit when <condition>;
End loop;
```

In the syntax exit when <*condition*> is equivalent to

```
If <condition> then
    Exit;
End if;
```

Ex:

```
DECLARE
    i number := 1;
BEGIN
    loop
       dbms_output.put_line('i  = ' || i);
       i := i + 1;
       exit when i > 5;
    end loop;
END;
```

Output:

```
i = 1
i = 2
i = 3
i = 4
i = 5
```

WHILE LOOP

Syntax:

```
While <condition> loop
Sequence of statements;
End loop;
```

Ex:

```
DECLARE
   i number := 1;
BEGIN
   While i <= 5 loop
         dbms_output.put_line('i  = ' || i);
         i := i + 1;
    end loop;
END;
```

Output:

```
i = 1
i = 2
i = 3
i = 4
i = 5
```

FOR LOOP

Syntax:

For <*loop_counter_variable*> in *low_bound..high_bound* loop
Sequence of statements;
End loop;

Ex1:

```
BEGIN
   For i in 1..5 loop
        dbms_output.put_line('i  = ' || i);
    end loop;
END;
```

Output:

```
i = 1
i = 2
i = 3
i = 4
i = 5
```

Ex2:

```
BEGIN
    For i in reverse 1..5 loop
        dbms_output.put_line('i  = ' || i);
    end loop;
END;
```

Output:

```
i = 5
i = 4
i = 3
i = 2
i = 1
```

NULL STATEMENT

Usually when you write a statement in a program, you want it to do something. There are cases, however, when you want to tell PL/SQL to do absolutely nothing, and that is where the NULL comes.

The NULL statement deos nothing except pass control to the next executable statement.

You can use NULL statement in the following situations.

1 Improving program readability.

Sometimes, it is helpful to avoid any ambiguity inherent in an IF statement that doesn't cover all possible cases. For example, when you write an IF statement, you do not have to include an ELSE clause.

2 Nullifying a raised exception.

When you don't want to write any special code to handle an exception, you can use the NULL statement to make sure that a raised exception halts execution of the current PL/SQL block but does not propagate any exceptions to enclosing blocks.

3 Using null after a label.

In some cases, you can pair NULL with GOTO to avoid having to execute

additional statements. For example, I use a GOTO statement to quickly move to the end of my program if the state of my data indicates that no further processing is required. Because I do not have to do anything at the termination of the program, I place a NULL statement after the label because at least one executable statement is required there. Even though NULL deos nothing, it is still an executable statement.

GOTO AND LABELS

Syntax:

Goto *label*;

Where *label* is a label defined in the PL/SQL block. Labels are enclosed in double angle brackets. When a goto statement is evaluated, control immediately passes to the statement identified by the label.

Ex:

```
BEGIN
   For i in 1..5 loop
        dbms_output.put_line('i  = ' || i);
        if i = 4 then
          goto exit_loop;
        end if;
   end loop;
   <<exit_loop>>
   Null;
END;
```

Output:

```
i = 1
i = 2
i = 3
i = 4
```

RESTRICTIONS ON GOTO

1 It is illegal to branch into an inner block, loop.

2 At least one executable statement must follow.

3 It is illegal to branch into an if statement.

4 It is illegal to branch from one if statement to another if statement.

5 It is illegal to branch from exception block to the current block.

PRAGMAS

Pragmas are compiler directives. They serve as instructions to the PL/SQL compiler. The compiler will act on the pragma during the compilation of the block.

Syntax:

PRGAMA *instruction_to_compiler*.

PL/SQL offers several pragmas:

1 AUTONOMOUS_TRANSACTION
2 EXCEPTION_INIT
3 RESTRICT_REFERENCES
4 SERIALLY_REUSABLE

SUBPROGRAMS

PROCEDURES

A procedure is a module that performs one or more actions.

Syntax:

```
Procedure [schema.]name [(parameter1 [,parameter2 …])]
        [authid definer | current_user] is
    -- [declarations]
Begin
    -- executable statements
[Exception
    -- exception handlers]
End [name];
```

In the above *authid* clause defines whether the procedure will execute under the authority of the definer of the procedure or under the authority of the current user.

FUNCTIONS

A function is a module that returns a value.

Syntax:

```
Function [schema.]name [(parameter1 [,parameter2 …])]
        Return return_datatype
        [authid definer | current_user]
        [deterministic]
        [parallel_enable] is
    -- [declarations]
Begin
    -- executable statements
[Exception
    -- exception handlers]
```

End [name];

In the above *authid* clause defines whether the procedure will execute under the authority of the definer of the procedure or under the authority of the current user.

***Deterministic* clause defines, an optimization hint that lets the system use a saved copy of the function's return result, if available. The quety optimizer can choose whether to use the saved copy or re-call the function.**

***Parallel_enable* clause defines, an optimization hint that enables the function to be executed in parallel when called from within SELECT statement.**

PARAMETER MODES

1	In (Default)
2	Out
3	In out

IN

In parameter will act as *pl/sql constant*.

OUT

1. **Out parameter will act as *unintialized variable*.**
2. **You cannot provide a default value to an *out* parameter.**
3. **Any assignments made to *out* parameter are rolled back when an exception is raised in the program.**
4. **An actual parameter corresponding to an *out* formal parameter must be a variable.**

IN OUT

5. **In out parameter will act as *initialized variable*.**
6. **An actual parameter corresponding to an *in out* formal parameter must be a variable.**

DEFAULT PARAMETERS

Default Parameters will not allow in the *beginning* and *middle*.
***Out* and *In Out* parameters can not have default values.**

Ex:

procedure p(a in number default 5, b in number default 6, c in number default 7) – valid
procedure p(a in number, b in number default 6, c in number default 7) – valild
procedure p(a in number, b in number, c in number default 7) – valild
procedure p(a in number, b in number default 6, c in number) – invalild
procedure p(a in number default 5, b in number default 6, c in number) – invalild
procedure p(a in number default 5, b in number, c in number) – invalild

NOTATIONS

Notations are of two types.

- **Positional notation**
- **Name notation**

We can combine positional and name notation but positional notation can not be followed by the name notation.

Ex:

Suppose we have a procedure proc(a number,b number,c number) and we have one
anonymous block which contains v1,v2, and v3;

```
SQL> exec proc (v1,v2,v3)          -- Positional notation
SQL> exec proc (a=>v1,b=>v2,c=>v3) -- Named notation
```

FORMAL AND ACTUAL PARAMETERS

1 Parametes which are in calling subprogram are *actual parameters*.

2 Parametes which are in called subprogram are *formal parameters*.

3 If any subprogram was called, once the call was completed then the values of formal

parameters are copied to the actual parameters.

Ex1:

```
CREATE OR REPLACE PROCEDURE SAMPLE(a in number,b out number,c in out number)
                is
               BEGIN
                  dbms_output.put_line('After call');
                  dbms_output.put_line('a = ' || a ||' b = ' || b || ' c = ' || c);
                  b := 10;
                  c := 20;
                  dbms_output.put_line('After assignment');
                  dbms_output.put_line('a = ' || a ||' b = ' || b || ' c = ' || c);
               END SAMPLE;
               DECLARE
                  v1 number := 4;
                  v2 number := 5;
                  v3 number := 6;
               BEGIN
                  dbms_output.put_line('Before call');
                 dbms_output.put_line('v1 = ' || v1 || ' v2 = ' || v2 || ' v3 = ' ||
               v3);
                  sample(v1,v2,v3);
                  dbms_output.put_line('After completion of call');
                 dbms_output.put_line('v1 = ' || v1 || ' v2 = ' || v2 || ' v3 = ' ||
               v3);
               END;
```

Output:

```
Before call
v1 = 4 v2 = 5 v3 = 6
After call
a = 4 b =  c = 6
After assignment
a = 4 b = 10 c = 20
```

```
After completion of call
v1 = 4 v2 = 10 v3 = 20
```

Ex2:

```
CREATE OR REPLACE FUN(a in number,b out number,c in out number)
return number
 IS
BEGIN
   dbms_output.put_line('After call');
   dbms_output.put_line('a  = ' || a || ' b = ' || b || ' c = ' || c);
   dbms_output.put_line('Before assignement Result = ' ||
(a*nvl(b,1)*c));
    b := 5;
    c := 7;
    dbms_output.put_line('After assignment');
    dbms_output.put_line('a  = ' || a || ' b = ' || b || ' c = ' || c);
    return (a*b*c);
 END FUN;
DECLARE
    v1 number := 1;
    v2 number := 2;
    v3 number := 3;
    v number;
BEGIN
    dbms_output.put_line('Before call');
   dbms_output.put_line('v1 = ' || v1 || ' v2 = ' || v2 || ' v3 = ' ||
v3);
    v := fun(v1,v2,v3);
    dbms_output.put_line('After call completed');
   dbms_output.put_line('v1 = ' || v1 || ' v2 = ' || v2 || ' v3 = ' ||
v3);
    dbms_output.put_line('Result = ' || v);
END;
```

Output:

```
Before call
v1 = 1 v2 = 2 v3 = 3
After call
```

a = 1 b = c = 3

Before assignement Result = 3

After assignment

a = 1 b = 5 c = 7

After call completed

v1 = 1 v2 = 5 v3 = 7

Result = 35

RESTRICTIONS ON FORMAL PARAMETERS

1. By declaring with specified size in actual parameters.
2. By declaring formal parameters with %type specifier.

USING NOCOPY

1. *Nocopy* is a hint, not a command. This means that the compiler might silently decide that it can't fulfill your request for a *nocopy* parameter.
2. The copying from formal to actual can be restricted by issuing *nocopy* qualifier.
3. To pass the out and in out parameters by reference use nocopy qualifier.

Ex:

```
CREATE OR REPLACE PROCEDURE PROC(a in out nocopy number)
IS
 BEGIN
   ----
END PROC;
```

CALL AND EXEC

Call is a SQL statement, which can be used to execute subprograms like exec.

Syntax:

Call *subprogram_name*([*argument_list*]) [into *host_variable*];

7. The parantheses are always required, even if the subprogram takes no arguments.

8 We can not use call with *out* and *in out* parameters.

9 Call is a SQL statement, it is not valid inside a PL/SQL block;

10 The INTO clause is used for the output variables of functions only.

11 We can not use 'exec' with *out* or *in out* parameters.

12 Exec is not valid inside a PL/SQL block;

Ex1:

```
CREATE OR REPLACE PROC
IS
BEGIN
    dbms_output.put_line('hello  world');
END PROC;
```

Output:

```
SQL> call proc();
      hello  world
```

Ex2:

```
CREATE OR REPLACE PROC(a in number,b  in number)
IS
BEGIN
    dbms_output.put_line('a  = ' || a || ' b = ' || b);
END PROC;
```

Output:

```
SQL> call proc(5,6);
      a = 5 b = 6
```

Ex3:

```
CREATE OR REPLACE FUNCTION FUN RETURN VARCHAR
IS
BEGIN
    return  'hello  world';
END FUN;
```

Output:

```
SQL> variable v varchar(20)
```

```
SQL> call fun() into :v;
SQL> print v
      hello world
```

CALL BY REFERENCE AND CALL BY VALUE

4 In parameters by default *call by reference* whereas out and in out *call by value*.
5 When parameter passed by reference, a pointer to the actual parameter is passed to the corresponding formal parameter.
6 When parameter passed by value it copies the value of the actual parameter to the formal parameter.
7 Call by reference is faster than the call by value because it avoids the copying.

SUBPROGRAMS OVERLOADING

1 Possible with different number of parameters.
2 Possible with different types of data.
3 Possible with same type with objects.
4 Can not be possible with different types of modes.
5 We can overload local subprograms also.

Ex:

```
SQL> create or replace type t1 as object(a number);/
SQL> create or replace type t2 as object(b number);/
      DECLARE
         i t1 := t1(5);
         j t2 := t2(5);
         PROCEDURE P(m t1) IS
         BEGIN
            dbms_output.put_line('a  = ' || m.a);
         END P;
         PROCEDURE P(n t2) IS
         BEGIN
            dbms_output.put_line('b  = ' || n.b);
         END P;
```

```
    PROCEDURE PRODUCT(a number,b number) IS
    BEGIN
       dbms_output.put_line('Product  of a,b = ' || a * b);
    END PRODUCT;
    PROCEDURE PRODUCT(a number,b number,c number) IS
    BEGIN
       dbms_output.put_line('Product  of a,b = ' || a * b * c);
    END PRODUCT;
BEGIN
    p(i);
    p(j);
    product(4,5);
    product(4,5,6);
END;
```

Output:

```
a = 5
b = 5
Product of a,b = 20
Product of a,b = 120
```

BENEFITS OF OVERLOADING

- **Supporting many data combinations**
- **Fitting the program to the user.**

RESTRICTIONS ON OVERLOADING

1. **Overloaded programs with parameter lists that differ only by name must be called using named notation.**
2. **The parameter list of overloaded programs must differ by more than parameter mode.**
3. **All of the overloaded programs must be defined within the same PL/SQL scope or block.**
4. **Overloaded functions must differ by more than their return type.**

IMPORTANT POINTS ABOUT SUBPROGRAMS

8 When a stored subprogram is created, it is stored in the *data dictionary*.
9 The subprogram is stored in compile form which is known as *p-code* in addition to the source text.
10 The p-code has all of the references in the subprogram evaluated, and the source code is translated into a form that is easily readable by PL/SQL engine.
11 When the subprogram is called, the p-code is read from the disk, if necessary, and executed.
12 Once it reads from the disk, the p-code is stored in the shared pool portion of the system global area (SGA), where it can be accessed by multiple users as needed.
13 Like all of the contents of the shared pool, p-code is aged out of the shared pool according to a least recently used (LRU) algorithm.
14 Subprograms can be *local*.
15 Local subprograms must be declared in the declarative section of PL/SQL block and called from the executable section.
16 Subprograms can not have the declarative section separately.
17 Stored subprograms can have local subprograms;
18 Local subprograms also can have local subprograms.
19 If the subprogram contains a variable with the same name as the column name of the table then use the dot method to differentiate (*subprogram_name*.sal).
20 Subprograms can be invalidated.

PROCEDURES V FUNCTIONS

13 Procedures may return through out and in out parameters where as function must return.
14 Procedures can not have return clause where as functions must.
15 We can use call statement directly for executing procedure where as we need to declare a variable in case of functions.
16 Functions can use in select statements where as procedures can not.
17 Functions can call from reports environment where as procedures can not.

18 We can use exec for executing procedures where as functions can not.

19 Function can be used in dbms_output where as procedure can not.

20 Procedure call is a standalone executable statement where as function call is a part of an executable statement.

STORED V LOCAL SUBPROGRAMS

1 The stored subprogram is stored in compiled p-code in the database, when the procedure is called it does not have to be compiled.

The local subprogram is compiled as part of its containing block. If the containing

block is anonymous and is run multiple times, the subprogram has to be compiled

each time.

1 Stored subprograms can be called from any block submitted by a user who has execute privileges on the subprogram.

Local subprograms can be called only from the block containing the subprogram.

2 By keeping the stored subprogram code separate from the calling block, the calling block is shorter and easier to understand.

The local subprogram and the calling block are one and the same, which can lead to

part confusion. If a change to the calling block is made, the subprogram will be

recompiled as of the recompilation of the containing block.

3 The compiled p-code can be pinned in the shared pool using the DBMS_SHARED_POOL Package. This can improve performance.

Local subprograms cannot be pinned in the shared pool by themselves.

4 Stand alone stored subprograms can not be overloaded, but packaged subprograms can be overloaded within the same package.

5 Local subprograms can be overloaded within the same block.

Ex1:

```
CREATE OR REPLACE PROCEDURE P IS
BEGIN
```

```
    dbms_output.put_line('Stored subprogram');
END;
```

Output:

```
SQL> exec p
Stored subprogram
```

Ex2:

```
DECLARE
    PROCEDURE P IS
    BEGIN
        dbms_output.put_line('Local subprogram');
    END;
BEGIN
    p;
END;
```

Output:

```
Local subprogram
```

COMPILING SUBPROGRAMS

6 SQL> Alter procedure P1 compile;

7 SQL> Alter function F1 compile;

SUBPROGRAMS DEPENDECIES

8 A stored subprogram is marked as invalid in the data dictionary if it has compile errors.

9 A stored subprogram can also become invalid if a DDL operation is performed on one of its dependent objects.

10 If a subprogram is invalidated, the PL/SQL engine will automatically attempt to recompile in the next time it is called.

11 If we have two procedures like P1 and P2 in which P1 depends on P2. If we compile P2 then P1 is invalidated.

SUBPROGRAMS DEPENDENCIES IN REMOTE DATABASES

1. We will call remote subprogram using connect string like P1@ORACLE;
2. If we have two procedures like P1 and P2 in which P1 depends on P2 but P2 was in remote database. If we compile P2 it will not invalidate P1 immediately because the data dictionary does not track remote dependencies.
3. Instead the validity of remote objects is checked at runtime. When P1 is called, the remote data dictionary is queried to determine the status of P2.
4. P1 and P2 are compared to see it P1 needs to be recompiled, there are two different methods of comparision
   - ✓ Timestamp Model
   - ✓ Signature Model

TIMESTAMP MODEL

- This is the default model used by oracle.
- With this model, the timestamps of the last modifications of the two objects are compared.
- The *last_ddl_time* field of *user_objects* contains the timestamp.
- If the base object has a newer timestamp than the dependent object, the dependent object will be recompiled.

ISSUES WITH THIS MODEL

1. If the objects are in different time zones, the comparison is invalid.
2. When P1 is in a client side PL/SQL engine such as oracle forms, in this case it may not possible to recompile P1, because the source for it may not be included with the forms.

SIGNATURE MODEL

1. When a procedure is created, a signature is stored in the data dictionary in addition to the p-code.
2. The signature encodes the types and order of the parametes.

3 When P1 is compiled the first time, the signature of P2 is included. Thus, P1 only needs to recompiled when the signature of P2 changes.

4 In order to use the signature model, the parameter REMOTE_DEPENDENCIES_MODE must be set to SIGNATURE. This is a parameter in the database initialization file.

THREE WAYS OF SETTING THIS MODE

1 Add the line REMOTE_DEPENDENCIES_MODE=SIGNATURE to the database initialization file. The next time the database is started, the mode will be set to SIGNATURE for all sessions.

2 Alter system set remote_dependencies_mode = signature;

This will affect the entire database (all sessions) from the time the statement is

issued. You must have the ALTER SYSTEM privilege to issue this command.

1 Alter session set remote_dependencies_mode = signature;

This will only affect your session

ISSUES WITH THIS MODEL

2 Signatures don't get modified if the default values of formal parameters are

changed.

3 Suppose P2 has a default value for one of its parameters, and P1 is using this

default value. If the default in the specification for P2 is changed, P1 will not be

recompiled by default. The old value for the default parameter will still be used until

P1 is manually recompiled.

2 If P1 is calling a packaged procedure P2, and a new overloaded version of P2 is added to the remote package, the signature is not changed. P1 will still use the old version(not the new overloaded one) until P1 is recompiled manually.

FORWARD DECLERATION

Before going to use the procedure in any other subprogram or other block , you must declare the prototype of the procedure in declarative section.

Ex1:

```
DECLARE
   PROCEDURE P1 IS
   BEGIN
      dbms_output.put_line('From  procedure p1');
      p2;
   END P1;
   PROCEDURE P2 IS
   BEGIN
      dbms_output.put_line('From  procedure p2');
      p3;
   END P2;
   PROCEDURE P3 IS
   BEGIN
      dbms_output.put_line('From  procedure p3');
   END P3;
BEGIN
   p1;
END;
```

Output:

```
p2;
*
ERROR at line 5:
ORA-06550:  line 5, column 1:
PLS-00313:  'P2' not declared in this scope
ORA-06550:  line 5, column 1:
PL/SQL:  Statement ignored
ORA-06550:  line 10, column 1:
PLS-00313:  'P3' not declared in this scope
ORA-06550:  line 10, column 1:
PL/SQL:  Statement ignored
```

Ex2:

```
DECLARE
   PROCEDURE P2;  -- forward declaration
   PROCEDURE P3;
   PROCEDURE P1 IS
   BEGIN
      dbms_output.put_line('From procedure p1');
      p2;
   END P1;
   PROCEDURE P2 IS
   BEGIN
      dbms_output.put_line('From procedure p2');
      p3;
   END P2;
   PROCEDURE P3 IS
   BEGIN
      dbms_output.put_line('From procedure p3');
   END P3;
BEGIN
   p1;
END;
```

Output:

```
From procedure p1
From procedure p2
From procedure p3
```

PRIVILEGES AND STORED SUBPROGRAMS

EXECUTE PREVILEGE

4 For stored subprograms and packages the relevant privilege is EXECUTE.

5 If user A had the procedure called emp_proc then user A grants execute privilege on procedure to user B with the following command.

SQL> Grant execute on emp_proc to user B.

6 Then user B can run the procedure by issuing

SQL> Exec user A.emp_proc

userA created the following procedure

```
CREATE OR REPLACE PROCEDURE P IS
    cursor is select *from student1;
BEGIN
    for v in c loop
        insert into student2 values(v.no,v.name,v.marks);
    end loop;
END P;
```

userA granted execute privilege to userB using

SQL> grant execute on p to userB

Then userB executed the procedure

SQL> Exec userA.p

If suppose userB also having student2 table then which table will populate whether userA's or userB's.

The answer is userA's student2 table only because by default the procedure will execute under the privlige set of its owner.

The above procedure is known as definer's procedure.

HOW TO POPULATE USER B's TABLE

7 Oracle introduces *Invoker's and Definer's rights.*

8 By default it will use the definer's rights.

9 An invoker's rights routine can be created by using AUTHID clause to populate the userB's table.

10 It is valid for stand-alone subprograms, package specifications, and object type specifications only.

userA created the following procedure

```
CREATE OR REPLACE PROCEDURE P
AUTHID CURRENT_USER IS
    cursor is select *from student;
BEGIN
    for v in c loop
        insert into student2 values(v.no,v.name,v.marks);
    end loop;
END P;
```

Then grant execute privilege on p to userB.
Executing the procedure by userB, which populates userB's table.

The above procedure is called invoker's procedure.
Instead of current_user of authid clause, if you use definer then it will be called definer' procedure.

STORED SUBPROGRAMS AND ROLES

we have two users saketh and sudha in which saketh has student table and sudha does not.
Sudha is going to create a procedure based on student table owned by saketh. Before doing this saketh must grant the permissions on this table to sudha.

```
SQL> conn saketh/saketh
SQL> grant all on student to sudha;
then sudha can create procedure
SQL> conn sudha/sudha

CREATE OR REPLACE PROCEDURE P IS
    cursor c is select *from saketh.student;
BEGIN
    for v in c loop
        dbms_output.put_line('No  = ' || v.no);
```

```
    end loop;
END P;
```

here procedure will be created.

If the same privilege was granted through a role it wont create the procedure.
Examine the following code

```
SQL> conn saketh/saketh
SQL> create role saketh_role;
SQL> grant all on student to saketh_role;
SQL> grant saketh_role to sudha;
then conn sudha/sudha

CREATE OR REPLACE PROCEDURE P IS
    cursor c is select *from saketh.student;
BEGIN
    for v in c loop
        dbms_output.put_line('No  = ' || v.no);
    end loop;
END P;
```

The above code will raise error instead of creating procedure .
This is because of early binding which PL/SQL uses by default in which references are evaluated in compile time but when you are using a role this will affect immediately.

ISSUES WITH INVOKER'S RIGHTS

1. In an invoker's rights routine, external references in SQL statements will be resolved using the caller's privilege set.
2. But references in PL/SQL statements are still resolved under the owner's privilege set.

TRIGGERS, VIEWS AND INVOKER'S RIGHTS

1. **A database trigger will always be executed with definer's rights and will execute under the privilege set of the schema that owns the triggering table.**
2. **This is also true for PL/SQL function that is called from a view. In this case, the function will execute under the privilege set of the view's owner.**

PACKAGES

A *package* is a container for related objects. It has specification and body. Each of them is stored separately in data dictionary.

PACKAGE SYNTAX

```
Create or replace package <package_name> is
   -- package specification includes subprograms signatures, cursors and global or
      public variables.
End <package_name>;

Create or replace package body <package_name> is
  -- package body includes body for all the subprograms declared in the spec, private
      Variables and cursors.
 Begin
   -- initialization section
 Exception
   -- Exception handling seciton
 End <package_name>;
```

IMPORTANT POINGS ABOUT PACKAGES

1. The first time a packaged subprogram is called or any reference to a packaged variable or type is made, the package is instantiated.
2. Each session will have its own copy of packaged variables, ensuring that two sessions executing subprograms in the same package use different memory locations.
3. In many cases initialization needs to be run the first time the package is instantiated within a session. This can be done by adding initialization section to the package body after all the objects.
4. Packages are stored in the data dictionary and can not be local.
5. Packaged subprograms has an advantage over stand alone subprogram.

6 When ever any reference to package, the whole package p-code was stored in shared pool of SGA.
7 Package may have local subprograms.
8 You can include authid clause inside the package spec not in the body.
9 The execution section of a package is know as initialization section.
10 You can have an exception section at the bottom of a package body.
11 Packages subprograms are not invalidated.

COMPILING PACKAGES

1 SQL> Alter package PKG compile;
2 SQL> Alter package PKG compile specification;
3 SQL> Alter package PKG compile body;

PACKAGE DEPENDENCIES

1 The package body depends on the some objects and the package header.
2 The package header does not depend on the package body, which is an advantage of packages.
3 We can change the package body with out changing the header.

PACKAGE RUNTIME STATE

Package runtime state is differ for the following packages.

1 Serially reusable packages
2 Non serially reusable packages

SERIALLY REUSABLE PACKAGES

To force the oracle to use serially reusable version then include PRAGMA SERIALLY_REUSABLE in both package spec and body, Examine the following package.

```
CREATE OR REPLACE PACKAGE PKG IS
    pragma serially_reusable;
    procedure emp_proc;
END PKG;
```

```
CREATE OR REPLACE PACKAGE BODY PKG IS
    pragma serially_reusable;
    cursor c is select ename from emp;
PROCEDURE EMP_PROC IS
    v_ename emp.ename%type;
    v_flag boolean := true;
    v_numrows  number := 0;
BEGIN
    if not c%isopen then
      open c;
    end if;
    while v_flag loop
          fetch c into v_ename;
          v_numrows := v_numrows + 1;
          if v_numrows = 5 then
            v_flag := false;
          end if;
          dbms_output.put_line('Ename  = ' || v_ename);
    end loop;
END EMP_PROC;
END PKG;
```

```
SQL> exec pkg.emp_proc
        Ename = SMITH
        Ename = ALLEN
        Ename = WARD
        Ename = JONES
        Ename = MARTIN


SQL> exec pkg.emp_proc
        Ename = SMITH
        Ename = ALLEN
        Ename = WARD
        Ename = JONES
```

Ename = MARTIN

1. The above package displays the same output for each execution even though the cursor is not closed.
2. Because the serially reusable version resets the state of the cursor each time it was called.

NON SERIALL Y REUSABLE PACKAGES

This is the default version used by the oracle, examine the following package.

```
CREATE OR REPLACE PACKAGE PKG IS
    procedure emp_proc;
END PKG;

CREATE OR REPLACE PACKAGE BODY PKG IS
    cursor c is select ename from emp;
PROCEDURE EMP_PROC IS
    v_ename emp.ename%type;
    v_flag boolean := true;
    v_numrows number := 0;
BEGIN
    if not c%isopen then
       open c;
   end if;
   while v_flag loop
          fetch c into v_ename;
          v_numrows := v_numrows + 1;
          if v_numrows = 5 then
            v_flag := false;
          end if;
          dbms_output.put_line('Ename = ' || v_ename);
    end loop;
END EMP_PROC;
END PKG;
```

```
SQL> exec pkg.emp_proc
```

```
Ename = SMITH
Ename = ALLEN
Ename = WARD
Ename = JONES
Ename = MARTIN
```

```
SQL> exec pkg.emp_proc

Ename = BLAKE
Ename = CLARK
Ename = SCOTT
Ename = KING
Ename = TURNER
```

1. The above package displays the different output for each execution even though the cursor is not closed.
2. Because the non-serially reusable version remains the state of the cursor over database calls.

DEPENDENCIES OF PACKAGE RUNTIME STATE

Dependencies can exists between package state and anonymous blocks.
Examine the following program

Create this package in first session

```
CREATE OR REPLACE PACKAGE PKG IS
    v number := 5;
    procedure p;
END PKG;

CREATE OR REPLACE PACKAGE BODY PKG IS
PROCEDURE P IS
BEGIN
dbms_output.put_line('v  = ' || v);
v := 10;
dbms_output.put_line('v  = ' || v);
END P;
```

```
END PKG;
```

Connect to second session, run the following code.

```
BEGIN
    pkg.p;
END;
```

The above code wil work.

Go back to first session and recreate the package using create.
Then connect to second session and run the following code again.

```
BEGIN
    pkg.p;
END;
```

This above code will not work because of the following.

1. **The anonymous block depends on pkg. This is compile time dependency.**
2. **There is also a runtime dependency on the packaged variables, since each session has its own copy of packaged variables.**
3. **Thus when pkg is recompiled the runtime dependency is followed, which invalidates the block and raises the oracle error.**
4. **Runtime dependencies exist only on package state. This includes variables and cursors declared in a package.**
5. **If the package had no global variables, the second execution of the anonymous block would have succeeded.**

PURITY LEVELS

In general, calls to subprograms are procedural, they cannot be called from SQL statements. However, if a stand-alone or packaged function meets certain restrictions, it can be called during execution of a SQL statement.

User-defined functions are called the same way as built-in functions but it must meet different restrictions. These restrictions are defined in terms of purity levels.

There are four types of purity levels.

WNDS	**--**	**Writes No Database State**
RNDS	**--**	**Reads No Database State**
WNPS	**--**	**Writes No Package State**
RNPS	**--**	**Reads No Package State**

In addition to the preceding restrictions, a user-defined function must also meet the following requirements to be called from a SQL statement.

- **The function has to be stored in the database, either stand-alone or as part of a package.**
- **The function can take only in parametes.**
- **The formal parameters must use only database types, not PL/SQL types such as boolean or record.**
- **The return type of the function must also be a database type.**
- **The function must not end the current transaction with commit or rollback, or rollback to a savepoint prior to the function execution.**
- **It also must not issue any alter session or alter system commands.**

RESTRICT_REFERENCES

For packaged functions, however, the RESTRICT_REFERENCES pragma is required to specify the purity level of a given function.

Syntax:

PRAGMA RESTRICT_REFERENCES(*subprogram_name or package_name*, WNDS [,WNPS]
[,RNDS] [,RNPS]);

Ex:

```
CREATE OR REPLACE PACKAGE PKG IS
    function fun1 return varchar;
    pragma restrict_references(fun1,wnds);
```

```
        function fun2 return varchar;
        pragma restrict_references(fun2,wnds);
END PKG;

CREATE OR REPLACE PACKAGE BODY PKG IS
        FUNCTION FUN1 return varchar IS
        BEGIN
            update dept set deptno = 11;
            return 'hello';
        END FUN1;
        FUNCTION FUN2 return varchar IS
        BEGIN
            update dept set dname ='aa';
            return 'hello';
       END FUN2;
END PKG;
```

The above package body will not created, it will give the following erros.

PLS-00452: Subprogram 'FUN1' violates its associated pragma
PLS-00452: Subprogram 'FUN2' violates its associated pragma

```
CREATE OR REPLACE PACKAGE BODY PKG IS
        FUNCTION FUN1 return varchar IS
        BEGIN
            return 'hello';
        END FUN1;
        FUNCTION FUN2 return varchar IS
        BEGIN
            return 'hello';
       END FUN2;
END PKG;
```

Now the package body will be created.

DEFAULT

If there is no RESTRICT_REFERENCES pragma associated with a given packaged function, it will not have any purity level asserted. However, you can change the

default purity level for a package. The DEFAULT keyword is used instead of the subprogram name in the pragma.

Ex:

```
CREATE OR REPLACE PACKAGE PKG IS
        pragma restrict_references(default,wnds);
        function fun1 return varchar;
        function fun2 return varchar;
END PKG;

CREATE OR REPLACE PACKAGE BODY PKG IS
        FUNCTION FUN1 return varchar IS
        BEGIN
             update dept set deptno = 11;
             return 'hello';
        END FUN1;
        FUNCTION FUN2 return varchar IS
        BEGIN
             update dept set dname ='aa';
             return 'hello';
       END FUN2;
END PKG;
```

The above package body will not created, it will give the following erros because the pragma will apply to all the functions.

PLS-00452: Subprogram 'FUN1' violates its associated pragma
PLS-00452: Subprogram 'FUN2' violates its associated pragma

```
CREATE OR REPLACE PACKAGE BODY PKG IS
        FUNCTION FUN1 return varchar IS
        BEGIN
             return 'hello';
        END FUN1;
        FUNCTION FUN2 return varchar IS
        BEGIN
             return 'hello';
       END FUN2;
```

```
END PKG;
```

Now the package body will be created.

TRUST

If the TRUST keyword is present, the restrictions listed in the pragma are not enforced. Rather, they are trusted to be true.

Ex:

```
CREATE OR REPLACE PACKAGE PKG IS
        function fun1 return varchar;
        pragma restrict_references(fun1,wnds,trust);
        function fun2 return varchar;
        pragma restrict_references(fun2,wnds,trust);
END PKG;

CREATE OR REPLACE PACKAGE BODY PKG IS
        FUNCTION FUN1 return varchar IS
        BEGIN
            update dept set deptno = 11;
            return 'hello';
        END FUN1;
        FUNCTION FUN2 return varchar IS
        BEGIN
            update dept set dname ='aa';
            return 'hello';
       END FUN2;
END PKG;
```

The above package will be created successfully.

IMPORTANT POINTS ABOUT RESTRICT_REFERENCES

- **This pragma can appear anywhere in the package specification, after the**

function declaration.

- It can apply to only one function definition.
- For overload functions, the pragma applies to the nearest definition prior to the Pragma.
- This pragma is required only for packages functions not for stand-alone functions.
- The Pragma can be declared only inside the package specification.
- The pragma is checked at compile time, not runtime.
- It is possible to specify without any purity levels when trust or combination of default and trust keywords are present.

PINNING IN THE SHARED POOL

The *shared pool* is the portion of the SGS that contains, among other things, the p-code of compiled subprograms as they are run. The first time a stored a store subprogram is called, the p-code is loaded from disk into the shared pool. Once the object is no longer referenced, it is free to be aged out. Objects are aged out of the shared pool using an LRU(Least Recently Used) algorithm.

The DBMS_SHARED_POOL package allows you to pin objects in the shared pool. When an object is pinned, it will never be aged out until you request it, no matter how full the pool gets or how often the object is accessed. This can improve performance, as it takes time to reload a package from disk.

DBMS_SHARED_POOL has four procedures

1. KEEP
2. UNKEEP
3. SIZES
4. ABORTED_REQUEST_THRESHOLD

KEEP

The DBMS_SHARED_POOL.KEEP procedure is used to pin objects in the pool.

Syntax:

PROCEDURE KEEP(*object_name* varchar2,*flag* char default 'P');

Here the flag represents different types of flag values for different types of objects.

P	**--**	**Package, function or procedure**
Q	**--**	**Sequence**
R	**--**	**Trigger**
C	**--**	**SQL Cursor**
T	**--**	**Object type**
JS	**--**	**Java source**
JC	**--**	**Java class**
JR	**--**	**Java resource**
JD	**--**	**Java shared data**

UNKEEP

UNKEEP is the only way to remove a kept object from the shared pool, without restarting the database. Kept objects are never aged out automatically.

Syntax:

PROCEDURE UNKEEP(*object_name* varchar2, *flag* char default 'P');

SIZES

SIZES will echo the contents of the shared pool to the screen.

Syntax:

PROCEDURE SIZES(*minsize* number);

Objects with greater than the *minsize* will be returned. SIZES uses DBMS_OUTPUT to return the data.

ABORTED_REQUEST_THRESHOLD

When the database determines that there is not enough memory in the shared pool to satisfy a given request, it will begin aging objects out until there is enough memory. It enough objects are aged out, this can have a performance impact on other database sessions. The ABORTED_REQUEST_THRESHOLD can be used to remedy this.

Syntax:

PROCEDURE ABORTED_REQUEST_THRESHOLD(*threshold_size* number);

Once this procedure is called, oracle will not start aging objects from the pool unless at least *threshold_size* bytes is needed.

DATA MODEL FOR SUBPROGRAMS AND PACKAGES

1. **USER_OBJECTS**
2. **USER_SOURCE**
3. **USER_ERRORS**
4. **DBA_OBJECTS**
5. **DBA_SOURCE**
6. **DBA_ERRORS**
7. **ALL_OBJECTS**
8. **ALL_SOURCE**
9. **ALL_ERRORS**

CURSORS

Cursor **is a pointer to memory location which is called as** ***context area*** **which contains the information necessary for processing, including the number of rows processed by the statement, a pointer to the parsed representation of the statement, and the** ***active set*** **which is the set of rows returned by the query.**

Cursor contains two parts

- ✓ **Header**
- ✓ **Body**

Header includes cursor name, any parameters and the type of data being loaded.
Body includes the select statement.

Ex:

Cursor c(dno in number) return dept%rowtype is select *from dept;

In the above

Header – cursor c(dno in number) return dept%rowtype

Body – select *from dept

CURSOR TYPES

1. **Implicit (SQL)**
2. **Explicit**
   - ✓ **Parameterized cursors**
   - ✓ **REF cursors**

CURSOR STAGES

1. **Open**
2. **Fetch**
3. **Close**

CURSOR ATTRIBUTES

1. %found
2. %notfound
3. %rowcount
4. %isopen
5. %bulk_rowcount
6. %bulk_exceptions

CURSOR DECLERATION

Syntax:

Cursor <*cursor_name*> is *select statement*;

Ex:

```
Cursor c is select *from dept;
```

CURSOR LOOPS

1. Simple loop
2. While loop
3. For loop

SIMPLE LOOP

Syntax:

```
Loop
    Fetch <cursor_name> into <record_variable>;
    Exit when <cursor_name> % notfound;
    <statements>;
End loop;
```

Ex:

```
DECLARE
   cursor c is select * from student;
   v_stud student%rowtype;
```

```
BEGIN
    open c;
    loop
      fetch c into v_stud;
      exit when c%notfound;
      dbms_output.put_line('Name  = ' || v_stud.name);
    end loop;
    close c;
END;
```

Output:

```
Name = saketh
Name = srinu
Name = satish
Name = sudha
```

WHILE LOOP

Syntax:

```
While <cursor_name>  % found loop
     Fetch <cursor_name>  nto <record_variable>;
      <statements>;
End loop;
```

Ex:

```
DECLARE
    cursor c is select * from student;
    v_stud student%rowtype;
BEGIN
    open c;
    fetch c into v_stud;
    while c%found loop
        fetch c into v_stud;
        dbms_output.put_line('Name  = ' || v_stud.name);
    end loop;
    close c;
END;
```

Output:

```
Name = saketh
Name = srinu
Name = satish
Name = sudha
```

FOR LOOP

Syntax:

```
for <record_variable> in <cursor_name> loop
    <statements>;
End loop;
```

Ex:

```
DECLARE
   cursor c is select * from student;
BEGIN
   for v_stud in c loop
      dbms_output.put_line('Name = ' || v_stud.name);
   end loop;
END;
```

Output:

```
Name = saketh
Name = srinu
Name = satish
Name = sudha
```

PARAMETARIZED CURSORS

1. This was used when you are going to use the cursor in more than one place with different values for the same where clause.
2. Cursor parameters must be *in* mode.
3. Cursor parameters may have default values.

4 The scope of cursor parameter is within the select statement.

Ex:

```
DECLARE
   cursor c(dno in number) is select * from dept where deptno = dno;
   v_dept dept%rowtype;
BEGIN
   open c(20);
   loop
      fetch c into v_dept;
      exit when c%notfound;
    dbms_output.put_line('Dname = ' || v_dept.dname || ' Loc = ' ||
v_dept.loc);
   end loop;
   close c;
END;
```

Output:

```
Dname = RESEARCH Loc = DALLAS
```

PACKAGED CURSORS WITH HEADER IN SPEC AND BODY IN PACKAGE BODY

1 cursors declared in packages will not close automatically.
2 In packaged cursors you can modify the select statement without making any changes to the cursor header in the package specification.
3 Packaged cursors with must be defined in the package body itself, and then use it as global for the package.
4 You can not define the packaged cursor in any subprograms.
5 Cursor declaration in package with out body needs the return clause.

Ex1:

```
CREATE OR REPLACE PACKAGE PKG IS
    cursor c return dept%rowtype is select * from dept;
    procedure proc is
END PKG;
```

```
CREATE OR REPLACE PAKCAGE BODY PKG IS
    cursor c return dept%rowtype is select * from dept;
PROCEDURE PROC IS
BEGIN
    for v in c loop
        dbms_output.put_line('Deptno  = ' || v.deptno || ' Dname = ' ||
                                          v.dname || ' Loc = ' || v.loc);
    end loop;
END PROC;
END PKG;
```

Output:

```
SQL> exec pkg.proc
     Deptno  = 10 Dname = ACCOUNTING Loc = NEW YORK
     Deptno  = 20 Dname = RESEARCH Loc = DALLAS
     Deptno  = 30 Dname = SALES Loc = CHICAGO
     Deptno  = 40 Dname = OPERATIONS Loc = BOSTON
```

Ex2:

```
CREATE OR REPLACE PAKCAGE BODY PKG IS
   cursor c return dept%rowtype is select * from dept where deptno
> 20;
PROCEDURE PROC IS
BEGIN
    for v in c loop
        dbms_output.put_line('Deptno  = ' || v.deptno || ' Dname = ' ||
                                  v.dname || ' Loc = ' || v.loc);
    end loop;
END PROC;
END PKG;
```

Output:

```
SQL> exec pkg.proc
     Deptno  = 30 Dname = SALES Loc = CHICAGO
     Deptno  = 40 Dname = OPERATIONS Loc = BOSTON
```

REF CURSORS AND CURSOR VARIABLES

1. This is unconstrained cursor which will return different types depends upon the user input.
2. Ref cursors can not be closed implicitly.
3. Ref cursor with return type is called *strong cursor*.
4. Ref cursor with out return type is called *weak cursor*.
5. You can declare ref cursor type in package spec as well as body.
6. You can declare ref cursor types in local subprograms or anonymous blocks.
7. Cursor variables can be assigned from one to another.
8. You can declare a cursor variable in one scope and assign another cursor variable with different scope, then you can use the cursor variable even though the assigned cursor variable goes out of scope.
9. Cursor variables can be passed as a parameters to the subprograms.
10. Cursor variables modes are in or out or in out.
11. Cursor variables can not be declared in package spec and package body (excluding subprograms).
12. You can not user remote procedure calls to pass cursor variables from one server to another.
13. Cursor variables can not use for update clause.
14. You can not assign nulls to cursor variables.
15. You can not compare cursor variables for equality, inequality and nullity.

Ex:

```
CREATE OR REPLACE PROCEDURE REF_CURSOR(TABLE_NAME IN VARCHAR) IS
    type t is ref cursor;
    c t;
    v_dept dept%rowtype;
   type r is record(ename emp.ename%type,job emp.job%type,sal
emp.sal%type);
    v_emp r;
    v_stud student.name%type;
BEGIN
    if table_name = 'DEPT' then
      open c for select * from dept;
```

```
        elsif table_name = 'EMP' then
          open c for select ename,job,sal from emp;
        elsif table_name = 'STUDENT' then
          open c for select name from student;
        end if;
        loop
          if table_name = 'DEPT' then
            fetch c into v_dept;
            exit when c%notfound;
            dbms_output.put_line('Deptno = ' || v_dept.deptno || ' Dname = ' ||
                                  v_dept.dname  || ' Loc = ' || v_dept.loc);
          elsif table_name = 'EMP' then
             fetch c into v_emp;
             exit when c%notfound;
           dbms_output.put_line('Ename = ' || v_emp.ename || ' Job = ' ||
   v_emp.job
                                 || ' Sal = ' || v_emp.sal);
          elsif table_name = 'STUDENT' then
             fetch c into v_stud;
             exit when c%notfound;
             dbms_output.put_line('Name = ' || v_stud);
          end if;
        end loop;
        close c;
     END;
```

Output:

```
SQL> exec ref_cursor('DEPT')

   Deptno = 10 Dname = ACCOUNTING Loc = NEW YORK
   Deptno = 20 Dname = RESEARCH Loc = DALLAS
   Deptno = 30 Dname = SALES Loc = CHICAGO
   Deptno = 40 Dname = OPERATIONS Loc = BOSTON

SQL> exec ref_cursor('EMP')
```

Ename = SMITH Job = CLERK Sal = 800
Ename = ALLEN Job = SALESMAN Sal = 1600
Ename = WARD Job = SALESMAN Sal = 1250
Ename = JONES Job = MANAGER Sal = 2975
Ename = MARTIN Job = SALESMAN Sal = 1250
Ename = BLAKE Job = MANAGER Sal = 2850
Ename = CLARK Job = MANAGER Sal = 2450
Ename = SCOTT Job = ANALYST Sal = 3000
Ename = KING Job = PRESIDENT Sal = 5000
Ename = TURNER Job = SALESMAN Sal = 1500
Ename = ADAMS Job = CLERK Sal = 1100
Ename = JAMES Job = CLERK Sal = 950
Ename = FORD Job = ANALYST Sal = 3000
Ename = MILLER Job = CLERK Sal = 1300

```
SQL> exec ref_cursor('STUDENT')
```

Name = saketh
Name = srinu
Name = satish
Name = sudha

CURSOR EXPRESSIONS

1. You can use cursor expressions in explicit cursors.
2. You can use cursor expressions in dynamic SQL.
3. You can use cursor expressions in REF cursor declarations and variables.
4. You can not use cursor expressions in implicit cursors.
5. Oracle opens the nested cursor defined by a cursor expression implicitly as soon as it fetches the data containing the cursor expression from the parent or outer cursor.
6. Nested cursor closes if you close explicitly.
7. Nested cursor closes whenever the outer or parent cursor is executed again or closed or canceled.
8. Nested cursor closes whenever an exception is raised while fetching data

from a parent cursor.

9 Cursor expressions can not be used when declaring a view.

10 Cursor expressions can be used as an argument to table function.

11 You can not perform bind and execute operations on cursor expressions when using the cursor expressions in dynamic SQL.

USING NESTED CURSORS OR CURSOR EXPRESSIONS

Ex:

```
DECLARE
    cursor c is select ename,cursor(select dname from dept d where e.empno =
    d.deptno)  from emp e;
    type t is ref cursor;
    c1 t;
    c2 t;
    v1 emp.ename%type;
    v2 dept.dname%type;
BEGIN
    open c;
    loop
        fetch c1 into v1;
        exit when c1%notfound;
        fetch c2 into v2;
        exit when c2%notfound;
        dbms_output.put_line('Ename  = ' || v1 || ' Dname = ' || v2);
    end loop;
    end loop;
    close c;
END;
```

CURSOR CLAUSES

1 Return
2 For update
3 Where current of
4 Bulk collect

RETURN

```
Cursor c return dept%rowtype is select *from dept;
                Or
Cursor c1 is select *from dept;
Cursor c  return c1%rowtype is select *from dept;
                Or
Type t is record(deptno dept.deptno%type, dname dept.dname%type);
Cursor c return t is select deptno, dname from dept;
```

FOR UPDATE AND WHERE CURRENT OF

Normally, a select operation will not take any locks on the rows being accessed. This will allow other sessions connected to the database to change the data being selected. The result set is still consistent. At open time, when the active set is determined, oracle takes a snapshot of the table. Any changes that have been committed prior to this point are reflected in the active set. Any changes made after this point, even if they are committed, are not reflected unless the cursor is reopened, which will evaluate the active set again.

However, if the FOR UPDATE caluse is pesent, exclusive row locks are taken on the rows in the active set before the open returns. These locks prevent other sessions from changing the rows in the active set until the transaction is committed or rolled back. If another session already has locks on the rows in the active set, then SELECT … FOR UPDATE operation will wait for these locks to be released by the other session. There is no time-out for this waiting period. The SELECT…FOR UPDATE will hang until the other session releases the lock. To handle this situation, the NOWAIT clause is available.

Syntax:

```
Select …from … for update of column_name [wait n];
```

If the cursor is declared with the FOR UPDATE clause, the WHERE CURRENT OF clause can be used in an update or delete statement.

Syntax:

Where current of cursor;

Ex:

```
DECLARE
    cursor c is select * from dept for update of dname;
BEGIN
    for v in c loop
        update dept set dname = 'aa' where current of c;
        commit;
    end loop;
END;
```

BULK COLLECT

1. This is used for array fetches
2. With this you can retrieve multiple rows of data with a single roundtrip.
3. This reduces the number of context switches between the pl/sql and sql engines.
4. Reduces the overhead of retrieving data.
5. You can use bulk collect in both dynamic and static sql.
6. You can use bulk collect in select, fetch into and returning into clauses.
7. SQL engine automatically initializes and extends the collections you reference in the bulk collect clause.
8. Bulk collect operation empties the collection referenced in the into clause before executing the query.
9. You can use the limit clause of bulk collect to restrict the no of rows retrieved.
10. You can fetch into multible collections with one column each.
11. Using the returning clause we can return data to the another collection.

BULK COLLECT IN FETCH

Ex:

```
DECLARE
    Type t is table of dept%rowtype;
    nt t;
    Cursor c is select *from dept;
BEGIN
    Open c;
    Fetch c bulk collect into nt;
    Close c;
    For i in nt.first..nt.last loop
        dbms_output.put_line('Dname = ' || nt(i).dname || ' Loc = ' ||
                                        nt(i).loc);
    end loop;
END;
```

Output:

```
Dname = ACCOUNTING Loc = NEW YORK
Dname = RESEARCH Loc = DALLAS
Dname = SALES Loc = CHICAGO
Dname = OPERATIONS Loc = BOSTON
```

BULK COLLECT IN SELECT

Ex:

```
DECLARE
    Type t is table of dept%rowtype;
    Nt t;
BEGIN
    Select * bulk collect into nt from dept;
    for i in nt.first..nt.last loop
        dbms_output.put_line('Dname = ' || nt(i).dname || ' Loc = ' ||
                                        nt(i).loc);
    end loop;
END;
```

Output:

```
Dname = ACCOUNTING Loc = NEW YORK
```

```
Dname = RESEARCH Loc = DALLAS
Dname = SALES Loc = CHICAGO
Dname = OPERATIONS Loc = BOSTON
```

LIMIT IN BULK COLLECT

You can use this to limit the number of rows to be fetched.

Ex:

```
DECLARE
    Type t is table of dept%rowtype;
    nt t;
    Cursor c is select *from dept;
BEGIN
    Open c;
    Fetch c bulk collect into nt limit 2;
    Close c;
    For i in nt.first..nt.last loop
        dbms_output.put_line('Dname = ' || nt(i).dname || ' Loc = ' ||
nt(i).loc);
    end loop;
END;
```

Output:

```
Dname = ACCOUNTING Loc = NEW YORK
Dname = RESEARCH Loc = DALLAS
```

MULTIPLE FETCHES IN INTO CLAUSE

Ex1:

```
DECLARE
    Type t is table of dept.dname%type;
    nt t;
    Type t1 is table of dept.loc%type;
    nt1 t;
    Cursor c is select dname,loc from dept;
```

```
BEGIN
    Open c;
    Fetch c bulk collect into nt,nt1;
    Close c;
    For i in nt.first..nt.last loop
        dbms_output.put_line('Dname  = ' || nt(i));
    end loop;
    For i in nt1.first..nt1.last loop
        dbms_output.put_line('Loc  = ' || nt1(i));
    end loop;
END;
```

Output:

```
Dname = ACCOUNTING
Dname = RESEARCH
Dname = SALES
Dname = OPERATIONS
Loc = NEW YORK
Loc = DALLAS
Loc = CHICAGO
Loc = BOSTON
```

Ex2:

```
DECLARE
    type t is table of dept.dname%type;
    type t1 is table of dept.loc%type;
    nt t;
    nt1 t1;
BEGIN
    Select dname,loc bulk collect into nt,nt1 from dept;
    for i in nt.first..nt.last loop
        dbms_output.put_line('Dname  = ' || nt(i));
    end loop;
    for i in nt1.first..nt1.last loop
        dbms_output.put_line('Loc  = ' || nt1(i));
    end loop;
```

```
END;
```

Output:

```
Dname = ACCOUNTING
Dname = RESEARCH
Dname = SALES
Dname = OPERATIONS
Loc = NEW YORK
Loc = DALLAS
Loc = CHICAGO
Loc = BOSTON
```

RETURNING CLAUSE IN BULK COLLECT

You can use this to return the processed data to the ouput variables or typed variables.

Ex:

```
DECLARE
    type t is table of number(2);
    nt t := t(1,2,3,4);
    type t1 is table of varchar(2);
    nt1 t1;
    type t2 is table of student%rowtype;
    nt2 t2;
BEGIN
    select name bulk collect into nt1 from student;
    forall v in nt1.first..nt1.last
          update student set no = nt(v) where name = nt1(v)
returning
                  no,name,marks bulk collect into nt2;
    for v in nt2.first..nt2.last loop
        dbms_output.put_line('Marks = ' || nt2(v));
    end loop;
END;
```

Output:

Marks = 100

Marks = 200

Marks = 300

Marks = 400

POINTS TO REMEMBER

1. **Cursor name can be up to 30 characters in length.**
2. **Cursors declared in anonymous blocks or subprograms closes automatically when that block terminates execution.**
3. **%bulk_rowcount and %bulk_exceptions can be used only with forall construct.**
4. **Cursor declarations may have expressions with column aliases.**
5. **These expressions are called virtual columns or calculated columns.**

SQL IN PL/SQL

The only statements allowed directly in pl/sql are DML and TCL.

BINDING

Binding a variable is the process of identifying the storage location associated with an identifier in the program.

Types of binding

1. **Early binding**
2. **Late binding**

3. **Binding during the compiled phase is early binding.**
4. **Binding during the runtime phase is late binding.**
5. **In early binding compile phase will take longer because of binding work but the execution is faster.**
6. **In late binding it will shorten the compile phase but lengthens the execution time.**
7. **PL/SQL by default uses early binding.**
8. **Binding also involves checking the database for permissions to access the object Referenced.**

DYNAMIC SQL

1. **If you use DDL in pl/sql it validates the permissions and existence if requires during compile time which makes invalid.**
2. **We can avoid this by using Dynamic SQL.**
3. **Dynamic SQL allows you to create a SQL statement dynamically at runtime.**

Two techniques are available for Dynamic SQL.

1 Native Dynamic SQL

2 DBMS_SQL package

USING NATIVE DYNAMIC SQL

USING EXECUTE IMMEDIATE

Ex:

```
BEGIN
   Execute immediate 'create table student(no number(2),name
varchar(10))';
                                    or
   Execute immediate ('create table student(no number(2),name
varchar(10))');
END;
```

USING EXECUTE IMMEDIATE WITH PL/SQL VARIABLES

Ex:

```
DECLARE
   v varchar(100);
BEGIN
   v := 'create table student(no number(2),name varchar(10))';
   execute immediate v;
END;
```

USING EXECUTE IMMEDIATE WITH BIND VARIABLES AND USING CLAUSE

Ex:

```
DECLARE
   v varchar(100);
BEGIN
   v := 'insert into student values(:v1,:v2,:v3)';
   execute immediate v using 6,'f',600;
END;
```

EXECUTING QUERIES WITH OPEN FOR AND USING CLAUSE

Ex:

```
CREATE OR REPLACE PROCEDURE P(smarks in number) IS
    s varchar(100) := 'select *from student where marks > :m';
    type t is ref cursor;
    c t;
    v student%rowtype;
BEGIN
    open c for s using smarks;
    loop
        fetch c into v;
        exit when c%notfound;
        dbms_output.put_line('Student  Marks = ' || v.marks);
    end loop;
    close c;
END;
```

Output:

```
SQL> exec p(100)

     Student Marks = 200
     Student Marks = 300
     Student Marks = 400
```

QUERIES WITH EXECUTE IMMEDIATE

Ex:

```
DECLARE
    d_name dept.dname%type;
    lc dept.loc%type;
    v varchar(100);
BEGIN
    v := 'select dname from dept where deptno = 10';
    execute immediate v into d_name;
    dbms_output.put_line('Dname  = '|| d_name);
    v := 'select loc from dept where dname = :dn';
```

```
        execute immediate v into lc using d_name;
        dbms_output.put_line('Loc  = ' || lc);
    END;
```

Output:

```
Dname = ACCOUNTING
Loc = NEW YORK
```

VARIABLE NAMES

Ex:

```
DECLARE
    Marks number(3) := 100;
BEGIN
    Delete student where marks = marks;      -- this will delete all the rows in
the
                                             -- student table
END;
```

This can be avoided by using the labeled blocks.

```
<<my_block>>
DECLARE
    Marks number(3) := 100;
BEGIN
   Delete student where marks = my_block.marks;      -- delete rows which
has
                                                  -- a marks of 100
END;
```

GETTING DATA INTO PL/SQL VARIABLES

Ex:

```
DECLARE
    V1 number;
    V2 varchar(2);
BEGIN
    Select no,name into v1,v2 from student where marks = 100;
END;
```

DML AND RECORDS

Ex:

```
CREATE OR REPLACE PROCEDURE P(srow in student%rowtype) IS
BEGIN
insert into student values srow;
END P;

DECLARE
    s student%rowtype;
BEGIN
    s.no := 11;
    s.name := 'aa';
    s.marks := 100;
    p(s);
END;
```

RECORD BASED INSERTS

Ex:

```
DECLARE
    srow student%rowtype;
BEGIN
    srow.no := 7;
    srow.name := 'cc';
    srow.marks := 500;
    insert into student values srow;
END;
```

RECORD BASED UPDATES

Ex:

```
DECLARE
     srow student%rowtype;
BEGIN
     srow.no := 6;
     srow.name := 'cc';
```

```
        srow.marks := 500;
        update student set row=srow  where no = srow.no;
    END;
```

USING RECORDS WITH RETURNING CLAUSE

Ex:

```
    DECLARE
        srow student%rowtype;
        sreturn student%rowtype;
    BEGIN
        srow.no := 8;
        srow.name := 'dd';
        srow.marks := 500;
        insert into student values srow returning  no,name,marks into sreturn;
        dbms_output.put_line('No  = ' || sreturn.no);
        dbms_output.put_line('No  = ' || sreturn.name);
        dbms_output.put_line('No  = ' || sreturn.marks);
    END;
```

Output:

```
No = 8
No = dd
No = 500
```

USING DBMS_SQL PACKAGE

DBMS_SQL is used to execute dynamic SQL from with in PL/SQL. Unlike native dynamic SQL, it is not built directly into the language, and thus is less efficient. The DBMS_SQL package allows you to directly control the processing of a statement within a cursor, with operations such as opening and closing a cursor, parsing a statement, binding input variable, and defining output variables.

Ex1:

```
    DECLARE
        cursor_id number;
```

```
        flag number;
        v_stmt varchar(50);
    BEGIN
        cursor_id := dbms_sql.open_cursor;
        v_stmt := 'create table stud(sno number(2),sname varchar(10))';
        dbms_sql.parse(cursor_id,v_stmt,dbms_sql.native);
        flag := dbms_sql.execute(cursor_id);
        dbms_sql.close_cursor(cursor_id);
        dbms_output.put_line('Table created');
    END;
```

Output:

```
Table created

SQL> desc stud
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------
 SNO                                                NUMBER(2)
 SNAME                                              VARCHAR2(10)
```

Ex2:

```
CREATE OR REPLACE PROCEDURE DBMS_SQL_PROC(v1 student.no%type,
                                          v2 student.marks%type) is
    cursor_id number;
    flag number;
    v_update varchar(50);
BEGIN
    cursor_id := dbms_sql.open_cursor;
    v_update := 'update student set marks = :smarks where no = :sno';
    dbms_sql.parse(cursor_id,v_update,dbms_sql.native);
    dbms_sql.bind_variable(cursor_id,':sno',v1);
    dbms_sql.bind_variable(cursor_id,':smarks',v2);
    flag := dbms_sql.execute(cursor_id);
    dbms_sql.close_cursor(cursor_id);
```

```
END DBMS_SQL_PROC;
```

Output:

```
SQL> select * from student;      -- before execution

      NO   NA      MARKS
      ---- ------ -- ----------
       1   a        100
       2   b        200
       3   c        300

SQL> exec dbms_sql_proc(2,222)

SQL> select * from student;      -- after execution

      NO   NA      MARKS
      ---- ------ -- ----------
       1   a        100
       2   b        222
       3   c        300
```

FORALL STATEMENT

This can be used to get the data from the database at once by reducting the number of context switches which is a transfer of control between PL/SQL and SQL engine.

Syntax:

```
Forall index_var in
        [ Lower_bound..upper_bound  |
        Indices of indexing_collection  |
        Values of indexing_collection  ]
SQL statement;
```

FORALL WITH NON-SEQUENTIAL ARRAYS

Ex:

```
DECLARE
    type t is table of student.no%type index by binary_integer;
    ibt t;
BEGIN
    ibt(1) := 1;
    ibt(10) := 2;
    forall i in ibt.first..ibt.last
            update student set marks = 900 where no = ibt(i);
END;
```

The above program will give error like 'element at index [2] does not exists. You can rectify it in one of the two following ways.

USGAGE OF INDICES OF TO AVOID THE ABOVE BEHAVIOUR

This will be used when you have a collection whose defined rows specify which rows in the binding array you would like to processed.

Ex:

```
DECLARE
    type t is table of student.no%type index by binary_integer;
    ibt t;
    type t1 is table of boolean index by binary_integer;
    ibt1 t1;
BEGIN
    ibt(1) := 1;
    ibt(10) := 2;
    ibt(100) := 3;
    ibt1(1) := true;
    ibt1(10) := true;
    ibt1(100) := true;
    forall i in indices of ibt1
            update student set marks = 900 where no = ibt(i);
END;
```

Ouput:

```
SQL> select * from student  -- before execution

        NO   NA    MARKS
---------- ----------
         1   a        100
         2   b        200
         3   c        300
SQL> select * from student  -- after execution

        NO   NA    MARKS
---------- ----------
         1   a        900
         2   b        900
         3   c        900
```

USGAGE OF VALUES OF TO AVOID THE ABOVE BEHAVIOUR

This will be used when you have a collection of integers whose content identifies the position in the binding array that you want to be processed by the FORALL statement.

Ex:

```
DECLARE
   type t is table of student.no%type index by binary_integer;
   ibt t;
   type t1 is table of pls_integer index by binary_integer;
   ibt1 t1;
BEGIN
   ibt(1) := 1;
   ibt(10) := 2;
   ibt(100) := 3;
   ibt1(11) := 1;
   ibt1(15) := 10;
   ibt1(18) := 100;
   forall i in values of ibt1
```

```
            update student set marks = 567 where no = ibt(i);
    END;
```

Ouput:

```
SQL> select * from student  -- before execution

        NO   NA    MARKS
---------- ----------
         1   a        100
         2   b        200
         3   c        300

SQL> select * from student  -- after execution

        NO   NA    MARKS
---------- ----------
         1   a        900
         2   b        900
         3   c        900
```

POINTS ABOUT BULK BINDS

1. Passing the entire PL/SQL table to the SQL engine in one step is known as bulk bind.
2. Bulk binds are done using the forall statement.
3. If there is an error processing one of the rows in bulk DML operation, only that row is rolled back.

POINTS ABOUT RETURING CLAUSE

1. This will be used only with DML statements to return data into PL/SQL variables.
2. This will be useful in situations like , when performing insert or update or delete if you want to know the data of the table which has been effected by the DML.

3 With out going for another SELECT using RETURNING clause we will get the data which will avoid a call to RDBMS kernel.

COLLECTIONS

Collections are also composite types, in that they allow you to treat several variables as a unit. A collection combines variables of the same type.

TYPES

1 Varrays
2 Nested tables
3 Index - by tables (Associate arrays)

VARRAYS

A varray is datatype very similar to an array. A varray has a fixed limit on its size, specified as part of the declaration. Elements are inserted into varray starting at index 1, up to maximum lenth declared in the varray type. The maximum size of the varray is 2 giga bytes.

Syntax:

Type <*type_name*> is varray | varying array (<*limit*>) of <*element_type*>;

Ex1:

```
DECLARE
    type t is varray(10) of varchar(2);
    va t := t('a','b','c','d');
    flag boolean;
BEGIN
    dbms_output.put_line('Limit  = ' || va.limit);
    dbms_output.put_line('Count  = ' || va.count);
    dbms_output.put_line('First  Index = ' || va.first);
    dbms_output.put_line('Last  Index = ' || va.last);
    dbms_output.put_line('Next  Index = ' || va.next(2));
    dbms_output.put_line('Previous  Index = ' || va.prior(3));
```

```
dbms_output.put_line('VARRAY ELEMENTS');
for i in va.first..va.last loop
    dbms_output.put_line('va[' || i || '] = ' || va(i));
end loop;
flag := va.exists(3);
if flag = true then
   dbms_output.put_line('Index 3 exists with an element ' || va(3));
else
   dbms_output.put_line('Index 3 does not exists');
end if;
va.extend;
dbms_output.put_line('After extend of one index, Count = ' || va.count);
flag := va.exists(5);
if flag = true then
   dbms_output.put_line('Index 5 exists with an element ' || va(5));
else
   dbms_output.put_line('Index 5 does not exists');
end if;
flag := va.exists(6);
if flag = true then
   dbms_output.put_line('Index 6 exists with an element ' || va(6));
else
   dbms_output.put_line('Index 6 does not exists');
end if;
va.extend(2);
dbms_output.put_line('After extend of two indexes, Count = ' ||
va.count);
dbms_output.put_line('VARRAY ELEMENTS');
for i in va.first..va.last loop
    dbms_output.put_line('va[' || i || '] = ' || va(i));
end loop;
va(5) := 'e';
va(6) := 'f';
va(7) := 'g';
dbms_output.put_line('AFTER ASSINGNING VALUES TO EXTENDED ELEMENTS,
```

```
                                        VARRAY ELEMENTS');
        for i in va.first..va.last loop
            dbms_output.put_line('va[' || i || '] = ' || va(i));
        end loop;
        va.extend(3,2);
       dbms_output.put_line('After extend of three indexes, Count = ' ||
    va.count);
        dbms_output.put_line('VARRAY ELEMENTS');
        for i in va.first..va.last loop
            dbms_output.put_line('va[' || i || '] = ' || va(i));
        end loop;
        va.trim;
        dbms_output.put_line('After trim of one index, Count = ' || va.count);
        va.trim(3);
        dbms_output.put_line('After trim of three indexs, Count = ' || va.count);
        dbms_output.put_line('AFTER TRIM, VARRAY ELEMENTS');
        for i in va.first..va.last loop
            dbms_output.put_line('va[' || i || '] = ' || va(i));
        end loop;
        va.delete;
       dbms_output.put_line('After delete of entire varray, Count = ' ||
    va.count);
    END;
```

Output:

```
Limit = 10
Count = 4
First Index = 1
Last Index = 4
Next Index = 3
Previous Index = 2
VARRAY ELEMENTS
va[1] = a
va[2] = b
va[3] = c
va[4] = d
```

Index 3 exists with an element c

After extend of one index, Count = 5

Index 5 exists with an element

Index 6 does not exists

After extend of two indexes, Count = 7

VARRAY ELEMENTS

va[1] = a

va[2] = b

va[3] = c

va[4] = d

va[5] =

va[6] =

va[7] =

AFTER ASSINGNING VALUES TO EXTENDED ELEMENTS, VARRAY ELEMENTS

va[1] = a

va[2] = b

va[3] = c

va[4] = d

va[5] = e

va[6] = f

va[7] = g

After extend of three indexes, Count = 10

VARRAY ELEMENTS

va[1] = a

va[2] = b

va[3] = c

va[4] = d

va[5] = e

va[6] = f

va[7] = g

va[8] = b

va[9] = b

va[10] = b

After trim of one index, Count = 9

```
After trim of three indexs, Count = 6
AFTER TRIM,  VARRAY  ELEMENTS
va[1] = a
va[2] = b
va[3] = c
va[4] = d
va[5] = e
va[6] = f
After delete of entire varray, Count = 0
```

Ex2:

```
DECLARE
    type t is varray(4) of student%rowtype;
    va t := t(null,null,null,null);
BEGIN
    for i in 1..va.count loop
        select * into va(i) from student where sno = i;
       dbms_output.put_line('Sno  =  '  ||  va(i).sno  ||  '  Sname  =  '  ||
va(i).sname);
    end loop;
END;
```

Output:

```
Sno = 1 Sname = saketh
Sno = 2 Sname = srinu
Sno = 3 Sname = divya
Sno = 4 Sname = manogni
```

Ex3:

```
DECLARE
     type t is varray(4) of student.smarks%type;
     va t := t(null,null,null,null);
BEGIN
     for i in 1..va.count loop
          select smarks into va(i) from student where sno = i;
          dbms_output.put_line('Smarks  = ' || va(i));
```

```
        end loop;
    END;
```

Output:

```
Smarks = 100
Smarks = 200
Smarks = 300
Smarks = 400
```

Ex4:

```
    DECLARE
        type r is record(c1 student.sname%type,c2 student.smarks%type);
        type t is varray(4) of r;
        va t := t(null,null,null,null);
    BEGIN
        for i in 1..va.count loop
            select sname,smarks into va(i) from student where sno = i;
           dbms_output.put_line('Sname = ' || va(i).c1 || ' Smarks = ' ||
    va(i).c2);
        end loop;
    END;
```

Output:

```
Sname = saketh Smarks = 100
Sname = srinu Smarks = 200
Sname = divya Smarks = 300
Sname = manogni Smarks = 400
```

Ex5:

```
    DECLARE
        type t is varray(1) of addr;
        va t := t(null);
        cursor c is select * from employ;
        i number := 1;
    BEGIN
        for v in c loop
            select address into va(i) from employ where ename = v.ename;
```

```
            dbms_output.put_line('Hno = ' || va(i).hno || ' City = ' ||
        va(i).city);
            end loop;
        END;
```

Output:

```
            Hno = 11 City = hyd
            Hno = 22 City = bang
            Hno = 33 City = kochi
```

Ex6:

```
        DECLARE
            type t is varray(5) of varchar(2);
            va1 t;
            va2 t := t();
        BEGIN
            if va1 is null then
              dbms_output.put_line('va1 is null');
            else
              dbms_output.put_line('va1 is not null');
            end if;
            if va2 is null then
              dbms_output.put_line('va2 is null');
            else
              dbms_output.put_line('va2 is not null');
            end if;
        END;
```

Output:

```
            va1 is null
            va2 is not null
```

NESTED TABLES

A nested table is thought of a database table which has no limit on its size. Elements are inserted into nested table starting at index 1. The maximum size of the varray is 2 giga bytes.

Syntax:

Type <*type_name*> is table of <*table_type*>;

Ex1:

```
DECLARE
    type t is table of varchar(2);
    nt t := t('a','b','c','d');
    flag boolean;
BEGIN
    if nt.limit is null then
       dbms_output.put_line('No  limit to Nested Tables');
    else
       dbms_output.put_line('Limit  = ' || nt.limit);
    end if;
    dbms_output.put_line('Count  = ' || nt.count);
    dbms_output.put_line('First  Index = ' || nt.first);
    dbms_output.put_line('Last  Index = ' || nt.last);
    dbms_output.put_line('Next  Index = ' || nt.next(2));
    dbms_output.put_line('Previous  Index = ' || nt.prior(3));
    dbms_output.put_line('NESTED TABLE ELEMENTS');
    for i in 1..nt.count loop
        dbms_output.put_line('nt[' || i || '] = ' || nt(i));
    end loop;
    flag := nt.exists(3);
    if flag = true then
      dbms_output.put_line('Index  3 exists with an element ' || nt(3));
    else
       dbms_output.put_line('Index  3 does not exists');
    end if;
    nt.extend;
   dbms_output.put_line('After  extend  of  one  index,  Count  =  ' ||
nt.count);
```

```
flag := nt.exists(5);
if flag = true then
   dbms_output.put_line('Index  5 exists with an element ' || nt(5));
else
   dbms_output.put_line('Index  5 does not exists');
end if;
flag := nt.exists(6);
if flag = true then
   dbms_output.put_line('Index  6 exists with an element ' || nt(6));
else
   dbms_output.put_line('Index  6 does not exists');
end if;
nt.extend(2);
dbms_output.put_line('After extend of two indexes, Count = ' ||
nt.count);
dbms_output.put_line('NESTED TABLE ELEMENTS');
for i in 1..nt.count loop
    dbms_output.put_line('nt[' || i || '] = ' || nt(i));
end loop;
nt(5) := 'e';
nt(6) := 'f';
nt(7) := 'g';
dbms_output.put_line('AFTER ASSINGNING VALUES TO EXTENDED ELEMENTS,
NESTED
                              TABLE ELEMENTS');
for i in 1..nt.count loop
     dbms_output.put_line('nt[' || i || '] = ' || nt(i));
end loop;
nt.extend(5,2);
dbms_output.put_line('After extend of five indexes, Count = ' ||
nt.count);
dbms_output.put_line('NESTED TABLE ELEMENTS');
for i in 1..nt.count loop
    dbms_output.put_line('nt[' || i || '] = ' || nt(i));
end loop;
nt.trim;
```

```
        dbms_output.put_line('After trim of one index, Count = ' || nt.count);
        nt.trim(3);
       dbms_output.put_line('After trim of three indexs, Count = ' ||
nt.count);
        dbms_output.put_line('AFTER TRIM, NESTED TABLE ELEMENTS');
        for i in 1..nt.count loop
             dbms_output.put_line('nt[' || i || '] = ' || nt(i));
        end loop;
        nt.delete(1);
       dbms_output.put_line('After delete of first index, Count = ' ||
nt.count);
        dbms_output.put_line('NESTED TABLE ELEMENTS');
        for i in 2..nt.count+1 loop
            dbms_output.put_line('nt[' || i || '] = ' || nt(i));
        end loop;
        nt.delete(4);
       dbms_output.put_line('After delete of fourth index, Count = ' ||
nt.count);
        dbms_output.put_line('NESTED TABLE ELEMENTS');
        for i in 2..3 loop
            dbms_output.put_line('nt[' || i || '] = ' || nt(i));
        end loop;
        for i in 5..nt.count+2 loop
            dbms_output.put_line('nt[' || i || '] = ' || nt(i));
        end loop;
        nt.delete;
        dbms_output.put_line('After delete of entire nested table, Count = ' ||
                                    nt.count);
END;
```

Output:

```
No limit to Nested Tables
Count = 4
First Index = 1
Last Index = 4
Next Index = 3
```

Previous Index = 2

NESTED TABLE ELEMENTS

nt[1] = a

nt[2] = b

nt[3] = c

nt[4] = d

Index 3 exists with an element c

After extend of one index, Count = 5

Index 5 exists with an element

Index 6 does not exists

After extend of two indexes, Count = 7

NESTED TABLE ELEMENTS

nt[1] = a

nt[2] = b

nt[3] = c

nt[4] = d

nt[5] =

nt[6] =

nt[7] =

AFTER ASSINGNING VALUES TO EXTENDED ELEMENTS, NESTED TABLE ELEMENTS

nt[1] = a

nt[2] = b

nt[3] = c

nt[4] = d

nt[5] = e

nt[6] = f

nt[7] = g

After extend of five indexes, Count = 12

NESTED TABLE ELEMENTS

nt[1] = a

nt[2] = b

nt[3] = c

nt[4] = d

nt[5] = e

nt[6] = f

nt[7] = g

nt[8] = b

nt[9] = b

nt[10] = b

nt[11] = b

nt[12] = b

After trim of one index, Count = 11

After trim of three indexs, Count = 8

AFTER TRIM, NESTED TABLE ELEMENTS

nt[1] = a

nt[2] = b

nt[3] = c

nt[4] = d

nt[5] = e

nt[6] = f

nt[7] = g

nt[8] = b

After delete of first index, Count = 7

NESTED TABLE ELEMENTS

nt[2] = b

nt[3] = c

nt[4] = d

nt[5] = e

nt[6] = f

nt[7] = g

nt[8] = b

After delete of fourth index, Count = 6

NESTED TABLE ELEMENTS

nt[2] = b

nt[3] = c

nt[5] = e

nt[6] = f

nt[7] = g

nt[8] = b

After delete of entire nested table, Count = 0

Ex2:

```
DECLARE
    type t is table of student%rowtype;
    nt t := t(null,null,null,null);
BEGIN
    for i in 1..nt.count loop
        select * into nt(i) from student where sno = i;
       dbms_output.put_line('Sno  =  '  ||  nt(i).sno  ||  '  Sname  =  '  ||
nt(i).sname);
    end loop;
END;
```

Output:

```
Sno = 1 Sname = saketh
Sno = 2 Sname = srinu
Sno = 3 Sname = divya
Sno = 4 Sname = manogni
```

Ex3:

```
DECLARE
     type t is table of student.smarks%type;
     nt t := t(null,null,null,null);
BEGIN
     for i in 1..nt.count loop
         select smarks into nt(i) from student where sno = i;
         dbms_output.put_line('Smarks  = ' || nt(i));
     end loop;
END;
```

Output:

```
Smarks = 100
Smarks = 200
Smarks = 300
Smarks = 400
```

Ex4:

```
DECLARE
    type r is record(c1 student.sname%type,c2 student.smarks%type);
    type t is table of r;
    nt t := t(null,null,null,null);
BEGIN
    for i in 1..nt.count loop
        select sname,smarks into nt(i) from student where sno = i;
       dbms_output.put_line('Sname = ' || nt(i).c1 || ' Smarks = ' ||
nt(i).c2);
    end loop;
END;
```

Output:

```
Sname = saketh Smarks = 100
Sname = srinu Smarks = 200
Sname = divya Smarks = 300
Sname = manogni Smarks = 400
```

Ex5:

```
DECLARE
    type t is table of addr;
    nt t := t(null);
    cursor c is select * from employ;
    i number := 1;
BEGIN
    for v in c loop
        select address into nt(i) from employ where ename = v.ename;
        dbms_output.put_line('Hno = ' || nt(i).hno || ' City = ' || nt(i).city);
    end loop;
END;
```

Output:

```
Hno = 11 City = hyd
Hno = 22 City = bang
```

Hno = 33 City = kochi

Ex6:

```
DECLARE
    type t is varray(5) of varchar(2);
    nt1 t;
    nt2 t := t();
BEGIN
    if nt1 is null then
      dbms_output.put_line('nt1  is null');
    else
      dbms_output.put_line('nt1  is not null');
    end if;
    if nt2 is null then
      dbms_output.put_line('nt2  is null');
    else
      dbms_output.put_line('nt2  is not null');
    end if;
END;
```

Output:

nt1 is null

nt2 is not null

SET OPERATIONS IN NESTED TABLES

You can perform set operations in the nested tables. You can also perform equality comparisions between nested tables.

Possible operations are

1. UNION
2. UNION DISTINCT
3. INTERSECT
4. EXCEPT (act like MINUS)

Ex:

```
DECLARE
    type t is table of varchar(2);
    nt1 t := t('a','b','c');
    nt2 t := t('c','b','a');
    nt3 t := t('b','c','a','c');
    nt4 t := t('a','b','d');
    nt5 t;
BEGIN
    nt5 := set(nt1);
    dbms_output.put_line('NESTED TABLE ELEMENTS');
    for i in nt5.first..nt5.last loop
        dbms_output.put_line('nt5[ ' || i || ' ] = ' || nt5(i));
    end loop;
    nt5 := set(nt3);
   dbms_output.put_line('NESTED TABLE ELEMENTS');
   for i in nt5.first..nt5.last loop
      dbms_output.put_line('nt5[ ' || i || ' ] = ' || nt5(i));
   end loop;
   nt5 := nt1 multiset union nt4;
   dbms_output.put_line('NESTED TABLE ELEMENTS');
   for i in nt5.first..nt5.last loop
      dbms_output.put_line('nt5[ ' || i || ' ] = ' || nt5(i));
   end loop;
   nt5 := nt1 multiset union nt3;
   dbms_output.put_line('NESTED TABLE ELEMENTS');
   for i in nt5.first..nt5.last loop
      dbms_output.put_line('nt5[ ' || i || ' ] = ' || nt5(i));
   end loop;
   nt5 := nt1 multiset union distinct nt3;
   dbms_output.put_line('NESTED TABLE ELEMENTS');
   for i in nt5.first..nt5.last loop
      dbms_output.put_line('nt5[ ' || i || ' ] = ' || nt5(i));
   end loop;
   nt5 := nt1 multiset except nt4;
   dbms_output.put_line('NESTED TABLE ELEMENTS');
```

```
    for i in nt5.first..nt5.last loop
        dbms_output.put_line('nt5[ ' || i || ' ] = ' || nt5(i));
    end loop;
    nt5 := nt4 multiset except nt1;
    dbms_output.put_line('NESTED TABLE ELEMENTS');
    for i in nt5.first..nt5.last loop
        dbms_output.put_line('nt5[ ' || i || ' ] = ' || nt5(i));
    end loop;
END;
```

Output:

```
NESTED TABLE ELEMENTS
nt5[ 1 ] = a
nt5[ 2 ] = b
nt5[ 3 ] = c
NESTED TABLE ELEMENTS
nt5[ 1 ] = b
nt5[ 2 ] = c
nt5[ 3 ] = a
NESTED TABLE ELEMENTS
nt5[ 1 ] = a
nt5[ 2 ] = b
nt5[ 3 ] = c
nt5[ 4 ] = a
nt5[ 5 ] = b
nt5[ 6 ] = d
NESTED TABLE ELEMENTS
nt5[ 1 ] = a
nt5[ 2 ] = b
nt5[ 3 ] = c
nt5[ 4 ] = b
nt5[ 5 ] = c
nt5[ 6 ] = a
nt5[ 7 ] = c
NESTED TABLE ELEMENTS
```

```
nt5[ 1 ] = a
nt5[ 2 ] = b
nt5[ 3 ] = c
NESTED TABLE ELEMENTS
nt5[ 1 ] = c
NESTED TABLE ELEMENTS
nt5[ 1 ] = d
```

INDEX-BY TABLES

An index-by table has no limit on its size. Elements are inserted into index-by table whose index may start non-sequentially including negative integers.

Syntax:

Type <*type_name*> is table of <*table_type*> index by binary_integer;

Ex:

```
DECLARE
    type t is table of varchar(2) index by binary_integer;
    ibt t;
    flag boolean;
BEGIN
    ibt(1) := 'a';
    ibt(-20) := 'b';
    ibt(30) := 'c';
    ibt(100) := 'd';
    if ibt.limit is null then
      dbms_output.put_line('No  limit to Index by Tables');
    else
      dbms_output.put_line('Limit  = ' || ibt.limit);
    end if;
    dbms_output.put_line('Count  = ' || ibt.count);
    dbms_output.put_line('First  Index = ' || ibt.first);
    dbms_output.put_line('Last  Index = ' || ibt.last);
    dbms_output.put_line('Next  Index = ' || ibt.next(2));
    dbms_output.put_line('Previous  Index = ' || ibt.prior(3));
```

```
        dbms_output.put_line('INDEX BY TABLE ELEMENTS');
        dbms_output.put_line('ibt[-20] = ' || ibt(-20));
        dbms_output.put_line('ibt[1] = ' || ibt(1));
        dbms_output.put_line('ibt[30] = ' || ibt(30));
        dbms_output.put_line('ibt[100] = ' || ibt(100));
        flag := ibt.exists(30);
        if flag = true then
          dbms_output.put_line('Index 30 exists with an element ' || ibt(30));
        else
          dbms_output.put_line('Index 30 does not exists');
        end if;
        flag := ibt.exists(50);
        if flag = true then
          dbms_output.put_line('Index 50 exists with an element ' || ibt(30));
        else
          dbms_output.put_line('Index 50 does not exists');
        end if;
        ibt.delete(1);
       dbms_output.put_line('After delete of first index, Count = ' ||
    ibt.count);
        ibt.delete(30);
       dbms_output.put_line('After delete of index thirty, Count = ' ||
    ibt.count);
        dbms_output.put_line('INDEX BY TABLE ELEMENTS');
        dbms_output.put_line('ibt[-20] = ' || ibt(-20));
        dbms_output.put_line('ibt[100] = ' || ibt(100));
        ibt.delete;
       dbms_output.put_line('After delete of entire index-by table, Count = '
    ||
                                        ibt.count);
    END;
```

Output:

```
No limit to Index by Tables
Count = 4
```

First Index = -20

Last Index = 100

Next Index = 30

Previous Index = 1

INDEX BY TABLE ELEMENTS

ibt[-20] = b

ibt[1] = a

ibt[30] = c

ibt[100] = d

Index 30 exists with an element c

Index 50 does not exists

After delete of first index, Count = 3

After delete of index thirty, Count = 2

INDEX BY TABLE ELEMENTS

ibt[-20] = b

ibt[100] = d

After delete of entire index-by table, Count = 0

DIFFERENCES AMONG COLLECTIONS

1. **Varrays has limit, nested tables and index-by tables has no limit.**
2. **Varrays and nested tables must be initialized before assignment of elements, in index-by tables we can directly assign elements.**
3. **Varrays and nested tables stored in database, but index-by tables can not.**
4. **Nested tables and index-by tables are PL/SQL tables, but varrays can not.**
5. **Keys must be positive in case of nested tables and varrays, in case of index-by tables keys can be positive or negative.**
6. **Referencing nonexistent elements raises SUBSCRIPT_BEYOND_COUNT in both nested tables and varrays, but in case of index-by tables NO_DATA_FOUND raises.**
7. **Keys are sequential in both nested tables and varrays, non-sequential in index-by tables.**
8. **Individual indexes can be deleted in both nested tables and index-by tables, but in varrays can not.**
9. **Individual indexes can be trimmed in both nested tables and varrays, but in**

index-by tables can not.

10 Individual indexes can be extended in both nested tables and varrays, but in index-by tables can not.

MULTILEVEL COLLECTIONS

Collections of more than one dimension which is a collection of collections, known as multilevel collections.

Syntax:

Type <*type_name1*> is table of <*table_type*> index by binary_integer;

Type <*type_name2*> is varray(<*limit*>) | table | of <*type_name1*> | index by

binary_integer;

Ex1:

```
DECLARE
    type t1 is table of varchar(2) index by binary_integer;
    type t2 is varray(5) of t1;
    va t2 := t2();
    c number := 97;
    flag boolean;
BEGIN
    va.extend(4);
    dbms_output.put_line('Count  = ' || va.count);
    dbms_output.put_line('Limit  = ' || va.limit);
    for i in 1..va.count loop
        for j in 1..va.count loop
            va(i)(j) := chr(c);
            c := c + 1;
        end loop;
    end loop;
    dbms_output.put_line('VARRAY ELEMENTS');
    for i in 1..va.count loop
        for j in 1..va.count loop
            dbms_output.put_line('va[' || i || '][' || j || '] = ' || va(i)(j));
```

```
            end loop;
        end loop;
        dbms_output.put_line('First  index = ' || va.first);
        dbms_output.put_line('Last  index = ' || va.last);
        dbms_output.put_line('Next  index = ' || va.next(2));
        dbms_output.put_line('Previous  index = ' || va.prior(3));
        flag := va.exists(2);
        if flag = true then
           dbms_output.put_line('Index  2 exists');
        else
           dbms_output.put_line('Index  2 exists');
        end if;
        va.extend;
        va(1)(5) := 'q';
        va(2)(5) := 'r';
        va(3)(5) := 's';
        va(4)(5) := 't';
        va(5)(1) := 'u';
        va(5)(2) := 'v';
        va(5)(3) := 'w';
        va(5)(4) := 'x';
        va(5)(5) := 'y';
       dbms_output.put_line('After  extend  of  one  index,  Count  =  ' ||
va.count);
        dbms_output.put_line('VARRAY ELEMENTS');
        for i in 1..va.count loop
            for j in 1..va.count loop
                dbms_output.put_line('va[' || i || '][' || j || '] = ' || va(i)(j));
            end loop;
        end loop;
        va.trim;
        dbms_output.put_line('After  trim of one index, Count = ' || va.count);
        va.trim(2);
       dbms_output.put_line('After  trim  of  two  indexes,  Count  =  ' ||
va.count);
```

```
        dbms_output.put_line('VARRAY ELEMENTS');
        for i in 1..va.count loop
            for j in 1..va.count loop
                dbms_output.put_line('va[' || i || '][' || j || '] = ' || va(i)(j));
            end loop;
        end loop;
        va.delete;
       dbms_output.put_line('After delete of entire varray, Count = ' ||
    va.count);
    END;
```

Output:

```
Count = 4
Limit = 5
VARRAY ELEMENTS
va[1][1] = a
va[1][2] = b
va[1][3] = c
va[1][4] = d
va[2][1] = e
va[2][2] = f
va[2][3] = g
va[2][4] = h
va[3][1] = i
va[3][2] = j
va[3][3] = k
va[3][4] = l
va[4][1] = m
va[4][2] = n
va[4][3] = o
va[4][4] = p
First index = 1
Last index = 4
Next index = 3
Previous index = 2
```

Index 2 exists

After extend of one index, Count = 5

VARRAY ELEMENTS

va[1][1] = a

va[1][2] = b

va[1][3] = c

va[1][4] = d

va[1][5] = q

va[2][1] = e

va[2][2] = f

va[2][3] = g

va[2][4] = h

va[2][5] = r

va[3][1] = i

va[3][2] = j

va[3][3] = k

va[3][4] = l

va[3][5] = s

va[4][1] = m

va[4][2] = n

va[4][3] = o

va[4][4] = p

va[4][5] = t

va[5][1] = u

va[5][2] = v

va[5][3] = w

va[5][4] = x

va[5][5] = y

After trim of one index, Count = 4

After trim of two indexes, Count = 2

VARRAY ELEMENTS

va[1][1] = a

va[1][2] = b

va[2][1] = e

va[2][2] = f

```
After delete of entire varray, Count = 0
```

Ex2:

```
DECLARE
    type t1 is table of varchar(2) index by binary_integer;
    type t2 is table of t1;
    nt t2 := t2();
    c number := 65;
    v number := 1;
    flag boolean;
BEGIN
    nt.extend(4);
    dbms_output.put_line('Count  = ' || nt.count);
    if nt.limit is null then
       dbms_output.put_line('No  limit to Nested Tables');
    else
       dbms_output.put_line('Limit  = ' || nt.limit);
    end if;
    for i in 1..nt.count loop
        for j in 1..nt.count loop
            nt(i)(j) := chr(c);
            c := c + 1;
            if c = 91 then
              c := 97;
            end if;
        end loop;
    end loop;
    dbms_output.put_line('NESTED TABLE ELEMENTS');
    for i in 1..nt.count loop
        for j in 1..nt.count loop
            dbms_output.put_line('nt[' || i || '][' || j || '] = ' || nt(i)(j));
        end loop;
    end loop;
    dbms_output.put_line('First  index = ' || nt.first);
    dbms_output.put_line('Last  index = ' || nt.last);
    dbms_output.put_line('Next  index = ' || nt.next(2));
```

```
        dbms_output.put_line('Previous  index = ' || nt.prior(3));
        flag := nt.exists(2);
        if flag = true then
           dbms_output.put_line('Index  2 exists');
        else
           dbms_output.put_line('Index  2 exists');
        end if;
        nt.extend(2);
        nt(1)(5) := 'Q';
        nt(1)(6) := 'R';
        nt(2)(5) := 'S';
        nt(2)(6) := 'T';
        nt(3)(5) := 'U';
        nt(3)(6) := 'V';
        nt(4)(5) := 'W';
        nt(4)(6) := 'X';
        nt(5)(1) := 'Y';
        nt(5)(2) := 'Z';
        nt(5)(3) := 'a';
        nt(5)(4) := 'b';
        nt(5)(5) := 'c';
        nt(5)(6) := 'd';
        nt(6)(1) := 'e';
        nt(6)(2) := 'f';
        nt(6)(3) := 'g';
        nt(6)(4) := 'h';
        nt(6)(5) := 'i';
        nt(6)(6) := 'j';
       dbms_output.put_line('After  extend  of  one  index,  Count  =  ' ||
nt.count);
        dbms_output.put_line('NESTED TABLE ELEMENTS');
        for i in 1..nt.count loop
            for j in 1..nt.count loop
                dbms_output.put_line('nt[' || i || '][' || j || '] = ' || nt(i)(j));
            end loop;
```

```
        end loop;
        nt.trim;
        dbms_output.put_line('After  trim of one indexe, Count = ' || nt.count);
        nt.trim(2);
       dbms_output.put_line('After  trim  of  two  indexes,  Count  =  ' ||
nt.count);
        dbms_output.put_line('NESTED TABLE ELEMENTS');
        for i in 1..nt.count loop
            for j in 1..nt.count loop
               dbms_output.put_line('nt['  || i || ']['  || j || '] = ' || nt(i)(j));
            end loop;
        end loop;
        nt.delete(2);
       dbms_output.put_line('After  delete  of  second  index,  Count  =  ' ||
nt.count);
        dbms_output.put_line('NESTED TABLE ELEMENTS');
        loop
            exit when v = 4;
            for j in 1..nt.count+1  loop
                dbms_output.put_line('nt['  || v || ']['  || j || '] = ' || nt(v)(j));
            end loop;
            v := v + 1;
            if v= 2 then
              v := 3;
            end if;
        end loop;
        nt.delete;
        dbms_output.put_line('After  delete of entire nested table, Count = ' ||
                                        nt.count);
END;
```

Output:

```
Count = 4
No limit to Nested Tables
```

NESTED TABLE ELEMENTS

nt[1][1] = A

nt[1][2] = B

nt[1][3] = C

nt[1][4] = D

nt[2][1] = E

nt[2][2] = F

nt[2][3] = G

nt[2][4] = H

nt[3][1] = I

nt[3][2] = J

nt[3][3] = K

nt[3][4] = L

nt[4][1] = M

nt[4][2] = N

nt[4][3] = O

nt[4][4] = P

First index = 1

Last index = 4

Next index = 3

Previous index = 2

Index 2 exists

After extend of one index, Count = 6

NESTED TABLE ELEMENTS

nt[1][1] = A

nt[1][2] = B

nt[1][3] = C

nt[1][4] = D

nt[1][5] = Q

nt[1][6] = R

nt[2][1] = E

nt[2][2] = F

nt[2][3] = G

nt[2][4] = H

nt[2][5] = S

nt[2][6] = T
nt[3][1] = I
nt[3][2] = J
nt[3][3] = K
nt[3][4] = L
nt[3][5] = U
nt[3][6] = V
nt[4][1] = M
nt[4][2] = N
nt[4][3] = O
nt[4][4] = P
nt[4][5] = W
nt[4][6] = X
nt[5][1] = Y
nt[5][2] = Z
nt[5][3] = a
nt[5][4] = b
nt[5][5] = c
nt[5][6] = d
nt[6][1] = e
nt[6][2] = f
nt[6][3] = g
nt[6][4] = h
nt[6][5] = i
nt[6][6] = j
After trim of one indexe, Count = 5
After trim of two indexes, Count = 3
NESTED TABLE ELEMENTS
nt[1][1] = A
nt[1][2] = B
nt[1][3] = C
nt[2][1] = E
nt[2][2] = F
nt[2][3] = G
nt[3][1] = I

```
nt[3][2] = J
nt[3][3] = K
After delete of second index, Count = 2
NESTED TABLE ELEMENTS
nt[1][1] = A
nt[1][2] = B
nt[1][3] = C
nt[3][1] = I
nt[3][2] = J
nt[3][3] = K
After delete of entire nested table, Count = 0
```

Ex3:

```
DECLARE
    type t1 is table of varchar(2) index by binary_integer;
    type t2 is table of t1 index by binary_integer;
    ibt t2;
    flag boolean;
BEGIN
    dbms_output.put_line('Count = ' || ibt.count);
    if ibt.limit is null then
       dbms_output.put_line('No limit to Index-by Tables');
    else
       dbms_output.put_line('Limit = ' || ibt.limit);
    end if;
    ibt(1)(1) := 'a';
    ibt(4)(5) := 'b';
    ibt(5)(1) := 'c';
    ibt(6)(2) := 'd';
    ibt(8)(3) := 'e';
    ibt(3)(4) := 'f';
    dbms_output.put_line('INDEX-BY TABLE ELEMENTS');
    dbms_output.put_line('ibt([1][1] = ' || ibt(1)(1));
    dbms_output.put_line('ibt([4][5] = ' || ibt(4)(5));
    dbms_output.put_line('ibt([5][1] = ' || ibt(5)(1));
    dbms_output.put_line('ibt([6][2] = ' || ibt(6)(2));
```

```
dbms_output.put_line('ibt([8][3] = ' || ibt(8)(3));
dbms_output.put_line('ibt([3][4] = ' || ibt(3)(4));
dbms_output.put_line('First Index = ' || ibt.first);
dbms_output.put_line('Last Index = ' || ibt.last);
dbms_output.put_line('Next Index = ' || ibt.next(3));
dbms_output.put_line('Prior Index = ' || ibt.prior(8));
ibt(1)(2) := 'g';
ibt(1)(3) := 'h';
ibt(1)(4) := 'i';
ibt(1)(5) := 'k';
ibt(1)(6) := 'l';
ibt(1)(7) := 'm';
ibt(1)(8) := 'n';
dbms_output.put_line('Count = ' || ibt.count);
dbms_output.put_line('INDEX-BY TABLE ELEMENTS');
for i in 1..8 loop
    dbms_output.put_line('ibt[1][' || i || '] = ' || ibt(1)(i));
end loop;
dbms_output.put_line('ibt([4][5] = ' || ibt(4)(5));
dbms_output.put_line('ibt([5][1] = ' || ibt(5)(1));
dbms_output.put_line('ibt([6][2] = ' || ibt(6)(2));
dbms_output.put_line('ibt([8][3] = ' || ibt(8)(3));
dbms_output.put_line('ibt([3][4] = ' || ibt(3)(4));
flag := ibt.exists(3);
if flag = true then
   dbms_output.put_line('Index 3 exists');
else
   dbms_output.put_line('Index 3 exists');
end if;
ibt.delete(1);
dbms_output.put_line('After delete of first index, Count = ' ||
ibt.count);
ibt.delete(4);
dbms_output.put_line('After delete of fourth index, Count = ' ||
ibt.count);
```

```
        dbms_output.put_line('INDEX-BY TABLE ELEMENTS');
        dbms_output.put_line('ibt([5][1] = ' || ibt(5)(1));
        dbms_output.put_line('ibt([6][2] = ' || ibt(6)(2));
        dbms_output.put_line('ibt([8][3] = ' || ibt(8)(3));
        dbms_output.put_line('ibt([3][4] = ' || ibt(3)(4));
        ibt.delete;
       dbms_output.put_line('After delete of entire index-by table, Count = '
||
                                        ibt.count);
END;
```

Output:

```
Count = 0
No limit to Index-by Tables
INDEX-BY TABLE ELEMENTS
ibt([1][1] = a
ibt([4][5] = b
ibt([5][1] = c
ibt([6][2] = d
ibt([8][3] = e
ibt([3][4] = f
First Index = 1
Last Index = 8
Next Index = 4
Prior Index = 6
Count = 6
INDEX-BY TABLE ELEMENTS
ibt[1][1] = a
ibt[1][2] = g
ibt[1][3] = h
ibt[1][4] = i
ibt[1][5] = k
ibt[1][6] = l
ibt[1][7] = m
ibt[1][8] = n
ibt([4][5] = b
```

```
ibt([5][1]  = c
ibt([6][2]  = d
ibt([8][3]  = e
ibt([3][4]  = f
Index 3 exists
After delete of first index, Count = 5
After delete of fourth index, Count = 4
INDEX-BY TABLE ELEMENTS
ibt([5][1]  = c
ibt([6][2]  = d
ibt([8][3]  = e
ibt([3][4]  = f
After delete of entire index-by table, Count = 0
```

Ex4:

```
DECLARE
    type t1 is table of varchar(2) index by binary_integer;
    type t2 is table of t1 index by binary_integer;
    type t3 is table of t2;
    nt t3 := t3();
    c number := 65;
BEGIN
    nt.extend(2);
    dbms_output.put_line('Count  = ' || nt.count);
    for i in 1..nt.count loop
        for j in 1..nt.count loop
            for k in 1..nt.count loop
                nt(i)(j)(k)  := chr(c);
                c := c + 1;
            end loop;
        end loop;
    end loop;
    dbms_output.put_line('NESTED TABLE ELEMENTS');
    for i in 1..nt.count loop
        for j in 1..nt.count loop
            for k in 1..nt.count loop
```

```
                    dbms_output.put_line('nt[' || i || '][' || j || '][' || k || '] = '
        ||
                                                        nt(i)(j)(k));
                end loop;
              end loop;
            end loop;
        END;
```

Output:

```
Count = 2
NESTED TABLE ELEMENTS
nt[1][1][1] = A
nt[1][1][2] = B
nt[1][2][1] = C
nt[1][2][2] = D
nt[2][1][1] = E
nt[2][1][2] = F
nt[2][2][1] = G
nt[2][2][2] = H
```

OBJECTS USED IN THE EXAMPLES

```
SQL> select * from student;

      SNO SNAME       SMARKS
--------- ---------- ----------
        1 saketh        100
        2 srinu         200
        3 divya         300
        4 manogni       400

SQL> create or replace type addr as object(hno number(2),city
varchar(10));/
```

```
SQL> select * from employ;

ENAME      JOB        ADDRESS(HNO,  CITY)
---------- ---------- ----------------------------
Ranjit     clerk          ADDR(11,  'hyd')
Satish     manager    ADDR(22,  'bang')
Srinu      engineer     ADDR(33,  'kochi')
```

ERROR HANDLING

PL/SQL implements error handling with exceptions and exception handlers. Exceptions can be associated with oracle errors or with your own user-defined errors. By using exceptions and exception handlers, you can make your PL/SQL programs robust and able to deal with both unexpected and expected errors during execution.

ERROR TYPES

1. **Compile-time errors**
2. **Runtime errors**

Errors that occur during the compilation phase are detected by the PL/SQL engine and reported back to the user, we have to correct them.

Runtime errors are detected by the PL/SQL runtime engine which can programmatically raise and caught by exception handlers.
Exceptions are designed for run-time error handling, rather than compile-time error handling.

HANDLING EXCEPTIONS

When exception is raised, control passes to the exception section of the block. The exception section consists of handlers for some or all of the exceptions. An exception handler contains the code that is executed when the error associated with the exception occurs, and the exception is raised.

Syntax:

```
EXCEPTION
        When exception_name then
                Sequence_of_statements;
        When exception_name then
                Sequence_of_statements;
        When others then
```

```
                Sequence_of_statements;
        END;
```

EXCEPTION TYPES

3 Predefined exceptions

4 User-defined exceptions

PREDEFINED EXCEPTIONS

Oracle has predefined several exceptions that corresponds to the most common oracle errors. Like the predefined types, the identifiers of these exceptions are defined in the STANDARD package. Because of this, they are already available to the program, it is not necessary to declare them in the declarative secion.

Ex1:

```
DECLARE
    a number;
    b varchar(2);
    v_marks number;
    cursor c is select * from student;
    type t is varray(3) of varchar(2);
    va t := t('a','b');
    va1 t;
BEGIN
    -- NO_DATA_FOUND
    BEGIN
        select smarks into v_marks from student where sno = 50;
    EXCEPTION
        when no_data_found then
            dbms_output.put_line('Invalid student number');
    END;
    -- CURSOR_ALREADY_OPEN
    BEGIN
        open c;
        open c;
    EXCEPTION
        when cursor_already_open then
```

```
            dbms_output.put_line('Cursor  is already opened');
END;

-- INVALID_CURSOR
BEGIN
    close c;
    open c;
    close c;
    close c;
EXCEPTION
    when invalid_cursor  then
            dbms_output.put_line('Cursor  is already closed');
END;
-- TOO_MANY_ROWS
BEGIN
    select smarks into v_marks from student where sno > 1;
EXCEPTION
    when too_many_rows  then
            dbms_output.put_line('Too  many values are coming to marks
                                        variable');
END;
-- ZERO_DIVIDE
BEGIN
     a := 5/0;
EXCEPTION
     when zero_divide then
             dbms_output.put_line('Divided  by zero - invalid operation');
END;
-- VALUE_ERROR
BEGIN
     b := 'saketh';
EXCEPTION
     when value_error then
             dbms_output.put_line('Invalid  string length');
END;
-- INVALID_NUMBER
BEGIN
     insert into student values('a','srinu',100);
EXCEPTION
     when invalid_number  then
```

```
                dbms_output.put_line('Invalid  number');
        END;

        -- SUBSCRIPT_OUTSIDE_LIMIT
        BEGIN
            va(4) := 'c';
        EXCEPTION
            when subscript_outside_limit  then
                dbms_output.put_line('Index  is greater than the limit');
        END;
        -- SUBSCRIPT_BEYOND_COUNT
        BEGIN
            va(3) := 'c';
        EXCEPTION
            when subscript_beyond_count  then
                dbms_output.put_line('Index  is greater than the count');
        END;
        -- COLLECTION_IS_NULL
        BEGIN
            va1(1) := 'a';
        EXCEPTION
            when collection_is_null  then
                dbms_output.put_line('Collection  is empty');
        END;
        --
    END;
```

Output:

```
Invalid student number
Cursor is already opened
Cursor is already closed
Too many values are coming to marks variable
Divided by zero - invalid operation
Invalid string length
Invalid number
Index is greater than the limit
Index is greater than the count
Collection is empty
```

Ex2:

```
DECLARE
      c number;
BEGIN
       c := 5/0;
EXCEPTION
       when zero_divide then
              dbms_output.put_line('Invalid  Operation');
       when others then
              dbms_output.put_line('From  OTHERS handler: Invalid
                                                  Operation');
END;
```

Output:

```
Invalid Operation
```

USER-DEFINED EXCEPTIONS

A user-defined exception is an error that is defined by the programmer. User-defined exceptions are declared in the declarative secion of a PL/SQL block. Just like variables, exeptions have a type EXCEPTION and scope.

RAISING EXCEPTIONS

User-defined exceptions are raised explicitly via the RAISE statement.

Ex:

```
DECLARE
     e exception;
BEGIN
     raise e;
EXCEPTION
     when e then
            dbms_output.put_line('e  is raised');
```

```
END;
```

Output:

```
e is raised
```

BULIT-IN ERROR FUNCTIONS

SQLCODE AND SQLERRM

1. SQLCODE returns the current error code, and SQLERRM returns the current error message text;
2. For user-defined exception SQLCODE returns 1 and SQLERRM returns "user-deifned exception".
3. SQLERRM wiil take only negative value except 100. If any positive value other than 100 returns non-oracle exception.

Ex1:

```
DECLARE
      e exception;
      v_dname varchar(10);
BEGIN
      -- USER-DEFINED EXCEPTION
      BEGIN
          raise e;
      EXCEPTION
          when e then
                dbms_output.put_line(SQLCODE || ' ' || SQLERRM);
      END;

      -- PREDEFINED EXCEPTION
      BEGIN
          select dname into v_dname from dept where deptno = 50;
      EXCEPTION
          when no_data_found then
                dbms_output.put_line(SQLCODE || ' ' || SQLERRM);
      END;
END;
```

Output:

```
1 User-Defined Exception
```

```
100 ORA-01403: no data found
```

Ex2:

```
BEGIN
    dbms_output.put_line(SQLERRM(100));
    dbms_output.put_line(SQLERRM(0));
    dbms_output.put_line(SQLERRM(1));
    dbms_output.put_line(SQLERRM(-100));
    dbms_output.put_line(SQLERRM(-500));
    dbms_output.put_line(SQLERRM(200));
    dbms_output.put_line(SQLERRM(-900));
END;
```

Output:

```
ORA-01403: no data found
ORA-0000: normal, successful completion
User-Defined Exception
ORA-00100: no data found
ORA-00500: Message 500 not found; product=RDBMS; facility=ORA
-200: non-ORACLE exception
ORA-00900: invalid SQL statement
```

DBMS_UTILITY.FORMAT_ERROR_STACK

1. The built-in function, like SQLERRM, returns the message associated with the current error.
2. It differs from SQLERRM in two ways:
3. Its length is not restricted; it will return the full error message string.
4. You can not pass an error code number to this function; it cannot be used to return the message for a random error code.

Ex:

```
DECLARE
   v number := 'ab';
BEGIN
   null;
EXCEPTION
```

```
        when others then
            dbms_output.put_line(dbms_utility.format_error_stack);
    END;
```

Output:

```
    declare
    *
    ERROR at line 1:
    ORA-06502: PL/SQL: numeric or value error: character to number
    conversion error
    ORA-06512: at line 2
```

DBMS_UTILITY.FORMAT_CALL_STACK

This function returns a formatted string showing the execution call stack inside your PL/SQL application. Its usefulness is not restricted to error management; you will also find its handy for tracing the exectution of your code. You may not use this function in exception block.

Ex:

```
    BEGIN
        dbms_output.put_line(dbms_utility.format_call_stack);
    END;
```

Output:

```
    ----- PL/SQL Call Stack -----
      Object_handle      line_number   object_name
        69760478              2         anonymous block
```

DBMS_UTILITY.FORMAT_ERROR_BACKTRACE

It displays the execution stack at the point where an exception was raised. Thus , you can call this function with an exception section at the top level of your stack and still find out where the error was raised deep within the call stack.

Ex:

```
    CREATE OR REPLACE PROCEDURE P1 IS
    BEGIN
```

```
    dbms_output.put_line('from  procedure  1');
    raise value_error;
END P1;
CREATE OR REPLACE PROCEDURE P2 IS
BEGIN
    dbms_output.put_line('from  procedure  2');
    p1;
END P2;

CREATE OR REPLACE PROCEDURE P3 IS
BEGIN
    dbms_output.put_line('from  procedure  3');
    p2;
EXCEPTION
    when others then

dbms_output.put_line(dbms_utility.format_error_backtrace);
END P3;
```

Output:

```
SQL> exec p3

from procedure 3
from procedure 2
from procedure 1
ORA-06512:  at "SAKETH.P1",  line 4
ORA-06512:  at "SAKETH.P2",  line 4
ORA-06512:  at "SAKETH.P3",  line 4
```

EXCEPTION_INIT PRAGMA

Using this you can associate a named exception with a particular oracle error. This gives you the ability to trap this error specifically, rather than via an OTHERS handler.

Syntax:

PRAGMA EXCEPTION_INIT(*exception_name*, *oracle_error_number*);

Ex:

```
DECLARE
    e exception;
    pragma exception_init(e,-1476);
    c number;
BEGIN
    c := 5/0;
EXCEPTION
    when e then
        dbms_output.put_line('Invalid  Operation');
END;
```

Output:

```
Invalid Operation
```

RAISE_APPLICATION_ERROR

You can use this built-in function to create your own error messages, which can be more descriptive than named exceptions.

Syntax:

RAISE_APPLICATION_ERROR(*error_number*, *error_message*,, [*keep_errors_flag*]);

The Boolean parameter *keep_errors_flag* is optional. If it is TRUE, the new error is added to the list of errors already raised. If it is FALSE, which is default, the new error will replace the current list of errors.

Ex:

```
DECLARE
    c number;
BEGIN
    c := 5/0;

EXCEPTION
    when zero_divide then
        raise_application_error(-20222,'Invalid  Operation');
```

```
END;
```

Output:

```
DECLARE
*
ERROR at line 1:
ORA-20222: Invalid Operation
ORA-06512: at line 7
```

EXCEPTION PROPAGATION

Exceptions can occur in the declarative, the executable, or the exception section of a PL/SQL block.

EXCEPTION RAISED IN THE EXECUATABLE SECTION

Exceptions raised in execuatable section can be handled in current block or outer block.

Ex1:

```
DECLARE
    e exception;
BEGIN
    BEGIN
       raise e;
    END;
    EXCEPTION
       when e then
              dbms_output.put_line('e  is raised');
END;
```

Output:

```
e is raised
```

Ex2:

```
DECLARE
    e exception;
BEGIN
```

```
        BEGIN
            raise e;
        END;
    END;
```

Output:

```
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 5
```

EXCEPTION RAISED IN THE DECLARATIVE SECTION

Exceptions raised in the declarative secion must be handled in the outer block.

Ex1:

```
DECLARE
    c number(3) := 'abcd';
BEGIN
    dbms_output.put_line('Hello');
EXCEPTION
    when others then
            dbms_output.put_line('Invalid  string length');
END;
```

Output:

```
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character to number
conversion error
ORA-06512: at line 2
```

Ex2:

```
BEGIN
   DECLARE
        c number(3) := 'abcd';
   BEGIN
        dbms_output.put_line('Hello');
   EXCEPTION
        when others then
```

```
                    dbms_output.put_line('Invalid string length');
    END;
EXCEPTION
    when others then
        dbms_output.put_line('From outer block: Invalid string
length');
END;
```

Output:

```
From outer block: Invalid string length
```

EXCEPTION RAISED IN THE EXCEPTION SECTION

Exceptions raised in the declarative secion must be handled in the outer block.

Ex1:

```
DECLARE
    e1 exception;
    e2 exception;
BEGIN
    raise e1;
EXCEPTION
    when e1 then
        dbms_output.put_line('e1 is raised');
        raise e2;
    when e2 then
        dbms_output.put_line('e2 is raised');
END;
```

Output:

```
e1 is raised
DECLARE
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 9
ORA-06510: PL/SQL: unhandled user-defined exception
```

Ex2:

```
DECLARE
    e1 exception;
    e2 exception;
BEGIN
    BEGIN
        raise e1;
    EXCEPTION
        when e1 then
                dbms_output.put_line('e1  is raised');
                raise e2;
        when e2 then
                dbms_output.put_line('e2  is raised');
     END;
EXCEPTION
     when e2 then
            dbms_output.put_line('From  outer block: e2 is raised');
END;
```

Output:

```
e1 is raised
From outer block: e2 is raised
```

Ex3:

```
DECLARE
     e exception;
BEGIN
     raise e;
EXCEPTION
     when e then
            dbms_output.put_line('e  is raised');
            raise e;
END;
```

Output:

```
e is raised
DECLARE
```

```
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 8
ORA-06510: PL/SQL: unhandled user-defined exception
```

RESTRICTIONS

You can not pass exception as an argument to a subprogram.

DATABASE TRIGGERS

Triggers are similar to procedures or functions in that they are named PL/SQL blocks with declarative, executable, and exception handling sections. A trigger is executed implicitly whenever the triggering event happens. The act of executing a trigger is known as firing the trigger.

RESTRICTIONS ON TRIGGERES

1. **Like packages, triggers must be stored as stand-alone objects in the database and cannot be local to a block or package.**
2. **A trigger does not accept arguments.**

USE OF TRIGGERS

1. **Maintaining complex integrity constraints not possible through declarative constraints enable at table creation.**
2. **Auditing information in a table by recording the changes made and who made them.**
3. **Automatically signaling other programs that action needs to take place when chages are made to a table.**
4. **Perform validation on changes being made to tables.**
5. **Automate maintenance of the database.**

TYPES OF TRIGGERS

1. **DML Triggers**
2. **Instead of Triggers**
3. **DDL Triggers**
4. **System Triggers**
5. **Suspend Triggers**

Buddha Reddy 9440885516 buddhareddy.polepelli@gmail.com

CATEGORIES

Timing -- Before or After
Level -- Row or Statement

Row level trigger fires once for each row affected by the triggering statement. Row level trigger is identified by the FOR EACH ROW clause.

Statement level trigger fires once either before or after the statement.

DML TRIGGER SYNTAX

```
Create or replace trigger <trigger_name>
{Before | after}  {insert or update or delete} on <table_name>
[For each row]
[When (...)]
[Declare]
        -- declaration
Begin
        -- trigger body
[Exception]
        -- exception section
End <trigger_name>;
```

DML TRIGGERS

A DML trigger is fired on an INSERT, UPDATE, or DELETE operation on a database table. It can be fired either before or after the statement executes, and can be fired once per affected row, or once per statement.

The combination of these factors determines the types of the triggers. These are a total of 12 possible types (3 statements * 2 timing * 2 levels).

STATEMENT LEVEL

Statement level trigger fires only once.

Ex:

```
SQL> create table statement_level(count  varchar(50));

    CREATE OR REPLACE TRIGGER STATEMENT_LEVEL_TRIGGER
        after update on student
    BEGIN
        insert into statement_level values('Statement level fired');
    END STATEMENT_LEVEL_TRIGGER;
```

Output:

```
SQL> update student set smarks=500;

      3 rows updated.

SQL> select * from statement_level;

            COUNT
            --------------------------
            Statement level fired
```

ROW LEVEL

Row level trigger fires once for each row affected by the triggering statement.

Ex:

```
SQL> create table row_level(count  varchar(50));

    CREATE OR REPLACE TRIGGER ROW_LEVEL_TRIGGER
        after update on student
    BEGIN
        insert into row_level values('Row level fired');
    END ROW_LEVEL_TRIGGER;
```

Output:

```
SQL> update student set smarks=500;

     3 rows updated.

SQL> select * from statement_level;

            COUNT
            ----------------------------
            Row level fired
            Row level fired
            Row level fired
```

ORDER OF DML TRIGGER FIRING

1. Before statement level
2. Before row level
3. After row level
4. After statement level

Ex:

Suppose we have a follwing table.

```
SQL> select * from student;

NO  NAME    MARKS
----- ------- ----------
 1     a       100
 2     b       200
 3     c       300
 4     d       400
```

```
SQL> create table firing_order(order  varchar(50));

CREATE OR REPLACE TRIGGER BEFORE_STATEMENT
```

```
    before insert on student
BEGIN
    insert into firing_order  values('Before Statement Level');
END BEFORE_STATEMENT;

CREATE OR REPLACE TRIGGER BEFORE_ROW
    before insert on student
    for each row
BEGIN
    insert into firing_order  values('Before Row Level');
END BEFORE_ROW;

CREATE OR REPLACE TRIGGER AFTER_STATEMENT
   after insert on student
BEGIN
   insert into firing_order  values('After Statement Level');
END AFTER_STATEMENT;

CREATE OR REPLACE TRIGGER AFTER_ROW
   after insert on student
   for each row
BEGIN
   insert into firing_order  values('After Row Level');
END AFTER_ROW;
```

Output:

```
SQL> select * from firing_order;

no rows selected

SQL> insert into student values(5,'e',500);

1 row created.

SQL> select * from firing_order;
```

ORDER

Before Statement Level

Before Row Level

After Row Level

After Statement Level

SQL> select * from student;

NO	NAME	MARKS
1	a	100
2	b	200
3	c	300
4	d	400
5	e	500

CORRELATION IDENTIFIERS IN ROW-LEVEL TRIGGERS

Inside the trigger, you can access the data in the row that is currently being processed. This is accomplished through two correlation identifiers - :old and :new.

A *correlation identifier* is a special kind of PL/SQL bind variable. The colon in front of each indicates that they are bind variables, in the sense of host variables used in embedded PL/SQL, and indicates that they are not regular PL/SQL variables. The PL/SQL compiler will treat them as records of type

Triggering_table%ROWTYPE.

Although syntactically they are treated as records, in reality they are not. :old and :new are also known as *pseudorecords*, for this reason.

TRIGGERING STATEMENT	:OLD	:NEW
INSERT	all fields are NULL.	values that will be inserted

When the statement is completed.

UPDATE original values for new values that will be updated

the row before the when the statement is completed.

update.

DELETE original values before all fields are NULL.

the row is deleted.

Ex:

```
SQL> create table marks(no number(2) old_marks number(3),new_marks
                        number(3));

CREATE OR REPLACE TRIGGER OLD_NEW
   before insert or update or delete on student
   for each row
BEGIN
   insert into marks values(:old.no,:old.marks,:new.marks);
END OLD_NEW;
```

Output:

```
SQL> select * from student;

        NO   NAME   MARKS
      ----- ------- ----------
         1      a       100
         2      b       200
         3      c       300
         4      d       400
         5      e       500

SQL> select * from marks;

no rows selected
```

```
SQL> insert into student values(6,'f',600);

1 row created.

SQL> select * from student;

        NO  NAME    MARKS
      ----- -------- ----------
         1  a        100
         2  b        200
         3  c        300
         4  d        400
         5  e        500
         6  f        600

SQL> select * from marks;

        NO  OLD_MARKS   NEW_MARKS
      ----- ---------- ----------
                            600

SQL> update student set marks=555 where no=5;

1 row updated.

SQL> select * from student;

        NO  NAME   MARKS
      ----- ------- ----------
         1  a        100
         2  b        200
         3  c        300
         4  d        400
         5  e        555
         6  f        600
```

```
SQL> select * from marks;

        NO   OLD_MARKS    NEW_MARKS
------ ---------- ----------
                              600
         5          500          555

SQL> delete student where no = 2;

1 row deleted.

SQL> select * from student;

      NO  NAME    MARKS
---- -------- ----------
       1       a        100
       3       c        300
       4       d        400
       5       e        555
       6       f        600

SQL> select * from marks;

      NO  OLD_MARKS   NEW_MARKS
----- ---------- ----------
                          600
       5          500          555
       2          200
```

REFERENCING CLAUSE

If desired, you can use the REFERENCING clause to specify a different name for :old ane :new. This clause is found after the triggering event, before the WHEN clause.

Syntax:

REFERENCING [old as old_name] [new as new_name]

Ex:

```
CREATE OR REPLACE TRIGGER REFERENCE_TRIGGER
    before insert or update or delete on student
    referencing old as old_student new as new_student
    for each row
BEGIN
    insert into marks

values(:old_student.no,:old_student.marks,:new_student.marks);
END REFERENCE_TRIGGER;
```

WHEN CLAUSE

WHEN clause is valid for row-level triggers only. If present, the trigger body will be executed only for those rows that meet the condition specified by the WHEN clause.

Syntax:

WHEN *trigger_condition*;

Where *trigger_condition* is a Boolean expression. It will be evaluated for each row. The *:new* and *:old* records can be referenced inside *trigger_condition* as well, but like REFERENCING, the colon is not used there. The colon is only valid in the trigger body.

Ex:

```
CREATE OR REPLACE TRIGGER WHEN_TRIGGER
    before insert or update or delete on student
    referencing old as old_student new as new_student
    for each row
    when (new_student.marks > 500)
BEGIN
    insert into marks

values(:old_student.no,:old_student.marks,:new_student.marks);
```

```
END WHEN_TRIGGER;
```

TRIGGER PREDICATES

There are three Boolean functions that you can use to determine what the operation is.

The predicates are

1 INSERTING
2 UPDATING
3 DELETING

Ex:

```
SQL> create table predicates(operation varchar(20));

CREATE OR REPLACE TRIGGER PREDICATE_TRIGGER
    before insert or update or delete on student
BEGIN
    if inserting then
      insert into predicates values('Insert');
    elsif updating then
        insert into predicates values('Update');
    elsif deleting then
        insert into predicates values('Delete');
    end if;
END PREDICATE_TRIGGER;
```

Output:

```
SQL> delete student where no=1;

1 row deleted.

SQL> select * from predicates;

MSG
--------------
Delete
```

```
SQL> insert into student values(7,'g',700);

1 row created.

SQL> select * from predicates;

MSG
---------------
Delete
Insert

SQL> update student set marks = 777 where no=7;

1 row updated.

SQL> select * from predicates;

MSG
---------------
Delete
Insert
Update
```

INSTEAD-OF TRIGGERS

Instead-of triggers fire instead of a DML operation. Also, instead-of triggers can be defined only on views. Instead-of triggers are used in two cases:

1. **To allow a view that would otherwise not be modifiable to be modified.**
2. **To modify the columns of a nested table column in a view.**

Ex:

```
SQL> create view emp_dept as select empno,ename,job,dname,loc,sal,e.deptno
from
```

```
        emp e, dept d where e.deptno = d.deptno;

CREATE OR REPLACE TRIGGER INSTEAD_OF_TRIGGER
    instead of insert on emp_dept
BEGIN
    insert into dept1 values(50,'rd','bang');
    insert into

emp1(empno,ename,job,sal,deptno)values(2222,'saketh','doctor',8000,50);
END INSTEAD_OF_TRIGGER;
```

Output:

```
SQL> insert into emp_dept values(2222,'saketh','doctor',8000,'rd','bang',50);
SQL> select * from emp_dept;
```

EMPNO	ENAME	JOB	SAL	DNAME	LOC	DEPTNO
7369	SMITH	CLERK	800	RESEARCH	DALLAS	20
7499	ALLEN	SALESMAN	1600	SALES	CHICAGO	30
7521	WARD	SALESMAN	1250	SALES	CHICAGO	30
7566	JONES	MANAGER	2975	RESEARCH	DALLAS	20
7654	MARTIN	SALESMAN	1250	SALES	CHICAGO	30
7698	BLAKE	MANAGER	2850	SALES	CHICAGO	30
7782	CLARK	MANAGER	2450	ACCOUNTING	NEW YORK	10
7788	SCOTT	ANALYST	3000	RESEARCH	DALLAS	20
7839	KING	PRESIDENT	5000	ACCOUNTING	NEW YORK	10
7844	TURNER	SALESMAN	1500	SALES	CHICAGO	30
7876	ADAMS	CLERK	1100	RESEARCH	DALLAS	20
7900	JAMES	CLERK	950	SALES	CHICAGO	30
7902	FORD	ANALYST	3000	RESEARCH	DALLAS	20
7934	MILLER	CLERK	1300	ACCOUNTING	NEW YORK	10
2222	saketh	doctor	8000	rd	bang	50

```
SQL> select * from dept;
```

DEPTNO	DNAME	LOC

```
---------- -------------- -----------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 rd             bang
```

```
SQL> select * from emp;

     EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM
DEPTNO
---------- ---------- --------------- -------- -------------- --------- --------- --
--------
      7369 SMITH      CLERK            7902 1 7-DEC-80         800
20
      7499 ALLEN      SALESMAN         7698 20-FEB-81         1600       300
30
      7521 WARD       SALESMAN         7698 22-FEB-81         1250       500
30
      7566 JONES      MANAGER          7839 02-APR-81         2975
20
      7654 MARTIN     SALESMAN         7698 28-SEP-81         1250      1400
30
      7698 BLAKE      MANAGER          7839 01-MAY-81         2850
30
      7782 CLARK      MANAGER          7839 09-JUN-81         2450
10
      7788 SCOTT      ANALYST          7566 19-APR-87         3000
20
      7839 KING       PRESIDENT                17-NOV-81         5000
10
      7844 TURNER     SALESMAN         7698 08-SEP-81         1500          0
30
```

```
    7876        ADAMS     CLERK            7788        23-MAY-87       1100
20
    7900        JAMES     CLERK            7698        03-DEC-81        950
30
    7902        FORD      ANALYST          7566        03-DEC-81       3000
20
    7934        MILLER    CLERK            7782        23-JAN-82       1300
10
    2222        saketh    doctor                                       8000
50
```

DDL TRIGGERS

Oracle allows you to define triggers that will fire when Data Definition Language statements are executed.

Syntax:

```
Create or replace trigger <trigger_name>
{Before | after}  {DDL event} on {database | schema}
[When (…)]
[Declare]
        -- declaration
Begin
        -- trigger body
[Exception]
        -- exception section
End <trigger_name>;
```

Ex:

```
SQL>  create  table  my_objects(obj_name  varchar(10),obj_type
varchar(10),obj_owner
      varchar(10),obj_time  date);

    CREATE OR REPLACE TRIGGER CREATE_TRIGGER
```

```
    after create on database
BEGIN
   insert                    into                    my_objects
values(sys.dictionary_obj_name,sys.dictionary_obj_type,
                              sys.dictionary_obj_owner,  sysdate);
END CREATE_TRIGGER;
```

Output:

```
SQL> select * from my_objects;

     no rows selected

SQL> create table stud1(no number(2));

SQL> select * from my_objects;

   OBJ_NAME   OBJ_TYPE   OBJ_OWNER  OBJ_TIME
   ---------- ---------- ---------- ----------
   STUD1         TABLE          SYS      21-JUL-07

SQL> create sequence ss;

SQL> create view stud_view as select * from stud1;

SQL> select * from my_objects;


   OBJ_NAME   OBJ_TYPE    OBJ_OWNER   OBJ_TIME
   ---------- ---------- ---------- ----------
   STUD1          TABLE        SYS        21-JUL-07
   SS             SEQUENCE     SYS        21-JUL-07
   STUD_VIEW      VIEW         SYS        21-JUL-07
```

WHEN CLAUSE

If WHEN present, the trigger body will be executed only for those that meet the condition specified by the WHEN clause.

Ex:

```
CREATE OR REPLACE TRIGGER CREATE_TRIGGER
    after create on database
    when (sys.dictionary_obj_type = 'TABLE')
BEGIN
   insert into my_objects
values(sys.dictionary_obj_name,sys.dictionary_obj_type,
                                    sys.dictionary_obj_owner, sysdate);
END CREATE_TRIGGER;
```

SYSTEM TRIGGERS

System triggers will fire whenever database-wide event occurs. The following are the database event triggers. To create system trigger you need ADMINISTER DATABASE TRIGGER privilege.

1. STARTUP
2. SHUTDOWN
3. LOGON
4. LOGOFF
5. SERVERERROR

Syntax:

```
Create or replace trigger <trigger_name>
{Before | after}  {Database event} on {database | schema}
[When (...)]
[Declare]
        -- declaration section
Begin
        -- trigger body
[Exception]
        -- exception section
End <trigger_name>;
```

Ex:

```
SQL> create table user_logs(u_name  varchar(10),log_time  timestamp);

    CREATE OR REPLACE TRIGGER AFTER_LOGON
        after logon on database
    BEGIN
        insert into user_logs values(user,current_timestamp);
    END AFTER_LOGON;
```

Output:

```
SQL> select * from user_logs;

      no rows selected

SQL> conn saketh/saketh
SQL> select * from user_logs;

    U_NAME     LOG_TIME
    ---------- ---------------------------------------------
    SAKETH     22-JUL-07 12.07.13.140000  AM

SQL> conn system/oracle
SQL> select * from user_logs;



    U_NAME     LOG_TIME
    ---------- ---------------------------------------------
    SAKETH     22-JUL-07 12.07.13.140000  AM
    SYSTEM     22-JUL-07 12.07.34.218000  AM

SQL> conn scott/tiger
SQL> select * from user_logs;
```

```
U_NAME     LOG_TIME
---------- ---------------------------------------
SAKETH     22-JUL-07 12.07.13.140000  AM
SYSTEM     22-JUL-07 12.07.34.218000  AM
SCOTT      22-JUL-07 12.08.43.093000  AM
```

SERVERERROR

The SERVERERROR event can be used to track errors that occur in the database. The error code is available inside the trigger through the SERVER_ERROR attribute function.

Ex:

```
SQL> create table my_errors(error_msg  varchar(200));

CREATE OR REPLACE TRIGGER SERVER_ERROR_TRIGGER
      after servererror on database
BEGIN
      insert into my_errors values(dbms_utility.format_error_stack);
END SERVER_ERROR_TRIGGER;
```

Output:

```
SQL> create table ss (no));
create table ss (no))
                *
ERROR at line 1:
ORA-00922: missing or invalid option

SQL> select * from my_errors;
ERROR_MSG
----------------------------------------------------------
ORA-00922: missing or invalid option

SQL> insert into student values(1,2,3);
insert into student values(1,2,3)
```

```
          *
ERROR at line 1:
ORA-00942: table or view does not exist

SQL> select * from my_errors;

ERROR_MSG
----------------------------------------------------------
ORA-00922: missing or invalid option
ORA-00942: table or view does not exist
```

SERVER_ERROR ATTRIBUTE FUNCTION

It takes a single number type of argument and returns the error at the position on the error stack indicated by the argument. The position 1 is the top of the stack.

Ex:

```
CREATE OR REPLACE TRIGGER SERVER_ERROR_TRIGGER
      after servererror on database
BEGIN
      insert into my_errors values(server_error(1));
END SERVER_ERROR_TRIGGER;
```

SUSPEND TRIGGERS

This will fire whenever a statement is suspended. This might occur as the result of a space issue such as exceeding an allocated tablepace quota. This functionality can be used to address the problem and allow the operatin to continue.

Syntax:

```
Create or replace trigger <trigger_name>
after suspend on {database | schema}
[When (...)]
[Declare]
```

```
        -- declaration section
    Begin
        -- trigger body
    [Exception]
        -- exception section
    End <trigger_name>;
```

Ex:

```
SQL> create tablespace my_space datafile 'f:\my_file.dbf' size 2m;
SQL> create table student(sno number(2),sname varchar(10)) tablespace my_space;

    CREATE OR REPLACE TRIGGER SUSPEND_TRIGGER
        after suspend on database
    BEGIN
        dbms_output.put_line(` No room to insert in your tablespace');
    END SUSPEND_TRIGGER;
```

Output:

Insert more rows in student table then , you will get

No room to insert in your tablespace

AUTONOMOUS TRANSACTION

Prior to Oracle8i, there was no way in which some SQL operations within a transaction could be committed independent of the rest of the operations. Oracle allows this, however, through *autonomous transactions*. An *autonomous transaction* is a transaction that is started within the context of another transaction, known as parent transaction, but is independent of it. The autonomous transaction can be committed or rolled back regardless ot the state of the parent transaction.

Ex:

```
CREATE OR REPLACE TRIGGER AUTONOMOUS_TRANSACTION_TRIGGER
        after insert on student
DECLARE
        pragma autonomous_transaction;
BEGIN
        update student set marks = 555;
        commit;
END AUTONOMOUS_TRANSACTION_TRIGGER;
```

Output:

```
SQL> select * from student;

        NO  NA      MARKS
       ----- ----- -- ----------
        1   a       111
        2   b       222
        3   c       300

SQL>  insert into student values(4,'d',444);

SQL> select * from student;

        NO  NA      MARKS
        ---- ------ -- ----------
        1   a       555
        2   b       555
        3   c       555
        4   d       444
```

RESTRICTIONS ON AUTONOMOUS TRANSACTION

1. If an autonomous transaction attempts to access a resource held by the main transaction, a deadlock can occur in you program.

2 You cannot mark all programs in a package as autonomous with a single PRAGMA declaration. You must indicate autonomous transactions explicity in each program.

3 To exit without errors from an autonomous transaction program that has executed at least one INSERT or UPDATE or DELETE, you must perform an explicit commit or rollback.

4 The COMMIT and ROLLBACK statements end the active autonomous transaction, but they do not force the termination of the autonomous routine. You can have multiple COMMIT and/or ROLLBACK statements inside your autonomous block.

5 You can not rollback to a savepoint set in the main transaction.

6 The TRANSACTIONS parameter in the oracle initialization file specifies the maximum number of transactions allowed concurrently in a session. The default value is 75 for this, but you can increase the limit.

MUTATING TABLES

There are restrictions on the tables and columns that a trigger body may access. In order to define these restrictions, it is necessary to understand mutating and constraining tables.

A mutating table is table that is currentlty being modified by a DML statement and the trigger event also DML statement. A mutating table error occurs when a row-level trigger tries to examine or change a table that is already undergoing change.

A constraining table is a table that might need to be read from for a referential integrity constraint.

Ex:

```
CREATE OR REPLACE TRIGGER MUTATING_TRIGGER
        before delete on student
        for each row

DECLARE
        ct number;
```

```
BEGIN
        select count(*) into ct from student where no = :old.no;
END MUTATING_TRIGGER;
```

Output:

```
SQL> delete student where no = 1;
delete student where no = 1
       *
ERROR at line 1:
ORA-04091: table SCOTT.STUDENT is mutating, trigger/function may not
see it
ORA-06512: at "SCOTT.T", line 4
ORA-04088: error during execution of trigger 'SCOTT.T'
```

HOW TO AVOID MUTATING TABLE ERROR ?

1. By using autonomous transaction
2. By using statement level trigger