TEMA 5 – PROGRAMACION DE BASE DE DATOS PLSQL.

INDICE

1. INTRODUCCCION

- PL/SQL significa Procedimental Language/SQL (Lenguaje Procedimental/SQL).

- PL/SQL es un lenguaje de 3ª generación (procedimental) diseñado por Oracle para trabajar con la base de datos.

- Soporta todas los comandos de consulta y manipulación de datos, aportando al lenguaje. SQL las estructuras de control (bucles, bifurcaciones, etc.) y otros elementos propios de los ljes. procedimentales de tercera generación.

- Desde PL/SQL se puede ejecutar cualquier orden de manipulación de datos (de sql).

- **PL/SQL es la extensión estructurada y procedimental (permite crear funciones y procedimientos) del lenguaje SQL implementada por Oracle junto a la versión 6.**

- Con los scripts de SQL se tienen limitaciones como uso de variables, modularidad,etc.

- Con PL/SQL se pueden usar sentencias SQL para acceder a bases de datos Oracle y sentencias de control de flujo para procesar los datos, se pueden declarar variables y constantes, definir procedimientos, funciones, subprogramas, capturar y tratar errores en tiempo de ejecución, etc...

- En un programa escrito en PL/SQL se pueden utilizar sentencias SQL de tipo LMD (Lenguaje de Manipulación de Datos) directamente y sentencias de tipo LDD (Lenguaje de Definición de Datos) mediante la utilización de paquetes.

- Oracle incorpora un gestor PL/SQL en el servidor de bbdd y en las principales herramientas, Forms , Reports, Graphics, etc.

- Basado en Ada incorpora todas las características propias de los lenguajes de tercera generación; manejo de variables, estructura modular (procedimientos y funciones) , estructuras de control, control de excepciones,etc

- El código PL/SQL puede estar almacenado en la base de datos (procedimientos, funciones, disparadores y paquetes) facilitando el acceso a todos los usuarios autorizados.

- La ejecución de los bloques PL/SQL puede realizarse interactivamente desde herramientas como SQL*Plus, Oracle Forms, etc..., o bien, cuando el S.G.B.D. detecte determinados eventos, también llamados disparadores.

- Los programas se ejecutan en el servidor, con el consiguiente ahorro de recursos en los clientes y disminución de tráfico en la red.

2. CARACTERÍSTICAS DEL LENGUAJE.

2.1. BLOQUES PL/SQL.

Con PL/SQL se pueden construir distintos tipos de programas: procedimientos, funciones, etc.; todos ellos tienen en común una estructura básica denominada **BLOQUE.**

Un bloque tiene tres zonas claramente definidas:

- Una zona de **declaraciones** donde se definen objetos (variables, constantes, etc.) locales. Suele ir precedida por la cláusula DECLARE (o IS/AS en los procedimientos y funciones). Esta zona es opcional.
- Un conjunto de **instrucciones** precedido por la cláusula BEGÍN.
- Una zona de tratamiento de **excepciones** precedido por la cláusula EXCEPTION. Esta zona también es opcional.

El formato genérico del bloque es:

```
[DECLARE
        <declaraciones>]
BEGIN
        <órdenes>
[EXCEPTION
        <gestión de excepciones>]
END;
```

Las únicas cláusulas obligatorias son BEGIN Y END. La zona de declaraciones puede empezar con DECLARE o IS, dependiendo del tipo de bloque.

Ejemplo 1: En este ejemplo se borra el dpto. nº 20.

```
BEGIN
        DELETE FROM depart
        WHERE dept_no = 20;
END;
```

Ejemplo 2: Con este programa actualizamos el sueldo del empleado con apellido 'MORENO', incrementándolo en 20.000 en caso de que dicho sueldo no supere 100.000 pesetas.

```
DECLARE
        sueldo NUMBER(8);
```

```
    BEGIN
        SELECT salario INTO sueldo
        FROM plantilla
        WHERE apellido LIKE 'MORENO';

        IF sueldo < 100000 THEN
            sueldo := sueldo + 20000;
        END IF;
        UPDATE plantilla
        SET salario = sueldo
        WHERE apellido LIKE 'MORENO';
    END;
/
```

El programa tiene parte declarativa, en la que se define una variable, y parte ejecutable.

Podemos observar que hay sentencias SQL, como SELECT y UPDATE, sentencias de control como IF... THEN y sentencias de asignación. El símbolo ':=' representa el operador de asignación.

El bloque de código se ejecuta al encontrar el operador de ejecución '/'.

2.2. USO DE CURSORES.

En PL/SQL el resultado de una consulta no va directamente al terminal del usuario, sino que se guarda en un área de memoria a la que se accede mediante una estructura denominada cursor. Por tanto los cursores sirven para guardar el resultado de una consulta.

El formato básico es:

SELECT <columna/s> INTO <variable/s> FROM <tabla> [WHERE ....etc];

Ejemplo:

```
DECLARE
        V_ape varchar2(10);
        V_oficio varchar2(10);
BEGIN
        SELECT apellido, oficio INTO v_ape, v_oficio
        FROM EMPLE WHERE EMP_NO = 7900;
END;
```

Las variables que siguen al INTO reciben el valor de la consulta. Por tanto, debe haber coincidencia en el tipo con las columnas especificadas en la cláusula SELECT.

A este tipo de cursores se les denomina cursores implícitos. Es el tipo más sencillo, pero tiene ciertas limitaciones, ya que la consulta deberá devolver una única fila, pues en caso contrario se producirá un error catalogado como TOO_MANY_ROWS.

2.3. GESTIÓN DE EXCEPCIONES.

Las excepciones sirven para tratar errores y mensajes de las diversas herramientas. Oracle tiene determinadas excepciones correspondientes a algunos de los errores más frecuentes que se producen al trabajar con la BD, como por ejemplo:

NO_DATA_FOUND. Una orden de tipo SELECT INTO no ha devuelto ningún valor.

TOO_MANY_ROWS. Una orden de tipo SELECT INTO ha devuelto más de una fila.

Nombre de la excepción	Número de error Oracle	Descripción
NO_DATA_FOUND	ORA-01403	La sentencia SELECT no devolvió datos.
TOO_MANY_ROWS	ORA-01422	La sentencia SELECT devolvió más de una fila.
INVALID_CURSOR	ORA-01001	Se produjo una operación de cursor ilegal.
ZERO_DIVIDE	ORA-01476	Se intentó dividir entre cero.
DUP_VAL_ON_INDEX	ORA-00001	Se intentó insertar un valor duplicado.
INVALID_NUMBER	ORA-01722	Falla la conversión de una cadena de caracteres a números.

Se disparan automáticamente al producirse los errores asociados. El ejemplo anterior con gestión de excepciones sería:

```
DECLARE
        V_ape varchar2(10);
        V_oficio varchar2(10);
BEGIN
        SELECT apellido, oficio INTO v_ape, v_oficio
        FROM EMPLE WHERE EMP_NO = 7900;
EXCEPTION
        WHEN NO_DATA_FOUND THEN
                Insert into temp (col1) values ('ERROR no hay datos');
        WHEN TOO_MANY_ROWS THEN
                Insert into temp (col1) values ('ERROR demasiados
datos');
END;
```

Si PL/SQL detecta una excepción, pasa el control a la cláusula WHEN correspondiente de la sección EXCEPTION del bloque PL, que lo tratará según lo establecido. Al finalizar este tratamiento, se devuelve el control al programa que llamó al bloque que trató la excepción.

2.4. INTERACCIÓN CON EL USUARIO.

PL/SQL no es un lje. creado para interactuar con el usuario, sino para trabajar con la BD.

No obstante Oracle, incorpora el paquete DBMS_OUTPUT con fines de depuración.

Éste incluye entre otros, el procedimiento PUT_LINE, que permite visualizar textos en la pantalla.

El formato genérico para invocar a este procedimiento es el siguiente:

DBMS_OUTPUT.PUT_LINE (<expresión>)

Para que funcione correctamente, la variable de entorno SERVEROUTPUT deberá estar en ON; en caso contrario no se visualizará nada. Para cambiar el
estado de la variable introduciremos al comienzo de la sesión:

SQL>SET SERVEROUTPUT ON

Ejemplo:

```
SQL> BEGIN
  2  DBMS_OUTPUT.PUT_LINE ('HOLA MUNDO');
  3  END;
  4  /
HOLA MUNDO
```

Procedimiento PL/SQL terminado con éxito.

2.5. TIPOS DE BLOQUES PL/SQL.

Podemos diferenciar tres tipos de bloques PL/SQL:

- **Bloques PL/SQL anónimos.** Son bloques que como su nombre indican no tienen nombre. Se pueden generar con diversas herramientas, como SQL*Plus. Desde el prompt de SQL*Plus, Oracle reconoce el comienzo de un bloque anónimo cuando encuentra la palabra reservada DECLARE o BEGIN. Cuando esto ocurre, limpia el buffer y entra en modo INPUT. La última línea del bloque debe ser un punto (.). Esto provoca que el bloque se guarde en el buffer y podremos ejecutarlo con la orden RUN. También se puede usar la barra oblicua (/) en lugar del punto. En este caso, además de almacenarse en el buffer, se ejecutará. El bloque del buffer se puede guardar en un fichero con la orden SAVE. Para cargar un fichero que contiene un bloque en el buffer SQL se usará la orden GET. Después se podrá ejecutar. También se puede utilizar la orden START para cargar y ejecutar el bloque con una sola orden.

```
SQL>   DECLARE
  2     v_precio NUMBER;
  3     BEGIN
  4         SELECT precio_uni INTO v_precio
  5             FROM productos
  6             WHERE COD_PRODUCTO = 7;
  7     DBMS_OUTPUT.PUT_LINE(v_precio);
  8     END;
  9     /
1000
Procedimiento PL/SQL terminado con éxito.
```

Este bloque muestra el precio del producto especificado.

```
BEGIN
    SELECT nombre INTO v_nom
            FROM clientes
            WHERE NIF = '&vs_nif';
    DBMS_OUTPUT.PUT_LINE (v_nom);
END;
```

- **Subprogramas almacenados.** Son bloques PL/SQL que tienen un nombre. Se trata de procedimientos y funciones que se compilan y almacenan en la BD, donde quedarán disponibles para ser ejecutados. Pueden recibir parámetros y en el caso de las funciones también pueden devolver un valor.
  En todo subprograma podemos distinguir 2 partes:
  - **La cabecera,** que contiene el nombre del subprograma, la definición de los parámetros con sus tipos y el tipo de valor de retorno, en el caso de las funciones.
  - **El cuerpo del subprograma.** Es un bloque PL/SQL que incluye declaraciones de variables (opcional), instrucciones y manejo de excepciones (opcional).

  A partir de ahora nos centraremos en los procedimientos.

  Tienen la siguiente estructura general:

```
PROCEDURE <nombre_procedimiento> [(<lista de parámetros>)]
[IS
   <declaraciones>];
BEGIN
   <instrucciones>;
[EXCEPTION
   <excepciones>];
END [<nombre_procedimiento>];
```

Los parámetros de la cabecera se definen sólo con el tipo de dato sin indicar tamaño y separados por comas:
<nombre_par>[IN|OUT|IN OUT]<tipo_dato>[{:=|DEFAULT}<valor>]

Por defecto u omisión un parámetro es IN.

Para crear un procedimiento lo haremos utilizando el comando siguiente:

```
CREATE [OR REPLACE]
PROCEDURE <nombre_procedimiento> [(<lista de parámetros>)]
[IS
   <declaraciones>];
BEGIN
   <instrucciones>;
[EXCEPTION
   <excepciones>];
END [<nombre_procedimiento>];
```

Con la opción OR REPLACE, si el procedimiento no existe lo crea y si ya existía lo sustituye o reemplaza por el nuevo.

```
SQL>  CREATE OR REPLACE
   2    PROCEDURE ver_cliente  (nomcli VARCHAR2)
   3    AS
   4        nifcli VARCHAR2 (10);
   5        domicli VARCHAR2 (15);
   6    BEGIN
   7        SELECT nif, domicilio INTO nifcli, domicli
   8           FROM clientes
   9           WHERE nombre=nomcli;
   10       DBMS_OUTPUT.PUT_LINE (‘Nombre:’ | |  nomcli | |  ‘Nif:’ | | Nifcli  | |
       ‘Domicilio:’ | | domicli);
   12   EXCEPTION
   13      WHEN NO_DATA_FOUND THEN
   14      DBMS_OUTPUT.PUT_LINE (‘No encontrado el cliente’
   15                                          | | nomcli);
   16   END ver_cliente;
```

Procedimiento creado.
El procedimiento se guardará en la BD.
NOTA: || sirve para concatenar. En PL/SQL una instrucción puede ocupar más de una línea, el ; indicará el final de la misma.

Si el compilador detecta errores, veremos el mensaje ‘Procedimiento creado con errores de compilación’. La orden SHOW ERRORS permite ver los errores detectados. Para corregirlos llamamos al editor para que nos muestre el procedimiento.

Cuando tenemos un procedimiento almacenado en la BD, para ejecutarlo lo hacemos con la orden EXECUTE .

SQL> SET SERVEROUTPUT ON
SQL> EXECUTE ver_cliente ('Pepe')

También podemos invocar al procedimiento desde un bloque PL/SQL.
SQL> BEGIN ver_cliente ('Antonio'); END;
Como cualquier otro lje., podemos insertar comentarios en cualquier parte del programa:

- Comentarios de una sola línea con --.
- Comentarios de varias líneas con /* <comentario> */.
  (Igual que en C).

- **Disparadores de bd.** Son subprogramas almacenados que se asocian a una tabla de la bd. Estos subprogramas se ejecutan automáticamente al producirse determinados cambios en la tabla (insercción, borrado o modificación). Son muy útiles para controlar los cambios que suceden en la bd, para implementar restricciones complejas, etc. Se verán más adelante.

2.6. TIPOS DE DATOS PL/SQL.

PL/SQL dispone de los mismos tipos de datos que SQL:

- CHAR(L).
- VARCHAR2(L)
- LONG(L).
- NUMBER(P,S).
- BOOLEAN. Almacena TRUE, FALSE y NULL.
- DATE.

Además dispone de otros tipos más complejos:

- Tipos compuestos: registros, tablas y arrays.
- Referencias: el equivalente a los punteros en C.
- LOB: objetos de gran tamaño.

Asimismo, permite que el usuario (programador) defina sus propios subtipos.

TIPOS DE DATOS EN ORACLE.

TIPO	CARACTERISTICAS	OBSERVACIONES
CHAR	Cadena de caracteres (alfanuméricos) de longitud fija	Entre 1 y 2000 bytes como máximo. Aunque se introduzca un valor más corto que el indicado en el tamaño, se rellenará al tamaño indicado. Es de longitud fija, siempre ocupará lo mismo, independientemente del valor que contenga
VARCHAR2	Cadena de caracteres de longitud variable	Entre 1 y 4000 bytes como máximo. El tamaño del campo dependerá del valor que contenga, es de longitud variable.
VARCHAR	Cadena de caracteres de longitud variable	En desuso, se utiliza VARCHAR2 en su lugar
NCHAR	Cadena de caracteres de longitud fija que sólo almacena caracteres Unicode	Entre 1 y 2000 bytes como máximo. El juego de caracteres del tipo de datos (datatype) NCHAR sólo puede ser AL16UTF16 ó UTF8. El juego de caracteres se especifica cuando se crea la base de datos Oracle

NVARCHAR2	Cadena de caracteres de longitud variable que sólo almacena caracteres Unicode	Entre 1 y 4000 bytes como máximo. El juego de caracteres del tipo de datos (datatype) NCHAR sólo puede ser AL16UTF16 ó UTF8. El juego de caracteres se especifica cuando se crea la base de datos Oracle
LONG	Cadena de caracteres de longitud variable	Como máximo admite hasta 2 GB (2000 MB). Los datos LONG deberán ser convertidos apropiadamente al moverse entre diversos sistemas. Este tipo de datos está obsoleto (en desuso), en su lugar se utilizan los datos de tipo LOB (CLOB,NCLOB). Oracle recomienda que se convierta el tipo de datos LONG a alguno LOB si aún se está utilizando. No se puede utilizar en claúsulas WHERE, GROUP BY, ORDER BY, CONNECT BY ni DISTINCT Una tabla sólo puede contener una columna de tipo LONG. Sólo soporta acceso secuencial.
LONG RAW	Almacenan cadenas binarias de ancho variable	Hasta 2 GB. En desuso, se sustituye por los tipos LOB.
RAW	Almacenan cadenas binarias de ancho variable	Hasta 32767 bytes. En desuso, se sustituye por los tipos LOB.
LOB (BLOG, CLOB, NCLOB, BFILE)	Permiten almacenar y manipular bloques grandes de datos no estructurados (tales como texto, imágenes, videos, sonidos, etc) en formato binario o del carácter	Admiten hasta 8 terabytes (8000 GB). Una tabla puede contener varias columnas de tipo LOB. Soportan acceso aleatorio. Las tablas con columnas de tipo LOB no pueden ser replicadas.
BLOB	Permite almacenar datos binarios no estructurados	Admiten hasta 8 terabytes
CLOB	Almacena datos de tipo carácter	Admiten hasta 8 terabytes
NCLOB	Almacena datos de tipo carácter	Admiten hasta 8 terabytes. Guarda los datos según el juego de caracteres Unicode nacional.
BFILE	Almacena datos binarios no estructurados en archivos del sistema operativo, fuera de la base de datos. Una columna BFILE almacena un localizador del archivo a uno externo que contiene los datos	Admiten hasta 8 terabytes. El administrador de la base de datos debe asegurarse de que exista el archivo en disco y de que los procesos de Oracle tengan permisos de lectura para el archivo .

ROWID	Almacenar la dirección única de cada fila de la tabla de la base de datos	ROWID físico almacena la dirección de fila en las tablas, las tablas en clúster, los índices, excepto en las índices-organizados (IOT). ROWID lógico almacena la dirección de fila en tablas de índice-organizado (IOT). Un ejemplo del valor de un campo ROWID podría ser: "AAAIugAAJAAC4AhAAI". El formato es el siguiente: Para "OOOOOOFFFBBBBBBRRR", donde: OOOOOO: segmento de la base de datos (AAAIug en el ejemplo). Todos los objetos que estén en el mismo esquema y en el mismo segmento tendrán el mismo valor. FFF: el número de fichero del tablespace relativo que contiene la fila (fichero AAJ en el ejemplo). BBBBBB: el bloque de datos que contiene a la fila (bloque AAC4Ah en el ejemplo). El número de bloque es relativo a su fichero de datos, no al tablespace. Por lo tanto, dos filas con números de bloque iguales podrían residir en diferentes datafiles del mismo tablespace. RRR: el número de fila en el bloque (fila AAI en el ejemplo). Este tipo de campo no aparece en los SELECT ni se puede modificar en los UPDATE, ni en los INSERT. Tampoco se puede utilizar en los CREATE. Es un tipo de datos utilizado exclusivamente por Oracle. Sólo se puede ver su valor utilizando la palabra reservada ROWID, por ejemplo: *select rowid, nombre, apellidos from clientes* Ejemplo 2: *SELECT ROWID, SUBSTR(ROWID,15,4) "Fichero", SUBSTR(ROWID,1,8) "Bloque", SUBSTR(ROWID,10,4) "Fila" FROM proveedores* Ejemplo 3: una forma de saber en cuántos ficheros de datos está alojada una tabla: SELECT COUNT(DISTINCT(SUBSTR(ROWID,7,3))) "Numero ficheros " FROM facturacion
UROWID	ROWID universal	Admite ROWID a tablas que no sean de Oracle, tablas externas. Admite tanto ROWID lógicos como físicos.

NUMBER	Almacena números fijos y en punto flotante	Se admiten hasta 38 dígitos de precisión y son portables a cualquier entre los diversos sistemas en que funcione Oracle. Para declarar un tipo de datos NUMBER en un CREATE ó UPDATE es suficiente con: nombre_columna NUMBER opcionalmente se le puede indicar la precisión (número total de dígitos) y la escala (número de dígitos a la derecha de la coma, decimales, los cogerá de la precisión indicada): nombre_columna NUMBER (precision, escala) Si no se indica la precisión se tomará en función del número a guardar, si no se indica la escala se tomará escala cero. Para no indicar la precisión y sí la escala podemos utilizar: nombre_columna NUMBER (*, escala) Para introducir números que no estén el el formato estándar de Oracle se puede utilizar la función TO_NUMBER.
FLOAT	Almacena tipos de datos numéricos en punto flotante	Es un tipo NUMBER que sólo almacena números en punto flotante
DATE	Almacena un punto en el tiempo (fecha y hora)	El tipo de datos DATE almacena el año (incluyendo el siglo), el mes, el día, las horas, los minutos y los segundos (después de medianoche). Oracle utiliza su propio formato interno para almacenar fechas. Los tipos de datos DATE se almacenan en campos de longitud fija de siete octetos cada uno, correspondiendo al siglo, año, mes, día, hora, minuto, y al segundo. Para entrada/salida de fechas, Oracle utiliza por defecto el formato *DD-MMM-AA*. Para cambiar este formato de fecha por defecto se utiliza el parámetro *NLS_DATE_FORMAT*. Para insertar fechas que no estén en el mismo formato de fecha estándar de Oracle, se puede utilizar la función *TO_DATE* con una máscara del formato: TO_DATE (el "13 de noviembre de 1992", "DD del MES, YYYY")
TIMESTAMP	Almacena datos de tipo hora, fraccionando los segundos	
TIMESTAMP WITH TIME ZONE	Almacena datos de tipo hora incluyendo la zona horaria (explícita), fraccionando los	

	segundos	
TIMESTAMP WITH LOCAL TIME ZONE	Almacena datos de tipo hora incluyendo la zona horaria local (relativa), franccionando los segundos	Cuando se usa un SELECT para mostrar los datos de este tipo, el valor de la hora será ajustado a la zona horaria de la sesión actual
XMLType	Tipo de datos abstracto. En realidad se trata de un CLOB.	Se asocia a un esquema XML para la definición de su estructura.

3. IDENTIFICADORES.

Los identificadores se utilizan para nombrar los objetos que intervienen en un programa: variables, constantes, cursores, excepciones, procedimientos, funciones, etc.

En PL/SQL pueden tener hasta 30 caracteres, empezando siempre por una letra, que puede ir seguida por letras, números, el signo de dólar, de almohadilla y el subguión.

Para dar nombre a los distintos tipos de objetos evitar siempre la ambigüedad y utilizar los siguientes prefijos que ayudan a distinguirlos:

- Nombres de cursores: **cur_name**. P.e. cur_emple.
- Nombres de variables que guardan datos de cursores: **v_name**. P.e. v_sal.
- Nombres de parámetros: **p_name**. P.e. p_deptno.
- Nombres de variables de registro: **vr_name**. P.e. vr_cli.

PL/SQL no diferencia entre mayúsculas y minúsculas en los identificadores.

4.- VARIABLES.

4.1. DECLARACIÓN E INICIALIZACIÓN DE VARIABLES.

Todas las variables PL/SQL deben declararse en la sección correspondiente antes de su uso. El formato genérico para declarar una variable es el siguiente:

<nombre_de_variable><tipo>[NOT NULL][{:= | DEFAULT}<valor>]

```
DECLARE
    Importe NUMBER (8,2);
    Contador NUMBER (2,0) := 0;
    Nombre VARCHAR2 (20) NOT NULL := ‘MIGUEL’; o bien
```

Nombre VARCHAR2 (20) NOT NULL DEFAULT ‘MIGUEL’;

Para cada variable se debe especificar el tipo. No se puede indicar una lista de variables del mismo tipo.
La opción NOT NULL fuerza a que la variable tenga siempre un valor. Si se usa, deberá inicializarse la variable en la declaración con DEFAULT o con := para la inicialización.
También se puede inicializar una variable que no tenga la opción NOT NULL.
Si no se inicializan las variables, en PL/SQL se garantiza que su valor es NULL. No obstante, no se debe hacer referencia a una variable antes de que tenga asignado un valor.

Uso de los atributos %TYPE y %ROWTYPE.

%TYPE declara una variable del mismo tipo que otra, o que una columna de una tabla.

Total Importe%TYPE;

Declara la variable total del mismo tipo que Importe, que se habrá definido previamente.

%ROWTYPE declara una variable de registro cuyos campos se corresponden con las columnas de una tabla o vista de la base de datos. Para hacer referencia a cada uno de los campos, indicaremos el nombre de la variable, un punto y el nombre del campo que coincide con el de la columna correspondiente.

Moroso Clientes%ROWTYPE;

Crea una variable de registro que podrá contener una fila de la tabla clientes.

Moroso.nombre.

Hace referencia al campo nombre de la variable de registro moroso.

4.2. CONSTANTES.

Se pueden declarar constantes mediante el siguiente formato:

<nombre_constante>CONSTANT<tipo>:=<valor>;

En el caso de las constantes, siempre se deberá asignar un valor en la declaración.

Porcentaje_iva CONSTANT NUMBER :=16;

4.3. ÁMBITO Y VISIBILIDAD DE LAS VARIABLES.

El ámbito de una variable es el bloque en el que se declara y los bloques "hijos" de dicho bloque. La variable será local para el bloque en el que ha sido declarada y global para los bloques hijos de éste. Sin embargo, las variables declaradas en los bloques hijos no son accesibles desde el bloque padre.

```
DECLARE  ---------------------------------Bloque padre
   v1 CHAR;
BEGIN
   …
    v1 := 1;
    DECLARE  -----------------------------Bloque hijo
       v2 CHAR;
       BEGIN
             v2 := 2;
             …
             v1 := v2;
             …
        END;  -----------------------------Fin bloque hijo
        V2 := v1;   Error v2 es desconocida en este ámbito.
END;  -------------------------------------Fin bloque padre
```

v1 es accesible para los dos bloques (es local al bloque padre y global para el bloque hijo), mientras que v2 solamente es accesible para el bloque hijo. Las variables se crean al comienzo del bloque y dejan de existir una vez finalizada la ejecución del bloque en el que han sido declaradas.

5.- OPERADORES.

Como cualquier otro lenguaje, PL/SQL dispone de operadores que se utilizan para asignar valores y formar expresiones de distintos tipos.

- **Asignación.** Para asignar un valor a una variable se usa el operador :=.
- **Lógicos.** AND, OR y NOT.
- **Concatenación.** Para unir dos o más cadenas se utiliza el operador de concatenación ||. Por ejemplo 'buenos'||'días' daría como resultado 'buenosdías'.
- **Comparación.** Igual =, distinto !=, <, >, <=, >=. Otros operadores de comparación son IS NULL, BETWEEN, LIKE, IN, etc (igual que en SQL).
- **Aritméticos.** Se emplean para realizar cálculos. +, -, *, /, **.

El orden de prioridad de los operadores determina el orden de evaluación de los operandos de una expresión.

Prioridad	Operador	Operación
1	**, NOT	Exponenciación, Negación
2	*, /	Multiplicación y División
3	+, -, ||	Suma, resta y concatenación
4	=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparación
5	AND	Conjunción
6	OR	Inclusión

Aunque esta es la prioridad establecida por defecto, podemos cambiarla utilizando paréntesis.

Los operadores que se encuentran en el mismo grupo tienen la misma prioridad. En estos casos no se garantiza el orden de evaluación. Si queremos que se evalúen en algún orden concreto, deberemos utilizar paréntesis.

6.- FUNCIONES.

En PL/SQL se pueden utilizar las funciones de SQL. Aunque hay que indicar que las funciones de grupo sólo se pueden utilizar dentro de cláusulas de sección (SELECT).

Veámoslo con un ejemplo: Función que pasándole un departamento nos devuelva el sueldo total del mismo

```
CREATE  OR  REPLACE  FUNCTION    SALARIO_DEPT
    (CODIGO        DEPARTMENT . DEPARTMENT_ID % TYPE)
      SALARIO      NUMBER (8);
      RETURN       NUMBER
      IS
BEGIN
    SELECT  SUM (salary) INTO   SALARIO
    FROM  EMPLOYEE
    WHERE  DEPARTMENT_ID  =  CODIGO;
    RETURN  ( SALARIO);
END SALARIO_DEPT ;
```

Llamar a la función como un bloque PLSQL

```
DECLARE
    SALARIO NUMBER;
    BEGIN
    SALARIO:= SALARIO_DEPT (13);
    DBMS_OUTPUT.PUT_LINE(SALARIO);
END;
```

Comprobamos que la salida es la misma que si hacemos la select

```
SELECT SUM(SALARY) FROM EMPLOYEE WHERE DEPARTMENT_ID=13;
```

7.- ESTRUCTURAS DE CONTROL.

La mayoría de las estructuras de control requieren evaluar una condición, que en PL/SQL puede dar tres resultados: TRUE, FALSE o NULL. Pues bien, a efectos de estas estructuras, el valor NULL es equivalente a FALSE, es decir, se considerará que se cumple la condición sólo si el resultado es TRUE. En caso contrario (FALSE o NULL), se considerará que no se cumple.

- **Alternativa simple.**

```
IF <condición> THEN
        Instrucciones;
END IF;
```

Si la condición se cumple, se ejecutan las instrucciones que siguen a la cláusula THEN.

- **Alternativa doble.**

```
IF <condición> THEN
        Instrucciones1;
ELSE
        Instrucciones2;
END IF;
```

Si la condición se cumple, se ejecutan las instrucciones que siguen a la cláusula THEN. En caso contrario, se ejecutarán las instrucciones que siguen a la cláusula ELSE.

- **Alternativa múltiple.**

```
IF <condición1> THEN
        Instrucciones1;
ELSIF <condición2> THEN
        Instrucciones2;
ELSIF <condición3> THEN
        Instrucciones3;
…
[ELSE
        InstruccionesN;]
END IF;
```

Evalúa, comenzando desde el principio, cada condición, hasta encontrar alguna que se cumpla, en cuyo caso ejecutará las instrucciones que siguen a la cláusula THEN correspondiente. La cláusula ELSE es opcional; en caso de que se utilice, se ejecuta cuando no se ha cumplido ninguna de las condiciones anteriores.

- **Mientras.**

WHILE <condición> LOOP
Instrucciones;
END LOOP;
Es un bucle que se ejecutará mientras se cumpla la condición. Se evalúa la condición y si se cumple se ejecutarán las instrucciones del bucle.

- **Para.**

Cuando se puede conocer a priori el número de veces que se debe ejecutar un bucle, se suele utilizar la estructura PARA o FOR, cuyo formato es el siguiente:

FOR <variablecontrol> IN <valorInicio> . . <valorFinal> LOOP
Instrucciones;
END LOOP;
Donde <variablecontrol> es la variable de control que cuenta las veces que se va realizando el bucle. No hay que declararla, es local al bucle y no es accesible desde el exterior del bucle, ni siquiera en el mismo bloque.

```
FOR i IN 1..100 LOOP
        Instrucciones;
        …;
END LOOP;
```

El incremento siempre es una unidad, pero puede ser negativo utilizando la opción REVERSE. En este caso, comenzará siempre por el valor especificado en segundo lugar e irá restando una unidad en cada iteración.

FOR <variable> IN REVERSE <valorInicio> . . <valorFinal> LOOP
Instrucciones:
…;
END LOOP;

```
SET SERVEROUTPUT ON;
BEGIN
FOR i IN REVERSE 1 . . 3 LOOP
        DBMS_OUTPUT.PUT_LINE( i );
END LOOP;
END;
/
3
2
1
Procedimiento PL/SQL terminado con éxito.
```

8.- CURSORES EXPLÍCITOS.

Hasta el momento hemos venido utilizando cursores implícitos. Este tipo de cursor es muy sencillo y cómodo de usar, pero plantea diversos problemas. El más importante es que la subconsulta debe devolver una sola fila, de lo contrario, se produciría un error. Por ello, dado que normalmente una consulta devolverá varias filas, se suelen manejar cursores explícitos.

Hay cuatro operaciones básicas para trabajar con un cursor explícito:

1. **Declaración del cursor.** El cursor se declara en la zona de declaraciones según el siguiente formato:

CURSOR <nombrecursor> IS SELECT <sentencia SELECT>;

2. **Apertura del cursor.** En la zona de instrucciones hay que abrir el cursor:

OPEN <nombrecursor>;

Al hacerlo se ejecuta automáticamente la sentencia SELECT asociada y sus resultados se almacenan en las estructuras internas de memoria manejadas por el cursor. Para acceder a la información debemos dar el paso siguiente.

3. **Recogida de información.** Para recoger la información almacenada en el cursor utilizamos el siguiente formato:

FETCH <nombrecursor> INTO {<variable_registro> | <listavariables>};

Después del INTO figurará una variable que recogerá la información de todas las columnas de una tabla (en este caso, la variable puede ser declarada con %ROWTYPE) o una lista de variables, cada una recogerá la columna correspondiente de la orden SELECT. Por tanto serán del mismo tipo que las columnas. Cada FETCH recupera una fila y el cursor avanza automáticamente a la fila siguiente.

4. **Cierre del cursor.** Cuando el cursor no se va a utilizar hay que cerrarlo:

CLOSE <nombrecursor>;

El cursor es un identificador, no una variable. Solamente se puede usar para hacer referencia a una consulta. No se le pueden asignar valores ni utilizar en expresiones.

```
DECLARE
    CURSOR cur1 IS
        SELECT dnombre, loc FROM depart;
    v_nombre VARCHAR2 (14);
    v_localidad VARCHAR2 (14);

BEGIN
    OPEN cur1;
    FETCH cur1 INTO v_nombre, v_localidad;
    WHILE cur1%FOUND LOOP
        DBMS_OUTPUT.PUT_LINE (v_nombre || '*' || v_localidad);
        FETCH cur1 INTO v_nombre, v_localidad;
    END LOOP;
    CLOSE cur1;
END;
```

8.1 ATRIBUTOS DEL CURSOR.

Para conocer detalles respecto a la situación del cursor hay cuatro atributos para consultar:

- **%FOUND.** Devuelve verdadero si el último FETCH ha recuperado algún valor; en caso contrario, devuelve falso. Si el cursor no estaba abierto devuelve error, y si estaba abierto pero no se había ejecutado aún ningún FETCH, devuelve NULL. Se suele utilizar como condición de continuación en bucles para recuperar información.

- **%NOTFOUND.** Hace lo contrario que el atributo anterior.

- **%ROWCOUNT.** Devuelve el número de filas recuperadas hasta el momento por el cursor (número de FETCH realizados satisfactoriamente).

- **%ISOPEN.** Devuelve verdadero si el cursor está abierto.

8. 2 CURSOR FOR . . LOOP.

Esta estructura simplifica las tareas realizadas por el cursor. Su formato es el siguiente:

1. **Se declara la información cursor** en la sección correspondiente.

2. **Se procesa el cursor** utilizando el siguiente formato:
   FOR nombrevareg IN nombrecursor LOOP
   …;
   END LOOP;

Al entrar en el bucle, se abre el cursor de manera automática, se declara implícitamente la variable *nombrevareg* de tipo *nombrecursor*%ROWTYPE y se ejecuta el primer FETCH, cuyo resultado quedará en *nombrevareg*. A continuación se realizarán las acciones que correspondan hasta llegar al END LOOP, que sube de nuevo al FOR … LOOP ejecutando el siguiente FETCH, y depositando otra vez el resultado en *nombrevareg* y así sucesivamente, hasta procesar la última fila de la consulta, momento en el que se producirá la salida del bucle y se cerrará automáticamente el cursor.
La variable *nombrevareg* es local al bucle, por tanto, al salir del bucle no estará disponible.

Dentro del bucle se puede hacer referencia a la variable de registro y a sus campos (cuyo nombre se corresponde con las columnas de la consulta) usando la notación del punto.

```
DECLARE
   CURSOR cur_emple IS
         SELECT apellido, fecha_alt FROM emple
         ORDER BY fecha_alt;
BEGIN
   FOR v_reg_emp IN cur_emple LOOP
      DBMS_OUTPUT.PUT_LINE (v_reg_emp.apellido || '*' ||
      v_reg_emple.fecha_alt);
   END LOOP;
END;
```

8.3 USO DE ALIAS EN CURSORES.

Cuando en las consultas de un cursor se usan funciones de grupo, hay que utilizar alias para dar nombre a esas funciones de grupo.

8.4 CURSORES CON PARÁMETROS.

Un cursor puede tener parámetros. En este caso se aplicará el siguiente formato genérico:

CURSOR *nombrecursor [(parámetro1, parámetro2,…)]*
IS SELECT <sentencia select en la que intervendrán los parámetros>;

Los parámetros tienen la siguiente sintaxis:

***Nombredevariable* [IN] *tipodedato* [{ := | DEFAULT } *valor*]**

El ámbito de estos parámetros es local al cursor.

```
DECLARE
    …
    CURSOR cur1 ( v_departamento NUMBER,
                  v_oficio VARCHAR2 DEFAULT 'DIRECTOR' )
          IS SELECT apellido, salario FROM emple
          WHERE dept_no = v_departamento AND oficio = v_oficio;
```

Para abrir un cursor pasándole parámetros lo haremos así:

OPEN *nombrecursor* [(***parámetro1, parámetro2, …***)];

Donde parámetro1, parámetro2, son expresiones que contienen los valores que se pasarán al cursor. No tienen por qué ser los mismos nombres de las variables indicadas como parámetros al declarar el cursor.

```
DECLARE

    CURSOR cur1 ( v_departamento NUMBER,
                  v_oficio VARCHAR2 DEFAULT 'DIRECTOR' )
          IS SELECT apellido, salario FROM emple
          WHERE dept_no = v_departamento AND oficio = v_oficio;
…
```

Cualquiera de los siguientes comandos abrirá el cursor:

```
BEGIN
…
OPEN cur1 (v_departamento);
OPEN cur1 (v_departamento, v_oficio);
OPEN cur1 (20, 'VENDEDOR');
…
```

9.- TRIGGER.

Un Trigger es un programa almacenado, creado para ejecutarse automaticamente cuando ocurra un evento en nuestra base de datos. Dichos eventos son generados por los comandos INSERT, UPDATE y DELETE, los cuales hacen parte del DML.

9.1 CREAR UN TRIGGER

Usaremos una sintaxis similar a la creación de Procedimientos y Funciones en MySQL. Observemos:

```
CREATE [DEFINER={usuario|CURRENT_USER}]
TRIGGER nombre_del_trigger {BEFORE|AFTER}
{UPDATE|INSERT|DELETE}
ON nombre_de_la_tabla
FOR EACH ROW
<bloque_de_instrucciones>
```

Obviamente la sentencia CREATE es conocidisima para crear nuevos objetos en la base de datos. Eso ya lo tienes claro. Enfoquemos nuestra atención en las otras partes de la definición:

- DEFINER={usuario|CURRENT_USER}

: Indica al gestor de bases de datos qué usuario tiene privilegios en su cuenta, para la invocación de los triggers cuando surjan los eventos DML. Por defecto este característica tiene el valor CURRENT_USER que hace referencia al usuario actual que esta creando el Trigger.

- nombre_del_trigger:

Indica el nombre de nuestro trigger. Existe una nomenclatura muy práctica para nombrar un trigger, la cual nos da mejor legibilidad en la administracion de la base de datos. Primero ponemos el nombre de tabla, luego especificamos con la inicial de la operación DML y seguido usamos la inicial del momento de ejecución(AFTER o BEFORE). Por ejemplo:
-- BEFORE INSERT
clientes_BI_TRIGGER

BEFORE|AFTER: Especifica si el Trigger se ejecuta antes o después del evento DML.

UPDATE|INSERT|DELETE:
Aquí eliges que sentencia usarás para que se ejecute el Trigger.

ON nombre_de_la_tabla:
En esta sección estableces el nombre de la tabla asociada.

FOR EACH ROW: Establece que el Trigger se ejecute por cada fila en la tabla asociada.

<bloque_de_instrucciones>: Define el bloque de sentencias que el Trigger ejecutará al ser invocado.

9.2 IDENTIFICADORES NEW Y OLD EN TRIGGERS

- Si queremos relacionar el trigger con columnas especificas de una tabla debemos usar los identificadores OLD y NEW.

- Con ellos podemos referenciar a los valores antes y después de la actualización a nivel de fila
- Solo se puede utilizar a nivel de fila
- Se utiliza :new.columna :old.columna
- Ejemplo: if :new.columna<:old.columna
- Cuando hacemos DELETE :new.columna es NULL
- Cuando hacemos INSERT :old.columna es NULL
- Cuando hacemos UPDATE ambas tienen valor
- Cuando utilicemos new y old la clausuala WHEN del trigger no hace falta

9.3 TRIGGERS

OLD indica el valor antiguo de la columna y NEW el valor nuevo que pudiese tomar. Por ejemplo: OLD.idproducto ó NEW.idproducto.

- Si usamos la sentencia UPDATE podremos referirnos a un valor OLD y NEW, ya que modificaremos registros existentes por nuevos valores.
- En cambio si usamos INSERT solo usaremos NEW, ya que su naturaleza es únicamente de insertar nuevos valores a las columnas.
- Y si usamos DELETE usaremos OLD debido a que borraremos valores que existen con anterioridad.

BEFORE Y AFTER

Estas clausulas indican si el Trigger se ejecuta **antes o después** del evento DML. Hay ciertos eventos que no son compatibles con estas sentencias.

Por ejemplo,

- si tuvieras un Trigger AFTER que se ejecuta en una sentencia UPDATE, sería ilógico editar valores nuevos NEW, sabiendo que el evento ya ocurrió.
- Igual sucedería con la sentencia INSERT, el Trigger tampoco podría referenciar valores NEW, ya que los valores que en algún momento fueron NEW, han pasado a ser OLD.

9.4 ¿QUÉ UTILIDADES TIENEN LOS TRIGGERS?

- Con los Triggers podemos implementar varios casos de uso que mantengan la integridad de la base de datos, como Validar información, Calcular atributos derivados, Seguimientos de movimientos en la base de datos, etc.

- Cuando surja una necesidad en donde veas que necesitas que se ejecute una acción implícitamente(sin que la ejecutes manualmente) sobre los registros de una tabla, entonces puedes considerar el uso de un Trigger.

9.5 EJEMPLO DE TRIGGER BEFORE EN LA SENTENCIA UPDATE

A continuación veremos un Trigger que valida la edad de un cliente antes de una sentencia UPDATE. Si por casualidad el nuevo valor es negativo, entonces asignaremos NULL a este atributo.

```
DELIMITER //
CREATE TRIGGER cliente_BU_Trigger
BEFORE UPDATE ON cliente FOR EACH ROW
BEGIN
-- La edad es negativa?
IF NEW.edad
```

Este Trigger se ejecuta antes de haber insertado el registro, lo que nos da el poder de verificar primero si el nuevo valor de la edad esta en el rango apropiado, si no es así entonces asignaremos NULL a ese campo. Grandes los Triggers!

9.6 EJEMPLO DE TRIGGER AFTER EN LA SENTENCIA UPDATE

Supongamos que tenemos una Tienda de accesorios para Gamers. Para la actividad de nuestro negocio hemos creado un sistema de facturación muy sencillo, que registra las ventas realizadas dentro de una factura que contiene el detalle de las compras.

Nuestra tienda tiene 4 vendedores de turno, los cuales se encargan de registrar las compras de los clientes en el horario de funcionamiento.
Implementaremos un Trigger que guarde los cambios realizados sobre la tabla DETALLE_FACTURA de la base de datos realizados por los vendedores.

Veamos la solución:

```
DELIMITER //
CREATE TRIGGER detalle_factura_AU_Trigger
AFTER UPDATE ON detalle_factura FOR EACH ROW
BEGIN
INSERT INTO log_updates
(idusuario, descripcion)
VALUES (user( ),
CONCAT('Se modificó el registro ','(',
OLD.iddetalle,',', OLD.idfactura,',',OLD.idproducto,',',
OLD.precio,',', OLD.unidades,') por ',
'(', NEW.iddetalle,',', NEW.idfactura,',',NEW.idproducto,',',
NEW.precio,',', NEW.unidades,')'));

END//

DELIMITER ;
```

Con este registro de logs podremos saber si algún vendedor “ocioso” esta alterando las facturas, lo que lógicamente sería atentar contra las finanzas de nuestro negocio. Cada registro nos informa el usuario que modificó la tabla DETALLE_FACTURA y muestra una descripción sobre los cambios en cada columna.

9.7 EJEMPLO DE TRIGGER BEFORE EN LA SENTENCIA INSERT

El siguiente ejemplo que te voy a mostrar ¡me encanta!, ya que muestra como mantener la integridad de una base de datos con respecto a una atributo derivado.

Supón que tienes una Tienda de electrodomésticos y que has implementado un sistema de facturación. En la base de datos que soporta la información de tu negocio, existen varias tablas, pero nos vamos a centrar en la tabla PEDIDO y la tabla TOTAL_VENTAS.

TOTAL_VENTAS almacena las ventas totales que se le han hecho a cada cliente del negocio. Es decir, si el cliente Armado Barreras en una ocasión compró 1000 dolares, luego compró 1250 dolares y hace poco ha vuelto a comprar 2000 dolares, entonces el total vendido a este cliente es de 4250 dolares.

Pero supongamos que eliminamos el ultimo pedido hecho por este cliente, ¿que pasaría con el registro en TOTAL_VENTAS ?,…¡exacto!, quedaría desactualizado.
Usaremos tres Triggers para solucionar esta situación. Para que cada vez que usemos un comando DML en la tabla PEDIDO, no tengamos que preocuparnos por actualizar manualmente TOTAL_VENTAS.

Veamos:

```
-- TRIGGER PARA INSERT

DELIMITER //
CREATE TRIGGER PEDIDO_BI_TRIGGER
BEFORE INSERT ON PEDIDO
FOR EACH ROW
BEGIN
DECLARE cantidad_filas INT;
SELECT COUNT(*)
INTO cantidad_filas
FROM TOTAL_VENTAS
WHERE idcliente=NEW.idcliente;
IF cantidad_filas > 0 THEN
UPDATE TOTAL_VENTAS
SET total=total+NEW.total
```

```
WHERE idcliente=NEW.idcliente;
ELSE
INSERT INTO TOTAL_VENTAS
(idcliente,total)
VALUES(NEW.idcliente,NEW.total);
END IF;
END//
```

-- TRIGGER PARA UPDATE

```
CREATE TRIGGER PEDIDO_BU_TRIGGER
BEFORE UPDATE ON PEDIDO
FOR EACH ROW
BEGIN
UPDATE TOTAL_VENTAS
SET total=total+(NEW.total-OLD.total)
WHERE idcliente=NEW.idcliente;
END//
```

-- TRIGGER PARA DELETE

```
CREATE TRIGGER PEDIDO_BD_TRIGGER
BEFORE DELETE ON PEDIDO
FOR EACH ROW
BEGIN
UPDATE TOTAL_VENTAS
SET total=total-OLD.total
WHERE idcliente=OLD.idcliente;
END//
```

Con todos ellos mantendremos el total de ventas de cada cliente actualizado dependiendo del evento realizado sobre un pedido.

Si insertamos un nuevo pedido generado por un cliente existente, entonces vamos rápidamente a la tabla TOTAL_VENTAS y actualizamos el total comprado por ese cliente con una sencilla suma acumulativa.

Ahora, si cambiamos el monto de un pedido, entonces vamos a TOTAL_VENTAS para descontar el monto anterior y adicionar el nuevo monto.

Y si eliminamos un pedido de un cliente simplemente descontamos del total acumulado el monto que con anterioridad habíamos acumulado. ¿Práctico cierto?

9.8 VER LA INFORMACIÓN DE UN TRIGGER EN MYSQL

Si!, usa el comando SHOW CREATE TRIGGER y rápidamente estarás viéndolas especificaciones de tu Trigger creado. Observa el siguiente ejemplo:

SHOW CREATE TRIGGER futbolista_ai_trigger;

También puedes ver los Triggers que hay en tu base de datos con: **SHOW TRIGGERS;**

9.9 ELIMINAR UN TRIGGER EN MYSQL

DROP, DROP y más DROP. Como ya sabes usamos este comando para eliminar casi cualquier cosa en nuestra base de datos:

DROP TRIGGER [IF EXISTS] nombre_trigger

Recuerda que podemos adicionar la condicion IF EXISTS para indica que si el Trigger ya existe, entonces que lo borre.