Oracle Advanced Topics

**What is PL/SQL?**

PL/SQL stands for Procedural Language extension of SQL.

PL/SQL is a combination of SQL along with the procedural features of programming languages.

It was developed by Oracle Corporation in the early 90’s to enhance the capabilities of SQL.

**The PL/SQL Engine:**

Oracle uses a PL/SQL engine to processes the PL/SQL statements.

A PL/SQL code can be stored in the client system (client-side) or in the database (server-side).

**A Simple PL/SQL Block:**

Each PL/SQL program consists of SQL and PL/SQL statements which from a PL/SQL block.

A PL/SQL Block consists of three sections:

* The Declaration section (optional).
* The Execution section (mandatory).
* The Exception (or Error) Handling section (optional).

**Declaration Section:**

* The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE.
* This section is optional and is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section.
* Placeholders may be any of Variables, Constants and Records, which stores data temporarily.
* Cursors are also declared in this section.

**Execution Section:**

* The Execution section of a PL/SQL Block starts with the reserved keyword BEGIN and ends with END.
* This is a mandatory section and is the section where the program logic is written to perform any task.
* The programmatic constructs like loops, conditional statement and SQL statements form the part of execution section.

**Exception Section:**

* The Exception section of a PL/SQL Block starts with the reserved keyword EXCEPTION.
* This section is optional. Any errors in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully.
* If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors.
* Every statement in the above three sections must end with a semicolon “ ; ” .
* PL/SQL blocks can be nested within other PL/SQL blocks. Comments can be used to document code.

This is how a sample PL/SQL Block looks.

**DECLARE**     Variable declaration  
**BEGIN**     Program Execution   
**EXCEPTION**     Exception handling  
**END;**

**Advantages of PL/SQL**

These are the advantages of PL/SQL.

* **Block Structures:**

PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module.

PL/SQL Blocks can be stored in the database and reused.

* **Procedural Language Capability:**

PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).

* **Better Performance:**

PL SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.

* **Error Handling:**

PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program. Once an exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.

**PL/SQL Placeholders**

* Placeholders are temporary storage area. Placeholders can be any of Variables, Constants and Records. Oracle defines placeholders to store data temporarily, which are used to manipulate data during the execution of a PL SQL block.
* Depending on the kind of data you want to store, you can define placeholders with a name and a datatype. Few of the datatypes used to define placeholders are as given below.

Number (n,m) , Char (n) , Varchar2 (n) , Date , Long , Long raw, Raw, Blob, Clob, Nclob, Bfile

**PL/SQL Variables**

These are placeholders that store the values that can change through the PL/SQL Block.

The General Syntax to declare a variable is:

VARIABLE\_NAME DATATYPE [NOT NULL := VALUE ];

* variable\_name is the name of the variable.
* datatype is a valid PL/SQL datatype.
* NOT NULL is an optional specification on the variable.
* Value or DEFAULT valueis also an optional specification, where you can initialize a variable.
* Each variable declaration is a separate statement and must be terminated by a semicolon.

**For example:**

If you want to store the current salary of an employee, you can use a variable.

**DECLARE**

**SALARY NUMBER (6);**

\* “salary” is a variable of datatype number and of length 6.

When a variable is specified as NOT NULL, you must initialize the variable when it is declared.

**For example:**

The below example declares two variables, one of which is a not null.

**DECLARE**

**SALARY NUMBER(4);**

**DEPT VARCHAR2(10) NOT NULL := “HR DEPT”;**

The value of a variable can change in the execution or exception section of the PL/SQL Block.

We can assign values to variables in the two ways given below.

1) We can directly assign values to variables.

The General Syntax is:

**VARIABLE\_NAME:= VALUE;**

2) We can assign values to variables directly from the database columns by using a SELECT.. INTO statement. The General Syntax is:

**SELECT COLUMN\_NAME**

**INTO VARIABLE\_NAME**

**FROM TABLE\_NAME**

**[WHERE CONDITION];**

**For Example:**

The below program will get the salary of an employee with id '1116' and display it on the screen.

**DECLARE**

**VAR\_SALARY NUMBER(6);**

**VAR\_EMP\_ID NUMBER(6) = 1116;**

**BEGIN**

**SELECT SALARY**

**INTO VAR\_SALARY**

**FROM EMPLOYEE**

**WHERE EMP\_ID = VAR\_EMP\_ID;**

**DBMS\_OUTPUT.PUT\_LINE(VAR\_SALARY);**

**DBMS\_OUTPUT.PUT\_LINE('THE EMPLOYEE ' || VAR\_EMP\_ID || ' HAS SALARY ' || VAR\_SALARY);**

**END;**

**/**

**NOTE:** The backward slash '/' in the above program indicates to execute the above PL/SQL Block.

**Scope of Variables**

* PL/SQL allows the nesting of Blocks within Blocks i.e, the Execution section of an outer block can contain inner blocks.
* Therefore, a variable which is accessible to an outer Block is also accessible to all nested inner Blocks.
* The variables declared in the inner blocks are not accessible to outer blocks.
* Based on their declaration we can classify variables into two types.
* **Local variables** - These are declared in a inner block and cannot be referenced by outside Blocks.
* **Global variables** - These are declared in a outer block and can be referenced by its itself and by its inner blocks.

**For Example:**

In the below example we are creating two variables in the outer block and assigning their product to the third variable created in the inner block.

The variable 'var\_mult' is declared in the inner block, so cannot be accessed in the outer block i.e.

It cannot be accessed after line 11. The variables 'var\_num1' and 'var\_num2' can be accessed anywhere in the block.

**DECLARE**

**VAR\_NUM1 NUMBER;**

**VAR\_NUM2 NUMBER;**

**BEGIN**

**VAR\_NUM1 := 100;**

**VAR\_NUM2 := 200;**

**DECLARE**

**VAR\_MULT NUMBER;**

**BEGIN**

**VAR\_MULT := VAR\_NUM1 \* VAR\_NUM2;**

**END;**

**END;**

**/**

**PL/SQL Constants**

* As the name implies a constant is a value used in a PL/SQL Block that remains unchanged throughout the program.
* A constant is a user-defined literal value.
* You can declare a constant and use it instead of actual value.

**For example:**

If you want to write a program which will increase the salary of the employees by 25%, you can declare a constant and use it throughout the program.

Next time when you want to increase the salary again you can change the value of the constant which will be easier than changing the actual value throughout the program.

The General Syntax to declare a constant is:

**CONSTANT\_NAME CONSTANT DATATYPE := VALUE;**

* Constant\_name is the name of the constant i.e. similar to a variable name.
* The word CONSTANT is a reserved word and ensures that the value does not change.
* VALUE - It is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

**For example:**

To declare salary\_increase, you can write code as follows:

**DECLARE**

**SALARY\_INCREASE CONSTANT NUMBER (3) := 10;**

You must assign a value to a constant at the time you declare it.

If you do not assign a value to a constant while declaring it and try to assign a value in the execution section, you will get a error.

If you execute the below Pl/SQL block you will get error.

**DECLARE**

**SALARY\_INCREASE CONSTANT NUMBER(3);**

**BEGIN**

**SALARY\_INCREASE := 100;**

**DBMS\_OUTPUT.PUT\_LINE (SALARY\_INCREASE);**

**END;**

**PL/SQL Records**

**What are records?**

* Records are another type of datatypes which oracle allows to be defined as a placeholder.
* Records are composite datatypes, which means it is a combination of different scalar datatypes like char, varchar, number etc.
* Each scalar data types in the record holds a value. A record can be visualized as a row of data.
* It can contain all the contents of a row.

**Declaring a record:**

To declare a record, you must first define a composite datatype; then declare a record for that type.   
The General Syntax to define a composite datatype is:

**TYPE RECORD\_TYPE\_NAME IS RECORD**

**(FIRST\_COL\_NAME COLUMN\_DATATYPE,**

**SECOND\_COL\_NAME COLUMN\_DATATYPE, ...);**

* *record\_type\_name* – it is the name of the composite type you want to define.
* *first\_col\_name, second\_col\_name, etc.,- it is the* names the fields/columns within the record.
* *column\_datatype* defines the scalar datatype of the fields.

There are different ways you can declare the datatype of the fields.

1. You can declare the field in the same way as you declare the fieds while creating the table.
2. If a field is based on a column from database table, you can define the field\_type as follows:

**COL\_NAME TABLE\_NAME.COLUMN\_NAME%TYPE**

By declaring the field datatype in the above method, the datatype of the column is dynamically applied to the field. This method is useful when you are altering the column specification of the table, because you do not need to change the code again.

**NOTE:** You can use also *%type* to declare variables and constants.   
The General Syntax to declare a record of a uer-defined datatype is:

**RECORD\_NAME RECORD\_TYPE\_NAME;**

The following code shows how to declare a record called *employee\_rec* based on a user-defined type.

**DECLARE**

**TYPE EMPLOYEE\_TYPE IS RECORD**

**(EMPLOYEE\_ID NUMBER(5),**

**EMPLOYEE\_FIRST\_NAME VARCHAR2(25),**

**EMPLOYEE\_LAST\_NAME EMPLOYEE.LAST\_NAME%TYPE,**

**EMPLOYEE\_DEPT EMPLOYEE.DEPT%TYPE);**

**EMPLOYEE\_SALARY EMPLOYEE.SALARY%TYPE;**

**EMPLOYEE\_REC EMPLOYEE\_TYPE;**

If all the fields of a record are based on the columns of a table, we can declare the record as follows:

**RECOR\_NAME TABLE\_NAME%ROWTYPE;**

For example, the above declaration of employee\_rec can as follows:

**DECLARE**

**EMPLOYEE\_REC EMPLOYEE%ROWTYPE;**

The advantages of declaring the record as a ROWTYPE are:

* You do not need to explicitly declare variables for all the columns in a table.
* If you alter the column specification in the database table, you do not need to update the code.

The disadvantage of declaring the record as a ROWTYPE is:

* When u create a record as a ROWTYPE, fields will be created for all the columns in the table and memory will be used to create the datatype for all the fields. So use ROWTYPE only when you are using all the columns of the table in the program.

**NOTE:** When you are creating a record, you are just creating a datatype, similar to creating a variable. You need to assign values to the record to use them.  
  
The following table consolidates the different ways in which you can define and declare a pl/sql record.

|  |  |
| --- | --- |
| **Syntax** | **Usage** |
| TYPE record\_type\_name IS RECORD  (column\_name1 datatype, column\_name2 datatype, ...); | Define a composite datatype, where each field is scalar. |
| col\_name table\_name.column\_name%type; | Dynamically define the datatype of a column based on a database column. |
| record\_name record\_type\_name; | Declare a record based on a user-defined type. |
| record\_name table\_name%ROWTYPE; | Dynamically declare a record based on an entire row  of a table. Each column in the table corresponds to a field in the record. |

**Passing Values To and From a Record**

When you assign values to a record, you actually assign values to the fields within it.   
The General Syntax to assign a value to a column within a record direclty is:

**RECORD\_NAME.COL\_NAME := VALUE;**

If you used %ROWTYPE to declare a record, you can assign values as shown:

**RECORD\_NAME.COLUMN\_NAME := VALUE;**

We can assign values to records using SELECT Statements as shown:

**SELECT COL1, COL2**

**INTO RECORD\_NAME.COL\_NAME1, RECORD\_NAME.COL\_NAME2**

**FROM TABLE\_NAME**

**[WHERE CLAUSE];**

If %ROWTYPE is used to declare a record then you can directly assign values to the whole record instead of each columns separately. In this case, you must SELECT all the columns from the table into the record as shown:

**SELECT \* INTO RECORD\_NAME**

**FROM TABLE\_NAME**

**[WHERE CLAUSE];**

Lets see how we can get values from a record.   
The General Syntax to retrieve a value from a specific field into another variable is:

|  |
| --- |
| **VAR\_NAME := RECORD\_NAME.COL\_NAME;** |

The following table consolidates the different ways you can assign values to and from a record:

|  |  |
| --- | --- |
| **Syntax** | **Usage** |
| record\_name.col\_name := value; | To directly assign a value to a specific column  of a record. |
| record\_name.column\_name := value; | To directly assign a value to a specific column  of a record, if the record is declared using %ROWTYPE. |
| SELECT col1, col2 INTO record\_name.col\_name1, record\_name.col\_name2 FROM table\_name  [WHERE clause]; | To assign values to each field of a record from the database table. |
| SELECT \* INTO record\_name FROM table\_name [WHERE clause]; | To assign a value to all fields in the record from  a database table. |
| variable\_name := record\_name.col\_name; | To get a value from a record column and assigning it to a variable. |

**Conditional Statements in PL/SQL**

As the name implies, PL/SQL supports programming language features like conditional statements, iterative statements.  
The programming constructs are similar to how you use in programming languages like Java and C++. In this section I will provide you syntax of how to use conditional statements in PL/SQL programming.

**IF THEN ELSE STATEMENT**

**Example:1**

**IF CONDITION**

**THEN**

**STATEMENT 1;**

**ELSE**

**STATEMENT 2;**

**END IF;**

**Example:2**

**IF CONDITION 1**

**THEN**

**STATEMENT 1;**

**STATEMENT 2;**

**ELSIF CONDTION2 THEN**

**STATEMENT 3;**

**ELSE**

**STATEMENT 4;**

**END IF;**

**Example:3**

**IF CONDITION 1**

**THEN**

**STATEMENT 1;**

**STATEMENT 2;**

**ELSIF CONDTION2 THEN**

**STATEMENT 3;**

**ELSE**

**STATEMENT 4;**

**END IF;**

**Example:4**

**IF CONDITION1 THEN**

**ELSE**

**IF CONDITION2 THEN**

**STATEMENT1;**

**END IF;**

**ELSIF CONDITION3 THEN**

**STATEMENT2;**

**END IF;**

**Iterative Statements in PL/SQL**

* An iterative control Statements are used when we want to repeat the execution of one or more statements for specified number of times. These are similar to those in

There are three types of loops in PL/SQL:

* Simple Loop
* While Loop
* For Loop

**Simple Loop**

* A Simple Loop is used when a set of statements is to be executed at least once before the loop terminates.
* An EXIT condition must be specified in the loop, otherwise the loop will get into an infinite number of iterations.
* When the EXIT condition is satisfied the process exits from the loop.

The General Syntax to write a Simple Loop is:

**LOOP**

**STATEMENTS;**

**EXIT;**

**{OR EXIT WHEN CONDITION;}**

**END LOOP;**

These are the important steps to be followed while using Simple Loop.

* Initialise a variable before the loop body.
* Increment the variable in the loop.
* Use a EXIT WHEN statement to exit from the Loop. If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once.

**While Loop**

* A WHILE LOOP is used when a set of statements has to be executed as long as a condition is true.
* The condition is evaluated at the beginning of each iteration. The iteration continues until the condition becomes false.

The General Syntax to write a WHILE LOOP is:

**WHILE <CONDITION>**

**LOOP STATEMENTS;**

**END LOOP;**

Important steps to follow when executing a while loop:

* Initialise a variable before the loop body.
* Increment the variable in the loop.
* EXIT WHEN statement and EXIT statements can be used in while loops but it's not done oftenly.

**FOR Loop**

* A FOR LOOP is used to execute a set of statements for a predetermined number of times.
* Iteration occurs between the start and end integer values given.
* The counter is always incremented by 1.
* The loop exits when the counter reachs the value of the end integer.

The General Syntax to write a FOR LOOP is:

**FOR COUNTER IN VAL1..VAL2**

**LOOP STATEMENTS;**

**END LOOP;**

* val1 - Start integer value.
* val2 - End integer value.

Important steps to follow when executing a while loop:

* The counter variable is implicitly declared in the declaration section, so it's not necessary to declare it explicity.
* The counter variable is incremented by 1 and does not need to be incremented explicitly.
* EXIT WHEN statement and EXIT statements can be used in FOR loops but it's not done oftenly.

**NOTE:** The above Loops are explained with a example when dealing with Explicit Cursors.

**What are Cursors?**

* A cursor is a temporary work area created in the system memory when a SQL statement is executed.
* A cursor contains information on a select statement and the rows of data accessed by it.
* This temporary work area is used to store the data retrieved from the database, and manipulate this data.
* A cursor can hold more than one row, but can process only one row at a time.
* The set of rows the cursor holds is called the active set.

There are two types of cursors in PL/SQL:

* **Implicit cursors:**
* These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed.
* They are also created when a SELECT statement that returns just one row is executed.
* **Explicit cursors:**
* They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row.
* When you fetch a row the current row position moves to next row.
* Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

**Implicit Cursors**

* When you execute DML statements like DELETE, INSERT, UPDATE and SELECT statements, implicit statements are created to process these statements.
* Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations.
* The cursor attributes available are %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.

**For example:**

* When you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected.
* When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement.
* PL/SQL returns an error when no data is selected.

The status of the cursor for each of these attributes are defined in the below table.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Return Value** | **Example** |
| %FOUND | The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECT ….INTO statement return at least one row. | SQL%FOUND |
| The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE do not affect row and if SELECT….INTO statement do not return a row. |
| %NOTFOUND | The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE at least one row and if SELECT ….INTO statement return at least one row. | SQL%NOTFOUND |
| The return value is TRUE, if a DML statement like INSERT, DELETE and UPDATE do not affect even one row and if SELECT ….INTO statement does not return a row. |
| %ROWCOUNT | Return the number of rows affected by the DML operations INSERT, DELETE, UPDATE, SELECT | SQL%ROWCOUNT |

**For Example:**

Consider the PL/SQL Block that uses implicit cursor attributes as shown below:

**DECLARE VAR\_ROWS NUMBER(5);**

**BEGIN**

**UPDATE EMPLOYEE**

**SET SALARY = SALARY + 1000;**

**IF SQL%NOTFOUND THEN**

**DBMS\_OUTPUT.PUT\_LINE('NONE OF THE SALARIES WHERE UPDATED');**

**ELSIF SQL%FOUND THEN**

**VAR\_ROWS := SQL%ROWCOUNT;**

**DBMS\_OUTPUT.PUT\_LINE('SALARIES FOR ' || VAR\_ROWS || 'EMPLOYEES ARE UPDATED');**

**END IF;**

**END;**

In the above PL/SQL Block, the salaries of all the employees in the ‘employee’ table are updated.

If none of the employee’s salary are updated we get a message 'None of the salaries where updated'. Else we get a message like for example, 'Salaries for 1000 employees are updated' if there are 1000 rows in ‘employee’ table.

**Explicit Cursors**

* An explicit cursor is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row. We can provide a suitable name for the cursor.

The General Syntax for creating a cursor is as given below:

**CURSOR CURSOR\_NAME IS SELECT\_STATEMENT;**

* cursor\_name – A suitable name for the cursor.
* select\_statement – A select query which returns multiple rows.

**How to use Explicit Cursor?**

There are four steps in using an Explicit Cursor.

* DECLARE the cursor in the declaration section.
* OPEN the cursor in the Execution Section.
* FETCH the data from cursor into PL/SQL variables or records in the Execution Section.
* CLOSE the cursor in the Execution Section before you end the PL/SQL Block.

Declaring a Cursor in the Declaration Section:

**DECLARE**

**CURSOR EMP\_CUR IS**

**SELECT \***

**FROM EMP\_TBL**

**WHERE SALARY > 5000;**

In the above example we are creating a cursor ‘emp\_cur’ on a query which returns the records of all the employees with salary greater than 5000. Here ‘emp\_tbl’ in the table which contains records of all the employees.

Accessing the records in the cursor:

Once the cursor is created in the declaration section we can access the cursor in the execution

section of the PL/SQL program.

**How to access an Explicit Cursor?**

These are the three steps in accessing the cursor.

* Open the cursor.
* Fetch the records in the cursor one at a time.
* Close the cursor.

General Syntax to open a cursor is:

**OPEN CURSOR\_NAME;**

General Syntax to fetch records from a cursor is:

**FETCH CURSOR\_NAME INTO RECORD\_NAME; OR**

**FETCH CURSOR\_NAME INTO VARIABLE\_LIST;**

General Syntax to close a cursor is:

**CLOSE CURSOR\_NAME;**

* When a cursor is opened, the first row becomes the current row.
* When the data is fetched it is copied to the record or variables and the logical pointer moves to the next row and it becomes the current row.
* On every fetch statement, the pointer moves to the next row.
* If you want to fetch after the last row, the program will throw an error.
* When there is more than one row in a cursor we can use loops along with explicit cursor attributes to fetch all the records.

Points to remember while fetching a row:

* We can fetch the rows in a cursor to a PL/SQL Record or a list of variables created in the PL/SQL Block.
* If you are fetching a cursor to a PL/SQL Record, the record should have the same structure as the cursor.
* If you are fetching a cursor to a list of variables, the variables should be listed in the same order in the fetch statement as the columns are present in the cursor.

General Form of using an explicit cursor is:

**DECLARE**

**VARIABLES;**

**RECORDS;**

**CREATE A CURSOR;**

**BEGIN**

**OPEN CURSOR;**

**FETCH CURSOR;**

**PROCESS THE RECORDS;**

**CLOSE CURSOR;**

**END;**

Lets Look at the example below

Example 1:

**DECLARE**

**EMP\_REC EMP\_TBL%ROWTYPE;**

**CURSOR EMP\_CUR IS**

**SELECT \***

**FROM**

**WHERE SALARY > 10;**

**BEGIN**

**OPEN EMP\_CUR;**

**FETCH EMP\_CUR INTO EMP\_REC;**

**DBMS\_OUTPUT.PUT\_LINE (EMP\_REC.FIRST\_NAME || ' ' || EMP\_REC.LAST\_NAME);**

**CLOSE EMP\_CUR;**

**END;**

In the above example, first we are creating a record ‘emp\_rec’ of the same structure as of table ‘emp\_tbl’ in line no 2. We can also create a record with a cursor by replacing the table name with the cursor name. Second, we are declaring a cursor ‘emp\_cur’ from a select query in line no 3 - 6. Third, we are opening the cursor in the execution section in line no 8. Fourth, we are fetching the cursor to the record in line no 9. Fifth, we are displaying the first\_name and last\_name of the employee in the record emp\_rec in line no 10. Sixth, we are closing the cursor in line no 11.

**What are Explicit Cursor Attributes?**

* Oracle provides some attributes known as Explicit Cursor Attributes to control the data processing while using cursors.
* We use these attributes to avoid errors while accessing cursors through OPEN, FETCH and CLOSE Statements.

**When does an error occur while accessing an explicit cursor?**

* When we try to open a cursor which is not closed in the previous operation.
* When we try to fetch a cursor after the last operation.

These are the attributes available to check the status of an explicit cursor.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Return values** | **Example** |
| %FOUND | TRUE, if fetch statement returns at least one row. | Cursor\_name%FOUND |
| FALSE, if fetch statement doesn’t return a row. |
| %NOTFOUND | TRUE, , if fetch statement doesn’t return a row. | Cursor\_name%NOTFOUND |
| FALSE, if fetch statement returns at least one row. |

|  |  |  |
| --- | --- | --- |
| %ROWCOUNT | The number of rows fetched by the fetch statement | Cursor\_name%ROWCOUNT |
| If no row is returned, the PL/SQL statement returns an error. |
| %ISOPEN | TRUE, if the cursor is already open in the program | Cursor\_name%ISNAME |
| FALSE, if the cursor is not opened in the program. |

**Using Loops with Explicit Cursors**

* Oracle provides three types of cursors namely SIMPLE LOOP, WHILE LOOP and FOR LOOP.
* These loops can be used to process multiple rows in the cursor.
* Here I will modify the same example for each loops to explain how to use loops with cursors.

Cursor with a Simple Loop:

**DECLARE**

**CURSOR EMP\_CUR IS**

**SELECT FIRST\_NAME, LAST\_NAME, SALARY FROM EMP\_TBL;**

**EMP\_REC EMP\_CUR%ROWTYPE;**

**BEGIN**

**IF NOT SALES\_CUR%ISOPEN THEN**

**OPEN SALES\_CUR;**

**END IF;**

**LOOP**

**FETCH EMP\_CUR INTO EMP\_REC;**

**EXIT WHEN EMP\_CUR%NOTFOUND;**

**DBMS\_OUTPUT.PUT\_LINE(EMP\_CUR.FIRST\_NAME || ' ' ||EMP\_CUR.LAST\_NAME 13> || ' ' ||EMP\_CUR.SALARY);**

**END LOOP;**

**END;**

**/**

In the above example we are using two cursor attributes %ISOPEN and %NOTFOUND.

In line no 6, we are using the cursor attribute %ISOPEN to check if the cursor is open, if the condition is true the program does not open the cursor again, it directly moves to line no 9.

In line no 11, we are using the cursor attribute %NOTFOUND to check whether the fetch returned any row. If there is no rows found the program would exit, a condition which exists when you fetch the cursor after the last row, if there is a row found the program continues.

We can use %FOUND in place of %NOTFOUND and vice versa. If we do so, we need to reverse the logic of the program. So use these attributes in appropriate instances.

**Cursor with a While Loop**

Lets modify the above program to use while loop.

**DECLARE**

**CURSOR EMP\_CUR IS**

**SELECT FIRST\_NAME, LAST\_NAME, SALARY FROM EMP\_TBL;**

**EMP\_REC EMP\_CUR%ROWTYPE;**

**BEGIN**

**IF NOT SALES\_CUR%ISOPEN THEN**

**OPEN SALES\_CUR;**

**END IF;**

**FETCH SALES\_CUR INTO SALES\_REC;**

**WHILE SALES\_CUR%FOUND THEN**

**LOOP**

**DBMS\_OUTPUT.PUT\_LINE(EMP\_CUR.FIRST\_NAME || ' ' ||EMP\_CUR.LAST\_NAME 13> || ' ' ||EMP\_CUR.SALARY); FETCH SALES\_CUR INTO SALES\_REC;**

**END LOOP;**

**END;**

**/**

In the above example, in line no 10 we are using %FOUND to evaluate if the first fetch statement in line no 9 returned a row, if true the program moves into the while loop. In the loop we use fetch statement again (line no 15) to process the next row.

If the fetch statement is not executed once before the while loop the while condition will return false in the first instance and the while loop is skipped. In the loop, before fetching the record again, always process the record retrieved by the first fetch statement, else you will skip the first row.

**Cursor with a FOR Loop**

When using FOR LOOP you need not declare a record or variables to store the cursor values, need not open, fetch and close the cursor. These functions are accomplished by the FOR LOOP automatically.

**General Syntax for using FOR LOOP:**

**FOR RECORD\_NAME IN CUSROR\_NAME**

**LOOP**

**PROCESS THE ROW...**

**END LOOP;**

Let’s use the above example to learn how to use for loops in cursors.

**DECLARE**

**CURSOR EMP\_CUR IS**

**SELECT FIRST\_NAME, LAST\_NAME, SALARY FROM EMP\_TBL;**

**EMP\_REC EMP\_CUR%ROWTYPE;**

**BEGIN**

**FOR EMP\_REC IN SALES\_CUR**

**LOOP**

**DBMS\_OUTPUT.PUT\_LINE(EMP\_CUR.FIRST\_NAME || ' ' ||EMP\_CUR.LAST\_NAME|| ' ' ||EMP\_CUR.SALARY);**

**END LOOP;**

**END;**

**/**

In the above example, when the FOR loop is processed a record ‘emp\_rec’of structure ‘emp\_cur’ gets created, the cursor is opened, the rows are fetched to the record ‘emp\_rec’ and the cursor is closed after the last row is processed. By using FOR Loop in your program, you can reduce the number of lines in the program.

**NOTE:** In the examples given above, we are using backward slash ‘/’ at the end of the program. This indicates the oracle engine that the PL/SQL program has ended and it can begin processing the statements.

**Stored Procedures**

**What is a Stored Procedure?**

* A stored procedure or in simple a proc is a named PL/SQL block which performs one or more specific task.
* This is similar to a procedure in other programming languages.
* A procedure has a header and a body.
* The header consists of the name of the procedure and the parameters or variables passed to the procedure.
* The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.
* A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

We can pass parameters to procedures in three ways.

* IN-parameters
* OUT-parameters
* IN OUT-parameters

A procedure may or may not return any value.

General Syntax to create a procedure is:

**CREATE [OR REPLACE] PROCEDURE PROC\_NAME [LIST OF PARAMETERS]**

**IS**

**DECLARATION SECTION**

**BEGIN**

**EXECUTION SECTION**

**EXCEPTION**

**EXCEPTION SECTION**

**END;**

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

The below example creates a procedure ‘employer\_details’ which gives the details of the employee.

**CREATE OR REPLACE PROCEDURE EMPLOYER\_DETAILS**

**IS**

**CURSOR EMP\_CUR IS**

**SELECT FIRST\_NAME, LAST\_NAME, SALARY FROM EMP\_TBL;**

**EMP\_REC EMP\_CUR%ROWTYPE;**

**BEGIN**

**FOR EMP\_REC IN SALES\_CUR**

**LOOP**

**DBMS\_OUTPUT.PUT\_LINE(EMP\_CUR.FIRST\_NAME || ' ' ||EMP\_CUR.LAST\_NAME10> || ' ' ||EMP\_CUR.SALARY);**

**END LOOP;**

**END;**

**/**

**How to execute a Stored Procedure?**

There are two ways to execute a procedure.

1) From the SQL prompt.

**EXECUTE [OR EXEC] PROCEDURE\_NAME;**

2) Within another procedure – simply use the procedure name.

**PROCEDURE\_NAME;**

**NOTE:** In the examples given above, we are using backward slash ‘/’ at the end of the program. This indicates the oracle engine that the PL/SQL program has ended and it can begin processing the statements.

**PL/SQL Functions**

**What is a Function in PL/SQL?**

* A function is a named PL/SQL Block which is similar to a procedure.
* The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

The General Syntax to create a function is:

**CREATE [OR REPLACE] FUNCTION FUNCTION\_NAME [PARAMETERS]**

**RETURN RETURN\_DATATYPE;**

**IS**

**DECLARATION\_SECTION**

**BEGIN**

**EXECUTION\_SECTION**

**RETURN RETURN\_VARIABLE;**

**EXCEPTION**

**EXCEPTION SECTION**

**RETURN RETURN\_VARIABLE;**

**END;**

1) **Return Type:** The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.

2) The execution and exception section both should return a value which is of the datatype defined in the header section.

**For example:**

Let’s create a frunction called ''employer\_details\_func' similar to the one created in stored proc

**CREATE OR REPLACE FUNCTION EMPLOYER\_DETAILS\_FUNC**

**RETURN VARCHAR(20);**

**IS**

**EMP\_NAME VARCHAR(20);**

**BEGIN**

**SELECT FIRST\_NAME INTO EMP\_NAME**

**FROM EMP\_TBL WHERE EMPID = '100';**

**RETURN EMP\_NAME;**

**END;**

**/**

In the example we are retrieving the ‘first\_name’ of employee with empID 100 to variable ‘emp\_name’.

The return type of the function is VARCHAR which is declared in line no 2.

The function returns the 'emp\_name' which is of type VARCHAR as the return value in line no 9.

**How to execute a PL/SQL Function?**

A function can be executed in the following ways.

* Since a function returns a value we can assign it to a variable.

**EMPLOYEE\_NAME := EMPLOYER\_DETAILS\_FUNC;**

If ‘employee\_name’ is of datatype varchar we can store the name of the employee by assigning the return type of the function to it.

* As a part of a SELECT statement

**SELECT EMPLOYER\_DETAILS\_FUNC FROM DUAL;**

* In a PL/SQL Statements like,

**DBMS\_OUTPUT.PUT\_LINE(EMPLOYER\_DETAILS\_FUNC);**

This line displays the value returned by the function.

**Parameters in Procedure and Functions**

**How to pass parameters to Procedures and Functions in PL/SQL ?**

In PL/SQL, we can pass parameters to procedures and functions in three ways.

* IN type parameter: These types of parameters are used to send values to stored procedures.
* OUT type parameter: These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.
* IN OUT parameter: These types of parameters are used to send values and get values from stored procedures.

**NOTE:** If a parameter is not explicitly defined a parameter type, then by default it is an IN type parameter.

**IN parameter:**

* This is similar to passing parameters in programming languages.
* We can pass values to the stored procedure through these parameters or variables.
* This type of parameter is a read only parameter. We can assign the value of IN type parameter to a variable or use it in a query, but we cannot change its value inside the procedure.

The General syntax to pass a IN parameter is:

**CREATE [OR REPLACE] PROCEDURE PROCEDURE\_NAME (**

**PARAM\_NAME1 IN DATATYPE, PARAM\_NAME12 IN DATATYPE ... )**

* param\_name1, param\_name2... are unique parameter names.
* datatype - defines the datatype of the variable.
* IN - is optional, by default it is a IN type parameter.

**OUT Parameter:**

* The OUT parameters are used to send the OUTPUT from a procedure or a function.
* This is a write-only parameter i.e, we cannot pass values to OUT paramters while executing the stored procedure, but we can assign values to OUT parameter inside the stored procedure and the calling program can recieve this output value.

The General syntax to create an OUT parameter is:

**CREATE [OR REPLACE] PROCEDURE PROC2 (PARAM\_NAME OUT DATATYPE)**

The parameter should be explicity declared as OUT parameter.

**IN OUT Parameter:**

* The IN OUT parameter allows us to pass values into a procedure and get output values from the procedure.
* This parameter is used if the value of the IN parameter can be changed in the calling program.
* By using IN OUT parameter we can pass values into a parameter and return a value to the calling program using the same parameter.
* But this is possible only if the value passed to the procedure and output value have a same datatype. This parameter is used if the value of the parameter will be changed in the procedure.

The General syntax to create an IN OUT parameter is:

**CREATE [OR REPLACE] PROCEDURE PROC3 (PARAM\_NAME IN OUT DATATYPE)**

The below examples show how to create stored procedures using the above three types of parameters.

**For Example:**

Using IN and OUT parameter:

Let’s create a procedure which gets the name of the employee when the employee id is passed.

**CREATE OR REPLACE PROCEDURE EMP\_NAME (ID IN NUMBER, EMP\_NAME OUT NUMBER)**

**IS**

**BEGIN**

**SELECT FIRST\_NAME INTO EMP\_NAME**

**FROM EMP\_TBL WHERE EMPID = ID;**

**END;**

**/**

We can call the procedure ‘emp\_name’ in this way from a PL/SQL Block.

**DECLARE**

**EMPNAME VARCHAR(20);**

**CURSOR ID\_CUR SELECT ID FROM EMP\_IDS;**

**BEGIN5> FOR EMP\_REC IN ID\_CUR6> LOOP**

**EMP\_NAME(EMP\_REC.ID, EMPNAME);**

**DBMS\_OUTPUT.PUTLINE('THE EMPLOYEE ' || EMPNAME || ' HAS ID ' || EMP-REC.ID);**

**END LOOP;**

**END;**

**/**

In the above PL/SQL Block

In line no 3; we are creating a cursor ‘id\_cur’ which contains the employee id.

In line no 7; we are calling the procedure ‘emp\_name’, we are passing the ‘id’ as IN parameter and ‘empName’ as OUT parameter.

In line no 8; we are displaying the id and the employee name which we got from the procedure ‘emp\_name’.

**For Example:**

**Using IN OUT parameter in procedures:**

**CREATE OR REPLACE PROCEDURE EMP\_SALARY\_INCREASE**

**(EMP\_ID IN EMPTBL.EMPID%TYPE, SALARY\_INC IN OUT EMPTBL.SALARY%TYPE)**

**IS**

**TMP\_SAL NUMBER;**

**BEGIN**

**SELECT SALARY**

**INTO TMP\_SAL**

**FROM EMP\_TBL**

**WHERE EMPID = EMP\_ID;**

**IF TMP\_SAL BETWEEN 10000 AND 20000 THEN**

**SALARY\_INOUT := TMP\_SAL \* 1.2;**

**ELSIF TMP\_SAL BETWEEN 20000 AND 30000 THEN**

**SALARY\_INOUT := TMP\_SAL \* 1.3;**

**ELSIF TMP\_SAL > 30000 THEN**

**SALARY\_INOUT := TMP\_SAL \* 1.4;**

**END IF;**

**END;**

**/**

The below PL/SQL block shows how to execute the above 'emp\_salary\_increase' procedure.

**DECLARE**

**CURSOR UPDATED\_SAL IS**

**SELECT EMPID,SALARY**

**FROM EMP\_TBL;**

**PRE\_SAL NUMBER;**

**BEGIN**

**FOR EMP\_REC IN UPDATED\_SAL LOOP**

**PRE\_SAL := EMP\_REC.SALARY;**

**EMP\_SALARY\_INCREASE(EMP\_REC.EMPID, EMP\_REC.SALARY);**

**DBMS\_OUTPUT.PUT\_LINE('THE SALARY OF '|| EMP\_REC.EMPID||' INCREASED FROM '|| PRE\_SAL || ' TO '||EMP\_REC.SALARY);**

**END LOOP;**

**END;**

**/**

**Exception Handling**

In this section we will discuss about the following,

* What is Exception Handling.
* Structure of Exception Handling.
* Types of Exception Handling.

**What is Exception Handling?**

* PL/SQL provides a feature to handle the Exceptions which occur in a PL/SQL Block known as exception Handling.
* Using Exception Handling we can test the code and avoid it from exiting abruptly.
* When an exception occurs a messages which explains its cause is recieved.

PL/SQL Exception message consists of three parts.

* Type of Exception
* An Error Code
* A message

By Handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

**Structure of Exception Handling.**

The General Syntax for coding the exception section

**DECLARE**

**DECLARATION SECTION**

**BEGIN**

**EXCEPTION SECTION**

**EXCEPTION**

**WHEN EX\_NAME1 THEN**

**-ERROR HANDLING STATEMENTS**

**WHEN EX\_NAME2 THEN**

**-ERROR HANDLING STATEMENTS**

**WHEN OTHERS THEN**

**-ERROR HANDLING STATEMENTS**

**END;**

General PL/SQL statments can be used in the Exception Block.

When an exception is raised, Oracle searches for an appropriate exception handler in the exception section.

For example in the above example, if the error raised is 'ex\_name1 ', then the error is handled according to the statements under it.

Since, it is not possible to determine all the possible runtime errors during testing fo the code, the 'WHEN Others' exception is used to manage the exceptions that are not explicitly handled.

Only one exception can be raised in a Block and the control does not return to the Execution Section after the error is handled.

If there are nested PL/SQL blocks like this.

**DELCARE**

**DECLARATION SECTION**

**BEGIN**

**DECLARE**

**DECLARATION SECTION**

**BEGIN**

**EXECUTION SECTION**

**EXCEPTION**

**EXCEPTION SECTION**

**END;**

**EXCEPTION**

**EXCEPTION SECTION**

**END;**

In the above case, if the exception is raised in the inner block it should be handled in the exception block of the inner PL/SQL block else the control moves to the Exception block of the next upper PL/SQL Block. If none of the blocks handle the exception the program ends abruptly with an error.

**Types of Exception.**

There are 3 types of Exceptions.

* Named System Exceptions
* Unnamed System Exceptions
* User-defined Exceptions

**Named System Exceptions**

* System exceptions are automatically raised by Oracle, when a program violates a RDBMS rule.
* There are some system exceptions which are raised frequently, so they are pre-defined and given a name in Oracle which are known as Named System Exceptions.

**For example:** NO\_DATA\_FOUND and ZERO\_DIVIDE are called Named System exceptions.

Named system exceptions are:

* Not Declared explicitly
* Raised implicitly when a predefined Oracle error occurs
* caught by referencing the standard name within an exception-handling routine.

|  |  |  |
| --- | --- | --- |
| **Exception Name** | **Reason** | **Error Number** |
| CURSOR\_ALREADY\_OPEN | When you open a cursor that is already open. | ORA-06511 |
| INVALID\_CURSOR | When you perform an invalid operation on a cursor like closing a cursor, fetch data from a cursor that is not opened. | ORA-01001 |
| NO\_DATA\_FOUND | When a SELECT...INTO clause does not return any row from a table. | ORA-01403 |
| TOO\_MANY\_ROWS | When you SELECT or fetch more than one row into a record or variable. | ORA-01422 |
| ZERO\_DIVIDE | When you attempt to divide a number by zero. | ORA-01476 |

**For Example**: Suppose a NO\_DATA\_FOUND exception is raised in a proc, we can write a code to handle the exception as given below.

**BEGIN**

**EXECUTION SECTION**

**EXCEPTION**

**WHEN NO\_DATA\_FOUND THEN**

**DBMS\_OUTPUT.PUT\_LINE ('A SELECT...INTO DID NOT RETURN ANY ROW.');**

**END;**

**Unnamed System Exceptions**

* Those system exception for which oracle does not provide a name is known as unamed system exception.
* These exception do not occur frequently. These Exceptions have a code and an associated message.

There are two ways to handle unnamed sysyem exceptions:

* By using the WHEN OTHERS exception handler, or
* By associating the exception code to a name and using it as a named exception.

We can assign a name to unnamed system exceptions using a **Pragma** called **EXCEPTION\_INIT**.

**EXCEPTION\_INIT** will associate a predefined Oracle error number to a programmer\_defined exception name.

Steps to be followed to use unnamed system exceptions are

* They are raised implicitly.
* If they are not handled in WHEN Others they must be handled explicity.
* To handle the exception explicity, they must be declared using Pragma EXCEPTION\_INIT as given above and handled referecing the user-defined exception name in the exception section.

The general syntax to declare unnamed system exception using EXCEPTION\_INIT is:

**DECLARE**

**EXCEPTION\_NAME**

**EXCEPTION;**

**PRAGMA**

**EXCEPTION\_INIT (EXCEPTION\_NAME, ERR\_CODE);**

**BEGIN**

**EXECUTION SECTION**

**EXCEPTION**

**WHEN EXCEPTION\_NAME THEN**

**HANDLE THE EXCEPTION**

**END;**

**For Example:**

Lets consider the product table and order\_items table from sql joins.

Here product\_id is a primary key in product table and a foreign key in order\_items table.

If we try to delete a product\_id from the product table when it has child records in order\_id table an exception will be thrown with oracle code number -2292.

We can provide a name to this exception and handle it in the exception section as given below.

**DECLARE**

**CHILD\_REC\_EXCEPTION EXCEPTION;**

**PRAGMA**

**EXCEPTION\_INIT (CHILD\_REC\_EXCEPTION, -2292);**

**BEGIN**

**DELETE FROM PRODUCT WHERE PRODUCT\_ID= 104;**

**EXCEPTION**

**WHEN CHILD\_REC\_EXCEPTION**

**THEN DBMS\_OUTPUT.PUT\_LINE('CHILD RECORDS ARE PRESENT FOR THIS PRODUCT\_ID.');**

**END;**

**/**

**User-defined Exceptions**

* Apart from sytem exceptions we can explicity define exceptions based on business rules.
* These are known as user-defined exceptions.

Steps to be followed to use user-defined exceptions:

* They should be explicitly declared in the declaration section.
* They should be explicitly raised in the Execution Section.
* They should be handled by referencing the user-defined exception name in the exception section.

**For Example:**

Lets consider the product table and order\_items table from sql joins to explain user-defined exception.

Lets create a business rule that if the total no of units of any particular product sold is more than 20, then it is a huge quantity and a special discount should be provided.

**DECLARE**

**HUGE\_QUANTITY EXCEPTION;**

**CURSOR PRODUCT\_QUANTITY IS**

**SELECT P.PRODUCT\_NAME AS NAME, SUM(O.TOTAL\_UNITS) AS UNITS**

**FROM ORDER\_TEMS O, PRODUCT P**

**WHERE O.PRODUCT\_ID = P.PRODUCT\_ID;**

**QUANTITY ORDER\_TEMS.TOTAL\_UNITS%TYPE;**

**UP\_LIMIT CONSTANT ORDER\_TEMS.TOTAL\_UNITS%TYPE := 20;**

**MESSAGE VARCHAR2(50);**

**BEGIN**

**FOR PRODUCT\_REC IN PRODUCT\_QUANTITY LOOP**

**QUANTITY := PRODUCT\_REC.UNITS;**

**IF QUANTITY > UP\_LIMIT THEN**

**MESSAGE := 'THE NUMBER OF UNITS OF PRODUCT ' || PRODUCT\_REC.NAME || ' IS MORE THAN 20. SPECIAL DISCOUNTS SHOULD BE PROVIDED. REST OF THE RECORDS ARE SKIPPED. '**

**RAISE HUGE\_QUANTITY;**

**ELSIF QUANTITY < UP\_LIMIT THEN**

**V\_MESSAGE:= 'THE NUMBER OF UNIT IS BELOW THE DISCOUNT LIMIT.';**

**END IF;**

**DBMS\_OUTPUT.PUT\_LINE (MESSAGE);**

**END LOOP;**

**EXCEPTION**

**WHEN HUGE\_QUANTITY THEN**

**DBMS\_OUTPUT.PUT\_LINE (MESSAGE);**

**END;**

**/**

**RAISE\_APPLICATION\_ERROR ( )**

**RAISE\_APPLICATION\_ERROR** is a built-in procedure in oracle which is used to display the user-defined error messages along with the error number whose range is in between -20000 and -20999.

Whenever a message is displayed using RAISE\_APPLICATION\_ERROR, all previous transactions which are not committed within the PL/SQL Block are rolled back automatically (i.e. change due to INSERT, UPDATE, or DELETE statements).

RAISE\_APPLICATION\_ERROR raises an exception but does not handle it.

RAISE\_APPLICATION\_ERROR is used for the following reasons.

* To create a unique id for an user-defined exception.
* To make the user-defined exception look like an Oracle error.

The General Syntax to use this procedure is:

**RAISE\_APPLICATION\_ERROR (ERROR\_NUMBER, ERROR\_MESSAGE);**

* The Error number must be between -20000 and -20999
* The Error\_message is the message you want to display when the error occurs.

Steps to be folowed to use RAISE\_APPLICATION\_ERROR procedure:

* Declare a user-defined exception in the declaration section.
* Raise the user-defined exception based on a specific business rule in the execution section.
* Finally, catch the exception and link the exception to a user-defined error number in RAISE\_APPLICATION\_ERROR.

Using the above example we can display a error message using RAISE\_APPLICATION\_ERROR.

**DECLARE**

**HUGE\_QUANTITY EXCEPTION;**

**CURSOR PRODUCT\_QUANTITY IS**

**SELECT P.PRODUCT\_NAME AS NAME, SUM(O.TOTAL\_UNITS) AS UNITS**

**FROM ORDER\_TEMS O, PRODUCT P**

**WHERE O.PRODUCT\_ID = P.PRODUCT\_ID;**

**QUANTITY ORDER\_TEMS.TOTAL\_UNITS%TYPE;**

**UP\_LIMIT CONSTANT ORDER\_TEMS.TOTAL\_UNITS%TYPE := 20;**

**MESSAGE VARCHAR2(50);**

**BEGIN**

**FOR PRODUCT\_REC IN PRODUCT\_QUANTITY LOOP**

**QUANTITY := PRODUCT\_REC.UNITS;**

**IF QUANTITY > UP\_LIMIT THEN**

**RAISE HUGE\_QUANTITY;**

**ELSIF QUANTITY < UP\_LIMIT THEN**

**V\_MESSAGE:= 'THE NUMBER OF UNIT IS BELOW THE DISCOUNT LIMIT.';**

**END IF;**

**DBMS\_OUTPUT.PUT\_LINE (MESSAGE);**

**END LOOP;**

**EXCEPTION**

**WHEN HUGE\_QUANTITY THEN**

**RAISE\_APPLICATION\_ERROR(-2100, 'THE NUMBER OF UNIT IS ABOVE THE DISCOUNT LIMIT.');**

**END;**

**/**

**What is a Trigger?**

* A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table.
* A trigger is triggered automatically when an associated DML statement is executed.

Syntax of Triggers

The Syntax for creating a trigger is:

**CREATE [OR REPLACE ] TRIGGER TRIGGER\_NAME**

**{BEFORE | AFTER | INSTEAD OF }**

**{INSERT [OR] | UPDATE [OR] | DELETE}**

**[OF COL\_NAME]**

**ON TABLE\_NAME**

**[REFERENCING OLD AS O NEW AS N]**

**[FOR EACH ROW]**

**WHEN (CONDITION)**

**BEGIN**

**--- SQL STATEMENTS**

**END;**

* **CREATE [OR REPLACE ] TRIGGER trigger\_name** - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
* **{BEFORE | AFTER | INSTEAD OF }** - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
* **{INSERT [OR] | UPDATE [OR] | DELETE}** - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
* **[OF col\_name]** - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
* **CREATE [OR REPLACE ] TRIGGER trigger\_name** - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
* **[ON table\_name]** - This clause identifies the name of the table or view to which the trigger is associated.
* **[REFERENCING OLD AS o NEW AS n]** - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column\_name or :new.column\_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.
* **[FOR EACH ROW]** - This clause is used to determine whether a trigger must fire when each row gets affected ( i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).
* **WHEN (condition)** - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

**For Example:**

The price of a product changes constantly. It is important to maintain the history of the prices of the products.

We can create a trigger to update the 'product\_price\_history' table when the price of the product is updated in the 'product' table.

1) Create the 'product' table and 'product\_price\_history' table

**CREATE TABLE PRODUCT\_PRICE\_HISTORY**

**(PRODUCT\_ID NUMBER(5),**

**PRODUCT\_NAME VARCHAR2(32),**

**SUPPLIER\_NAME VARCHAR2(32),**

**UNIT\_PRICE NUMBER(7,2) );**

**CREATE TABLE PRODUCT**

**(PRODUCT\_ID NUMBER(5),**

**PRODUCT\_NAME VARCHAR2(32),**

**SUPPLIER\_NAME VARCHAR2(32),**

**UNIT\_PRICE NUMBER(7,2) );**

2) Create the price\_history\_trigger and execute it.

**CREATE OR REPLACE TRIGGER PRICE\_HISTORY\_TRIGGER**

**BEFORE UPDATE OF UNIT\_PRICE**

**ON PRODUCT**

**FOR EACH ROW**

**BEGIN**

**INSERT INTO PRODUCT\_PRICE\_HISTORY VALUES (:OLD.PRODUCT\_ID, :OLD.PRODUCT\_NAME, :OLD.SUPPLIER\_NAME, :OLD.UNIT\_PRICE);**

**END;**

**/**

3) Lets update the price of a product.

**UPDATE PRODUCT SET UNIT\_PRICE = 800 WHERE PRODUCT\_ID = 100**

Once the above update query is executed, the trigger fires and updates the 'product\_price\_history' table.

4) If you ROLLBACK the transaction before committing to the database, the data inserted to the table is also rolled back.

**Types of PL/SQL Triggers**

There are two types of triggers based on the which level it is triggered.

* Row level trigger - An event is triggered for each row upated, inserted or deleted.
* Statement level trigger - An event is triggered for each sql statement executed.

**PL/SQL Trigger Execution Hierarchy**

The following hierarchy is followed when a trigger is fired.

* BEFORE statement trigger fires first.
* Next BEFORE row level trigger fires, once for each row affected.
* Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.
* Finally the AFTER statement level trigger fires.

**For Example:**

Let's create a table 'product\_check' which we can use to store messages when triggers are fired.

**CREATE TABLE PRODUCT**

**(MESSAGE VARCHAR2(50),**

**CURRENT\_DATE NUMBER(32)**

**);**

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

**1) BEFORE UPDATE, Statement Level:** This trigger will insert a record into the table 'product\_check' before a sql update statement is executed, at the statement level.

**CREATE OR REPLACE TRIGGER BEFORE\_UPDATE\_STAT\_PRODUCT**

**BEFORE**

**UPDATE ON PRODUCT**

**BEGIN**

**INSERT INTO PRODUCT\_CHECK**

**VALUES('BEFORE UPDATE, STATEMENT LEVEL',SYSDATE);**

**END;**

**/**

**2) BEFORE UPDATE, Row Level:** This trigger will insert a record into the table 'product\_check' before each row is updated.

**CREATE OR REPLACE TRIGGER BEFORE\_UPDDATE\_ROW\_PRODUCT**

**BEFORE**

**UPDATE ON PRODUCT**

**FOR EACH ROW**

**BEGIN**

**INSERT INTO PRODUCT\_CHECK VALUES('BEFORE UPDATE ROW LEVEL',SYSDATE);**

**END;**

**/**

**3) AFTER UPDATE, Statement Level:** This trigger will insert a record into the table 'product\_check' after a sql update statement is executed, at the statement level.

**CREATE OR REPLACE TRIGGER AFTER\_UPDATE\_STAT\_PRODUCT**

**AFTER**

**UPDATE ON PRODUCT**

**BEGIN**

**INSERT INTO PRODUCT\_CHECK**

**VALUES('AFTER UPDATE, STATEMENT LEVEL', SYSDATE);**

**END;**

**/**

**4) AFTER UPDATE, Row Level:** This trigger will insert a record into the table 'product\_check' after each row is updated.

**CREATE OR REPLACE TRIGGER AFTER\_UPDATE\_ROW\_PRODUCT**

**AFTER**

**INSERT ON PRODUCT**

**FOR EACH ROW**

**BEGIN**

**INSERT INTO PRODUCT\_CHECK**

**VALUES('AFTER UPDATE, ROW LEVEL',SYSDATE);**

**END;**

**/**

Now lets execute a update statement on table product.

**UPDATE PRODUCT SET UNIT\_PRICE = 800**

**WHERE PRODUCT\_ID IN (100,101);**

Lets check the data in 'product\_check' table to see the order in which the trigger is fired.

**SELECT \* FROM PRODUCT\_CHECK;**

**Output:**

Mesage Current\_Date

Before update, statement level 26-Nov-2008

Before update, row level 26-Nov-2008

After update, Row level 26-Nov-2008

Before update, row level 26-Nov-2008

After update, Row level 26-Nov-2008

After update, statement level 26-Nov-2008

The above result shows 'before update' and 'after update' row level events have occured twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per sql statement.

The above rules apply similarly for INSERT and DELETE statements.

**How to know Information about Triggers**

We can use the data dictionary view 'USER\_TRIGGERS' to obtain information about any trigger.

The below statement shows the structure of the view 'USER\_TRIGGERS'

**DESC USER\_TRIGGERS;**

NAME Type

TRIGGER\_NAME VARCHAR2(30)

TRIGGER\_TYPE VARCHAR2(16)

TRIGGER\_EVENT VARCHAR2(75)

TABLE\_OWNER VARCHAR2(30)

BASE\_OBJECT\_TYPE VARCHAR2(16)

TABLE\_NAME VARCHAR2(30)

COLUMN\_NAME VARCHAR2(4000)

REFERENCING\_NAMES VARCHAR2(128)

WHEN\_CLAUSE VARCHAR2(4000)

STATUS VARCHAR2(8)

DESCRIPTION VARCHAR2(4000)

ACTION\_TYPE VARCHAR2(11)

TRIGGER\_BODY LONG

13 rows selected.

This view stores information about header and body of the trigger.

**SELECT \* FROM USER\_TRIGGERS WHERE TRIGGER\_NAME = 'BEFORE\_UPDATE\_STAT\_PRODUCT';**

The above sql query provides the header and body of the trigger 'Before\_Update\_Stat\_product'.

You can drop a trigger using the following command.

**DROP TRIGGER TRIGGER\_NAME;**

**CYCLIC CASCADING in a TRIGGER**

* This is an undesirable situation where more than one trigger enter into an infinite loop.
* While creating a trigger we should ensure the such a situtation does not exist.

The below example shows how Trigger's can enter into cyclic cascading.

Let's consider we have two tables 'abc' and 'xyz'. Two triggers are created.

* The INSERT Trigger, triggerA on table 'abc' issues an UPDATE on table 'xyz'.
* The UPDATE Trigger, triggerB on table 'xyz' issues an INSERT on table 'abc'.

In such a situation, when there is a row inserted in table 'abc', triggerA fires and will update table 'xyz'.

When the table 'xyz' is updated, triggerB fires and will insert a row in table 'abc'.

This cyclic situation continues and will enter into a infinite loop, which will crash the database.

**Hierarchical queries**

* A relational database does not store data in a hierarchical way.
* Then how do I get the data in a hierarchical manner? Here we get to know about how to use the hierarchical querying feature which Oracle has given.
* How you can interpret the hierarchical query conceptually and build hierarchical queries catering your needs.
* Using hierarchical queries, you can retrieve records from a table by their natural relationship.
* Be it a family tree or a employee/manager tree or what ever.
* Tree walking enables you to construct a hierarchical tree if the relationship lie in the same table.
* For instance, a manager column which exists in the emp table which defines the managerial hierarchy.

**For example:** The emp table in Scott schema.

**SELECT EMPNO,**

**ENAME,**

**JOB,**

**MGR,**

**HIREDATE**

**FROM EMP ;**

empno ename job mgr hiredate

7369 SMITH CLERK 7902 17-Dec-80

7499 ALLEN SALESMAN 7698 20-Feb-81

7521 WARD SALESMAN 7698 22-Feb-81

7566 JONES MANAGER 7839 02-Apr-81

7654 MARTIN SALESMAN 7698 28-Sep-81

7698 BLAKE MANAGER 7839 01-May-81

7782 CLARK MANAGER 7839 09-Jun-81

7788 SCOTT ANALYST 7566 19-Apr-87

7839 KING PRESIDENT 17-Nov-81

7844 TURNER SALESMAN 7698 08-Sep-81

7876 ADAMS CLERK 7788 23-May-87

7900 JAMES CLERK 7698 03-Dec-81

7902 FORD ANALYST 7566 03-Dec-81

7934 MILLER CLERK 7782 23-Jan-82

14 rows selected

If we have to query the employees reporting to King directly,

**SELECT EMPNO,**

**ENAME,**

**JOB,**

**MGR,**

**HIREDATE**

**FROM EMP**

**WHERE MGR = 7839**

7566 JONES MANAGER 7839 2-Apr-81

7698 BLAKE MANAGER 7839 1-May-81

7782 CLARK MANAGER 7839 9-Jun-81

But if we have to walk down the tree and check who all are reporting to Jones, Blake and Clark (recursively).

**SELECT EMPNO, ENAME,**

**JOB,**

**MGR,**

**HIREDATE**

**FROM EMP**

**START WITH MGR IS NULL**

**CONNECT BY PRIOR EMPNO = MGR;**

We will quickly see what are all the key words used in this query.

**START WITH** – Specifies the root rows of the hierarchy or in other words, where to start parsing from. This clause is necessary for true hierarchical queries

**CONNECT BY PRIOR** – This explains the relationship between the parent and the child.

**PRIOR** – This is used to achieve the recursive condition (The actual walking)

**If walking from top to bottom**

col\_1 is the parent Key(One which identifies the parent) and col\_2 is the child key (this identifies the child) And here it is

CONNECT BY PRIOR empno = mgr

**SELECT EMPNO,**

**ENAME,**

**JOB,**

**MGR,**

**HIREDATE,**

**LEVEL**

**FROM EMP**

**START WITH MGR IS NULL**

**CONNECT BY PRIOR EMPNO = MGR**

Gets me this result

7839 KING PRESIDENT 17-Nov-81 1

7566 JONES MANAGER 7839 02-Apr-81 2

7788 SCOTT ANALYST 7566 19-Apr-87 3

7876 ADAMS CLERK 7788 23-May-87 4

7902 FORD ANALYST 7566 03-Dec-81 3

7369 SMITH CLERK 7902 17-Dec-80 4

7698 BLAKE MANAGER 7839 01-May-81 2

7499 ALLEN SALESMAN 7698 20-Feb-81 3

7521 WARD SALESMAN 7698 22-Feb-81 3

7654 MARTIN SALESMAN 7698 28-Sep-81 3

7844 TURNER SALESMAN 7698 08-Sep-81 3

7900 JAMES CLERK 7698 03-Dec-81 3

7782 CLARK MANAGER 7839 09-Jun-81 2

7934 MILLER CLERK 7782 23-Jan-82 3

**If walking from bottom to top**

Col\_1 should be the child key and col\_2 should be the parent key

CONNECT BY PRIOR mgr = empno

**Using Level**

LEVEL psedo column shows the level or rank of the particular row in the hierarchical tree. If you see the below query, It shows the level of KING and the level of the guys reporting directly to him

**SELECT EMPNO,**

**ENAME,**

**JOB,**

**MGR,**

**HIREDATE,**

**LEVEL**

**FROM EMP**

**WHERE LEVEL <= 2**

**START WITH MGR IS NULL**

**CONNECT BY PRIOR EMPNO = MGR;**

empno ename job mgr hiredate level

7839 KING PRESIDENT 17-Nov-81 1

7566 JONES MANAGER 7839 02-Apr-81 2

7698 BLAKE MANAGER 7839 01-May-81 2

7782 CLARK MANAGER 7839 09-Jun-81 2

**Analytic Functions**

* Analytic functions compute an aggregate value based on a group of rows.
* They differ from aggregate functions in that they return multiple rows for each group.
* The group of rows is called a window and is defined by the analytic\_clause.
* For each row, a sliding window of rows is defined.
* The window determines the range of rows used to perform the calculations for the current row. Window sizes can be based on either a physical number of rows or a logical interval such as time.
* Analytic functions are the last set of operations performed in a query except for the final ORDER BY clause.
* All joins and all WHERE, GROUP BY, and HAVING clauses are completed before the analytic functions are processed.
* Therefore, analytic functions can appear only in the select list or ORDER BY clause.
* Analytic functions are commonly used to compute cumulative, moving, centered, and reporting aggregates.

The general syntax of analytic function is:

* Function(arg1,..., argn) OVER ( [PARTITION BY <...>] [ORDER BY <....>] [<window\_clause>] )
* <window\_clause> is like "ROW <?>" or "RANK <?>"
* All the keywords will be dealt in details as we walk through the examples.
* The script for creating the schema (SCOTT) on which the example queries of this article are run can be obtained in ORACLE\_HOME/sqlplus/demo/demobld.sql of any standard Oracle installation.

**How are analytic functions different from group or aggregate functions?**

**SELECT DEPTNO,**

**COUNT(\*) DEPT\_COUNT**

**FROM EMP**

**WHERE DEPTNO IN (20, 30)**

**GROUP BY DEPTNO;**

DEPTNO DEPT\_COUNT

20 5

30 6

2 rows selected

* Consider the above Query and its result. The Query returns departments and their employee count.
* Most importantly it groups the records into departments in accordance with the GROUP BY clause.
* As such any non-"group by" column is not allowed in the select clause.

**SELECT EMPNO, DEPTNO,**

**COUNT(\*) OVER (PARTITION BY**

**DEPTNO) DEPT\_COUNT**

**FROM EMP**

**WHERE DEPTNO IN (20, 30);**

EMPNO DEPTNO DEPT\_COUNT

---------- ---------- ----------

7369 20 5

7566 20 5

7788 20 5

7902 20 5

7876 20 5

7499 30 6

7900 30 6

7844 30 6

7698 30 6

7654 30 6

7521 30 6

11 rows selected.

* Now consider the analytic function in above query and its result. Note the repeating values of DEPT\_COUNT column.
* This brings out the main difference between aggregate and analytic functions.
* Though analytic functions give aggregate result they do not group the result set.
* They return the group value multiple times with each record.
* As such any other non-"group by" column or expression can be present in the select clause, for example, the column EMPNO in above Query.
* Analytic functions are computed after all joins, WHERE clause, GROUP BY and HAVING are computed on the query. The main ORDER BY clause of the query operates after the analytic functions.
* So analytic functions can only appear in the select list and in the main ORDER BY clause of the query.
* In absence of any PARTITION or <window\_clause> inside the OVER( ) portion, the function acts on entire record set returned by the where clause.
* Note the results of below Query and compare it with the result of aggregate function query.

**SELECT EMPNO, DEPTNO,**

**COUNT(\*) OVER ( ) CNT**

**FROM EMP**

**WHERE DEPTNO IN (10, 20)**

**ORDER BY 2, 1;**

EMPNO DEPTNO CNT

7782 10 8

7839 10 8

7934 10 8

7369 20 8

7566 20 8

7788 20 8

7876 20 8

7902 20 8

8 rows selected.

**SELECT COUNT(\*) FROM EMP**

**WHERE DEPTNO IN (10, 20);**

COUNT(\*)

----------

8

**How to break the result set in groups or partitions?**

* It might be obvious from the previous example that the clause PARTITION BY is used to break the result set into groups.
* PARTITION BY can take any non-analytic SQL expression.
* Some functions support the <window\_clause> inside the partition to further limit the records they act on.
* In the absence of any <window\_clause> analytic functions are computed on all the records of the partition clause.
* The functions SUM, COUNT, AVG, MIN, MAX are the common analytic functions the result of which does not depend on the order of the records.
* Functions like LEAD, LAG, RANK, DENSE\_RANK, ROW\_NUMBER, FIRST, FIRST VALUE, LAST, LAST VALUE depends on order of records. In the next example we will see how to specify that.

**How to specify the order of the records in the partition?**

* The answer is simple, by the "ORDER BY" clause inside the OVER ( ) clause.
* This is different from the ORDER BY clause of the main query which comes after WHERE.
* In this section we go ahead and introduce each of the very useful functions LEAD, LAG, RANK, DENSE\_RANK, ROW\_NUMBER, FIRST, FIRST VALUE, LAST, LAST VALUE and show how each depend on the order of the record.

The general syntax of specifying the ORDER BY clause in analytic function is:

**ORDER BY <SQL\_EXPR> [ASC OR DESC] NULLS [FIRST OR LAST]**

The syntax is self-explanatory.

**ROW\_NUMBER, RANK and DENSE\_RANK**

* All the above three functions assign integer values to the rows depending on their order. That is the reason of clubbing them together.
* **ROW\_NUMBER ( )** gives a running serial number to a partition of records.
* It is very useful in reporting, especially in places where different partitions have their own serial numbers.
* In below Query, the function ROW\_NUMBER ( ) is used to give separate sets of running serial to employees of departments 10 and 20 based on their HIREDATE.

**SELECT EMPNO, DEPTNO, HIREDATE,**

**ROW\_NUMBER( ) OVER (PARTITION BY**

**DEPTNO ORDER BY HIREDATE**

**NULLS LAST) SRLNO**

**FROM EMP**

**WHERE DEPTNO IN (10, 20)**

**ORDER BY DEPTNO, SRLNO;**

EMPNO DEPTNO HIREDATE SRLNO

7782 10 09-JUN-81 1

7839 10 17-NOV-81 2

7934 10 23-JAN-82 3

7369 20 17-DEC-80 1

7566 20 02-APR-81 2

7902 20 03-DEC-81 3

7788 20 09-DEC-82 4

7876 20 12-JAN-83 5

8 rows selected.

* RANK and DENSE\_RANK both provide rank to the records based on some column value or expression.
* In case of a tie of 2 records at position N, RANK declares 2 positions N and skips position N+1 and gives position N+2 to the next record.
* While DENSE\_RANK declares 2 positions N but does not skip position N+1.

The below Query shows the usage of both RANK and DENSE\_RANK.

For DEPTNO 20 there are two contenders for the first position (EMPNO 7788 and 7902).

Both RANK and DENSE\_RANK declares them as joint toppers.

RANK skips the next value that is 2 and next employee EMPNO 7566 is given the position 3.

For DENSE\_RANK there are no such gaps.

**SELECT EMPNO, DEPTNO, SAL,**

**RANK() OVER (PARTITION BY DEPTNO**

**ORDER BY SAL DESC NULLS LAST) RANK,**

**DENSE\_RANK() OVER (PARTITION BY**

**DEPTNO ORDER BY SAL DESC NULLS**

**LAST) DENSE\_RANK**

**FROM EMP**

**WHERE DEPTNO IN (10, 20)**

**ORDER BY 2, RANK;**

EMPNO DEPTNO SAL RANK DENSE\_RANK

7839 10 5000 1 1

7782 10 2450 2 2

7934 10 1300 3 3

7788 20 3000 1 1

7902 20 3000 1 1

7566 20 2975 3 2

7876 20 1100 4 3

7369 20 800 5 4

8 rows selected.

**LEAD and LAG Functions**

LEAD has the ability to compute an expression on the next rows (rows which are going to come after the current row) and return the value to the current row.

The general syntax of LEAD is shown below:

**LEAD (<SQL\_EXPR>, <OFFSET>, <DEFAULT>) OVER (<ANALYTIC\_CLAUSE>)**

**<sql\_expr>** is the expression to compute from the leading row.

**<offset>** is the index of the leading row relative to the current row.

**<offset>** is a positive integer with default 1.

**<default>** is the value to return if the <offset> points to a row outside the partition range.

The syntax of LAG is similar except that the offset for LAG goes into the previous rows.

Below Query and its result show simple usage of LAG and LEAD function.

**SELECT DEPTNO, EMPNO, SAL,**

**LEAD(SAL, 1, 0) OVER (PARTITION BY DEPT ORDER BY SAL DESC NULLS LAST) NEXT\_LOWER\_SAL,**

**LAG(SAL, 1, 0) OVER (PARTITION BY DEPT ORDER BY SAL DESC NULLS LAST) PREV\_HIGHER\_SAL**

**FROM EMP WHERE DEPTNO IN (10, 20)**

**ORDER BY DEPTNO, SAL DESC;**

DEPTNO EMPNO SAL NEXT\_LOWER\_SAL PREV\_HIGHER\_SAL

------- ------ ----- -------------- ---------------

10 7839 5000 2450 0

10 7782 2450 1300 5000

10 7934 1300 0 2450

20 7788 3000 3000 0

20 7902 3000 2975 3000

20 7566 2975 1100 3000

20 7876 1100 800 2975

20 7369 800 0 1100

8 rows selected.

**FIRST VALUE and LAST VALUE function**

The general syntax is:

**FIRST\_VALUE(<SQL\_EXPR>) OVER (<ANALYTIC\_CLAUSE>)**

The FIRST\_VALUE analytic function picks the first record from the partition after doing the ORDER BY. The <sql\_expr> is computed on the columns of this first record and results are returned.

The LAST\_VALUE function is used in similar context except that it acts on the last record of the partition.

* How many days after the first hire of each department were the next employees hired?

**SELECT EMPNO, DEPTNO, HIREDATE ? FIRST\_VALUE(HIREDATE)**

**OVER (PARTITION BY DEPTNO ORDER BY HIREDATE) DAY\_GAP**

**FROM EMP**

**WHERE DEPTNO IN (20, 30)**

**ORDER BY DEPTNO, DAY\_GAP;**

EMPNO DEPTNO DAY\_GAP

---------- ---------- -------------------

7369 20 0

7566 20 106

7902 20 351

7788 20 722

7876 20 756

7499 30 0

7521 30 2

7698 30 70

7844 30 200

7654 30 220

7900 30 286

11 rows selected.

**FIRST and LAST function (Continued…)**

* The FIRST function (or more properly KEEP FIRST function) is used in a very special situation.
* Suppose we rank a group of record and found several records in the first rank.
* Now we want to apply an aggregate function on the records of the first rank. KEEP FIRST enables that.

The general syntax is:

**FUNCTION( ) KEEP (DENSE\_RANK FIRST ORDER BY <EXPR>) OVER (<PARTITIONING\_CLAUSE>)**

* Please note that FIRST and LAST are the only functions that deviate from the general syntax of analytic functions.
* They do not have the ORDER BY inside the OVER clause.
* Neither do they support any <window> clause. The ranking done in FIRST and LAST is always DENSE\_RANK.

The query below shows the usage of FIRST function.

The LAST function is used in similar context to perform computations on last ranked records.

* How each employee's salary compare with the average salary of the first year hires of their department?

**SELECT EMPNO, DEPTNO, TO\_CHAR(HIREDATE,'YYYY') HIRE\_YR, SAL,**

**TRUNC(**

**AVG(SAL) KEEP (DENSE\_RANK FIRST**

**ORDER BY TO\_CHAR(HIREDATE,'YYYY') )**

**OVER (PARTITION BY DEPTNO)) AVG\_SAL\_YR1\_HIRE**

**FROM EMP**

**WHERE DEPTNO IN (20, 10)**

**ORDER BY DEPTNO, EMPNO, HIRE\_YR;**

EMPNO DEPTNO HIRE SAL AVG\_SAL\_YR1\_HIRE

7782 10 1981 2450 3725

7839 10 1981 5000 3725

7934 10 1982 1300 3725

7369 20 1980 800 800

7566 20 1981 2975 800

7788 20 1982 3000 800

7876 20 1983 1100 800

7902 20 1981 3000 800

8 rows selected.

**How to specify the Window clause (ROW type or RANGE type windows)?**

* Some analytic functions (AVG, COUNT, FIRST\_VALUE, LAST\_VALUE, MAX, MIN and SUM among the ones we discussed) can take a window clause to further sub-partition the result and apply the analytic function.
* An important feature of the windowing clause is that it is dynamic in nature.

The general syntax of the <window\_clause> is

**[ROW OR RANGE] BETWEEN <START\_EXPR> AND <END\_EXPR>**

**<start\_expr>** can be any one of the following

* UNBOUNDED PECEDING
* CURRENT ROW
* <sql\_expr> PRECEDING or FOLLOWING.

**<end\_expr>** can be any one of the following

* UNBOUNDED FOLLOWING or
* CURRENT ROW or
* <sql\_expr> PRECEDING or FOLLOWING.

For ROW type windows the definition is in terms of row numbers before or after the current row.

* So for ROW type windows <sql\_expr> must evaluate to a positive integer.
* For RANGE type windows the definition is in terms of values before or after the current ORDER.
* We will take this up in details latter.
* The ROW or RANGE window cannot appear together in one OVER clause.
* The window clause is defined in terms of the current row.
* But may or may not include the current row.
* The start point of the window and the end point of the window can finish before the current row or after the current row.
* Only start point cannot come after the end point of the window.
* In case any point of the window is undefined the default is UNBOUNDED PRECEDING for <start\_expr> and UNBOUNDED FOLLOWING for <end\_expr>.
* If the end point is the current row, syntax only in terms of the start point can be can be
* [ROW or RANGE] [<start\_expr> PRECEDING or UNBOUNDED PRECEDING ]
* [ROW or RANGE] CURRENT ROW is also allowed but this is redundant.
* In this case the function behaves as a single-row function and acts only on the current row.

**ROW Type Windows**

For analytic functions with ROW type windows, the general syntax is:

**FUNCTION( ) OVER (PARTITIN BY <EXPR1> ORDER BY <EXPR2,..> ROWS BETWEEN <START\_EXPR> AND <END\_EXPR>)**

OR

**FUNCTION( ) OVER (PARTITON BY <EXPR1> ORDER BY <EXPR2,..> ROWS [<START\_EXPR> PRECEDING OR UNBOUNDED PRECEDING]**

For ROW type windows the windowing clause is in terms of record numbers.

* The below query has no apparent real life description (except column FROM\_PU\_C) but the various windowing clause are illustrated by a COUNT(\*) function.
* The count simply shows the number of rows inside the window definition. Note the build up of the count for each column for the YEAR 1981.
* The column FROM\_P3\_TO\_F1 shows an example where start point of the window is before the current row and end point of the window is after current row.
* This is a 5 row window; it shows values less than 5 during the beginning and end.
* The query below has no apparent real life description (except column FROM\_PU\_C) but is remarkable in illustrating the various windowing clause by a COUNT(\*) function.

**SELECT EMPNO, DEPTNO, TO\_CHAR(HIREDATE, 'YYYY') YEAR,**

**COUNT(\*) OVER (PARTITION BY TO\_CHAR(HIREDATE, 'YYYY')**

**ORDER BY HIREDATE ROWS BETWEEN 3 PRECEDING AND 1 FOLLOWING) FROM\_P3\_TO\_F1,**

**COUNT(\*) OVER (PARTITION BY TO\_CHAR(HIREDATE, 'YYYY')**

**ORDER BY HIREDATE ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM\_PU\_TO\_C,**

**COUNT(\*) OVER (PARTITION BY TO\_CHAR(HIREDATE, 'YYYY')**

**ORDER BY HIREDATE ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING) FROM\_P2\_TO\_P1,**

**COUNT(\*) OVER (PARTITION BY TO\_CHAR(HIREDATE, 'YYYY')**

**ORDER BY HIREDATE ROWS BETWEEN 1 FOLLOWING AND 3 FOLLOWING) FROM\_F1\_TO\_F3**

**FROM EMP**

**ORDEDR BY HIREDATE**

EMPNO DEPTNO YEAR FROM\_P3\_TO\_F1 FROM\_PU\_TO\_C FROM\_P2\_TO\_P1 FROM\_F1\_TO\_F3

7369 20 1980 1 1 0 0

<font bgcolor=yellow>7499 30 1981 2 1 0 3

7521 30 1981 3 2 1 3

7566 20 1981 4 3 2 3

7698 30 1981 5 4 3 3

7782 10 1981 5 5 3 3

7844 30 1981 5 6 3 3

7654 30 1981 5 7 3 3

7839 10 1981 5 8 3 2

7900 30 1981 5 9 3 1

7902 20 1981 4 10 3 0</font>

7934 10 1982 2 1 0 1

7788 20 1982 2 2 1 0

7876 20 1983 1 1 0 0

14 rows selected.

* The column FROM\_PU\_TO\_CURR shows an example where start point of the window is before the current row and end point of the window is the current row.
* This column only has some real world significance.
* It can be thought of as the yearly employee build-up of the organization as each employee is getting hired.
* The column FROM\_P2\_TO\_P1 shows an example where start point of the window is before the current row and end point of the window is before the current row.
* This is a 3 row window and the count remains constant after it has got 3 previous rows.
* The column FROM\_F1\_TO\_F3 shows an example where start point of the window is after the current row and end point of the window is after the current row.
* This is a reverse of the previous column. Note how the count declines during the end.

**RANGE Windows**

For RANGE windows the general syntax is same as that of ROW:

**FUNCTION( ) OVER (PARTITION BY <EXPR1> ORDER BY <EXPR2> RANGE BETWEEN <START\_EXPR> AND <END\_EXPR>)**

OR

**FUNCTION( ) OVER (PARTITION BY <EXPR1> ORDER BY <EXPR2> RANGE [<START\_EXPR> PRECEDING OR UNBOUNDED PRECEDING]**

For <start\_expr> or <end\_expr> we can use UNBOUNDED PECEDING, CURRENT ROW or **<sql\_expr>** PRECEDING or FOLLOWING.

However for RANGE type windows <sql\_expr> must evaluate to value compatible with ORDER BY expression <expr1>.

**<sql\_expr>** is a logical offset. It must be a constant or expression that evaluates to a positive numeric value or an interval literal. Only one ORDER BY expression is allowed.

If **<sql\_expr>** evaluates to a numeric value, then the ORDER BY expr must be a NUMBER or DATE datatype. If <sql\_expr> evaluates to an interval value, then the ORDER BY expr must be a DATE datatype.

Note the example below which uses RANGE windowing. The important thing here is that the size of the window in terms of the number of records can vary.

For each employee give the count of employees getting half more that their salary and also the count of employees in the departments 20 and 30 getting half less than their salary.

**SELECT DEPTNO, EMPNO, SAL,**

**COUNT(\*) OVER (PARTITION BY DEPTNO ORDER BY SAL RANGE**

**BETWEEN UNBOUNDED PRECEDING AND (SAL/2) PRECEDING) CNT\_LT\_HALF,**

**COUNT(\*) OVER (PARTITION BY DEPTNO ORDER BY SAL RANGE**

**BETWEEN (SAL/2) FOLLOWING AND UNBOUNDED FOLLOWING) CNT\_MT\_HALF**

**FROM EMP WHERE DEPTNO IN (20, 30) ORDER BY DEPTNO, SAL;**

DEPTNO EMPNO SAL CNT\_LT\_HALF CNT\_MT\_HALF

20 7369 800 0 3

20 7876 1100 0 3

20 7566 2975 2 0

20 7788 3000 2 0

20 7902 3000 2 0

30 7900 950 0 3

30 7521 1250 0 1

30 7654 1250 0 1

30 7844 1500 0 1

30 7499 1600 0 1

30 7698 2850 3 0

11 rows selected.

**Order of computation and performance tips**

* Defining the PARTITOIN BY and ORDER BY clauses on indexed columns (ordered in accordance with the PARTITION CLAUSE and then the ORDER BY clause in analytic function) will provide optimum performance.
* It is advisable to always use CBO for queries using analytic functions.
* The tables and indexes should be analyzed and optimizer mode should be CHOOSE.
* Even in absence of indexes analytic functions provide acceptable performance but need to do sorting for computing partition and order by clause.
* If the query contains multiple analytic functions, sorting and partitioning on two different columns should be avoided if they are both not indexed.

**GROUP BY with ROLLUP and CUBE Operators**

* Use ROLLUP or CUBE with GROUP BY to produce superaggregate rows by cross-referencing columns.
* ROLLUP grouping produces a results set containing the regular grouped rows and the subtotal values.
* CUBE grouping produces a results set containing the rows from ROLLUP and cross-tabulation rows.

**GROUP BY with the ROLLUP and CUBE Operators**

* You specify ROLLUP and CUBE operators in the GROUP BY clause of a query. ROLLUP grouping produces a results set containing the regular grouped rows and subtotal rows.
* The CUBE operation in the GROUP BY clause groups the selected rows based on the values of all possible combinations of expressions in the specification and returns a single row of summary information for each group.
* You can use the CUBE operator to produce cross-tabulation rows.

**ROLLUP Operator**

**SELECT [COLUMN,] GROUP\_FUNCTION(COLUMN). . .**

**FROM TABLE**

**[WHERE CONDITION]**

**[GROUP BY [ROLLUP] GROUP\_BY\_EXPRESSION]**

**[HAVING HAVING\_EXPRESSION];**

**[ORDER BY COLUMN];**

* ROLLUP is an extension to the GROUP BY clause.
* Use the ROLLUP operation to produce cumulative aggregates, such as subtotals.

**The ROLLUP Operator**

* The ROLLUP operator delivers aggregates and superaggregates for expressions within a GROUP BY statement.
* The ROLLUP operator can be used by report writers to extract statistics and summary information from results sets.
* The cumulative aggregates can be used in reports, charts, and graphs.
* The ROLLUP operator creates groupings by moving in one direction, from right to left, along the list of columns specified in the GROUP BY clause.
* It then applies the aggregate function to these groupings.

**SELECT DEPARTMENT\_ID, JOB\_ID, SUM(SALARY)**

**FROM EMPLOYEES**

**WHERE DEPARTMENT\_ID < 60**

**GROUP BY ROLLUP(DEPARTMENT\_ID, JOB\_ID);**

****

**Example of a ROLLUP Operator**

In the example above:

* Total salaries for every job ID within a department for those departments whose department ID is less than 60 are displayed by the GROUP BY clause (labeled 1)
* The ROLLUP operator displays:
* Total salary for those departments whose department ID is less than 60 (labeled 2)
* Total salary for all departments whose department ID is less than 60, irrespective of the job IDs (labeled 3)
* All rows indicated as 1 are regular rows and all rows indicated as 2 and 3 are super aggregate rows.

**CUBE Operator**

* CUBE is an extension to the GROUP BY clause.
* You can use the CUBE operator to produce crosstabulation values with a single SELECT statement.

**SELECT [COLUMN,] GROUP\_FUNCTION(COLUMN)...**

**FROM TABLE**

**[WHERE CONDITION]**

**[GROUP BY [CUBE] GROUP\_BY\_EXPRESSION]**

**[HAVING HAVING\_EXPRESSION]**

**[ORDER BY COLUMN];**

**The CUBE Operator**

* The CUBE operator is an additional switch in the GROUP BY clause in a SELECT statement.
* The CUBE operator can be applied to all aggregate functions, including AVG, SUM, MAX, MIN, and COUNT. It is used to produce results sets that are typically used for cross-tabular reports.
* While ROLLUP produces only a fraction of possible subtotal combinations, CUBE produces subtotals for all possible combinations of groupings specified in the GROUP BY clause, and a grand total.
* The CUBE operator is used with an aggregate function to generate additional rows in a results set. Columns included in the GROUP BY clause are cross-referenced to produce a superset of groups.
* The aggregate function specified in the select list is applied to these groups to produce summary values for the additional super aggregate rows.
* The number of extra groups in the results set is determined by the number of columns included in the GROUP BY clause.

**CUBE Operator: Example**

**SELECT DEPARTMENT\_ID, JOB\_ID, SUM(SALARY)**

**FROM EMPLOYEES**

**WHERE DEPARTMENT\_ID < 60**

**GROUP BY CUBE (DEPARTMENT\_ID, JOB\_ID) ;**

****

**Example of a CUBE Operator**

The output of the SELECT statement in the example can be interpreted as follows:

* The total salary for every job within a department (for those departments whose department ID is less than 60) is displayed by the GROUP BY clause (labeled 1)
* The total salary for those departments whose department ID is less than 60 (labeled 2)
* The total salary for every job irrespective of the department (labeled 3)
* Total salary for those departments whose department ID is less than 60, irrespective of the job

titles (labeled 4)

**What Is a Subquery?**

A subquery is a SELECT statement embedded in a

clause of another SQL statement.

**SELECT ...**

**FROM ...**

**WHERE ... (SELECT ...**

**FROM ...**

**WHERE ...)**

* A subquery is a SELECT statement that is embedded in a clause of another SQL statement, called the parent statement.
* The subquery (inner query) returns a value that is used by the parent statement. Using a nested subquery is equivalent to performing two sequential queries and using the result of the inner query as the search value in the outer query (main query).

Sub queries can be used for the following purposes:

* To provide values for conditions in WHERE, HAVING, and START WITH clauses of SELECT statements
* To define the set of rows to be inserted into the target table of an INSERT or CREATE TABLE statement
* To define the set of rows to be included in a view or snapshot in a CREATE VIEW or CREATE

SNAPSHOT statement.

* To define one or more values to be assigned to existing rows in an UPDATE statement
* To define a table to be operated on by a containing query. (You do this by placing the subquery in the FROM clause. This can be done in INSERT, UPDATE, and DELETE statements as well.)

**Note:** A subquery is evaluated once for the entire parent statement.

**Pair wise Comparison Sub query**

Display the details of the employees who are managed by the same manager and work in the same department as the employees with EMPLOYEE\_ID 178 or 174.

**SELECT EMPLOYEE\_ID, MANAGER\_ID, DEPARTMENT\_ID**

**FROM EMPLOYEES**

**WHERE (MANAGER\_ID, DEPARTMENT\_ID) IN**

**(SELECT MANAGER\_ID, DEPARTMENT\_ID**

**FROM EMPLOYEES**

**WHERE EMPLOYEE\_ID IN (178,174))**

**AND EMPLOYEE\_ID NOT IN (178,174);**

* The example in the slide is that of a multiple-column subquery because the subquery returns more than one column.
* It compares the values in the MANAGER\_ID column and the DEPARTMENT\_ID column of each row in the EMPLOYEES table with the values in the MANAGER\_ID column and the DEPARTMENT\_ID column for the employees with the EMPLOYEE\_ID 178 or 174.
* First, the subquery to retrieve the MANAGER\_ID and DEPARTMENT\_ID values for the employees with the EMPLOYEE\_ID 178 or 174 is executed.
* These values are compared with the MANAGER\_ID column and the DEPARTMENT\_ID column of each row in the EMPLOYEES table.
* If the values match, the row is displayed. In the output, the records of the employees with the EMPLOYEE\_ID 178 or 174 will not be displayed.

The output of the query in the slide follows.

****

**Pivot Query**

Here is an example of a pivot query. Say you have the following set of data:

**SELECT JOB, DEPTNO, COUNT(\*)**

**FROM EMP**

**GROUP BY JOB, DEPTNO**

**/**

JOB DEPTNO COUNT(\*)

ANALYST 20 2

CLERK 10 1

CLERK 20 2

CLERK 30 1

MANAGER 10 1

MANAGER 20 1

MANAGER 30 1

PRESIDENT 10 1

SALESMAN 30 4

9 rows selected.

And you would like to make DEPTNO be a column. We have 4 deptno's in EMP, 10,20,30,40.

We can make columns dept\_10, dept\_20, dept\_30, dept\_40 that have the values that are currently in the count column.

It would look like this:

**SELECT JOB,**

**MAX( DECODE( DEPTNO, 10, CNT, NULL ) ) DEPT\_10,**

**MAX( DECODE( DEPTNO, 20, CNT, NULL ) ) DEPT\_20,**

**MAX( DECODE( DEPTNO, 30, CNT, NULL ) ) DEPT\_30,**

**MAX( DECODE( DEPTNO, 40, CNT, NULL ) ) DEPT\_40**

**FROM ( SELECT JOB, DEPTNO, COUNT(\*) CNT**

**FROM EMP**

**GROUP BY JOB, DEPTNO )**

**GROUP BY JOB**

**/**

JOB DEPT\_10 DEPT\_20 DEPT\_30 DEPT\_40

ANALYST 2

CLERK 1 2 1

MANAGER 1 1 1

PRESIDENT 1

SALESMAN 4

That has pivoted the CNT column by deptno across job.

**Correlated Subqueries**

* Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.

**GET**

candidate row from outer query

**EXECUTE**

inner query using candidate row value

**USE**

values from inner query to qualify or disqualify candidate row

* The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement.
* A correlated subquery is evaluated once for each row processed by the parent statement.
* The parent statement can be a SELECT, UPDATE, or DELETE statement.

**Correlated Subquery Execution**

* Get a candidate row (fetched by the outer query).
* Execute the inner query using the value of the candidate row.
* Use the values resulting from the inner query to qualify or disqualify the candidate.
* Repeat until no candidate row remains.

**Correlated Subquerie syntax:**

**SELECT column1, column2, ...**

**FROM table1 outer**

**WHERE column1 operator**

**(SELECT colum1, column2**

**FROM table2**

**WHERE expr1 =**

**outer.expr2);**

* A correlated subquery is one way of reading every row in a table and comparing values in each row against related data.
* It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.
* The Oracle Server performs a correlated subquery when the subquery references a column from a table in the parent query.

**Note:** You can use the ANY and ALL operators in a correlated subquery.

Correlated Subquery Example:

**SELECT e.employee\_id, last\_name,e.job\_id**

**FROM employees e**

**WHERE 2 <= (SELECT COUNT(\*)**

**FROM job\_history**

**WHERE employee\_id = e.employee\_id);**

****

**The EXISTS Operator**

* The EXISTS operator tests for existence of rows in the results set of the subquery.
* If a subquery row value is found:
* The search does not continue in the inner query
* The condition is flagged TRUE
* If a subquery row value is not found:
* The condition is flagged FALSE
* The search continues in the inner query

**The EXISTS Operator**

* With nesting SELECT statements, all logical operators are valid. In addition, you can use the EXISTS operator.
* This operator is frequently used with correlated subqueries to test whether a value retrieved by the outer query exists in the results set of the values retrieved by the inner query.
* If the subquery returns at least one row, the operator returns TRUE. If the value does not exist, it returns FALSE.
* Accordingly, NOT EXISTS tests whether a value retrieved by the outer query is not a part of the results set of the values retrieved by the inner query.
* Using the EXISTS Operator Find employees who have at least one person reporting to them.

**SELECT employee\_id, last\_name, job\_id, department\_id**

**FROM employees outer**

**WHERE EXISTS ( SELECT ’X’**

**FROM employees**

**WHERE manager\_id =**

**outer.employee\_id);**

****

**Using the NOT EXISTS Operator**

Find all departments that do not have any employees.

**SELECT department\_id, department\_name**

**FROM departments d**

**WHERE NOT EXISTS (SELECT ’X’**

**FROM employees**

**WHERE department\_id = d.department\_id);**

****

**Correlated UPDATE**

In the case of the UPDATE statement, you can use a correlated subquery to update rows in one table based on rows from another table.

Correlated UPDATE Syntax:

**UPDATE table1 alias1**

**SET column = (SELECT expression**

**FROM table2 alias2**

**WHERE alias1.column = alias2.column);**

* Denormalize the EMPLOYEES table by adding a column to store the department name.
* Populate the table by using a correlated update.

**ALTER TABLE employees ADD(department\_name VARCHAR2(14));**

**UPDATE employees e**

**SET department\_name =**

**(SELECT department\_name**

**FROM departments d**

**WHERE e.department\_id = d.department\_id);**

* The example in the above denormalizes the EMPLOYEES table by adding a column to store the department name and then populates the table by using a correlated update.

**Correlated DELETE**

Use a correlated subquery to delete rows in one table based on rows from another table.

Correlated DELETE Syntax:

**DELETE FROM table1 alias1**

**WHERE column operator**

**(SELECT expression**

**FROM table2 alias2**

**WHERE alias1.column = alias2.column);**

Use a correlated subquery to delete only those rows from the EMPLOYEES table that also exist in the EMP\_HISTORY table.

**DELETE FROM employees E**

**WHERE employee\_id =**

**(SELECT employee\_id**

**FROM emp\_history**

**WHERE employee\_id = E.employee\_id);**

Two tables are used in this example. They are:

* The EMPLOYEES table, which gives details of all the current employees
* The EMP\_HISTORY table, which gives details of previous employees

EMP\_HISTORY contains data regarding previous employees, so it would be erroneous if the same

employee’s record existed in both the EMPLOYEES and EMP\_HISTORY tables. You can delete such erroneous records by using the correlated subquery.

**Hierarchical Queries**

* Using hierarchical queries, you can retrieve data based on a natural hierarchical relationship between rows in a table.
* A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, a process called tree walking enables the hierarchy to be constructed.
* A hierarchical query is a method of reporting, in order, the branches of a tree.
* A hierarchical query is possible when a relationship exists between rows in a table.

**SELECT [LEVEL], column, expr...**

**FROM table**

**[WHERE condition(s)]**

**[START WITH condition(s)]**

**[CONNECT BY PRIOR condition(s)] ;**

Hierarchical queries can be identified by the presence of the CONNECT BY and START WITH clauses.

In the syntax:

* **SELECT:** Is the standard SELECT clause.
* **LEVEL:** For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on.
* **FROM table:** Specifies the table, view, or snapshot containing the columns. You can select from only one table.
* **WHERE:** Restricts the rows returned by the query without affecting other rows of the hierarchy.
* **condition:** Is a comparison with expressions.
* **START WITH:** Specifies the root rows of the hierarchy (where to start). This clause is required for a true hierarchical query.
* **CONNECT BY PRIOR:** Specifies the columns in which the relationship between parent and child rows exist. This clause is required for a hierarchical query.

The SELECT statement cannot contain a join or query from a view that contains a join.

**MERGE Statement**

* Use the MERGE statement to select rows from one or more sources for update or insertion into a table or view.
* You can specify conditions to determine whether to update or insert into the target table or view.
* This statement is a convenient way to combine multiple operations. It lets you avoid multiple INSERT, UPDATE, and DELETE DML statements.
* MERGE is a deterministic statement. You cannot update the same row of the target table multiple times in the same MERGE statement.

**Prerequisites**

* You must have the INSERT and UPDATE object privileges on the target table and the SELECT object privilege on the source table.
* To specify the DELETE clause of the merge\_update\_clause, you must also have the DELETE object privilege on the target table.

**INTO Clause:**

* Use the INTO clause to specify the target table or view you are updating or inserting into.
* In order to merge data into a view, the view must be updatable. Refer to "Notes on Updatable Views" for more information.

**USING Clause:**

* Use the USING clause to specify the source of the data to be updated or inserted. The source can be a table, view, or the result of a subquery.

**ON Clause:**

* Use the ON clause to specify the condition upon which the MERGE operation either updates or inserts.
* For each row in the target table for which the search condition is true, Oracle Database updates the row with corresponding data from the source table.
* If the condition is not true for any rows, then the database inserts into the target table based on the corresponding source table row.

**Restrictions on the ON Clause :**

* Oracle Database does not implement fine-grained access control during MERGE statements.
* If you are using the fine-grained access control feature on the target table or tables, then use equivalent INSERT and UPDATE statements instead of MERGE to avoid error messages and to ensure correct access control.

**merge\_update\_clause :**

* The merge\_update\_clause specifies the new column values of the target table. Oracle performs this update if the condition of the ON clause is true.
* If the update clause is executed, then all update triggers defined on the target table are activated.
* Specify the where\_clause if you want the database to execute the update operation only if the specified condition is true.
* The condition can refer to either the data source or the target table.
* If the condition is not true, then the database skips the update operation when merging the row into the table.

Restrictions on the merge\_update\_clause:

* You cannot update a column that is referenced in the ON condition clause.
* You cannot specify DEFAULT when updating a view.

**DELETE where Clause:**

* Specify the DELETE where\_clause to clean up data in a table while populating or updating it.
* The only rows affected by this clause are those rows in the destination table that are updated by the merge operation.
* The DELETE WHERE condition evaluates the updated value, not the original value that was evaluated by the UPDATE SET ... WHERE condition.
* If a row of the destination table meets the DELETE condition but is not included in the join defined by the ON clause, then it is not deleted.
* Any delete triggers defined on the target table will be activated for each row deletion.

**merge\_insert\_clause:**

* The merge\_insert\_clause specifies values to insert into the column of the target table if the condition of the ON clause is false.
* If the insert clause is executed, then all insert triggers defined on the target table are activated.
* If you omit the column list after the INSERT keyword, then the number of columns in the target table must match the number of values in the VALUES clause.
* Specify the where\_clause if you want Oracle Database to execute the insert operation only if the specified condition is true.
* The condition can refer only to the data source table. Oracle Database skips the insert operation for all rows for which the condition is not true. You can specify this clause by itself or with the merge\_update\_clause.
* If you specify both, then they can be in either order.

Restriction on Merging into a View :

* You cannot specify DEFAULT when updating a view.

**CREATE TABLE bonuses (employee\_id NUMBER, bonus NUMBER DEFAULT 100);**

**INSERT INTO bonuses(employee\_id)**

**(SELECT e.employee\_id FROM employees e, orders o**

**WHERE e.employee\_id = o.sales\_rep\_id**

**GROUP BY e.employee\_id);**

**SELECT \* FROM bonuses ORDER BY employee\_id;**

EMPLOYEE\_ID BONUS

153 100

154 100

155 100

156 100

158 100

159 100

160 100

161 100

163 100

9 rows selected.

**MERGE INTO bonuses D**

**USING (SELECT employee\_id, salary, department\_id FROM employees**

**WHERE department\_id = 80) S**

**ON (D.employee\_id = S.employee\_id)**

**WHEN MATCHED THEN UPDATE SET D.bonus = D.bonus + S.salary\*.01**

**DELETE WHERE (S.salary > 8000)**

**WHEN NOT MATCHED THEN INSERT (D.employee\_id, D.bonus)**

**VALUES (S.employee\_id, S.salary\*.01)**

**WHERE (S.salary <= 8000);**

**SELECT \* FROM bonuses ORDER BY employee\_id;**

EMPLOYEE\_ID BONUS

153 180

154 175

155 170

159 180

160 175

161 170

179 620

173 610

165 680

166 640

164 720

172 730

167 620

171 740

14 rows selected.

**Sequences**

Sequences are special database objects that provide numbers in sequence for input to a table.

They are useful for providing generated primary key values and for input of number type columns such as purchase order, employee number, sample number, and sales order number, where the input must be unique and in some form of numerical sequence.

**Creation of Sequences**

Sequences are created by use of the CREATE SEQUENCE command.

**CREATE SEQUENCE [schema.]sequence**

**[INCREMENT BY integer]**

**[START WITH integer]**

**[MAXVALUE integer | NOMAXVALUE]**

**[MINVALUE integer | NOMINVALUE]**

**[CYCLE | NOCYCLE]**

**[CACHE integer | NOCACHE]**

**[ORDER | NOORDER] ;**

Where:

**Schema:**

* Is the schema to contain the sequence. If you omit schema, Oracle creates the sequence in your own schema.

**Sequence:**

* Is the name of the sequence to be created.

**INCREMENT BY:**

* Specifies the interval between sequence numbers. This value can be any positive or negative Oracle integer, but it cannot be 0. If this value is negative, then the sequence descends. If the increment is positive, then the sequence ascends. If you omit this clause, the interval defaults to 1.

**MINVALUE:**

* Specifies the sequence's minimum value.

**NOMINVALUE:**

* Specifies a minimum value of 1 for an ascending sequence or -10 for a descending sequence. The default is NOMINVALUE.

**MAXVALUE:**

* Specifies the maximum value the sequence can generate.

**NOMAXVALUE:**

* Specifies a maximum value of 10 for a descending sequence. The default is NOMAXVALUE.

**START WITH:**

* Specifies the first sequence number to be generated.
* You can use this option to start an ascending sequence at a value greater than its minimum or to start a descending sequence at a value less than its maximum.
* For ascending sequences, the default value is the sequence's minimum value.
* For descending sequences, the default value is the sequence's maximum value.

**CYCLE:**

* Specifies that the sequence continues to generate values after reaching either its maximum or minimum value.
* After an ascending sequence reaches its maximum value, it generates its minimum value.
* After a descending sequence reaches its minimum, it generates its maximum.

**NOCYCLE:**

* Specifies that the sequence cannot generate more values after reaching its maximum or minimum value. The default is NOCYCLE.

**CACHE:**

* Specifies how many values of the sequence Oracle preallocates and keeps in memory for faster access. The minimum value for this parameter is 2.
* For sequences that cycle, this value must be less than the number of values in the cycle.

**NOCACHE:**

* Specifies that values of the sequence are not preallocated.
* If you omit both the CACHE parameter and the NOCACHE option, Oracle caches 20 sequence numbers by default.
* However, if you are using Oracle with the Parallel Server option in parallel mode and you specify the ORDER option, sequence values are never cached, regardless of whether you specify the CACHE parameter or the NOCACHE option.

**ORDER:**

* Guarantees that sequence numbers are generated in order of request.
* You may want to use this option if you are using the sequence numbers as timestamps.
* Guaranteeing order is usually not important for sequences used to generate primary keys.

**NOORDER:**

* Does not guarantee sequence numbers are generated in order of request.
* If you omit both the ORDER and NOORDER options, Oracle chooses NOORDER by default.
* Note that the ORDER option is only necessary to guarantee ordered generation if you are using Oracle with the Parallel Server option in parallel mode.
* If you are using exclusive mode, sequence numbers are always generated in order.

**Materialized View**

* A database object that stores the results of a query.
* Marries the query rewrite features found in Oracle Discoverer with the data refresh capabilities of snapshots

**Features/Capabilities:**

* Can be partitioned and indexed
* Can be queried directly
* Can have DML applied against it
* Several refresh options are available
* Best in read-intensive environments

**Advantages:**

* Useful for summarizing, pre-computing, replicating and distributing data
* Faster access for expensive and complex joins
* Transparent to end-users
* MVs can be added/dropped without invalidating coded SQL

**Disadvantages:**

* Performance costs of maintaining the views
* Storage costs of maintaining the views

**Syntax:**

**CREATE MATERIALIZED VIEW <name>**

**TABLESPACE <tbs name> {<storage parameters>}**

**<build option>**

**REFRESH <refresh option> <refresh mode>**

**[ENABLE|DISABLE] QUERY REWRITE**

**AS**

**SELECT <select clause>;**

The <build option> determines when MV is built

* BUILD IMMEDIATE: view is built at creation time
* BUILD DEFFERED: view is built at a later time
* ON PREBUILT TABLE: use an existing table as view source
* Must set QUERY\_REWRITE\_INTEGRITY to TRUSTED

**Refresh Options:**

* COMPLETE - totally refreshes the view
* Can be done at any time; can be time consuming
* FAST – incrementally applies data changes
* A materialized view log is required on each detail table.
* Data changes are recorded in MV logs or direct loader logs.
* Many other requirements must be met for fast refreshes
* FORCE – does a FAST refresh in favor of a COMPLETE
* The default refresh option

**Refresh Modes:**

* ON COMMIT – refreshes occur whenever a commit is performed on one of the view’s underlying detail table(s).
* Available only with single table aggregate or join based views
* Keeps view data transactionally accurate
* Need to check alert log for view creation errors
* ON DEMAND – refreshes are initiated manually using one of the procedures in the DBMS\_MVIEW package.
* Can be used with all types of materialized views
* Manual Refresh Procedures
* DBMS\_MVIEW.REFRESH(<mv\_name>, <refresh\_option>)
* DBMS\_MVIEW.REFRESH\_ALL\_MVIEWS()
* START WITH [NEXT] <date> - refreshes start at a specified date/time and continue at regular intervals

**Materialized View Example**

**CREATE MATERIALIZED VIEW items\_summary\_mv**

**ON PREBUILT TABLE**

**REFRESH FORCE AS**

**SELECT a.PRD\_ID, a.SITE\_ID, a.TYPE\_CODE, a.CATEG\_ID,**

**sum(a.GMS) GMS,**

**sum(a.NET\_REV) NET\_REV,**

**sum(a.BOLD\_FEE) BOLD\_FEE,**

**sum(a.BIN\_PRICE) BIN\_PRICE,**

**sum(a.GLRY\_FEE) GLRY\_FEE,**

**sum(a.QTY\_SOLD) QTY\_SOLD,**

**count(a.ITEM\_ID) UNITS**

**FROM items a**

**GROUP BY a.PRD\_ID, a.SITE\_ID, a.TYPE\_CODE, a.CATEG\_ID;**