Oracle for Developers (PL/SQL)

Introduction to PL/SQL

Overview

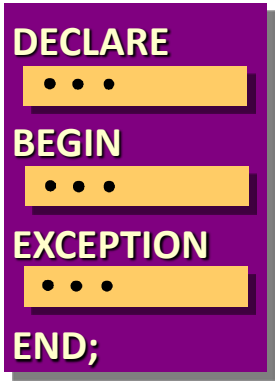
- **PL/SQL is a procedural extension to SQL.**
  - The “data manipulation” capabilities of “SQL” are combined with the “processing capabilities” of a “procedural language”.
  - PL/SQL provides features like conditional execution, looping and branching.
    - PL/SQL supports subroutines, as well.
  - PL/SQL program is of block type, which can be “sequential” or “nested” (one inside the other).

Salient Features

- **PL/SQL provides the following features:**
  - Tight Integration with SQL
  - Better performance
    - Several SQL statements can be bundled together into one PL/SQL block and sent to the server as a single unit.
  - Standard and portable language
    - Although there are a number of alternatives when it comes to writing software to run against the Oracle Database, it is easier to run highly efficient code in PL/SQL, to access the Oracle Database, than in any other language.

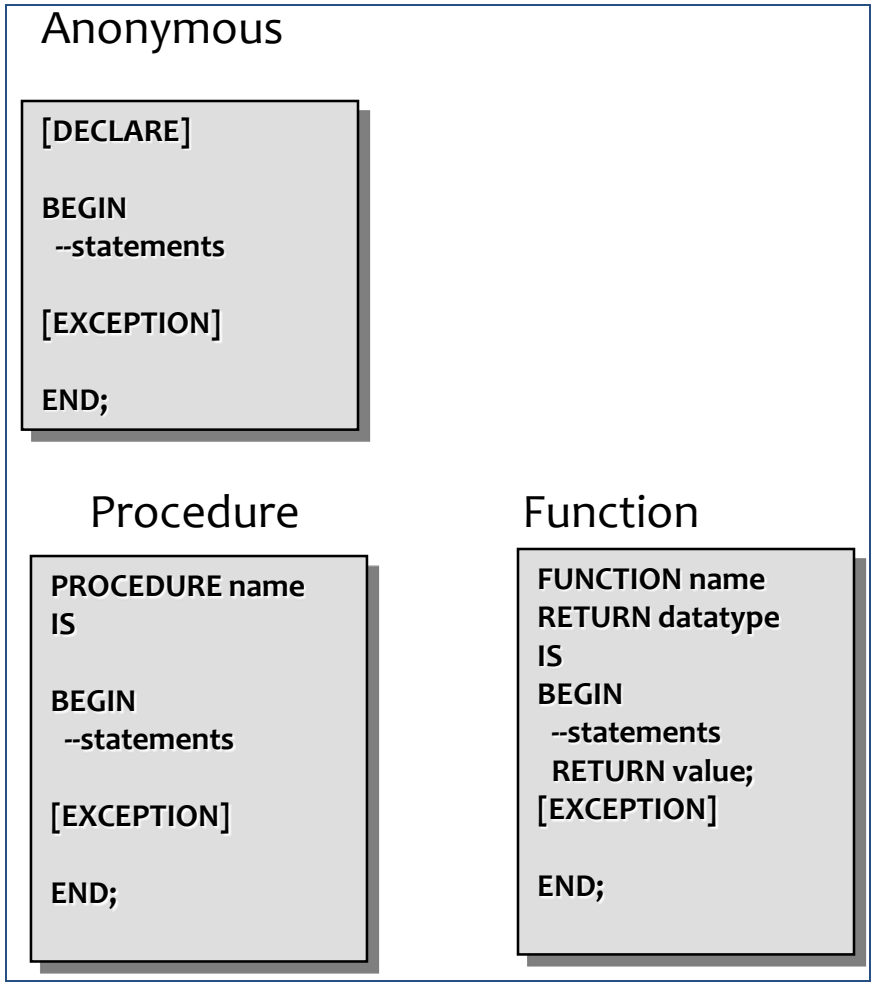
PL/SQL Block Structure

- **A PL/SQL block comprises of the following structures:**
  - DECLARE – Optional
    - Variables, cursors, user-defined exceptions
  - BEGIN – Mandatory
    - SQL statements
    - PL/SQL statements
  - EXCEPTION – Optional
    - Actions to perform when errors occur
  - END; – Mandatory



Block Types

- **There are three types of blocks in PL/SQL:**
  - Anonymous
  - Named:
    - Procedure
    - Function



Points to Remember

- **While handling variables in PL/SQL:**
  - declare and initialize variables within the declaration section
  - assign new values to variables within the executable section
  - pass values into PL/SQL blocks through parameters
  - view results through output variables

Guidelines for declaring variables

- **Given below are a few guidelines for declaring variables:**
  - follow the naming conventions
  - initialize the variables designated as NOT NULL
  - initialize the identifiers by using the assignment operator (:=) or by using the DEFAULT reserved word
  - declare at most one Identifier per line

Types of Variables

- **PL/SQL variables**
  - Scalar
  - Composite
  - Reference
  - LOB (large objects)
- **Non-PL/SQL variables**
  - Bind and host variables

Declaring PL/SQL variables

- **Syntax**

```
identifier [CONSTANT] datatype [NOT NULL]
                    [:= | DEFAULT expr];
```

Example

```
DECLARE
      v_hiredate          DATE;
      v_deptno            NUMBER(2) NOT NULL := 10;
      v_location          VARCHAR2(13) := 'Atlanta';
      c_comm  CONSTANT NUMBER := 1400;
```

Base Scalar Data Types

- **Base Scalar Datatypes:**
  - Given below is a list of Base Scalar Datatypes:
    - VARCHAR2 (maximum_length)
    - NUMBER [(precision, scale)]
    - DATE
    - CHAR [(maximum_length)]
    - LONG
    - LONG RAW
    - BOOLEAN
    - BINARY_INTEGER
    - PLS_INTEGER

Base Scalar Data Types - Example

- **Here are a few examples of Base Scalar Datatypes:**

```
v_job          VARCHAR2(9);
v_count        BINARY_INTEGER := 0;
v_total_sal    NUMBER(9,2) := 0;
v_orderdate    DATE := SYSDATE + 7;
c_tax_rate     CONSTANT NUMBER(3,2) := 8.25;
v_valid        BOOLEAN NOT NULL := TRUE;
```

Declaring Datatype by using %TYPE Attribute

Example:

```
...
v_name                  staff_master.staff_name%TYPE;
v_balance               NUMBER(7,2);
v_min_balance  v_balance%TYPE := 10;
...
```

Declaring Datatype by using %ROWTYPE

Example:

```
DECLARE
        nRecord staff_master%rowtype;
BEGIN
        SELECT * into nrecord
                        FROM staff_master
                        WHERE staff_code = 100001;

                        UPDATE staff_master
                        SET staff_sal = staff_sal + 101
                        WHERE emp_code = 100001;
END;
```

Composite Data Types

- **Composite Datatypes in PL/SQL:**
  - Two composite datatypes are available in PL/SQL:
    - records
    - tables
  - A composite type contains components within it. A variable of a composite type contains one or more scalar variables.

Record Data Types

- **Record Datatype:**
  - A record is a collection of individual fields that represents a row in the table.
  - They are unique and each has its own name and datatype.
  - The record as a whole does not have value.
- **Defining and declaring records:**
  - Define a RECORD type, then declare records of that type.
  - Define in the declarative part of any block, subprogram, or package.

Record Data Types

- **Syntax:**

```
TYPE type_name IS RECORD (field_declaration [,field_ declaration] ...);
```

Record Data Types - Example

- **Here is an example for declaring Record datatype:**

```
DECLARE
TYPE DeptRec IS RECORD (
Dept_id         department_master.dept_code%TYPE,
Dept_name       varchar2(15),
```

Record Data Types - Example

- **Here is an example for declaring and using Record datatype:**

```
DECLARE
        TYPE recname is RECORD
        (customer_id number,
         customer_name varchar2(20));
        var_rec    recname;
BEGIN
        var_rec.customer_id:=20;
        var_rec.customer_name:='Smith';
        dbms_output.put_line(var_rec.customer_id||'
'||var_rec.customer_name);
END;
```

Table Data Type

- **A PL/SQL table is:**
  - a one-dimensional, unbounded, sparse collection of homogeneous elements
  - indexed by integers
  - In technical terms, a PL/SQL table:
    - is like an array
    - is like a SQL table; yet it is not precisely the same as either of those data structures
    - is one type of collection structure
    - is PL/SQL's way of providing arrays

Table Data Type

- **Declaring a PL/SQL table:**
  - There are two steps to declare a PL/SQL table:
    - Declare a TABLE type.
    - Declare PL/SQL tables of that type.

```
TYPE type_name is TABLE OF
{Column_type | table.column%type} [NOT NULL]
INDEX BY BINARY_INTEGER;
```

  - If the column is defined as NOT NULL, then PL/SQL table will reject NULLs.

Table Data Type - Examples

Example 1:

- To create a PL/SQL table named as "student_table" of char column.

```
DECLARE
TYPE student_table is table of char(10)
INDEX BY BINARY_INTEGER;
```

Example 2:

- To create "student_table" based on the existing column of "student_name" of EMP table.

```
DECLARE
TYPE student_table is table of student_master.student_name%type
INDEX BY BINARY_INTEGER;
```

Table Data Type - Examples

- **After defining type emp_table, define the PL/SQL tables of that type.**

For Example:

```
Student_tab   student_table;
```

- These tables are unconstrained tables.
- You cannot initialize a PL/SQL table in its declaration.

For Example:

```
Student_tab :=('SMITH','JONES','BLAKE');  --Illegal
```

Referencing PL/SQL Tables

- **Here is an example of referencing PL/SQL tables:**

```
DECLARE
   TYPE staff_table is table of
    staff_master.staff_name%type
    INDEX BY BINARY_INTEGER;
staff_tab staff_table;
BEGIN
    staff_tab(1) := 'Smith'; --update Smith's salary
    UPDATE staff_master
    SET staff_sal = 1.1 * staff_sal
    WHERE staff_name = staff_tab(1);
END;
```

Referencing PL/SQL Tables - Examples

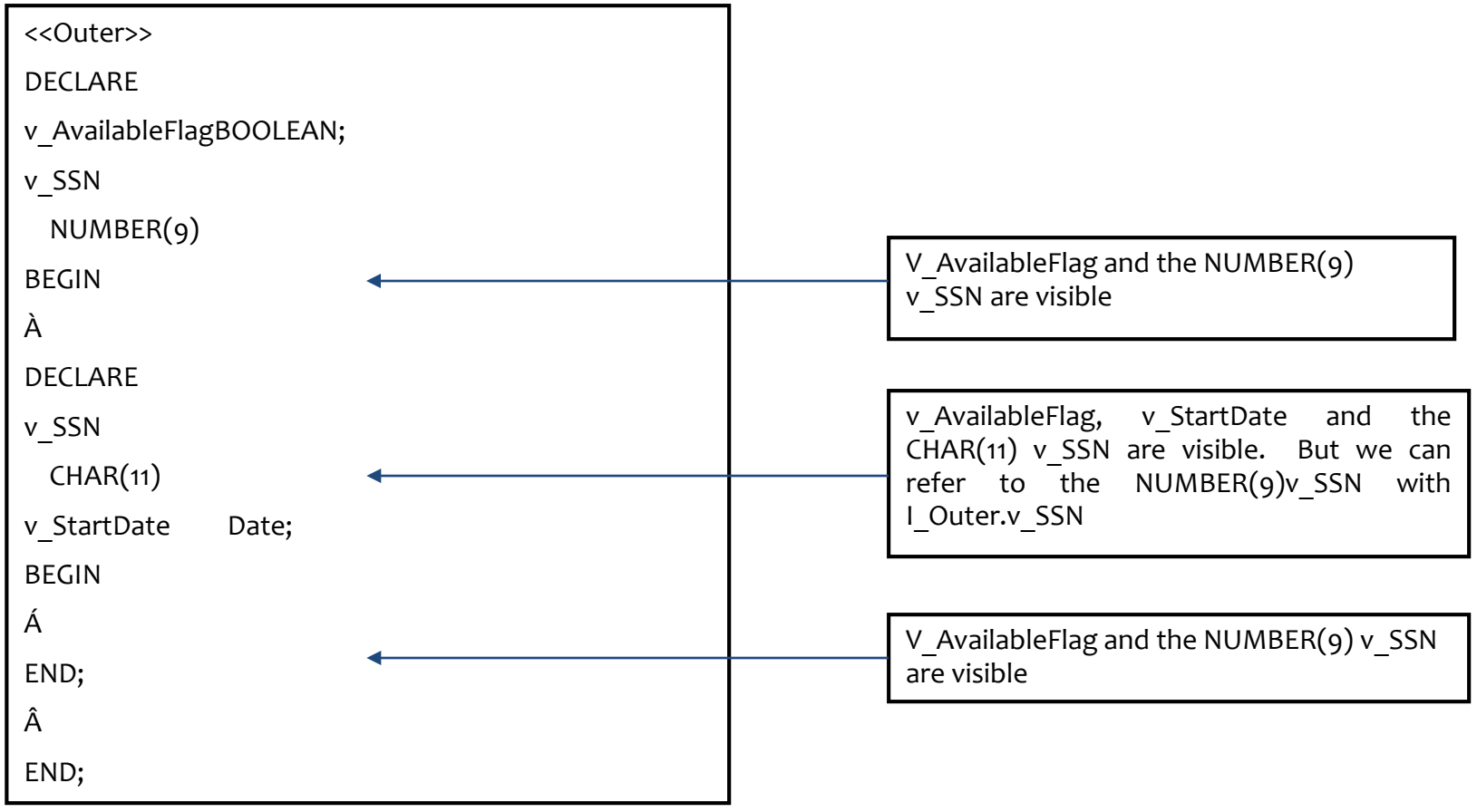
- To assign values to specific rows, the following syntax is used:

```
PLSQL_table_name(primary_key_value) := PLSQL expression;
```

- From ORACLE 7.3, the PL/SQL tables allow records as their columns.

Scope and Visibility of Variables

- **Pictorial representation of visibility of a variable:**



Scope and Visibility of Variables

```
<<OUTER>>
DECLARE
V_Flag   BOOLEAN ;
V_Var1  CHAR(9);
BEGIN
<<INNER>>
DECLARE
V_Var1  NUMBER(9);
V_Date  DATE;
BEGIN
NULL;
END;
NULL;
END;
```

Types of Statements

- **Given below are some of the SQL statements that are used in PL/SQL:**
  - INSERT statement
    - The syntax for the INSERT statement remains the same as in SQL-INSERT.
    - For example:

```
DECLARE
        v_dname varchar2(15) := 'Accounts';
BEGIN
        INSERT into department_master
      VALUES (50, v_dname);
END;
```

Types of Statements

- DELETE statement

For Example:

```
DECLARE
        v_sal_cutoff  number  := 2000;
BEGIN
        DELETE FROM staff_master
     WHERE  staff_sal < v_sal_cutoff;
END;
```

Types of Statements

- UPDATE statement

For Example:

```
DECLARE
        v_sal_incr  number(5) := 1000;
BEGIN
        UPDATE staff_master
     SET staff_sal = staff_sal + v_sal_incr
     WHERE staff_name='Smith';
END;
```

Types of Statements

- SELECT statement
  - Syntax:

```
SELECT Column_List INTO Variable_List
    FROM Table_List
    [WHERE expr1]
    CONNECT BY expr2 [START WITH expr3]]
    GROUP BY expr4] [HAVING expr5]
    [UNION | INTERSECT | MINUS SELECT ...]
    [ORDER BY expr | ASC | DESC]
    [FOR UPDATE [OF Col1,...] [NOWAIT]]
    INTO Variable_List;
```

Types of Statements

- **The column values returned by the SELECT command must be stored in variables.**
- **The Variable_List should match Column_List in both COUNT and DATATYPE.**
- **Here the variable lists are PL/SQL (Host) variables. They should be defined before use.**

Types of Statements

Example: <<BLOCK1>>

```
DECLARE
        deptno   number(10)  := 30;
        dname   varchar2(15) ;
BEGIN
        SELECT dept_name INTO dname FROM department_master
WHERE dept_code = Block1. deptno;
        DELETE FROM department_master
                                WHERE dept_code = Block1. deptno ;
END;
```

IF Construct

- **Given below is a list of Programmatic Constructs which are used in PL/SQL:**
  - Conditional Execution:
    - This construct is used to execute a set of statements only if a particular condition is TRUE or FALSE.
    - Syntax:

```
IF Condition_Expr
THEN
        PL/SQL_Statements
END IF;
```

IF Construct - Example

For Example:

```
IF v_staffno = 100003
THEN
        UPDATE staff_master
        SET staff_sal = staff_sal + 100
        WHERE staff_code = 100003 ;
END IF;
```

IF Construct - Example

- **To take alternate action if condition is FALSE, use the following syntax:**

```
IF Condition_Expr THEN
        PL/SQL_Statements_1 ;
ELSE
        PL/SQL_Statements_2 ;
END IF;
```

IF Construct - Example

- **To check for multiple conditions, use the following syntax.**

```
IF Condition_Expr_1
      THEN
            PL/SQL_Statements_1 ;
      ELSIF Condition_Expr_2
      THEN
            PL/SQL_Statements_2 ;
      ELSIF Condition_Expr_3
      THEN
            PL/SQL_Statements_3 ;
      ELSE
            PL/SQL_Statements_n ;
END IF;
```

- Note: Conditions for NULL are checked through IS NULL and IS NOT NULL predicates.

Simple Loop

- **Looping**
  - A LOOP is used to execute a set of statements more than once.
  - Syntax:

```
LOOP
      PL/SQL_Statements;
END LOOP ;
```

Simple Loop

For example:

```
DECLARE
        v_counter  number := 50 ;
BEGIN
LOOP
        INSERT INTO department_master
                        VALUES(v_counter,'new dept');
        v_counter := v_counter + 10 ;
END LOOP;
        COMMIT ;
END ;
/
```

Simple Loop – EXIT statement

- **EXIT**
  - Exit path is provided by using EXIT or EXIT WHEN commands.
  - EXIT is an unconditional exit. Control is transferred to the statement following END LOOP, when the execution flow reaches the EXIT statement.

contd.

Simple Loop – EXIT statement

- **Syntax:**

```
BEGIN
        .....
        .....
        LOOP
                IF <Condition> THEN
        . . . . .
                EXIT ;                  -- Exits loop immediately
                END IF ;

        END LOOP;
        LOOP
                ……..
                ……..
                EXIT WHEN <condition>
        END LOOP;

        . . . . .                                   -- Control resumes here
        COMMIT ;
        END ;
```

Simple Loop – EXIT statement

For example:

```
DECLARE
      v_counter number := 50 ;
BEGIN
      LOOP
         INSERT INTO department_master
        VALUES(v_counter,'NEWDEPT');
      DELETE  FROM emp WHERE deptno = v_counter;
                  v_counter := v_counter + 10 ;
                 EXIT WHEN v_counter >100 ;
      END LOOP;
      COMMIT ;
END ;
```

- Note: As long as v_counter has a value less than or equal to 100, the loop continues.

For Loop

- **FOR Loop:**
  - Syntax:

```
FOR Variable IN [REVERSE] Lower_Bound..Upper_Bound
LOOP
      PL/SQL_Statements
END LOOP ;
```

While Loop

- **WHILE Loop**
  - The WHILE loop is used as shown below.
  - Syntax:

```
WHILE Condition
LOOP
      PL/SQL  Statements;
END LOOP;
```

- EXIT OR EXIT WHEN can be used inside the WHILE loop to prematurely exit the loop.

Labeling of Loops

- **Labeling of Loops:**
  - The label can be used with the EXIT statement to exit out of a particular loop.

```
BEGIN
      <<Outer_Loop>>
      LOOP
                  PL/SQL
                  << Inner_Loop>>
                  LOOP
                              PL/SQL  Statements ;
                              EXIT Outer_Loop WHEN <Condition Met>
                  END LOOP  Inner_Loop
      END LOOP  Outer_Loop
END ;
```

Oracle for Developers (PL/SQL)

Introduction to Cursors

Concept of a Cursor

- **A cursor is a "handle" or "name" for a private SQL area**
  - An SQL area (context area) is an area in the memory in which a parsed statement and other information for processing the statement are kept
  - PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return "only one row"
  - For queries that return "more than one row", you must declare an explicit cursor
  - Thus the two types of cursors are:
    - implicit
    - explicit

Implicit Cursors

- **Implicit Cursor:**
  - The PL/SQL engine takes care of automatic processing
  - PL/SQL implicitly declares cursors for all DML statements
  - They are simple SELECT statements and are written in the BEGIN block (executable section) of the PL/SQL
  - They are easy to code, and they retrieve exactly one row

Processing Implicit Cursor

- **Processing Implicit Cursors:**
  - Oracle implicitly opens a cursor to process each SQL statement that is not associated with an explicitly declared cursor
  - This implicit cursor is known as SQL cursor
    - Program cannot use the OPEN, FETCH, and CLOSE statements to control the SQL cursor. PL/SQL implicitly does those operations
    - You can use cursor attributes to get information about the most recently executed SQL statement
    - Implicit Cursor is used to process INSERT, UPDATE, DELETE, and single row SELECT INTO statements

Processing Implicit Cursor - Examples

```
BEGIN
   UPDATE dept SET dname =‘Production' WHERE deptno= 50;
   IF SQL%NOTFOUND THEN
    INSERT into department_master VALUES ( 50, ‘Production');
   END IF;
END;
```

```
BEGIN
   UPDATE dept SET dname =‘Production' WHERE deptno = 50;
   IF SQL%ROWCOUNT = 0 THEN
     INSERT into  department_master VALUES ( 50, ‘Production');
END IF;
END;
```

Explicit Cursor

- **Explicit Cursor:**
  - The set of rows returned by a query can consist of zero, one, or multiple rows, depending on how many rows meet your search criteria
  - When a query returns multiple rows, you can explicitly declare a cursor to process the rows
  - You can declare a cursor in the declarative part of any PL/SQL block, subprogram, or package
  - Processing has to be done by the user

Processing Explicit Cursor

- **While processing Explicit Cursors you have to perform the following four steps:**
  - Declare the cursor
  - Open the cursor for a query
  - Fetch the results into PL/SQL variables
  - Close the cursor

Processing Explicit Cursor

- **Declaring a Cursor:**
  - Syntax:

```
CURSOR Cursor_Name IS Select_Statement;
```

  - Any SELECT statements are legal including JOINS, UNION, and MINUS clauses.
    - SELECT statement should not have an INTO clause.
  - Cursor declaration can reference PL/SQL variables in the WHERE clause.
    - The variables (bind variables) used in the WHERE clause must be visible at the point of the cursor.

Processing Explicit Cursor

- **Opening a Cursor**
  - Syntax:

```
OPEN  Cursor_Name;
```

- When a cursor is opened, the following events occur:
  - The values of bind variables are examined.
  - The active result set is determined.
  - The active result set pointer is set to the first row.

Processing Explicit Cursor

- **Fetching from a Cursor**
  - Syntax:

```
FETCH Cursor_Name  INTO List_Of_Variables;
FETCH  Cursor_Name INTO PL/SQL _Record;
```

- The "list of variables" in the INTO clause should match the "column names list" in the SELECT clause of the CURSOR declaration, both in terms of count as well as in datatype.
- After each FETCH, the active set pointer is increased to point to the next row.
  - The end of the active set can be found out by using %NOTFOUND attribute of the cursor.

Processing Explicit Cursor

- **Closing a Cursor**
  - Syntax

```
CLOSE  Cursor_Name;
```

- Closing a Cursor frees the resources associated with the Cursor.
  - You cannot FETCH from a closed Cursor.
  - You cannot close an already closed Cursor.

Cursor Attributes

- **Cursor Attributes:**
  - Explicit cursor attributes return information about the execution of a multi-row query.
  - When an “Explicit cursor” or a “cursor variable” is opened, the rows that satisfy the associated query are identified and form the result set.
  - Rows are fetched from the result set.
  - Examples: %ISOPEN, %FOUND, %NOTFOUND, %ROWCOUNT, etc.

Types of Cursor Attributes

- **The different types of cursor attributes are described in brief, as follows:**
  - %ISOPEN
    - %ISOPEN returns TRUE if its cursor or cursor variable is open. Otherwise it returns FALSE.
    - Syntax:

```
Cur_Name%ISOPEN
```

Types of Cursor Attributes

Example:

```
DECLARE
                cursor c1 is
          select_statement ;
 BEGIN
      IF c1%ISOPEN THEN
                pl/sql_statements ;
      END IF;
  END ;
```

Types of Cursor Attributes

- **%FOUND**
  - %FOUND yields NULL after a cursor or cursor variable is opened but before the first fetch.
  - Thereafter, it yields:
    - TRUE if the last fetch has returned a row, or
    - FALSE if the last fetch has failed to return a row
  - Syntax:

```
cur_Name%FOUND
```

Types of Cursor Attributes

Example:

```
DECLARE section;
   open c1 ;
   fetch c1 into var_list ;
IF c1%FOUND THEN
      pl/sql_statements ;
END IF ;
```

Types of Cursor Attributes

- **%NOTFOUND**
  - %NOTFOUND is the logical opposite of %FOUND.
  - %NOTFOUND yields:
    - FALSE if the last fetch has returned a row, or
    - TRUE if the last fetch has failed to return a row
  - It is mostly used as an exit condition.
  - Syntax:

```
cur_Name%NOTFOUND
```

Types of Cursor Attributes

- **%ROWCOUNT**
  - %ROWCOUNT returns number of rows fetched from the cursor area using FETCH command
  - %ROWCOUNT is zeroed when its cursor or cursor variable is opened.
    - Before the first fetch, %ROWCOUNT yields 0
    - Thereafter, it yields the number of rows fetched at that point of time
  - The number is incremented if the last FETCH has returned a row
  - Syntax:

```
cur_Name%NOTFOUND
```

Cursor FETCH loops

- They are examples of simple loop statements
- The FETCH statement should be followed by the EXIT condition to avoid infinite looping
- Condition to be checked is cursor%NOTFOUND

Examples: LOOP .. END LOOP, WHILE LOOP, etc

Cursor using LOOP … END LOOP:

```
DECLARE
     cursor c1 is ……..
BEGIN
    open cursor c1;  /* open the cursor and identify the active result set.*/
LOOP
    fetch c1 into  variable_list ;
    -- exit out of the loop when there are no more rows.
    /* exit is done before processing to prevent handling of null rows.*/
    EXIT WHEN C1%NOTFOUND ;
    /* Process the fetched rows using variables and  PL/SQLstatements */
END LOOP;
    -- Free resources used by the cursor
    close  c1;
    -- commit
    commit;
END;
```

FOR Cursor Loop

- **FOR Cursor Loop**

```
FOR Variable in Cursor_Name
      LOOP
                  Process the variables
   END LOOP;
```

- **You can pass parameters to the cursor in a CURSOR FOR loop.**

```
FOR Variable in Cursor_Name ( PARAM1 , PARAM 2 ….)
      LOOP
                  Process the variables
      END LOOP;
```

SELECT… FOR UPDATE

- **SELECT … FOR UPDATE cursor:**
  - The method of locking records which are selected for modification, consists of two parts:
    - The FOR UPDATE clause in CURSOR declaration.
    - The WHERE CURRENT OF clause in an UPDATE or DELETE statement.
    - Syntax: FOR UPDATE

```
CURSOR  Cursor_Name IS  SELECT  ….. FROM … WHERE  .. ORDER BY
FOR UPDATE  [OF column names ] [ NOWAIT]
```

- where column names are the names of the columns in the table against which the query is fired. The column names are optional.

SELECT… FOR UPDATE

- If the cursor is declared with a FOR UPDATE clause, the WHERE CURRENT OF clause can be used in an UPDATE or DELETE statement.
  - Syntax: WHERE CURRENT OF

```
WHERE CURRENT OF Cursor_Name
```

- The WHERE CURRENT OF clause evaluates up to the row that was just retrieved by the cursor.
- When querying multiple tables Rows in a table are locked only if the FOR UPDATE OF clause refers to a column in that table.

contd.

SELECT… FOR UPDATE

For example: Following query locks the staff_master table but not the department_master table.

```
CURSOR C1 is SELECT staff_code, job, dname from emp, dept WHERE
emp.deptno=dept.deptno FOR UPDATE OF sal;
```

- Using primary key simulates the WHERE CURRENT OF clause but does not create any locks.

Examples

- To promote professors who earn more than 20000

```
DECLARE
CURSOR c_staff  is SELECT staff_code, staff_master.design_code
 FROM staff_master,designation_master
 WHERE design_name = 'Professor' and staff_sal > 20000
and staff_master.design_code =designation_master.design_code
FOR UPDATE OF design_code NOWAIT;
d_code  designation_master.design_code%type;
BEGIN
      SELECT design_code into d_code FROM designation_master
          WHERE design_name='Director';
        FOR v_rec in c_staff
        LOOP
            UPDATE staff_master SET design_code = d_code
             WHERE current of c_staff;
        END LOOP;
END;
```

Parameterized Cursor

- **You must use the OPEN statement to pass parameters to a cursor.**
  - Unless you want to accept default values, each "formal parameter" in the Cursor declaration must have a corresponding "actual parameter" in the OPEN statement.
  - The scope of parameters is local to the cursor.
  - Syntax:

```
OPEN Cursor-name(param1, param2…..)
```

Parameterized Cursor - Examples

- **Parameters are passed to a parametric cursor using the syntax OPEN (param1, param2 …)as shown in the following example:**

```
OPEN C_Select_staff( 800,5000);
    Query → SELECT  * from staff_master
     WHERE staff_sal BETWEEN 800 AND 5000;
```

Usage of Cursor Variables

- **Like a Cursor, a Cursor Variable points to the current row in the result set of a multi-row query**
  - A Cursor is static whereas a Cursor Variable is dynamic because it is not tied to a specific query
  - You can open a Cursor Variable for any type-compatible query
    - This offers more flexibility
  - You can assign new values to a Cursor Variable and pass it as a parameter to subprograms, including those in database
    - This offers an easy way to centralize data retrieval

Cursor Variables - Example

- **Defining REF CURSOR types:**
  - Syntax:

```
TYPE ref_type_name IS REF CURSOR RETURN return_type;
    DECLARE
        TYPE DeptCurTyp IS REF CURSOR RETURN
department_master%ROWTYPE;
```

- where:
  - ref_type_name is a type specifier used in subsequent declarations of cursor variables
  - Return_type must represent a record or a row in a database table.

Cursor Variables - Example

- REF CURSOR types are strong (restrictive), or weak (non-restrictive)

```
DECLARE
      TYPE staffCurTyp IS REF CURSOR
      RETURN staff_master%ROWTYPE;  -- Strong types


      TYPE GenericCurTyp IS REF CURSOR;  -- Weak types
```

Cursor Variables - Example

- **Declaring Cursor Variables:**

Example 1:

```
DECLARE
TYPE DeptCurTyp IS REF CURSOR RETURN
department_master%ROWTYPE;
dept_cv   DeptCurTyp;  -- Declare cursor variable
```

- You cannot declare cursor variables in a package.

Example 2:

```
TYPE TmpCurTyp IS REF CURSOR RETURN staff_master%ROWTYPE;
tmp_cv TmpCurTyp;  -- Declare cursor variable
```

Cursor Variables - Example

```
DECLARE
        TYPE staffcurtyp is REF CURSOR RETURN
         staff_master%rowtype;
        staff_cv    staffcurtyp;  -- declare cursor variable
        staff_cur       staff_master%rowtype;
BEGIN
         open staff_cv for  select * from staff_master;
 LOOP
              EXIT WHEN staff_cv%notfound;
              FETCH staff_cv into staff_cur;
              INSERT into temp_table  VALUES (staff_cv.staff_code,
                 staff_cv.staff_name,staff_cv.staff_sal);
          END LOOP;
          CLOSE staff_cv;
      END;
```

Oracle for Developers PL/SQL)

Procedures, Functions, and Packages

Introduction

- **A subprogram is a named block of PL/SQL**
- **There are two types of subprograms in PL/SQL, namely: Procedures and Functions**
- **Each subprogram has:**
  - A declarative part
  - An executable part or body, and
  - An exception handling part (which is optional)
- **A function is used to perform an action and return a single value**

Anonymous Blocks & Stored Subprograms Comparison

Anonymous Blocks	Stored Subprograms/Named Blocks
1. Anonymous Blocks do not have names.	1. Stored subprograms are named PL/SQL blocks.
2. They are interactively executed. The block needs to be compiled every time it is run.	2. They are compiled at the time of creation and stored in the database itself. Source code is also stored in the database.
3. Only the user who created the block can use the block.	3. Necessary privileges are required to execute the block.

Procedures

- **A procedure is used to perform an action.**
- **It is illegal to constrain datatypes.**
- **Syntax:**

```
CREATE PROCEDURE  Proc_Name
   (Parameter {IN | OUT | IN OUT} datatype := value,...) AS
   Variable_Declaration ;
   Cursor_Declaration ;
   Exception_Declaration ;
BEGIN
    PL/SQL_Statements ;
EXCEPTION
    Exception_Definition ;
END Proc_Name ;
```

Subprogram Parameter Modes

IN	OUT	IN OUT
The default	Must be specified	Must be specified
Used to pass values to the procedure.	Used to return values to the caller.	Used to pass initial values to the procedure and return updated values to the caller.
Formal parameter acts like a constant.	Formal parameter acts like an uninitialized variable.	Formal parameter acts like an uninitialized variable.
Formal parameter cannot be assigned a value.	Formal parameter cannot be used in an expression, but should be assigned a value.	Formal parameter should be assigned a value.
Actual parameter can be a constant, literal, initialized variable, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.
Actual parameter is passed by reference (a pointer to the value is passed in).	Actual parameter is passed by value (a copy of the value is passed out) unless NOCOPY is specified.	Actual parameter is passed by value (a copy of the value is passed in and out) unless NOCOPY is specified.

Example on Procedures

Example 1:

```
CREATE OR REPLACE PROCEDURE Raise_Salary
  ( s_no IN number, raise_sal IN number) IS
  v_cur_salary   number ;
   missing_salary exception;
BEGIN
     SELECT staff_sal INTO v_cur_salary FROM staff_master
     WHERE staff_code=s_no;
  IF  v_cur_salary IS NULL THEN
     RAISE  missing_salary;
  END IF ;
        UPDATE staff_master SET staff_sal = v_cur_salary + raise_sal
        WHERE staff_code =  s_no ;
EXCEPTION
           WHEN  missing_salary THEN
           INSERT into emp_audit VALUES( sno, 'salary is missing');
 END raise_salary;
```

Example on Procedures

Example 2:

```
CREATE OR REPLACE PROCEDURE
    Get_Details(s_code IN number,
    s_name OUT varchar2,s_sal OUT number ) IS
BEGIN
  SELECT staff_name, staff_sal INTO s_name, s_sal
  FROM staff_master WHERE staff_code=s_code;
EXCEPTION
          WHEN no_data_found  THEN
          INSERT into auditstaff
          VALUES( 'No employee with id ' || s_code);
          s_name := null;
          s_sal := null;
    END get_details ;
```

Executing a Procedure

- **Executing the Procedure from SQL*PLUS environment,**
  - Create a bind variables salary and name SQLPLUS by using VARIABLE command as follows:

```
variable salary number
variable name varchar2(20)
```

  - Execute the procedure with EXECUTE command

```
EXECUTE Get_Details(100003,:Salary, :Name)
```

  - After execution, use SQL*PLUS PRINT command to view results.

```
print salary
print name
```

Positional notation: Example

```
CREATE OR REPLACE PROCEDURE Create_Dept(deptno number,dname
varchar2,location varchar2) as
BEGIN
INSERT INTO  dept VALUES(deptno,dname,location);
END;
```

Executing a procedure using positional parameter notation is as follows:

```
SQL>execute  Create_Dept(90,'sales','mumbai');
```

Named notation: Example

```
CREATE OR REPLACE PROCEDURE Create_Dept(deptno number,dname
varchar2,location varchar2) as
BEGIN
INSERT INTO  dept VALUES(deptno,dname,location);
END;
```

Executing a procedure using named parameter notation is as follows:

```
SQL>execute  Create_Dept(deptno=>90,dname=>'sales',location=>'mumbai');
```

Following procedure call is also valid :

```
SQL>execute  Create_Dept(location=>'mumbai‘, deptno=>90,dname=>'sales');
```

Mixed Notation Example:

```
CREATE OR REPLACE PROCEDURE Create_Dept(deptno number,dname
varchar2,location varchar2) as
BEGIN
INSERT INTO  dept VALUES(deptno,dname,location);
END;
```

Executing a procedure using mixed parameter notation is as follows:

```
SQL>execute  Create_Dept(90, location=>'mumbai‘, dname=>'sales');
```

Functions

- **A function is similar to a procedure.**
- **A function is used to compute a value.**
  - A function accepts one or more parameters, and returns a single value by using a return value.
  - A function can return multiple values by using OUT parameters.
  - A function is used as part of an expression, and can be called as Lvalue = Function_Name(Param1, Param2, …….).
  - Functions returning a single value for a row can be used with SQL statements.

Functions

- **Syntax :**

```
CREATE  FUNCTION Func_Name(Param datatype :=
  value,..) RETURN datatype1 AS
        Variable_Declaration ;
        Cursor_Declaration ;
        Exception_Declaration ;
  BEGIN
          PL/SQL_Statements ;
          RETURN Variable_Or_Value_Of_Type_Datatype1 ;
  EXCEPTION
          Exception_Definition ;
  END Func_Name ;
```

Examples on Functions

Example 1:

```
CREATE FUNCTION Crt_Dept(dno number,
  dname varchar2)  RETURN number AS
BEGIN
        INSERT into department_master
        VALUES (dno,dname) ;
                return 1 ;
EXCEPTION
        WHEN others THEN
                return 0 ;
END crt_dept ;
```

Executing a Function

- **Executing functions from SQL*PLUS:**
  - Create a bind variable Avg salary in SQLPLUS by using VARIABLE command as follows:
    - Execute the Function with EXECUTE command:
    - After execution, use SQL*PLUS PRINT command to view results.

```
variable flag number
```

```
EXECUTE :flag:=Crt_Dept(60,'Production');
```

```
PRINT flag;
```

Exceptions handling in Procedures and Functions

- **If procedure has no exception handler for any error, the control immediately passes out of the procedure to the calling environment.**
- **Values of OUT and IN OUT formal parameters are not returned to actual parameters.**
- **Actual parameters will retain their old values.**

Packages

- **A package is a schema object that groups all the logically related PL/SQL types, items, and subprograms.**
  - Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary.
    - The specification (spec for short) is the interface to your applications. It declares the types, variables, constants, exceptions, cursors, and subprograms available for use.
    - The body fully defines cursors and subprograms, and so implements the spec.
  - Each part is separately stored in a Data Dictionary.

Packages

- **Note that:**
  - Packages variables ~ global variables
  - Functions and Procedures ~ accessible to users having access to the package
  - Private Subprograms ~ not accessible to users

Packages

- **Syntax of Package Specification:**

```
CREATE PACKAGE Package_Name  AS
        variable_declaration ;
        cursor_declaration ;
    FUNCTION Func_Name(param datatype,..) return datatype1 ;
    PROCEDURE Proc_Name(param {IN|OUT|INOUT}
    datatype,...);
END package_name ;
```

Example of Package

- **Creating Package Specification**

```
CREATE OR REPLACE PACKAGE Pack1 AS
        PROCEDURE Proc1;
        FUNCTION Fun1 return varchar2;
END pack1;
```

Example of Package

- **Creating Package Body**

```
CREATE OR REPLACE PACKAGE BODY Pack1 AS
   PROCEDURE Proc1 IS
   BEGIN
            dbms_output.put_line('hi a message frm procedure');
   END Proc1;
            function Fun1 return varchar2 IS
   BEGIN
            return ('hello from fun1');
   END Fun1;
END Pack1;
```

Executing a Package

- **Executing Procedure from a package:**

```
EXEC Pack1.Proc1
Hi a message frm procedure
```

- **Executing Function from a package:**

```
SELECT Pack1.Fun1 FROM dual;

FUN1
--------------------------------------
hello from fun1
```

Package Instantiation

- **Package Instantiation:**
  - The packaged procedures and functions have to be prefixed with package names.
  - The first time a package is called, it is instantiated.

Subprograms and Ref Type Cursors

- **You can declare Cursor Variables as the formal parameters of Functions and Procedures.**

```
CREATE OR REPLACE PACKAGE Staff_Data AS
   TYPE staffcurtyp is ref cursor return
     staff_master%rowtype;
 PROCEDURE Open_Staff_Cur(staff_cur IN OUT
  staffcurtyp);
END Staff_Data;
```

Subprograms and Ref Type Cursors

```
CREATE OR REPLACE PACKAGE BODY Staff_Data AS
PROCEDURE Open_Staff_Cur (staff_cur IN OUT staffcurtyp) IS
BEGIN
      OPEN staff_cur  for SELECT * FROM staff_master;
      end Open_Staff_Cur;
END Staff_Data;
```

- **Note: Cursor Variable as the formal parameter should be in IN OUT mode.**

Subprograms and Ref Type Cursors

- **Execution in SQL*PLUS:**
  - Step 1: Declare a bind variable in a PL/SQL host environment of type REFCURSOR.

```
SQL> VARIABLE cv REFCURSOR
```

  - Step 2: SET AUTOPRINT ON to automatically display the query results.

```
SQL> set autoprint on
```

Subprograms and Ref Type Cursors

- Step 3: Execute the package with the specified procedure along with the cursor as follows:

```
SQL> execute Staff_Data.Open_Staff_Cur(:cv);
```

Subprograms and Ref Type Cursors

- **Passing a Cursor Variable as IN parameter to a stored procedure:**
  - Step 1: Create a Package Specification

```
CREATE OR REPLACE PACKAGE StaffData AS
        TYPE cur_type is REF CURSOR;
        TYPE staffcurtyp is REF CURSOR
        return staff%rowtype;
        PROCEDURE Ret_Data (staff_cur INOUT staffcurtyp,
        choice in number);
 END StaffData;
```

Subprograms and Ref Type Cursors

- Step 2: Create a Package Body:

```
CREATE OR REPLACE PACKAGE BODY StaffData AS
   PROCEDURE Ret_Data (staff_cur INOUT staffcurtyp,
        choice IN number) is
BEGIN
   IF choice = 1 THEN
            OPEN staff_cur for select * FROM staff_master
              WHERE staff_dob is not null;
   ELSIF choice = 2 THEN
             OPEN staff_cur for SELECT * FROM staff_master
          WHERE staff_sal > 2500;
```

Subprograms and Ref Type Cursors

- **Step 2 (contd.):**

```
        ELSIF choice = 3 THEN
                    OPEN staff_cur for SELECT * FROM
          staff_master WHERE dept_code = 20;
        END IF;
 END Ret_Data;
END StaffData;
```

Oracle for Developers (PL/SQL)

Built-in Packages in Oracle

DBMS_OUTPUT package

- **PL/SQL has no input/output capability**
- **However, built-in package DBMS_OUTPUT is provided to generate reports**
- **The procedure PUT_LINE is also provided that places the contents in the buffer**

PUT_LINE (VARCHAR2 OR NUMBER OR DATE)

Displaying Output

- **Syntax:**

```
SQL>SET SERVEROUTPUT ON
DECLARE
 V_Variable VARCHAR2(25) :=' Used for'
|| 'Debugging ';
BEGIN
DBMS_OUTPUT.PUT_LINE(V_Variable);
END;
```

DBMS_OUTPUT - Example

- **In this example, the following anonymous PL/SQL block uses DBMS_OUTPUT to display the name and salary of each staff member in department 10:**

```
DECLARE
      CURSOR emp_cur IS SELECT staff_name, staff_sal
       FROM staff_master  WHERE dept_code = 10
            ORDER BY staff_sal DESC;
BEGIN FOR emp_rec IN emp_cur
      LOOP
           DBMS_OUTPUT.PUT_LINE ('Employee ' ||
           emp_rec.staff_name || ' earns ' ||
            TO_CHAR (emp_rec.staff_sal) || 'rupees');
      END LOOP;
      END;
```

DBMS_LOB

- **Handling LOBs (Large Objects)**
  - Large Objects (LOBs) are a set of datatypes that are designed to hold large amounts of data.
  - LOBs are designed to support Unstructured kind of data.
  - In short:
    - LOBs are used to store Large Objects (LOBs).
    - LOBs support random access to data and has maximum size of 4 GB
    - For example: Hospital database

Types of LOBs

```
SQL> Create table Leave
  2  (Empno number(4),
  3   S_date date,
  4   E_date date,
  5   snap   blob,
  6   msg    clob);
Table created
```

Oracle for Developers (PL/SQL)

Database Triggers

Concept of Database Triggers

- **Database Triggers:**
  - Database Triggers are procedures written in PL/SQL, Java, or C that run (fire) implicitly:
    - whenever a table or view is modified, or
    - when some user actions or database system actions occur
  - They are stored subprograms.

Concept of Database Triggers

- You can write triggers that fire whenever one of the following operations occur:
  - User events:
    - DML statements on a particular schema object
    - DDL statements issued within a schema or database
    - user logon or logoff events
  - System events:
    - server errors
    - database startup
    - instance shutdown

Usage of Triggers

- Triggers can be used for:
  - maintaining complex integrity constraints.
  - auditing information, that is the Audit trail.
  - automatically signaling other programs that action needs to take place when changes are made to a table.

Syntax of Triggers

- **Syntax:**

```
CREATE TRIGGER Trg_Name
{BEFORE | AFTER} {event} OF Column_Names ON Table_Name

[FOR EACH ROW]
[WHEN restriction]
BEGIN
        PL/SQL statements;
END Trg_Name ;
```

Types of Triggers

- **Type of Trigger is determined by the triggering event, namely:**
  - INSERT
  - UPDATE
  - DELETE
- Triggers can be fired:
  - before or after the operation.
  - on row or statement operations.
- Trigger can be fired for more than one type of triggering statement.

More on Triggers

Category	Values	Comments
Statement	INSERT, DELETE, UPDATE	Defines which kind of DML statement causes the trigger to fire.
Timing	BEFORE, AFTER	Defines whether the trigger fires before the statement is executed or after the statement is executed.
Level	Row or Statement	• If the trigger is a row-level trigger, it fires once for each row affected by the triggering statement. • If the trigger is a statement-level trigger, it fires once, either before or after the statement. • A row-level trigger is identified by the FOR EACH ROW clause in the trigger definition.

Example 1

```
CREATE TABLE Account_log
(
deleteInfo VARCHAR2(20),
logging_date DATE
)
```

Trigger creation code

```
CREATE or REPLACE TRIGGER

After_Delete_Row_product

 AFTER   delete On Account_masters
 FOR EACH ROW
 BEGIN
 INSERT INTO  Account_log
 Values('After delete, Row level',sysdate);
 END;
```

Restrictions on Triggers

- **The use of Triggers has the following restrictions:**
  - Triggers should not issue transaction control statements (TCL) like COMMIT, SAVEPOINT.
  - Triggers cannot declare any long or long raw variables.
  - :new and :old cannot refer to a LONG datatype.

Disabling and Dropping Triggers

- **To disable a trigger:**

```
ALTER  TRIGGER Trigger_Name DISABLE/ENABLE
```

- **To drop a trigger (by using drop trigger command):**

```
DROP TRIGGER Trigger_Name
```

Order of Trigger Firing

- **Order of Trigger firing is arranged as:**
  - Execute the “before statement level” trigger.
  - For each row affected by the triggering statement:
    - Execute the “before row level” trigger.
    - Execute the statement.
    - Execute the “after row level” trigger.
  - Execute the “after statement level” trigger.

Using :Old & :New values in Triggers

Triggering statement	:Old	:New
INSERT	**Undefined – all fields are null.**	**Values that will be inserted when the statement is complete.**
UPDATE	**Original values for the row before the update.**	**New values that will be updated when the statement is complete.**
DELETE	**Original values before the row is deleted.**	**Undefined – all fields are NULL.**

- **Note:** They are valid only within row level triggers and not in statement level triggers.

Using WHEN clause

- **Use of WHEN clause is valid for row-level triggers only.**
- **Trigger body is executed for rows that meet the specified condition.**

Example 2

```
CREATE TABLE Account_masters
(
account_no NUMBER(6) PRIMARY KEY,
cust_id NUMBER(6) ,
account_type CHAR(3) CONSTRAINT chk_acc_type CHECK(account_type IN
('SAV','SAL')) ,
Ledger_balance NUMBER(10)
)
```