Unidad 6

Lenguaje de programación: construcción de guiones

Gestión de bases de datos

Índice

Introducción

En este tema veremos los conceptos de PL/SQL, con el fin de que podamos interactuar con las bases de datos de Oracle.

Estudiaremos los bloques anónimos y las funciones que podemos realizar con ellos, desde la simple consulta de la base de datos hasta la modificación de esta al añadir, borrar o actualizar información.

Veremos distintas funciones que podemos realizar junto con elementos que podemos emplear para mejorar el rendimiento, como la inclusión de bucles, invocaciones de procedimientos y llamadas, etc.

Veremos sentencias de SQL que son empleables en PL/SQL y diversos elementos que nos serán de ayuda como cursores, disparadores y triggers.

Al finalizar esta unidad

- Conoceremos los lenguajes derivados de SQL, en especial nos centraremos en PL/SQL creado por ORACLE.
- Habremos estudiado las distintas sentencias que podemos realizar con PL/SQL.
- Sabremos llevar a cabo operaciones en las bases de datos con PL/SQL.
- Habremos distinguido los diferentes tipos de estructuras de control y funcionales.
- Habremos explicado cursores, disparadores y triggers.

6.1. ¿Por qué PL/SQL?

SQL es un lenguaje con una gran eficacia cuando se emplea con bases de datos, pero no es ni mucho menos perfecto. Ante las carencias de este a la hora de realizar ciertas consultas, se desarrolla por ORACLE un nuevo lenguaje PL/SQL, Procedural Language/Structured Query Language, el cual, en Oracle, nos otorga mayores prestaciones que su antecesor. Como desventaja este lenguaje no soporta no DDL, ni DCL, pero pose otras cualidades como:

- Posee las características de un lenguaje procedimental.
  - Empleo de variables.
  - Estructuras de control de flujo.
  - Tomas de decisiones.
  - Control de excepciones.
  - Reutilización de código.
- Elección entre:
  - Procedimientos o funciones.
  - Scripts anónimos en SQL*Plus.
- Posibilidad de almacenar el código como objetos de la propia base de datos.
- Ejecución desde el servidor.
- Empleo de disparadores o triggers para llevar a cabo acciones.

6.2. Otros lenguajes de programación

La creación de PL/SQL ayudo a Oracle, pero no es compatible con todos los SGBD, los cuales tuvieron que desarrollar sus propios lenguajes procedimentales, como ocurrió con PostgreSQL que creo PL/pgSQL. A pesar de esto PL/SQL es compatible con algunos de ellos, como son DB2 y Times Ten in-memory.

6.3. Bloques de Código Anónimos en PL/SQL

El empleo de bloques de código anónimos es imprescindible para PL/SQL, ya que son la base de este lenguaje, estos son instrucciones escritas en la consola y ejecutadas con el carácter "/". El código no se guarda, por lo que se debe escribir tantas veces como se desee realizar.

```
[DECLARE]
--Variables, cursores, excepciones
 BEGIN
        -- sentencia SQL deseadas
        -- sentencias de control PL/SQL deseadas;
[EXCEPTION]
        -- Excepciones implementadas
END;
/
```

Imagen 1. Modelo de estructura de bloque anónimo en PL/SQL

Las claves "BEGIN" y "END" son las únicas obligatorias junto a la implementación "/".

La estructura es la siguiente:

DECLARE:

Contiene las variables, constantes, cursores y excepciones que el usuario defina.

BEGIN y END:

Acotaran las sentencias que deseamos implementar. Tipos de sentencias:

- **Secuencias:** Ordenes consecutivas separadas por un punto y coma, ";".
- **Alternativas:** Rompen la secuencia, de modo que se ejecutan en función del valor de la sentencia.
- **Iteración o bucle:** Repeticiones de una sentencia hasta que una condición se cumpla o deje de cumplirse.

6.4. Tipos de datos en PL/SQL

Es importante que tanto las variables, datos que pueden cambiar durante la ejecución, y constantes, datos invariables, sean compatibles con los datos que el lenguaje soporta. Por supuesto esta es una decisión bastante importante para el desarrollo de la base de datos, ya que esta elección condicionará otros elementos como el formato de almacenamiento, restricciones y rangos de valores.

PL/SQL, además de los datos soportados por SQL, añade los suyos propios.

Entre los datos más comunes encontramos:

- **NUMBER, numérico:** Aunque pueden añadirse restricciones por el usuario, este tipo de valor permite soportar números de cualquier longitud, tanto números enteros como de punto flotante.
- **CHAR, carácter:** Almacena caracteres en un rango entre 1 y 32 767.
- **VARCHAR2, carácter de longitud variable:** Almacena los mínimos caracteres necesarios, reduciendo si es necesario su longitud original.
- **BOOLEAN, lógico:** Valores de TRUE o FALSE.
- **DATE, fecha:** Almacena fechas como datos numéricos, permitiendo las operaciones.
- **Atributos de tipo:** Permite obtener información de un objeto de la base de datos.
  - %TYPE: Reconoce tipos de variable, constante o campos empleados por la base de datos.
  - %ROWTYPE: Reconoce tipos de campos en una tabla, vista o curso de la base de datos.
  - Se pueden crear con PL/SQL tipos personalizados.

6.4.1. Declaración de variables

Con el fin de incluir una variable es necesario incluirla en DECLARE, ya que esta acción permitirá generar el espacio de memoria necesario para que esta se cree dentro del bloque de código anónimo. Podemos inicializar, incluir un valor, a la variable.

La sintaxis correcta es la siguiente:

```
Nombre [CONSTANT] tipo_de_dato [NOT NULL] [:= | DEFAULT |
Expresión];
```

La asignación de valores debe tener en cuenta el espacio determinado para ello, CHAR(30) solo aceptará 30 caracteres, ya que solo reservará memoria para estos 30.

La asignación se realiza empleando el operador ":=".

Podemos llevar a cabo la asignación de valores tanto de variables como de constantes directamente en el bloque de DECLARE mediante una inicialización.

Podemos ver ejemplos de esta sintaxis a continuación:

```
Variable tipo fecha no inicializada.
-- fechaIngreso DATE;
Variable numérica inicializada.
-- NEdad NUMBER(18) NOT NULL := 20;
Variable de carácter inicializada.
-- LocalUsuario VARCHAR2(20) := 'Murcia';
Constante numérica.
-- NHogar CONSTANT NUMBER := 82;
```

Imagen 2. Ejemplos de sintaxis

Las declaraciones más elaboradas se pueden realizar mediante %TYPE y %ROWTYPE los cuales permiten declaraciones relacionales, asociando su tipo a otro, de modo que si uno cambia el otro también lo haga automáticamente.

```
NUsuario Usuarios.Nombre%TYPE
```

También podemos asociarlo a una fila completa con ROWTYPE:

```
IngresosUsuarios Usuarios%ROWTYPE
```

Es posible tomar valores, en PL/SQL desde una consulta empleando SELECT, INTO, FROM, WHERE.

```
SELECT SUM(FechaAlquiler+Diaspagados) INTO FechaDevol FROM
Registro;
```

Si existen valores ROWTYPE podemos asignar valores a la consulta:

```
SELECT * INTO RegUsuario FROM Usuarios WHERE Edad=18
```

6.5. Operadores y expresiones

Podemos realizar diversas operaciones con los valores, independientemente de que se realicen con constantes o variables. Estas operaciones se realizan mediante operadores:

Operador	Acción
`**`	Potencia
`+- (unarios)`	Signo positivo o negativo
`*`	Multiplicación
`/`	División
`+`	Suma
`-`	Resta
`||`	Concatenación
`=,<,>,<=`	Comparaciones: igual, menor, mayor, menor o igual
`>=,<>,!=`	Mayor o igual, distinto, distinto
`IS NULL, LIKE`	Es nulo, como
`BETWEEN, IN`	Entre, in
`NOT`	Negación lógica (boolean)
`AND`	Operador AND lógico entre datos boolean
`OR`	Operador OR lógico entre datos boolean

Imagen 3. Operadores y sus significados.

Valores posibles de la operación `AND`:

A	B	A AND B
FALSE	FALSE	FALSE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	TRUE	TRUE
TRUE	NULL	NULL
FALSE	NULL	FALSE
NULL	TRUE	NULL
NULL	FALSE	FALSE

Imagen 4. Valores lógicos de AND

Valores posibles de la operación **OR**:

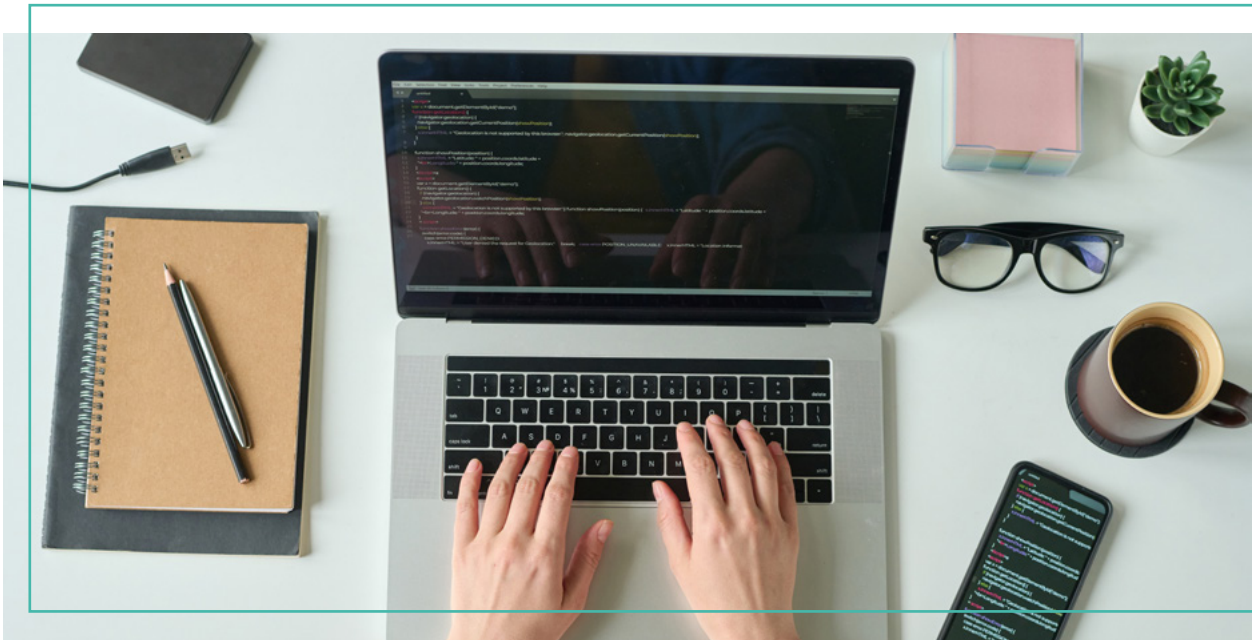
A	B	A AND B
FALSE	FALSE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
TRUE	TRUE	TRUE
TRUE	NULL	TRUE
FALSE	NULL	NULL
NULL	TRUE	TRUE
NULL	FALSE	NULL

Imagen 5. Valores lógicos de OR

Valores posibles de la operación **NOT**:

A	NOT A
FALSE	TRUE
TRUE	FALSE
NULL	NULL

Imagen 6. Valores lógicos de NOT



6.6. Entrada y salida para la depuración

Dado que PL/SQL no está destinado a los usuarios solo se puede trabajar mediante código, ya que no reconoce entradas y salidas de información que no sean elaboradas desde el código.

6.6.1. La salida

Si trabajamos con ORACLE deberemos emplear la variante **SERVEROUTPUT** para poder llevar a cabo las operaciones necesarias, tan solo es necesario realizar esta operación una sola vez por sesión.

```
SET SERVEROUTPUT ON
```

Una vez hecho esto el servidor mostrará, por pantalla, información con la siguiente sintaxis:

```
dbms_output.put_line (Datos de salida);
```

Si se desea mostrar un texto se debe intercalar con comillas simples, si se desea mostrar un valor se debe introducir el nombre de la variable, y se desea mostrar diversos valores se deben separar con dos barras "||". Los valores pueden ser caracteres, números, operaciones o variables.

```
dbms_output.put_line('Importe' ||
PrecioUnida*Cantidad || 'A fecha de' || sysdate)
```

Imagen 7. Ejemplo de dbms_output.put_line

6.6.2. La entrada

Con el fin de llevar a cabo una entrada se debe llevar a cabo una asignación de la variable donde se producirá la entrada y, tras el operador ":=" introducir "&" y aquello que se desea introducir. No se pueden incluir espacios por lo que las separaciones se deben realizar con un guion bajo "_". Independientemente del lugar donde se introduzca, esta entrada afectará a todas las variables del documento que hayamos designado.

```
BEGIN
VentasDia:=&INTRUDUCE_VALOR_VENTA_DIARIA;
END;
/
```

→

```
BEGIN
VentasDia:=&1465.32;
END;
/
```

Imagen 8. Ejemplo de entrada

6.7. Estructuras de Control

Con las variables y operadores que hemos aprendido podemos desarrollar operaciones ejecutadas automáticamente, claro está a un nivel elemental, como calcular una factura mediante un precio por unidad y una cantidad comprada.

Con el fin de seguir el teorema del programa estructurado debemos restringirnos a tres estructuras lógicas, llamadas estructuras de control:

- **Secuencia:** Las instrucciones se ejecutan sucesivamente.
- **Selección:** Según el valor de una variable se elige entre una acción u otra.
- **Iteración:** Realización de una acción hasta que un valor, booleano, cambie.

```
DECLARE
PRE INT:=4;
VAL INT:=2;
BEGIN
DBMS_OUTPUT.PUT_LINE(PRE ||' + '|| VAL ||' = '||A+B);
DBMS_OUTPUT.PUT_LINE(PRE ||' - '|| VAL ||' = '||A-B);
DBMS_OUTPUT.PUT_LINE(PRE ||' * '|| VAL ||' = '||A*B);
DBMS_OUTPUT.PUT_LINE(PRE ||' / '|| VAL ||' = '||A/B);
END;
/
```

Imagen 9. Ejemplo de secuencia

6.7.1. La Selección

La sentencia IF

La selección se basa en el empleo de la sentencia **IF**, la cual nos permite incorporar condiciones a nuestro propio código.

```
IF nota >=0 AND nota < 5 THEN
DBMS_OUTPUT.PUT_LINE('SUSPENSO: INSUFICIENTE');
ELSIF nota >=5 AND nota < 6 THEN
DBMS_OUTPUT.PUT_LINE('APROBADO: SUFICIENTE');
ELSIF nota >=6 AND nota < 7 THEN
DBMS_OUTPUT.PUT_LINE('APROBADO: BIEN');
ELSIF nota >=7 AND nota < 9 THEN
DBMS_OUTPUT.PUT_LINE('APROBADO: NOTABLE');
ELSIF nota >=9 AND nota <= 10 THEN
DBMS_OUTPUT.PUT_LINE ('APROBADO: SOBRESALIENTE');
ELSE
DBMS_OUTPUT.PUT_LINE('ERROR')
END IF;
```

Imagen 10. Ejemplo de selección con IF

ELSIF Señala cada uno de los resultados individuales predefinidos y deseados, mientras que **ELSE** engloba el resto de los casos no recogidos por los **ELSIF**.

La sentencia CASE

Este tipo de sentencia evalúa cada condición hasta encontrar una condición, de la misma manera que "switch" en lenguaje C.

```
CASE
WHEN nota >=0 AND nota < 5 THEN
DBMS_OUTPUT.PUT_LINE('SUSPENSO: INSUFICIENTE');
WHEN nota >=5 AND nota < 6 THEN
DBMS_OUTPUT.PUT_LINE('APROBADO: SUFICIENTE');
WHEN nota >=6 AND nota < 7 THEN
DBMS_OUTPUT.PUT_LINE('APROBADO: BIEN');
WHEN nota >=7 AND nota < 9 THEN
DBMS_OUTPUT.PUT_LINE('APROBADO: NOTABLE');
WHEN nota >=9 AND nota <= 10 THEN
DBMS_OUTPUT.PUT_LINE ('APROBADO: SOBRESALIENTE');
ELSE
DBMS_OUTPUT.PUT_LINE('ERROR')
END CASE;
```

Imagen 11. Ejemplo de selección con CASE

Se buscan los resultados esperados en **WHEN**, si no se encuentran pasa a funcionar **ELSE** como último recurso, generalmente con un mensaje de error.

6.7.2. La Iteración

La iteración, o bucles, repiten una o varias sentencias hasta una conclusión. Podemos encontrar diversos tipos de iteraciones en PL/SQL:

- **Bucle básico.** LOOP. Acciones repetitivas sin condiciones en el bucle.
- **Bucle FOR.** Acciones repetitivas con un contador y, por tanto, un número predefinido de repeticiones.
- **Bucle WHILE.** Acciones repetitivas basadas en una condición.

Bucle básico LOOP

Si no se especifica una condición el bucle será eterno, podemos fijar este límite mediante **IF** o **WHEN**, de modo que, aunque extenso en el tiempo, sea una operación que posea un final predefinido.

```
SET SERVEROUTPUT ON
DECLARE
NUM INT:=0;
LOOP
      NUM:=NUM+1;
      EXIT WHEN NUM >=100;
END LOOP;
```

```
SET SERVEROUTPUT ON
DECLARE
NUM INT:=0;
LOOP
      NUM:=NUM+1;
      IF NUM >=100 THEN
            EXIT;
      END IF;
END LOOP;
END;
```

Imagen 12. Ejemplos de iteración con WHEN e IF

Bucle con WHILE

El bucle se ejecuta de manera continua **mientras que** una condición específica no se cumpla. Esto implica que poseeremos una línea con un determinado valor, y hasta que este no cambie el bucle seguirá ejecutándose.

```
SET SERVEROUTPUT ON
DECLARE
     GIROS INT := 0;
BEGIN
WHILE GIROS <= 100 LOOP
     DBMS_OUTPUT.PUT_LINE(GIROS);
     GIROS := GIROS + 1;
END LOOP;
END;
```

Imagen 13. Ejemplo de bucle con WHILE

Bucle FOR

Bucle preparado con un numero de repeticiones predeterminadas. Se basa en la creación de un bucle, no que termine al llegar a cierto punto, sino que, de un determinado número de repeticiones, contándose estas de manera automática sin necesidad de programarlo.

Sintaxis: FOR Nombre IN valor inicial .. valor final LOOP

```
SET SERVEROUPUT ON
BEGIN
FOR NUM IN o .. 100 LOOP
DBMS_OUTPUT.PUT_LINE(NUM);
END LOOP;
END;
```

Imagen 14. Ejemplos de iteración con FOR

Podemos hacer que el contador retroceda con **REVERSE**.

```
FOR NUM IN REVERSE 0 .. 100 LOOP
```

6.8. Estructuras funcionales: procedimientos y funciones

La escritura en el código de los bloques no es la única forma de llevar a cabo acciones, con el fin de evitar un exceso de repetición de la escritura se implementaron las llamadas o invocaciones de procedimiento o función. Estas invocaciones nos permiten desarrollar una función tan solo con su nombre, evitándonos la tediosa tarea de escribirlo. Podemos emplear, con el fin de modificar estos procedimientos podemos emplear diversos valores denominados parámetros.

La creación de los procedimientos necesarios nos permite reutilizar código y por tanto ahorrar tiempo y evitar errores. También podemos crear procedimientos especiales llamados Triggers, que cuentan con la particularidad de invocarse a sí mismos cuando ocurre un evento específico a modo de respuesta.

La incorporación de invocaciones y Triggers, permite un código mucho más ligero y ordenado, evitando la repetición excesiva de un mismo código. Como segunda ventaja la reutilización implica un código más corto y, por tanto, menos errores y una codificación y depuración más sencilla.

6.8.1. Procedimientos

El procedimiento permite a que un bloque de código pueda ser llamado con una invocación, su sintaxis se caracteriza por el inicio con la siguiente frase.

```
CREATE (OR REPLACE) PROCEDURE Nombre del proceso
Parámetros (con IN | OUT | IN OUT)
IS | AS
Declaraciones
BEGIN
...
```

Imagen 15. Ejemplo de sintaxis de un procedimiento

Esta sintaxis permite la creación del procedimiento, el inicio en **`CREATE PROCEDURE`** o **`CREATE OR REPLACE PROCEDURE`** es indispensable, ya que iniciará la creación o la creación y sustitución de un procedimiento. Tras esto se debe incluir el nombre de dicho procedimiento, sin ningún tipo de palabra clave o variable para que sea válido. Tras esto se añadirán los parámetros entre paréntesis y una de las siguientes palabras clave, **`AS`** o **`IS`**.

Tras esta introducción se añadirá el bloque que se desea que se produzca normalmente.

```
CREATE OR REPLACE PROCEDURE EJEMPLO AS
BEGIN
FOR NUM IN 0 .. 100 LOOP
DBMS_OUTPUT.PUT_LINE(NUM)
END LOOP;
END;
```

Imagen 16. Ejemplo de creación de un procedimiento

Parámetros en procedimientos

Los parámetros nos permitirán modificar el procedimiento al llamarlo, lo cual nos dará versatilidad sin tener que escribir el código de nuevo.

Todos los parámetros deben precederse, para su identificación de **`IN, OUT, IN OUT`**. Tras estos también se deben incluir los tipos de datos empleados.

Para poder entender las explicaciones de estos primero debemos conocer los conceptos de parámetros formales y actuales:

- **Parámetro formal:** Parámetro definido por la función como una variable local de este, pero se inician con el valor actual que posean.
- **Parámetro actual:** Valores de los parámetros al llamar la función. Se asignan a los parámetros formales en la invocación.

La diferencia entre **`IN, OUT, IN OUT`** es la siguiente.

- **`IN`** = Un parámetro de entrada, no modifica el resto de los parámetros, ni a sí mismo fuera del parámetro.

- **OUT** = Un parámetro de salida, el valor asignado afecta al parámetro, pero no emplea un procedimiento formal para introducir un valor.
- **IN OUT** = Un parámetro de entrada y salida, de modo que su valor inicial no es igual que su valor de salida, modificando este, al iniciar P1:=G1, al terminar el procedimiento G1:=P1.

Uno de los procedimientos comúnmente empleados es el de intercambio, permitiendo intercambiar dos valores entre sí. Con el fin de llevar a cabo este proceso podemos emplear variables no existentes como auxiliares, en este caso se denominarán **AX**. **INT** designa el tipo de dato, en este caso número entero.

```
CREATE OR REPLACE PROCEDURE INTERCAMBIO
(VALORA IN OUT INT, VALORB IN OUT INT) AS
AX INT;
BEGIN
      AX:=VALORA
      VALORA:=VALORB
      VALORB:=AX

END;
/

DECLARE
NUM1 INT:= 1
NUM2 INT:=0
BEGIN
      DBMS_OUTPUT.PUT_LINE('NUM1: ' || NUM1 || 'NUM2: ' ||NUM2);
            INTERCAMBIO (NUM1,NUM2);
      DBMS_OUTPUT.PUT_LINE('NUM1: ' || NUM1 || 'NUM2: ' ||NUM2);
END;
/
```

Imagen 17. Ejemplo de creación de procedimiento en invocación

6.8.2. Funciones

A diferencia de los procedimientos, las funciones tienen como final devolver un resultado concreto tras su invocación. Es imprescindible indicar el tipo de dato de la devolución. Su sintaxis es la siguiente:

```
CREATE OR REPLACE FUNCTION Nombre
RETURN Tipos de Dato
IS || AS
Declaraciones
BEGIN
…
END;
/
```

Imagen 18. Ejemplo de sintaxis de una función

Las líneas de creación y RETURN son imprescindibles

Una función de permite introducir parámetros y obtener los resultados deseados.

```
CREATE OR REPLACE FUNCTION
EJEMPLO (PRE IN NUMBER, DIV IN NUMBER)
RETURN NUMBER AS
AX NUMBER:=0;
RES NUMBER:=0;
BEGIN
PRE/100:=AX;
AX*DIV:=RES;
RETURN(RES);
END;
/
```

Imagen 19. Ejemplo de una función para cálculo porcentajes

En este ejemplo al introducir los valores **PRE** (precio) y **DIV** (divisor) podremos obtener el cálculo de un porcentaje obteniendo como resultado el porcentaje **DIV** de **PRE**, por ejemplo, el 20% de 50 obtendríamos el resultado 10, lo cual obtendríamos al añadir los valores 50 y 20, en ese orden.

Llamadas a funciones y procedimientos

Dentro de un bloque anónimo, tanto para un procedimiento como para una función, solo es necesario escribir el nombre se estos y sus parámetros.

Fuera de un bloque podemos llamar un procedimiento mediante **CALL** o **EXEC** y una función en una expresión **CALL DBMS_OUTPUT.PUT_LINE**. Ejemplos:

```
BEGIN
EJEMPLO (100,20);
END;
/


CALL INTERCAMBIO (100,20);
EXEC INTERCAMBIO (100,20);


CALL DBMS_OUTPUT.PUT_LINE(EJEMPLO (100,20));
```

Imagen 20. Ejemplos de los tipos de llamadas

6.9. Sentencias SQL en PL/SQL

En PL/SQL podemos incluir sentencias de SQL, ya que PL/SQL descendiente de SQL y comparte muchas de sus reglas.

El empleo de ambos nos permitirá un resultado mucho mayor.

6.9.1. Recuperar datos de la BD con SELECT

Permite asignar como valores a las variables resultados de consultas con la sentencia **SELECT**. El resultado se debe realizar fila a fila. Su sintaxis es la siguiente:

```
SELECT Operación(lista de columnas)
INTO Nombre de la variable
FROM Tabla WHERE Condición;
```

```
DECLARE
ABONADO NUMBER;
BEGIN
      SELECT SUM(UNIDADES) INTO ABONADO
      FROM ABONOS WHERE USUARIOCOD=321;
      DBMS_OUTPUT.PUT_LINE('El abono recibido del cliente '
      || USUARIOCOD || ' es un total de:' || ABONADO);
END;
/
```

Imagen 21. Ejemplo de empleo del valor de SELECT en una variable

6.9.2. Inserción de datos en PL/SQL

Podemos emplear **INSERT** de SQL para añadir registros a una tabla. Emplearemos esta sintaxis:

```
INSERT INTO Nombre de la tabla
[(casilla 1[, casilla 2…])]
Valores
(ValorA, ValorB, …);

DECLARE
PRESTAMOSPENDIENTES VARCHAR2:=NO;
BEGIN
     INSERT INTO USUARIOS VALORES
     (US:52635, '27-sep-1996','-', PRESTAMOSPENDIENTES);
END;
/
```

Imagen 22. Ejemplo de INSERT

6.9.3. Actualización de datos en PL/SQL

Es posible sobre escribir datos con otros más recientes, ya sea borrando o escribiendo o modificando directamente los datos, en especial si estos son numéricos.

```
CREATE OR REPLACE PROCEDURE
MODIFICACION(CAMBIO IN INT,
SELECCION IN PRODUCTOS.TIPO%TYPE)AS
BEGIN
UPDATE PRODUCTOS
SET CANTIDAD=CANTIDAD+10
FROM PRODUCTOS WHERE TIPE='LIBRETAS';
END;
/
```

Imagen 23. Ejemplo de actualización

6.10. Acceso a la Base de Datos. Cursores

Las áreas de memoria destinadas a guardar datos extraídos son denominadas cursores. Estos datos se pueden extraer mediante **SELECT** o con **DML**, actualización e inserción. Podemos encontrar dos tipos: Implícitos y explícitos.

- **Implícitos:** No requieren ser declarados y se generan con **DML** y **SELECT** en PL/SQL, solo devuelven una fila.
- **Explícitos:** Se deben ingresar mediante el comando:

```
CURSOR Nombre IS Sentencia SELECT;

DECLARE
ENOMBRE USUARIOS.Nombre%TYPE;
EAPELLIDO USUARIO.Apellido%TYPE;
CURSOR CURSORUSUARIO IS SELECT Nombre, Apellido
FROM USUARIOS WHERE CODREGISTRO >= 1000;
BEGIN
```

Ilustración 12. Ejemplo de CURSOR explícito

En el caso de querer trabajar con el cursor dentro de un bloque será necesario realizar las siguientes operaciones:

- **Abrir el cursor:** Debemos emplear **OPEN** y el nombre del cursor.
- **Recuperar los datos:** Debemos emplear **FETCH** junto con el número de campos de las variables. Los campos y variables deben coincidir.
- **Cierre del cursor:** Debemos emplear **CLOSE** y el nombre del cursor.

Siempre debemos cerrar el cursor si no lo necesitamos ya que solo podemos tener abiertos un número limitado de estos.

Podemos comprobar los cursores en sus atributos al escribir el nombre del cursor % y el atributo deseado.

Los atributos son los siguientes:

- **%ISOPEN:** Valor booleano:
  - Cursor abierto: TRUE
  - Cursor cerrado: FALSE
- **%NOTFOUND:TRUE:** No se ha encontrado ninguna fila.
- **%FOUND:TRUE:** Si se ha recuperado una fila.
- **%ROWCOUNT:** Número de filas recuperadas.

```
DECLARE
CURSOR EjemploCursor IS SELECT Nombre, Apellido, Ciudad
FROM Usuarios;
RegUsuario EjemploCursor%ROWTYPE;
BEGIN
OPEN EjemploCursor;
FETCH EjemploCursor INTO RegUsuario;
WHILE EjemploCursor%FOUND LOOP
DBMS_OUTPUT.PUT_LINE(RegUsuario.Nombre||' '||
RegUsuario.Apellido||' '||
RegUsuario.Ciudad);
FETCH EjemploCursor INTO RegUsuario;
END LOOP;
CLOSE EjemploCursor;
END;
/
```

Ilustración 13. Ejemplo de CURSOR empleado en un bloque

Podemos realizar esta misma acción con **FOR**.

```
FOR Variable del Registro IN Nombre del Cursor LOOP
Instrucción 1;
Instrucción 2;
...
END LOOP;
END;
```

Ilustración 14. Sintaxis del uso de FOR para la simplificación de cursores

Las ventajas de esta sintaxis son:

- Se elimina la necesidad de abrir y cerrar cursores.
- La variable no requiere ser declarada previamente.
- Cada iteración del bucle realiza un **FETCH** sobre la variable.
- El bucle se cierra automáticamente al finalizar las casillas.

```
DECLARE
CURSOR EjemploCursor IS SELECT Nombre, Apellido, Ciudad
FROM Usuarios;
BEGIN
FOR RegUsuario IN EjemploCursor LOOP
DBMS_OUTPUT.PUT_LINE(RegUsuario.Nombre||' '||
RegUsuario.Apellido||' '||
RegUsuario.Ciudad);
END LOOP;
END;
/
```

Ilustración 15. Ejemplos de uso de FOR para la simplificación de cursores

6.11. Excepciones en PL/SQL

Una excepción se puede crear por un error en la ejecución o ser creada intencionadamente por el desarrollador.

Una excepción, si es capturada, no causará problemas, pero si no lo hace ORACLE mostrará un mensaje de error.

Tratamiento de excepciones implícitas

Mediante **EXCEPTION** podemos controlar los errores previsibles, de modo que estos no interrumpan el desarrollo normal de la ejecución.

La sintaxis de esta es muy sencilla:

```
[EXCEPTION
WHEN Exception1 THEN …]
[WHEN Exception2 [OR Exception3] THEN …]
[WHEN OTHERS THEN …]
```

Ilustración 16. Ejemplos de EXCEPTION

Podemos programar una respuesta para uno o varios errores con cada **WHEN** e integrar cualquier otro error en **WHEN OTHERS**.

Al capturar las excepciones se nos mostrarán o desarrollarán las acciones correspondientes sin provocar la interrupción.

ORACLE posee las funciones SQLCODE Y SQLERR que permiten mostrar un código concreto en función del error, generalmente se emplea con **OTHERS**.

```
EXCEPTION
WHEN OTHERS TREN
DBMS_OUTPUT.PUT_LINE('Código de error n':'||SQLCODE);
DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
```

Ilustración 17. Ejemplo de SQLCODE y SQLERRM

Entre los valores de excepciones que podemos interceptar destacan:

- **CASE_NOT_FOUND**: en la estructura CASE ninguna de la sentencia WHEN se corresponde con el valor evaluado y no existe cláusula ELSE.
- **CURSOR_ALREADY_OPEN**: Se intenta abrir un curso ya abierto.
- **INVALID_CURSOR**: La operación no es válida.
- **INVALID_NUMBER** o **VALUE_ERROR**: El valor numérico no es válido.
- **VALUE_ERROR**: Error en operaciones aritméticas.
- **LOGIN_DENIED**: Usuario o password incorrectos
- **NOT_LOGGED_ON**: Necesita iniciar sesión.
- **NO-DATA-FOUND**: SELECT INTO no devuelve registros.

- **TOO-MANY-ROWS**: SELECT INTO devuelve más de un registro.
- **TIMEOUT-ON-RESOURCE**: Tiempo de espera finalizado sin resultados.
- **ZERO-DIVIDE**: División de un número entre cero.
- **OTHERS**: Todos los errores no tenidos en cuenta.

Excepciones lanzadas por el programador

Usando **RAISE** podemos crear una excepción. Su sintaxis es la siguiente:

```
DECLARE
BEGIN
...
RAISE Nombre de la excepción
...
EXCEPTION
WHEN Nombre de la excepción THEN ...;
END;
/
```

Ilustración 18. Ejemplo de excepción con RAISE

6.12. Disparadores o Triggers

Un trigger es un procedimiento o función automático, asociado a una tabla, se activa ante una operación DML como **INSERT**, **UPDATE** o **DELETE**, sobre la tabla a la que se asocia.

```
CREATE (OR REPLACE) TRIGGER NomDisp
(BEFORE|AFTER) (DELETE|INSERT|UPDATE (OF columnas))
(OR (DELETE|INSERT|UPDATE (OF columnas))...)
ON tabla
(FOR EACH ROW( WHEN Condición)
DECLARE
BEGIN
EXCEPTION
END;
/
```

Ilustración 19. Ejemplo de la sintaxis de un trigger

La temporalidad se marca con uno de estos elementos:

- **BEFORE**: El código se ejecuta antes que la operación DML.
- **AFTER**: El código se ejecuta tras la operación DML.

Las operaciones que deseamos realizar irán justo después, se puede seleccionar más de una o unirlas con OR.

Después debemos especificar el tipo de disparador. Con **FOR EACH ROW** el disparador se activa sobre cada fila de la tabla, podemos añadir restricciones con **WHEN**, mientras que sin él solo se implementa en las filas donde se llevan a cabo operaciones **DML**.

Si existen varios disparadores podemos encontrar el responsable con IF junto a:

- **Inserting:** Si el disparador es `INSERT`, devuelve TRUE.
- **Deleting:** Si el disparador es `DELETE`, devuelve TRUE.
- **Updating:** Si el disparador es `UPDATE`, devuelve TRUE.
- **Updating (columna):** Si el disparador es `UPDATE` y la columna se ha modificado, devuelve TRUE.

Podemos emplear los pseudo-registros :old y :new, para comprobar los cambios en una tabla.

Orden	:old	:new
INSERT	Valor NULL	Valor insertado
UPDATE	Valor original	Nuevos valores introducidos
DELETE	Valores originales	Valor NULL

Cuadro 13. Valores de: old y :new

PL/SQL establece el orden en la ejecución de los triggers cuando se produce el disparador y existen varios de estos siguiendo el siguiente orden:

- Ejecuta, si existe, el trigger de tipo `BEFORE`.
- En cada fila a la que afecte dicha orden:
  - El disparador de tipo `BEFORE`.
  - La orden.
  - El disparador de tipo `AFTER`.
- El disparador de tipo `AFTER`.

www.universae.com