(5) PL/SQL

(5.1) introducción al SQL procedimental

Casi todos los grandes Sistemas Gestores de Datos incorporan utilidades que permiten ampliar el lenguaje **SQL** para producir pequeñas utilidades que añaden al SQL mejoras de la programación estructurada (bucles, condiciones, funciones,....). La razón es que hay diversas acciones en la base de datos para las que SQL no es suficiente.

Por ello todas las bases de datos incorporan algún lenguaje de tipo procedimental (de tercera generación) que permite manipular de forma más avanzada los datos de la base de datos.

PL/SQL es el lenguaje procedimental que es implementado por el precompilador de **Oracle**. Es una extensión procedimental del lenguaje SQL; es decir, se trata de un lenguaje creado para dar a SQL nuevas posibilidades. Esas posibilidades permiten utilizar condiciones y bucles al estilo de los lenguajes de tercera generación (como **Basic**, **Cobol**, **C++, Java**, etc.).

En otros sistemas gestores de bases de datos existen otros lenguajes procedimentales: **SQL Server** utiliza **Transact SQL**, **Informix** usa **Informix 4GL**,...

Lo interesante del lenguaje PL/SQL es que integra SQL por lo que gran parte de su sintaxis procede de dicho lenguaje.

PL/SQL es un lenguaje pensado para la gestión de datos. La creación de aplicaciones sobre la base de datos se realiza con otras herramientas (**Oracle Developer**) o lenguajes externos como **Visual Basic** o **Java**. El código PL/SQL puede almacenarse:

- En la propia base de datos
- En archivos externos

(5.1.2) funciones que pueden realizar los programas PL/SQL

Las más destacadas son:

- Facilitar la realización de tares administrativas sobre la base de datos (copia de valores antiguos, auditorías, control de usuarios,…)
- Validación y verificación avanzada de usuarios
- Consultas muy avanzadas

- Tareas imposibles de realizar con SQL

(5.1.3) conceptos básicos

bloque PL/SQL

Se trata de un trozo de código que puede ser interpretado por Oracle. Se encuentra inmerso dentro de las palabras **BEGIN** y **END**.

programa PL/SQL

Conjunto de bloques que realizan una determinada labor.

procedimiento

Programa PL/SQL almacenado en la base de datos y que puede ser ejecutado si se desea con solo saber su nombre (y teniendo permiso para su acceso).

función

Programa PL/SQL que a partir de unos datos de entrada obtiene un resultado (datos de salida). Una función puede ser utilizada desde cualquier otro programa PL/SQL e incluso desde una instrucción SQL.

trigger (*disparador*)

Programa PL/SQL que se ejecuta automáticamente cuando ocurre un determinado suceso a un objeto de la base de datos.

paquete

Colección de procedimientos y funciones agrupados dentro de la misma estructura. Similar a las bibliotecas y librerías de los lenguajes convencionales.

(5.2) escritura de PL/SQL

(5.2.1) estructura de un bloque PL/SQL

Ya se ha comentado antes que los programas PL/SQL se agrupan en estructuras llamadas **bloques**. Cuando un bloque no tiene nombre, se le llama **bloque anónimo**. Un bloque consta de tres secciones:

- **Declaraciones.** Define e inicializa las variables, constantes, excepciones de usuario y cursores utilizados en el bloque. Va precedida de la palabra **DECLARE**
- **Comandos ejecutables.** Sentencias para manipular la base de datos y los datos del programa. Todas estas sentencias van precedidas por la palabra **BEGIN.**
- **Tratamiento de excepciones.** Para indicar las acciones a realizar en caso de error. Van precedidas por la palabra **EXCEPTION**
- **Final del bloque.** La palabra **END** da fin al bloque.

La estructura en sí es:

```
[DECLARE
    declaraciones ]
BEGIN
    instrucciones ejecutables
[EXCEPTION
    instrucciones de manejo de errores ]
END;
```

A los bloques se les puede poner nombre usando (así se declara un procedimiento):

```
PROCEDURE nombre IS
bloque
```

para una función se hace:

```
FUNCTION nombre
RETURN tipoDedatos IS
bloque
```

Cuando un bloque no se declara como procedimiento o función, se trata de un bloque anónimo.

(5.2.2) escritura de instrucciones PL/SQL

normas básicas

La mayor parte de las normas de escritura en PL/SQL proceden de SQL, por ejemplo:

- Las palabras clave, nombres de tabla y columna, funciones,… no distinguen entre mayúsculas y minúsculas
- Todas las instrucciones finalizan con el signo del punto y coma (;), excepto las encabezan un bloque
- Los bloques comienzan con la palabra BEGIN y terminan con END
- Las instrucciones pueden ocupar varias líneas

comentarios

Pueden ser de dos tipos:

- **Comentarios de varias líneas.** Comienzan con /* y terminan con */
- **Comentarios de línea simple.** Son los que utilizan los signos -- (doble guión). El texto a la derecha de los guiones se considera comentario (el de la izquierda no)

Ejemplo:

```
DECLARE
   v NUMBER := 17;
BEGIN
   /* Este es un comentario que
   ocupa varias líneas */
   v:=v*2; -- este sólo ocupa esta línea
   DBMS_OUTPUT.PUT_LINE(v) -- escribe 34
END;
```

(5.3) variables

(5.3.1) uso de variables

declarar variables

Las variables se declaran en el apartado DECLARE del bloque. PL/SQL no acepta entrada ni salida de datos por sí mismo (para conseguirlo se necesita software auxiliar). La sintaxis de la declaración de variables es:

```
DECLARE
   identificador [CONSTANT] tipoDeDatos [:= valorInicial];
   [siguienteVariable...]
```

Ejemplos:

```
DECLARE
   pi CONSTANT NUMBER(9,7):=3.1415927;
   radio NUMBER(5);
   area NUMBER(14,2) := 23.12;
```

El operador := sirve para asignar valores a una variable. Este operador permite inicializar la variable con un valor determinado. La palabra CONSTANT indica que la variable no puede ser modificada (es una constante). Si no se inicia la variable, ésta contendrá el valor NULL.

Los identificadores de Oracle deben de tener 30 caracteres, empezar por letra y continuar con letras, números o guiones bajos (_) (también vale el signo de dólar ($) y la almohadilla (#)<9. No debería coincidir con nombres de columnas de las tablas ni con palabras reservadas (como SELECT).

En PL/SQL sólo se puede declarar una variable por línea.

tipos de datos para las variables

Las variables PL/SQL pueden pertenecer a uno de los siguientes datos (sólo se listan los tipos básicos, los llamados **escalares**), la mayoría son los mismos del SQL de Oracle.

tipo de datos	descripción
CHAR(n)	Texto de anchura fija
VARCHAR2(n)	Texto de anchura variable
NUMBER[(p[,s])]	Número. Opcionalmente puede indicar el tamaño del número (*p*) y el número de decimales (s)
DATE	Almacena fechas
TIMESTAMP	Almacena fecha y hora
INTERVAL YEAR TO MONTH	Almacena intervalos de años y meses
INTERVAL DAY TO SECOND	Almacena intervalos de días, horas, minutos y segundos
LONG	Para textos de más de 32767 caracteres
LONG RAW	Para datos binarios. PL/SQL no puede mostrar estos datos directamente
INTEGER	Enteros de -32768 a 32767
BINARY_INTEGER	Enteros largos (de -2.147.483.647 a -2.147.483.648)
PLS_INTEGER	Igual que el anterior pero ocupa menos espacio
BOOLEAN	Permite almacenar los valores **TRUE** (verdadero) y **FALSE** (falso)
BINARY_DOUBLE	Disponible desde la versión 10g, formato equivalente al **double** del lenguaje C. Representa números decimales en coma flotante.
BINARY_FLOAT	Otro tipo añadido en la versión 10g, equivalente al **float** del lenguaje C.

expresión %TYPE

Se utiliza para dar a una variable el mismo tipo de otra variable o el tipo de una columna de una tabla de la base de datos. La sintaxis es:

```
identificador variable | tabla.columna%TYPE;
```

Ejemplo:

```
nom personas.nombre%TYPE;
precio NUMBER(9,2);
precio_iva precio%TYPE;
```

La variable *precio_iva* tomará el tipo de la variable precio (es decir *NUMBER(9,2)*) la variable *nom* tomará el tipo de datos asignado a la columna *nombre* de la tabla *personas*.

(5.3.2) DBMS_OUTPUT.PUT_LINE

Para poder mostrar datos (fechas, textos y números), Oracle proporciona una función llamada **put_line** en el paquete **dbms_output**. Ejemplo:

```
DECLARE
   a NUMBER := 17;
BEGIN
    DBMS_OUTPUT.PUT_LINE(a);
END;
```

Eso escribiría el número 17 en la pantalla. Pero para ello se debe habilitar primero el paquete en el entorno de trabajo que utilicemos. En el caso de iSQL*Plus hay que colocar la orden interna (no lleva punto y coma):

```
SET SERVEROUTPUT ON
```

hay que escribirla antes de empezar a utilizar la función.

(5.3.3) alcance de las variables

Ya se ha comentado que en PL/SQL puede haber un bloque dentro de otro bloque. Un bloque puede anidarse dentro de:

- Un apartado **BEGIN**
- Un apartado **EXCEPTION**

Hay que tener en cuenta que las variables declaradas en un bloque concreto, son eliminadas cuando éste acaba (con su END correspondiente).

Ejemplo:

```
DECLARE
   v NUMBER := 2;
BEGIN
   v:=v*2;
     DECLARE
        z NUMBER := 3;
     BEGIN
         z:=v*3;
         DBMS_OUTPUT.PUT_LINE(z); --escribe 12
         DBMS_OUTPUT.PUT_LINE(v); --escribe 4
     END;
     DBMS_OUTPUT.PUT_LINE(v*2); --escribe 8
     DBMS_OUTPUT.PUT_LINE(z); --error
END;
```

En el ejemplo anterior, se produce un error porque **z** no es accesible desde ese punto, el bloque interior ya ha finalizado. Sin embargo desde el bloque interior sí se puede acceder a *v*

(5.3.4) operadores y funciones

operadores

En PL/SQL se permiten utilizar todos los operadores de SQL: los operadores aritméticos (+-*/), condicionales (> < != <> >= <= OR AND NOT) y de cadena (||).

A estos operadores, PL/SQL añade el operador de potencia **. Por ejemplo *4**3* es 4^3.

funciones

Se pueden utilizar las funciones de Oracle procedentes de SQL (TO_CHAR, SYSDATE, NVL, SUBSTR, SIN, etc., etc.) excepto la función DECODE y las funciones de grupo (SUM, MAX, MIN, COUNT,...)

A estas funciones se añaden diversas procedentes de paquetes de Oracle o creados por los programadores y las funciones GREATEST y LEAST

(5.3.5) instrucciones SQL permitidas

instrucciones SELECT en PL/SQL

PL/SQL admite el uso de un SELECT que permite almacenar valores en variables. Es el llamado SELECT INTO.

Su sintaxis es:

```
SELECT listaDeCampos
INTO listaDeVariables
FROM tabla
[JOIN ...]
[WHERE condición]
```

La cláusula INTO es obligatoria en PL/SQL y además la expresión SELECT **sólo puede devolver una única fila;** de otro modo, ocurre un error.

Ejemplo:

```
DECLARE
   v_salario NUMBER(9,2);
 V_nombre VARCHAR2(50);
BEGIN
   SELECT salario,nombre INTO v_salario, v_nombre
   FROM empleados WHERE id_empleado=12344;
   SYSTEM_OUTPUT.PUT_LINE('El nuevo salario será de ' ||
            salario*1.2 || 'euros');
END;
```

instrucciones DML y de transacción

Se pueden utilizar instrucciones DML dentro del código ejecutable. Se permiten las instrucciones INSERT, UPDATE, DELETE y MERGE; con la ventaja de que en PL/SQL pueden utilizar variables.

Las instrucciones de transacción ROLLBACK y COMMIT también están permitidas para anular o confirmar instrucciones.

(5.3.6) paquetes estándar

Oracle incorpora una serie de paquetes para ser utilizados dentro del código PL/SQL. Es el caso del paquete DBMS_OUTPUT que sirve para utilizar funciones y procedimientos de escritura como PUT_LINE. Por ejemplo DBMS_OUTPUT.NEW_LINE() sirve para escribir una línea en blanco en el buffer de datos.

números aleatorios

El paquete DBMS_RANDOM contiene diversas funciones para utilizar número aleatorios. Quizá la más útil es la función DBMS_RANDOM.RANDOM que devuelve un número entero (positivo o negativo) aleatorio (y muy grande). Por ello si deseáramos un número aleatorio entre 1 y 10 se haría con la expresión:

```
MOD(ABS(DBMS_RANDOM.RANDOM),10)+1
```

Entre 20 y 50 sería:

```
MOD(ABS(DBMS_RANDOM.RANDOM),31)+20
```

(5.3.7) instrucciones de control de flujo

Son las instrucciones que permiten ejecutar un bloque de instrucciones u otro dependiendo de una condición. También permiten repetir un bloque de instrucciones hasta cumplirse la condición (es lo que se conoce como bucles).

La mayoría de estructuras de control PL/SQL son las mismas que las de los lenguajes tradicionales como C o Pascal. En concreto PL/SQL se basa en el lenguaje Ada.

instrucción IF

Se trata de una sentencia tomada de los lenguajes estructurados. Desde esta sentencia se consigue que ciertas instrucciones se ejecuten o no dependiendo de una condición

sentencia IF simple

Sintaxis:

```
IF condicion THEN
   instrucciones
END IF;
```

Las instrucciones se ejecutan en el caso de que la condición sea verdadera. La condición es cualquier expresión que devuelva verdadero o falso. Ejemplo:

```
IF departamento=134 THEN
   salario := salario * 13;
 departamento := 123;
END IF;
```

sentencia IF-THEN-ELSE

Sintaxis:

```
IF condición THEN
   instrucciones
ELSE
   instrucciones
END IF;
```

En este caso las instrucciones bajo el ELSE se ejecutan si la condición es falsa.

sentencia IF-THEN-ELSIF

Cuando se utilizan sentencias de control es común desear anidar un IF dentro de otro IF.

Ejemplo:

```
IF saldo>90 THEN
   DBMS_OUTPUT.PUT_LINE('Saldo mayor que el esperado');
ELSE
   IF saldo>0 THEN
         DBMS_OUTPUT.PUT_LINE('Saldo menor que el esperado');
   ELSE
         DBMS_OUTPUT.PUT_LINE('Saldo NEGATIVO');
   END IF;
END IF;
```

Otra solución es utilizar esta estructura:

```
IF condición1 THEN
   instrucciones1
ELSIF condición2 THEN
   instrucciones3
[ELSIF.... ]
[ELSE
   instruccionesElse ]
END IF;
```

En este IF (que es el más completo) es evalúa la primera condición; si es verdadera se ejecutan las primeras instrucciones y se abandona el IF; si no es así

se mira la siguiente condición y si es verdadera se ejecutan las siguientes instrucciones, si es falsa se va al siguiente ELSIF a evaluar la siguiente condición, y así sucesivamente. La cláusula ELSE se ejecuta sólo si no se cumple ninguna de las anteriores condiciones.

Ejemplo (equivalente al anterior):

```
IF saldo>90 THEN
   DBMS_OUTPUT.PUT_LINE('Saldo mayor que el esperado');
ELSIF saldo>0 THEN
   DBMS_OUTPUT.PUT_LINE('Saldo menor que el esperado');
ELSE
   DBMS_OUTPUT.PUT_LINE('Saldo NEGATIVO');
END IF;
```

sentencia CASE

La sentencia CASE devuelve un resultado tras evaluar una expresión. Sintaxis:

```
CASE selector
   WHEN expresion1 THEN resultado1
   WHEN expresion2 THEN resultado2
   ...
   [ELSE resultadoElse]
END;
```

Ejemplo:

```
texto:= CASE actitud
            WHEN 'A' THEN 'Muy buena'
            WHEN 'B' THEN 'Buena'
            WHEN 'C' THEN 'Normal'
            WHEN 'D' THEN 'Mala'
            ELSE 'Desconocida'
   END;
```

Hay que tener en cuenta que la sentencia CASE sirve para devolver un valor y no para ejecutar una instrucción.

también se pueden escribir sentencias CASE más complicadas. Por ejemplo:

```
aprobado:= CASE
   WHEN actitud='A' AND nota>=4 THEN TRUE
   WHEN nota>=5 AND (actitud='B' OR actitud='C') THEN TRUE
   WHEN nota>=7  THEN TRUE
   ELSE FALSE
END;
```

bucles

bucle LOOP

Se trata de una instrucción que contiene instrucción que se repiten indefinidamente (bucle infinito). Se inicia con la palabra **LOOP** y finaliza con la palabra **END LOOP** y dentro de esas palabras se colocan las instrucciones que se repetirán.

Lógicamente no tiene sentido utilizar un bucle infinito, por eso existe una instrucción llamada **EXIT** que permite abandonar el bucle. Cuando Oracle encuentra esa instrucción, el programa continua desde la siguiente instrucción al END LOOP.

Lo normal es colocar EXIT dentro de una sentencia IF a fin de establecer una condición de salida del bucle. También se puede acompañar a la palabra EXIT de la palabra **WHEN** seguida de una condición. Si se condición es cierta, se abandona el bucle, sino continuamos dentro.

Sintaxis

```
LOOP
   instrucciones
   ...
   EXIT [WHEN condición]
END LOOP;
```

Ejemplo (bucle que escribe los números del 1 al 10):

```
DECLARE
  cont NUMBER :=1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE(cont);
    EXIT WHEN cont=10;
    cont:=cont+1;
  END LOOP;
END;
```

bucle WHILE

Genera un bucle cuyas instrucciones se repiten mientras la condición que sigue a la palabra WHILE sea verdadera. Sintaxis:

```
WHILE condición LOOP
   instrucciones
END LOOP;
```

En este bucle es posible utilizar (aunque no es muy habitual en este tipo de bucle) la instrucción **EXIT** o **EXIT WHEN**. La diferencia con el anterior es que este es más estructurado (más familiar para los programadores de lenguajes como Basic, Pascal, C, Java,...)

Ejemplo (escribir números del 1 al 10):

```
DECLARE
  cont NUMBER :=1;
BEGIN
  WHILE cont<=10 LOOP
    DBMS_OUTPUT.PUT_LINE(cont);
    cont:=cont+1;
  END LOOP;
END;
```

bucle FOR

Se utilizar para bucles con contador, bucles que se recorren un número concreto de veces. Para ello se utiliza una variable (contador) que no tiene que estar declarada en el DECLARE, esta variable es declarada automáticamente en el propio FOR y se elimina cuando éste finaliza.

Se indica el valor inicial de la variable y el valor final (el incremento irá de uno en uno). Si se utiliza la cláusula REVERSE, entonces el contador cuenta desde el valor alto al bajo restando 1.

Sintaxis:

```
FOR contador IN [REVERSE] valorBajo..valorAlto
   instrucciones
END LOOP;
```

bucles anidados

Se puede colocar un bucle dentro de otro sin ningún problema, puede haber un WHILE dentro de un FOR, un LOOP dentro de otro LOOP, etc.

Hay que tener en cuenta que en ese caso, la sentencia EXIT abandonaría el bucle en el que estamos:

```
FOR i IN 1..10 LOOP
   FOR j IN 1..30 LOOP
         EXIT WHEN j=5;
         ...
   END LOOP;
   ...
END LOOP;
```

El bucle más interior sólo cuenta hasta que j vale 5 ya que la instrucción EXIT abandona el bucle más interior cuando *j* llega a ese valor.

No obstante hay una variante de la instrucción EXIT que permite salir incluso del bucle más exterior. Eso se consigue poniendo una etiqueta a los bucles que se deseen. Una etiqueta es un identificador que se coloca dentro de los signos << y >> delante del bucle. Eso permite poner nombre al bucle.

Por ejemplo:

```
<<buclei>>
FOR i IN 1..10 LOOP
   FOR j IN 1..30 LOOP
         EXIT buclei WHEN j=5;
         ...
   END LOOP;
   ...
END LOOP buclei;
```

En este caso cuando *j* vale 5 se abandonan ambos bucles. No es obligatorio poner la etiqueta en la instrucción END LOOP (en el ejemplo en la instrucción *END LOOP buclei*), pero se suele hacer por dar mayor claridad al código.

(5.4) cursores

(5.4.1) introducción

Los cursores representan consultas SELECT de SQL que devuelven más de un resultado y que permiten el acceso a cada fila de dicha consulta. Lo cual significa que el cursor siempre tiene un puntero señalando a una de las filas del SELECT que representa el cursor.

Se puede recorrer el cursor haciendo que el puntero se mueva por las filas. Los cursores son las herramientas fundamentales de PL/SQL

(5.4.2) procesamiento de cursores

Los cursores se procesan en tres pasos:

(1) Declarar el cursor

(2) Abrir el cursor. Tras abrir el cursor, el puntero del cursor señalará a la primera fila (si la hay)

(3) Procesar el cursor. La instrucción FETCH permite recorrer el cursor registro a registro hasta que el puntero llegue al final (se dice que hasta que el cursor esté vacío)

(4) Cerrar el cursor

(5.4.3) declaración de cursores

Sintaxis:

```
CURSOR nombre IS sentenciaSELECT;
```

La sentencia SELECT indicada no puede tener apartado INTO. Lógicamente esta sentencia sólo puede ser utilizada en el apartado DECLARE.

Ejemplo:

```
CURSOR cursorProvincias IS
  SELECT p.nombre, SUM(poblacion) AS poblacion
  FROM localidades l
  JOIN provincias p USING (n_provincia)
  GROUP BY p.nombre;
```

(5.4.4) apertura de cursores

```
OPEN cursor;
```

Esta sentencia abre el cursor, lo que significa:

(1) Reservar memoria suficiente para el cursor

(2) Ejecutar la sentencia SELECT a la que se refiere el cursor

(3) Colocar el puntero de recorrido de registros en la primera fila

Si la sentencia SELECT del cursor no devuelve registros, Oracle no devolverá una excepción. Hasta intentar leer no sabremos si hay resultados o no.

(5.4.5) instrucción FETCH

La sentencia FETCH es la encargada de recorrer el cursor e ir procesando los valores del mismo:

```
FETCH cursor INTO listaDeVariables;
```

Esta instrucción almacena el contenido de la fila a la que apunta actualmente el puntero en la lista de variables indicada. La lista de variables tiene tener el mismo tipo y número que las columnas representadas en el cursor (por supuesto el orden de las variables se tiene que corresponder con la lista de columnas). Tras esta instrucción el puntero de registros avanza a la siguiente fila (si la hay).

Ejemplo:

```
FETCH cursorProvincias INTO v_nombre, v_poblacion;
```

Una instrucción FETCH lee una sola fila y su contenido lo almacena en variables. Por ello se usa siempre dentro de bucles a fin de poder leer todas las filas de un cursor:

```
LOOP
   FETCH cursorProvincias INTO (v_nombre, v_poblacion);
   EXIT WHEN... --aquí se pondría la condición de salida
   ... --instrucciones de proceso de los datos del cursor
END LOOP;
```

(5.4.6) cerrar el cursor

```
CLOSE cursor;
```

Al cerrar el cursor se libera la memoria que ocupa y se impide su procesamiento (no se podría seguir leyendo filas). Tras cerrar el cursor se podría abrir de nuevo.

(5.4.7) atributos de los cursores

Para poder procesar adecuadamente los cursores se pueden utilizar una serie de atributos que devuelven **verdadero** o **falso** según la situación actual del cursor. Estos atributos facilitan la manipulación del cursor. Se utilizan indicando el nombre del cursor, el símbolo % e inmediatamente el nombre del atributo a valorar (por ejemplo *cursorProvincias%ISOPEN*)

%ISOPEN

Devuelve verdadero si el cursor ya está abierto.

%NOTFOUND

Devuelve verdadero si la última instrucción FETCH no devolvió ningún valor. Ejemplo:

```
DECLARE
   CURSOR cursorProvincias IS
     SELECT p.nombre, SUM(poblacion) AS poblacion
     FROM LOCALIDADES l
     JOIN PROVINCIAS p USING (n_provincia)
     GROUP BY p.nombre;

   v_nombre PROVINCIAS.nombre%TYPE;
   v_poblacion LOCALIDADES.poblacion%TYPE;

BEGIN
    OPEN cursorProvincias;
   LOOP
          FETCH cursorProvincias INTO v_nombre,
                v_poblacion;
          EXIT WHEN cursorProvincias%NOTFOUND;
          DBMS_OUTPUT.PUT_LINE(v_nombre || ',' ||
                v_poblacion);
   END LOOP;
   CLOSE cursorProvincias;
END;
```

En el ejemplo anterior se recorre el cursor hasta que el FETCH no devuelve ninguna fila. Lo que significa que el programa anterior muestra el nombre de cada provincia seguida de una coma y de la población de la misma.

%FOUND

Instrucción contraria a la anterior, devuelve verdadero si el último FETCH devolvió una fila.

%ROWCOUNT

Indica el número de filas que se han recorrido en el cursor (inicialmente vale cero). Es decir, indica cuántos FETCH se han aplicado sobre el cursor.

(5.4.8) variables de registro

introducción

Los registros son una estructura estática de datos presente en casi todos los lenguajes clásicos (**record** en Pascal o **struct** en C). Se trata de un tipo de datos que se compone de datos más simple. Por ejemplo el registro *persona* se compondría de los datos simples *nombre, apellidos, dirección, fecha de nacimiento,* etc.

En PL/SQL su interés radica en que cada fila de una tabla o vista se puede interpretar como un registro, ya que cada fila se compone de datos simples. Gracias a esta interpretación, los registros facilitan la manipulación de los cursores ya que podemos entender que un cursor es un conjunto de registros (cada registro sería una fila del cursor).

declaración

Para utilizar registros, primero hay que definir los datos que componen al registro. Así se define el tipo de registro (por eso se utiliza la palabra TYPE). Después se declarará una variable de registro que sea del tipo declarado (es decir, puede haber varias variables del mismo tipo de registro).

Sintaxis:

```
TYPE nombreTipoRegistro IS RECORD(
    campo1 tipoCampo1 [:= valorInicial],
    campo2 tipoCampo2 [:= valorInicial],
    ...
    campoN tipoCampoN [:= valorInicial]
);

nombreVariableDeRegistro nombreTipoRegistro;
```

Ejemplo:

```
TYPE regPersona IS RECORD(
    nombre VARCHAR2(25),
    apellido1 VARCHAR2(25),
    apellido2 VARCHAR2(25),
    fecha_nac DATE
);
alvaro regPersona;
laura regPersona;
```

uso de registros

Para rellenar los valores de los registros se indica el nombre de la variable de registro seguida de un punto y el nombre del campo a rellenar:

```
alvaro.nombre := 'Alvaro';
alvaro.fecha_nac := TO_DATE('2/3/2004');
```

%ROWTYPE

Al declarar registros, se puede utilizar el modificador %ROWTYPE que sirve para asignar a un registro la estructura de una tabla. Por ejemplo:

```
DECLARE
    regPersona personas%ROWTYPE;
```

personas debe ser una tabla. *regPersona* es un registro que constará de los mismos campos y tipos que las columnas de la tabla *personas*.

(5.4.9) cursores y registros

uso de FETCH con registros

Una de las desventajas, con lo visto hasta ahora, de utilizar FETCH reside en que necesitamos asignar todos los valores de cada fila del cursor a una variable. Por lo que si una fila tiene 10 columnas, habrá que declarar 10 variables.

En lugar de ello se puede utilizar una variable de registro y asignar el resultado de FETCH a esa variable. Ejemplo (equivalente al de la página 154):

```
DECLARE
   CURSOR cursorProvincias IS
     SELECT p.nombre, SUM(poblacion) AS poblacion
     FROM LOCALIDADES l
     JOIN PROVINCIAS p USING (n_provincia)
     GROUP BY p.nombre;

   rProvincias cursorProvincias%ROWTYPE;
BEGIN
    OPEN cursorProvincias;
   LOOP
          FETCH cursorProvincias INTO rProvincias;
          EXIT WHEN cursorProvincias%NOTFOUND;
          DBMS_OUTPUT.PUT_LINE(rProvincias.nombre || ',' ||
                                  rProvincias.poblacion);
   END LOOP;
   CLOSE cursorProvincias;
END;
```

bucle FOR de recorrido de cursores

Es la forma más habitual de recorrer todas las filas de un cursor. Es un bucle FOR que se encarga de realizar tres tareas:

(1) Abre un cursor (realiza un OPEN implícito sobre el cursor antes de empezar el bucle)

(2) Recorre todas las filas de un cursor (cada vez que se entra en el interior del FOR se genera un FETCH implícito) y en cada vuelta del bucle almacena el contenido de cada fila en una variable de registro. La variable de registro utilizada en el bucle FOR no se debe declarar en la zona DECLARE; se crea al inicio del bucle y se elimina cuando éste finaliza.

(3) Cierra el cursor (cuando finaliza el FOR)

Sintaxis:

```
FOR variableRegistro IN cursor LOOP
   ..instrucciones
END LOOP;
```

Esa sintaxis es equivalente a:

```
OPEN cursor;
LOOP
   FETCH cursor INTO variableRegistro;
   EXIT WHEN cursor%NOTFOUND;
   ...instrucciones
END LOOP;
```

Ejemplo (equivalente al ejemplo comentado en los apartados anteriores):

```
DECLARE
   CURSOR cursorProvincias IS
     SELECT p.nombre, SUM(poblacion) AS poblacion
     FROM LOCALIDADES l
     JOIN PROVINCIAS p USING (n_provincia)
     GROUP BY p.nombre;
BEGIN
   FOR rProvincias IN cursorProvincias LOOP
         DBMS_OUTPUT.PUT_LINE(rProvincias.nombre || ',' ||
                                 rProvincias.poblacion);
   END LOOP;
END;
```

Naturalmente este código es más sencillo de utilizar y más corto que los anteriores.

(5.4.10) cursores avanzados

cursores con parámetros

En muchas ocasiones se podría desear que el resultado de un cursor dependa de una variable. Por ejemplo al presentar una lista de personal, hacer que aparezca el cursor de un determinado departamento y puesto de trabajo.

Para hacer que el cursor varíe según esos parámetros, se han de indicar los mismos en la declaración del cursor. Para ello se pone entre paréntesis su nombre y tipo tras el nombre del cursor en la declaración.

Ejemplo:

```
DECLARE
   CURSOR cur_personas(dep NUMBER, pue VARCHAR2(20)) IS
         SELECT nombre, apellidos
         FROM empleados
         WHERE departamento=dep AND puesto=pue;
BEGIN
   OPEN cur_personas(12,'administrativo');
   .....
   CLOSE cur_personas;
END
```

Es al abrir el. cursor cuando se indica el valor de los parámetros, lo que significa que se puede abrir varias veces el cursor y que éste obtenga distintos resultados dependiendo del valor del parámetro.

Se pueden indicar los parámetros también en el bucle FOR:

```
DECLARE
   CURSOR cur_personas(dep NUMBER, pue VARCHAR2(20)) IS
          SELECT nombre, apellidos
          FROM empleados
          WHERE departamento=dep AND puesto=pue;
BEGIN
   FOR r IN cur_personas(12,'administrativo') LOOP
          .....
   END LOOP;
END
```

actualizaciones al recorrer registros

En muchas ocasiones se realizan operaciones de actualización de registros sobre el cursor que se está recorriendo. Para evitar problemas se deben bloquear los registros del cursor a fin de detener otros procesos que también desearan modificar los datos.

Esta cláusula se coloca al final de la sentencia SELECT del cursor (iría detrás del ORDER BY). Opcionalmente se puede colocar el texto NOWAIT para que el programa no se quede esperando en caso de que la tabla esté bloqueada por otro usuario. Se usa el texto OF seguido del nombre del campo que se modificará (no es necesaria esa cláusula, pero se mantiene para clarificar el código).

Sintaxis:

```
CURSOR ...
   SELECT...
   FOR UPDATE [OF campo] [NOWAIT]
```

Ejemplo:

```
DECLARE
   CURSOR c_emp IS
          SELECT id_emp, nombre, n_departamento, salario
          FROM empleados, departamentos
          WHERE empleados.id_dep=departamentos.id_dep
                AND empleados.id_dep=80
          FOR UPDATE OF salario NOWAIT;
```

A continuación en la instrucción UPDATE que modifica los registros se puede utilizar una nueva cláusula llamada WHERE CURRENT OF seguida del nombre de un cursor, que hace que se modifique sólo el registro actual del cursor.

Ejemplo:

```
FOR r_emp IN c_emp LOOP
   IF r_emp.salario<1500 THEN
         UPDATE empleados SET salario = salario *1.30
         WHERE CURRENT OF c_emp;
```

(5.5) excepciones

(5.5.1) introducción

Se llama excepción a todo hecho que le sucede a un programa que causa que la ejecución del mismo finalice. Lógicamente eso causa que el programa termine de forma anormal.

Las excepciones se debe a:

- Que ocurra un error detectado por Oracle (por ejemplo si un **SELECT** no devuelve datos ocurre el error *ORA-01403* llamado *NO_DATA_FOUND*).
- Que el propio programador las lance (comando **RAISE**).

Las excepciones se pueden capturar a fin de que el programa controle mejor la existencia de las mismas.

(5.5.2) captura de excepciones

La captura se realiza utilizando el bloque **EXCEPTION** que es el bloque que está justo antes del END del bloque. Cuando una excepción ocurre, se comprueba el bloque EXCEPTION para ver si ha sido capturada, si no se captura, el error se propaga a Oracle que se encargará de indicar el error existente.

Las excepciones pueden ser de estos tipos:

- **Excepciones predefinidas de Oracle.** Que tienen ya asignado un nombre de excepción.
- **Excepciones de Oracle sin definir.** No tienen nombre asignado pero se les puede asignar.
- **Definidas por el usuario.** Las lanza el programador.

La captura de excepciones se realiza con esta sintaxis:

```
DECLARE
   sección de declaraciones
BEGIN
   instrucciones
EXCEPTION
   WHEN excepción1 [OR excepción2 ...] THEN
         instrucciones que se ejcutan si suceden esas excepciones
   [WHEN excepción3 [OR...] THEN
   instrucciones que se ejcutan si suceden esas excepciones]
```

```
[WHEN OTHERS THEN
instrucciones que se ejecutan si suceden otras
        excepciones]
END;
```

Cuando ocurre una determinada excepción, se comprueba el primer WHEN para comprobar si el nombre de la excepción ocurrida coincide con el que dicho WHEN captura; si es así se ejecutan las instrucciones, si no es así se comprueba el siguiente **WHEN** y así sucesivamente.

Si existen cláusula **WHEN OTHERS**, entonces las excepciones que no estaban reflejadas en los demás apartados WHEN ejecutan las instrucciones del WHEN OTHERS. Ésta cláusula debe ser la última.

(5.5.3) excepciones predefinidas

Oracle tiene las siguientes excepciones predefinidas. Son errores a los que Oracle asigna un nombre de excepción. Están presentes los más comunes:

Nombre de excepción	Número de error	Ocurre cuando..
ACCESS_INTO_NULL	ORA-06530	Se intentan asignar valores a un objeto que no se había inicializado
CASE_NOT_FOUND	ORA-06592	Ninguna opción WHEN dentro de la instrucción CASE captura el valor, y no hay instrucción ELSE
COLLECTION_IS_NULL	ORA-06531	Se intenta utilizar un *varray* o una tabla anidada que no estaba inicializada
CURSOR_ALREADY_OPEN	ORA-06511	Se intenta abrir un cursor que ya se había abierto
DUP_VAL_ON_INDEX	ORA-00001	Se intentó añadir una fila que provoca que un índice único repita valores
INVALID_CURSOR	ORA-01001	Se realizó una operación ilegal sobre un cursor
INVALID_NUMBER	ORA-01722	Falla la conversión de carácter a número
LOGIN_DEINED	ORA-01017	Se intenta conectar con Oracle usando un nombre de usuario y contraseña inválidos
NO_DATA_FOUND	ORA-01403	El SELECT de fila única no devolvió valores
PROGRAM_ERROR	ORA-06501	Error interno de Oracle
ROWTYPE_MISMATCH	ORA-06504	Hay incompatibilidad de tipos entre el cursor y las variables a las que se intentan asignar sus valores
STORAGE_ERROR	ORA-06500	No hay memoria suficiente
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Se hace referencia a un elemento de un *varray* o una tabla anidada usando un índice mayor que los elementos que poseen
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Se hace referencia a un elemento de un *varray* o una tabla anidada usando un índice cuyo valor está fuera del rango legal

Nombre de excepción	Número de error	Ocurre cuando..
SYS_INVALID_ROWID	ORA-01410	Se convierte un texto en un número de identificación de fila (ROWID) y el texto no es válido
TIMEOUT_ON_RESOURCE	ORA-00051	Se consumió el máximo tiempo en el que Oracle permite esperar al recurso
TOO_MANY_ROWS	ORA-01422	El SELECT de fila única devuelve más de una fila
VALUE_ERROR	ORA-06502	Hay un error aritmético, de conversión, de redondeo o de tamaño en una operación
ZERO_DIVIDE	ORA-01476	Se intenta dividir entre el número cero.

Ejemplo:

```
DECLARE
   x NUMBER :=0;
   y NUMBER := 3;
   res NUMBER;
BEGIN
    res:=y/x;
    DBMS_OUTPUT.PUT_LINE(res);
EXCEPTION
    WHEN ZERO_DIVIDE THEN
            DBMS_OUTPUT.PUT_LINE('No se puede dividir por cero') ;
    WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error inesperado') ;
END;
```

(5.5.4) excepciones sin definir

Pueden ocurrir otras muchas excepciones que no están en la lista anterior. En ese caso aunque no tienen un nombre asignado, sí tienen un número asignado. Ese número es el que aparece cuando Oracle muestra el mensaje de error tras la palabra ORA.

Por ejemplo en un error por restricción de integridad Oracle lanza un mensaje encabezado por el texto: ORA-02292 Por lo tanto el error de integridad referencia es el –02292.

Si deseamos capturar excepciones sin definir hay que:

(1) Declarar un nombre para la excepción que capturaremos. Eso se hace en el apartado DECLARE con esta sintaxis:

```
nombreDeExcepción EXCEPTION;
```

(2) Asociar ese nombre al número de error correspondiente mediante esta sintaxis en el apartado DECLARE (tras la instrucción del paso 1):

```
PRAGMA EXCEPTION_INIT(nombreDeExcepción, nºDeExcepción);
```

(3) En el apartado EXCEPTION capturar el nombre de la excepción como si fuera una excepción normal.

Ejemplo:

```
DECLARE
   e_integridad EXCEPTION;
   PRAGMA EXCEPTION_INIT(e_integridad, -2292);
BEGIN
    DELETE FROM piezas WHERE tipo='TU' AND modelo=6;
EXCEPTION
   WHEN e_integridad THEN
          DBMS_OUTPUT.PUT_LINE('No se puede borrar esa pieza' ||
          ' porque tiene existencias relacionadas');
END;
```

(5.5.5) funciones de uso con excepciones

Se suelen usar dos funciones cuando se trabaja con excepciones:

- SQLCODE. Retorna el código de error del error ocurrido
- SQLERRM. Devuelve el mensaje de error de Oracle asociado a ese número de error.

Ejemplo:

```
EXCEPTION
   ...
   WHEN OTHERS THEN
          DBMS_OUTPUT.PUT_LINE('Ocurrió el error ' ||
                SQLCODE ||'mensaje: ' || SQLERRM);
END;
```

(5.5.6) excepciones de usuario

El programador puede lanzar sus propias excepciones simulando errores del programa. Para ello hay que:

(1) Declarar un nombre para la excepción en el apartado DECLARE, al igual que para las excepciones sin definir:

```
miExcepcion EXCEPTION;
```

(2) En la sección ejecutable (BEGIN) utilizar la instrucción RAISE para lanzar la excepción:

```
RAISE miExcepcion;
```

(3) En el apartado de excepciones capturar el nombre de excepción declarado:

```
EXCEPTION
   ...
   WHEN miExcepcion THEN
         ....
```

Otra forma es utilizar la función RAISE_APPLICATION_ERROR que simplifica los tres pasos anteriores. Sintaxis:

```
RAISE_APPLICATION_ERROR(nºDeError, mensaje, [,{TRUE|FALSE}]);
```

Esta instrucción se coloca en la sección ejecutable o en la de excepciones y sustituye a los tres pasos anteriores. Lo que hace es lanzar un error cuyo número debe de estar entre el -20000 y el -20999 y hace que Oracle muestre el mensaje indicado. El tercer parámetro opciones puede ser TRUE o FALSE (por defecto TRUE) e indica si el error se añade a la pila de errores existentes.

Ejemplo:

```
DECLARE

BEGIN
    DELETE FROM piezas WHERE tipo='ZU' AND modelo=26;
    IF SQL%NOTFOUND THEN
          RAISE_APPLICATION_ERROR(-20001,'No existe esa pieza');
    END IF;
END;
```

En el ejemplo, si la pieza no existe, entonces SQL%NOTFOUND devuelve verdadero ya que el DELETE no elimina ninguna pieza. Se lanza la excepción de usuario -20001 haciendo que Oracle utilice el mensaje indicado. Oracle lanzará el mensaje: ORA-20001: No existe esa pieza

(5.6) procedimientos

(5.6.1) introducción

Un procedimiento es un bloque PL/SQL al que se le asigna un nombre. Un procedimiento se crea para que realice una determinada tarea de gestión.

Los procedimientos son compilados y almacenados en la base de datos. Gracias a ellos se consigue una reutilización eficiente del código, ya que se puede invocar al procedimiento las veces que haga falta desde otro código o desde una herramienta de desarrollo como Oracle Developer. Una vez almacenados pueden ser modificados de nuevo.

(5.6.2) estructura de un procedimiento

La sintaxis es:

```
CREATE [OR REPLACE] PROCEDURE nombreProcedimiento
[(parámetro1 [modelo] tipoDatos
  [,parámetro2 [modelo] tipoDatos [,...])]
{IS|AS}
    secciónDeDeclaraciones
BEGIN
    instrucciones
[EXCEPTION
    controlDeExcepciones]
END;
```

La opción **REPLACE** hace que si ya existe un procedimiento con ese nombre, se reemplaza con el que se crea ahora. Los parámetros son la lista de variables que necesita el procedimiento para realizar su tarea.

Al declarar cada parámetro se indica el tipo de los mismos, pero no su tamaño; es decir sería *VARCHAR2* y no *VARCHAR2(50)*.

El apartado opcional **modelo**, se elige si el parámetro es de tipo **IN**, **OUT** o **IN OUT** (se explica más adelante).

No se utiliza la palabra **DECLARE** para indicar el inicio de las declaraciones. No obstante la sección de declaraciones figura tras las palabras **IS** o **AS** (es decir justo antes del BEGIN es donde debemos declarar las variables).

(5.6.3) desarrollo de procedimientos

Los pasos para desarrollar procedimientos son:

(1) Escribir el código en un archivo **.sql** desde cualquier editor.

(2) Compilar el código desde un editor como **iSQL*Plus** o cualquier otro que realice esa tarea (como **toad** por ejemplo). El resultado es el llamado **código P,** el procedimiento estará creado.

(3) Ejecutar el procedimiento para realizar su tarea, eso se puede hacer las veces que haga falta (en **iSQL*Plus**, el comando que ejecuta un procedimiento es el comando **EXECUTE,** en otros entornos se suele crear un bloque anónimo que incorpore una llamada al procedimiento).

(5.6.4) parámetros

Los procedimientos permiten utilizar parámetros para realizar su tarea. Por ejemplo supongamos que queremos crear el procedimiento *ESCRIBIR* para escribir en el servidor (como hace *DBMS_OUTPUT.PUT_LINE*) lógicamente dicho procedimiento necesita saber lo que queremos escribir. Ese sería el parámetro, de esa forma el procedimiento sería:

```
CREATE OR REPLACE PROCEDURE
Escribir(texto VARCHAR)
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(texto);
END;
```

Para invocarle:

```
BEGIN
   ...
   Escribir('Hola');
```

Cuando se invoca a un procedimiento, si éste no tiene parámetros, se pueden omitir los paréntesis (es decir la llamada al procedimiento *actualizar()* se puede hacer simplemente escribiendo actualizar, sin paréntesis)

parámetros IN y parámetros OUT

Hay tres tipos de parámetros en PL/SQL:

- **Parámetros IN.** Son los parámetros que en otros lenguajes se denominan como parámetros por valor. El procedimiento recibe una copia del valor o variable que se utiliza como parámetro al llamar al procedimiento. Estos parámetros pueden ser: valores literales (*18* por ejemplo), variables (*v_num* por ejemplo) o expresiones (como *v_num+18*). A estos parámetros se les puede asignar un valor por defecto.
- **Parámetros OUT.** Relacionados con el paso por variable de otros lenguajes. Sólo pueden ser variables y no pueden tener un valor por defecto. Se utilizan para que el procedimiento almacene en ellas algún valor. Es decir, los parámetros OUT son variables sin declarar que se envían al procedimiento de modo que si en el procedimiento cambian su valor, ese valor permanece en ellas cuando el procedimiento termina,
- **Parámetros IN OUT.** Son una mezcla de los dos anteriores. Se trata de variables declaradas anteriormente cuyo valor puede ser utilizado por el procedimiento que, además, puede almacenar un valor en ellas. No se las puede asignar un valor por defecto.

Se pueden especificar estas palabras en la declaración del procedimiento (es el modo del procedimiento). Si no se indica modo alguno, se supone que se está utilizando IN (que es el que más se usa).

Ejemplo:

```
CREATE OR REPLACE PROCEDURE consultarEmpresa
(v_Nombre VARCHAR2, v_CIF OUT VARCHAR2, v_dir OUT
VARCHAR2)
IS
BEGIN
    SELECT cif, direccion INTO v_CIF, v_dir
    FROM EMPRESAS
    WHERE nombre LIKE '%'||v_nombre||'%';
EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('No se encontraron datos');
        WHEN TOO_MANY_ROWS THEN
            DBMS_OUTPUT.PUT_LINE ('Hay más de una fila con esos'
                ||' datos');
END;
```

El procedimiento consulta las empresas cuyo nombre tenga el texto envíado en *v_nombre*, captura los posibles errores y en caso de que la consulta sea buena almacena el cif y la dirección de la empresa en los parámetros *v_CIF* y *v_dir*.

La llamada al procedimiento anterior podría ser:

```
DECLARE
    v_c VARCHAR2(50);
    v_d VARCHAR2(50);
BEGIN
   consultarEmpresa('Hernández',v_c,v_d);
   DBMS_OUTPUT.PUT_LINE(v_c);
   DBMS_OUTPUT.PUT_LINE(v_d);
END;
```

Las variables *v_c* y *v_d* almacenarán (si existe una sola empresa con el texto *Hernández*) el CIF y la dirección de la empresa buscada.

Los procedimientos no pueden leer los valores que posean las variables OUT, sólo escribir en ellas. Si se necesitan ambas cosas es cuando hay que declararlas con IN OUT.

(5.6.5) borrar procedimientos

El comando DROP PROCEDURE seguido del nombre del procedimiento que se elimina es el encargado de realizar esta tarea.

(5.7) funciones

(5.7.1) introducción

Las funciones son un tipo especial de procedimiento que sirven para calcular un determinado valor. Todo lo comentado en el apartado anterior es válido para las funciones, la diferencia estriba **sólo** en que éstas devuelven un valor.

(5.7.2) sintaxis

```
CREATE [OR REPLACE] FUNCTION nombreFunción
[(parámetro1 [modelo] tipoDatos
[,parámetro2 [modelo] tipoDatos [,...]])]
RETURN tipoDeDatos
IS|AS
   secciónDeDeclaraciones
BEGIN
   instrucciones
[EXCEPTION
   controlDeExcepciones]
END;
```

Si comparamos con la declaración de las funciones, la palabra **PROCEDURE** se modifica por la palabra **FUNCTION** (indicando que es una función y no un procedimiento) y aparece la cláusula **RETURN** justo antes de la palabra IS que sirve para indicar el tipo de datos que poseerá el valor retornado por la función.

(5.7.3) uso de funciones

Las funciones se crean igual que los procedimientos y, al igual que éstos, se almacenan en la base de datos. Toda función ha de devolver un valor, lo cual implica utilizar la instrucción **RETURN** seguida del valor que se devuelve.

Ejemplo:

```
CREATE OR REPLACE FUNCTION cuadrado
(x NUMBER)
RETURN NUMBER
IS
BEGIN
    RETURN x*x;
END;
```

La función descrita calcula el cuadrado de un número. Una llamada podría ser:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE(cuadrado(9));
END;
```

Las funciones de PL/SQL se utilizan como las funciones de cualquier lenguaje estructurado, se pueden asignar a una variable, utilizar para escribir, etc. Además dentro de una función se puede invocar a otra función.

(5.7.4) utilizar funciones desde SQL

Una ventaja fantástica de las funciones es la posibilidad de utilizarlas desde una instrucción SQL. Por ejemplo:

```
CREATE OR REPLACE FUNCTION precioMedio
RETURN NUMBER
IS
v_precio NUMBER(11,4);
BEGIN
    SELECT AVG(precio_venta) INTO v_precio
    FROM PIEZAS;
    RETURN v_precio;
END;
```

Esta función devuelve el precio medio de la tabla de piezas. Una vez compilada y almacenada la función, se puede invocar desde una instrucción SQL cualquiera. Por ejemplo:

```
SELECT * FROM PIEZAS
WHERE precioMedio>precio_venta;
```

Esa consulta obtiene los datos de las piezas cuyo precio sea menor que el precio medio.
Hay que tener en cuenta que para que las funciones puedan ser invocadas desde SQL, éstas tienen que cumplir que:

- Sólo valen funciones que se hayan almacenado
- Sólo pueden utilizar parámetros de tipo IN

- Sus parámetros deben ser de tipos compatibles con el lenguaje SQL (no valen tipos específicos de PL/SQL como **BOOLEAN** por ejemplo)
- El tipo devuelto debe ser compatible con SQL
- No pueden contener instrucciones **DML**
- Si una instrucción DML modifica una determinada tabla, en dicha instrucción no se puede invocar a una función que realice consultas sobre la misma tabla
- No pueden utilizar instrucciones de transacciones (**COMMIT**, **ROLLBACK**,...)
- La función no puede invocar a otra función que se salte alguna de las reglas anteriores.

(5.7.5) eliminar funciones

Sintaxis:

```
DROP FUNCTION nombreFunción;
```

(5.7.6) recursividad

En PL/SQL la recursividad (el hecho de que una función pueda llamarse a sí misma) está permitida. Este código es válido:

```
CREATE FUNCTION Factorial
(n NUMBER)
IS
BEGIN
   IF (n<=1) THEN
         RETURN 1
   ELSE
         RETURN n * Factorial(n-1);
   END IF;
END;
```

(5.7.7) mostrar procedimientos almacenados

La vista **USER_PROCEDURES**, contiene una fila por cada procedimiento o función que tenga almacenado el usuario actual.

(5.8) paquetes

(5.8.1) introducción

Los paquetes sirven para agrupar bajo un mismo nombre funciones y procedimientos. Facilitan la modularización de programas y su mantenimiento.

Los paquetes constan de dos partes:

- **Especificación.** Que sirve para declarar los elementos de los que consta el paquete. En esta especificación se indican los procedimientos, funciones y variables **públicos** del paquete (los que se podrán invocar desde fuera del paquete). De los procedimientos sólo se indica su nombre y parámetros (sin el cuerpo).
- **Cuerpo.** En la que se especifica el funcionamiento del paquete. Consta de la definición de los procedimientos indicados en la especificación. Además se pueden declarar y definir variables y procedimientos **privados** (sólo visibles para el cuerpo del paquete, no se pueden invocar desde fuera del mismo).

Los paquetes se editan, se compilan (obteniendo su código P) y se ejecutan al igual que los procedimientos y funciones

(5.8.2) creación de paquetes

Conviene almacenar la especificación y el cuerpo del paquete en dos archivos de texto (**.sql**) distintos para su posterior mantenimiento.

especificación

Sintaxis:

```
CREATE [OR REPLACE] PACKAGE nombrePaquete
{IS|AS}
    variables, constantes, cursores y excepciones públicas
    cabecera de procedimientos y funciones
END nombrePaquete;
```

Ejemplo:

```
CREATE OR REPLACE PACKAGE paquete1 IS
    v_cont NUMBER := 0;
    PROCEDURE reset_cont(v_nuevo_cont NUMBER);
    FUNCTION devolver_cont
            RETURN NUMBER;
END paquete1;
```

De las funciones hay que indicar sus parámetros y el tipo de datos que devuelve.

cuerpo

```
CREATE [OR REPLACE] PACKAGE BODY nombrePaquete
IS|AS
    variables, constantes, cursores y excepciones privadas
    cuerpo de los procedimientos y funciones
END nombrePaquete;
```

Ejemplo:

```
CREATE OR REPLACE PACKAGE BODY paquete1 IS
  PROCEDURE reset_cont(v_nuevo_cont NUMBER)
  IS
  BEGIN
    v_cont:=v_new_cont;
  END reset_cont;

  FUNCTION devolver_cont
  IS
  BEGIN
    RETURN v_cont;
  END devolver_cont;
END paquete1;
```

uso de los objetos definidos en los paquetes

Desde dentro del paquete, para utilizar otra función o procedimiento o variable dentro del mismo paquete, basta con invocarla por su nombre.

Si queremos utilizar un objeto de un paquete, fuera del mismo, entonces se antepone el nombre del paquete a la función. Por ejemplo *paquete1.reset_cont(4)* (en el ejemplo anterior).

Para ejecutar un paquete desde SQL*Plus, se usa la orden **EXECUTE**. Por ejemplo: *EXECUTE paquete1.reset_cont(4)*

uso de cursores en los paquetes

Además las variables, en la cabecera del paquete se pueden definir cursores. Para ello se indica el nombre del cursor, los parámetros (si hay) y el tipo devuelto (normalmente con **%ROWTYPE**).

En el cuerpo del paquete el cursor se usa como habitualmente. La única razón para declarar un cursor en un paquete, es por si necesitamos acceder a él desde fuera.

Eso mismo ocurre con las variables, las variables declaradas en la especificación de un paquete, son accesibles desde fuera del paquete (habría que invocarlas escribiendo *paquete.variable*)

(5.9) triggers

(5.9.1) introducción

Se llama **trigger** (o **disparador)** al código que se ejecuta automáticamente cuando se realiza una determinada acción sobre la base de datos. El código se ejecuta independientemente de la aplicación que realizó dicha operación.

De esta forma tenemos tres tipos triggers:

- **Triggers de tabla.** Se trata de triggers que se disparan cuando ocurre una acción DML sobre una tabla.
- **Triggers de vista.** Se lanzan cuando ocurre una acción DML sobre una vista.
- **Triggers de sistema.** Se disparan cuando se produce un evento sobre la base de datos (conexión de un usuario, borrado de un objeto,…)

En este manual sólo se da cabida a los del primer y segundo tipo. Por lo que se dará por hecho en todo momento que nos referiremos siempre a ese tipo de triggers.

Los triggers se utilizan para:

- Ejecutar acciones relacionadas con la que *dispara* el trigger
- Centralizar operaciones globales
- Realizar tareas administrativas de forma automática
- Evitar errores
- Crear reglas de integridad complejas

El código que se lanza con el trigger es **PL/SQL**. No es conveniente realizar excesivos triggers, sólo los necesarios, de otro modo se ralentiza en exceso la base de datos.

(5.9.2) creación de triggers

elementos de los triggers

Puesto que un trigger es un código que se dispara, al crearle se deben indicar las siguientes cosas:

(1) El evento que da lugar a la ejecución del trigger (**INSERT, UPDATE** o **DELETE**)

(2) Cuando se lanza el evento en relación a dicho evento (**BEFORE** (antes), **AFTER** (después) o **INSTEAD OF** (en lugar de))

(3) Las veces que el trigger se ejecuta (tipo de trigger: de instrucción o de fila)

(4) El cuerpo del trigger, es decir el código que ejecuta dicho trigger

cuándo ejecutar el trigger

En el apartado anterior se han indicado los posibles tiempos para que el trigger se ejecute. Éstos pueden ser:

- **BEFORE.** El código del trigger se ejecuta antes de ejecutar la instrucción DML que causó el lanzamiento del trigger.
- **AFTER.** El código del trigger se ejecuta después de haber ejecutado la instrucción DML que causó el lanzamiento del trigger.
- **INSTEAD OF.** El trigger sustituye a la operación DML Se utiliza para vistas que no admiten instrucciones DML.

tipos de trigger

Hay dos tipos de trigger

- **De instrucción.** El cuerpo del trigger se ejecuta una sola vez por cada evento que lance el trigger. Esta es la opción por defecto. El código se ejecuta aunque la instrucción DML no genere resultados.
- **De fila.** El código se ejecuta una vez por cada fila afectada por el evento. Por ejemplo si hay una cláusula **UPDATE** que desencadena un trigger y dicho UPDATE actualiza 10 filas; si el trigger es de fila se ejecuta una vez por cada fila, si es de instrucción se ejecuta sólo una vez.

(5.9.3) sintaxis de la creación de triggers

triggers de instrucción

```
CREATE [OR REPLACE] TRIGGER nombreDeTrigger
cláusulaDeTiempo evento1 [OR evento2[,...]]
ON tabla
[DECLARE
   declaraciones
]
BEGIN
    cuerpo
[EXCEPTION
    captura de excepciones
]
END;
```

La cláusula de tiempo es una de estas palabras: **BEFORE** o **AFTER**. Por su parte el evento tiene esta sintaxis:

```
{INSERT|UPDATE [OF columna1 [,columna2,...]]|DELETE}
```

Los eventos asocian el trigger al uso de una instrucción DML. En el caso de la instrucción **UPDATE**, el apartado **OF** hace que el trigger se ejecute sólo cuando se modifique la columna indicada (o columnas si se utiliza una lista de columnas

separada por comas). En la sintaxis del trigger, el apartado **OR** permite asociar más de un evento al trigger (se puede indicar **INSERT OR UPDATE** por ejemplo).

Ejemplo:

```
CREATE OR REPLACE TRIGGER ins_personal
BEFORE INSERT ON personal
BEGIN
   IF(TO_CHAR(SYSDATE,'HH24') NOT IN ('10','11','12')THEN
         RAISE_APPLICATION_ERROR(-20001,'Sólo se puede ' ||
               '' añadir personal entre las 10 y las 12:59');
   END IF;
END;
```

Este trigger impide que se puedan añadir registros a la tabla de personal si no estamos entre las 10 y las 13 horas.

triggers de fila

Sintaxis:

```
CREATE [OR REPLACE] TRIGGER nombreDeTrigger
cláusulaDeTiempo evento1 [OR evento2[,...]]
ON tabla
[REFERENCING {OLD AS nombreViejo | NEW AS nombreNuevo}]
FOR EACH ROW [WHEN condición]
[WHEN (condición)]
[declaraciones]
cuerpo
```

La diferencia con respecto a los triggers de instrucción está en la línea **REFERENCING** y en **FOR EACH ROW**. Ésta última es la que hace que el trigger sea de fila, es decir que se repita su ejecución por cada fila afectada en la tabla por la instrucción DML.

El apartado **WHEN** permite colocar una condición que deben de cumplir los registros para que el trigger se ejecute. Sólo se ejecuta el trigger para las filas que cumplan dicha condición.

El apartado **REFERENCING** es el que permite indicar un nombre para los valores antiguos y otro para los nuevos.

(5.9.4) referencias NEW y OLD

Cuando se ejecutan instrucciones **UPDATE**, hay que tener en cuenta que se modifican valores antiguos (**OLD**) para cambiarles por valores nuevos (**NEW**). Las palabras NEW y OLD permiten acceder a los valores nuevos y antiguos respectivamente.

El apartado **REFERENCING** de la creación de triggers, permite asignar nombres a las palabras **NEW** y **OLD** (en la práctica no se suele utilizar esta posibilidad). Así **NEW**.*nombre* haría referencia al nuevo nombre que se asigna a una determinada tabla y **OLD**.*nombre* al viejo.

En el apartado de instrucciones del trigger (el BEGIN) hay que adelantar el símbolo "**:**" a las palabra NEW y OLD (serían **:NEW.*nombre*** y **:OLD.*nombre***)

Imaginemos que deseamos hacer una auditoria sobre una tabla en la que tenemos un listado de las piezas mecánicas que fabrica una determinada empresa. Esa tabla es PIEZAS y contiene el tipo y el modelo de la pieza (los dos campos forman la clave de la tabla) y el precio de venta de la misma. Deseamos almacenar en otra tabla diferente los cambios de precio que realizamos a las piezas, para lo cual creamos la siguiente tabla:

```
CREATE TABLE PIEZAS_AUDIT(
  precio_viejo NUMBER(11,4),
  precio_nuevo NUMBER(11,4),
  tipo VARCHAR2(2),
  modelo NUMBER(2),
  fecha DATE
);
```

Como queremos que la tabla se actualice automáticamente, creamos el siguiente trigger:

```
CREATE OR REPLACE TRIGGER crear_audit_piezas
BEFORE UPDATE OF precio_venta
ON PIEZAS
FOR EACH ROW
WHEN ( OLD.precio_venta<NEW.precio_venta )
BEGIN
    INSERT INTO PIEZAS_AUDIT
    VALUES(:OLD.precio_venta, :NEW.precio_vent
                              :OLD.tipo,:OLD.modelo,SYSDATE);
END;
```

Con este trigger cada vez que se modifiquen un registros de la tabla de piezas, siempre y cuando se esté incrementado el precio, se añade una nueva fila por registro modificado en la tabla de aditorías, observar el uso de NEW y de OLD y el uso de los dos puntos (:NEW y :OLD) en la sección ejecutable.

Cuando se añaden registros, los valores de OLD son todos nulos. Cuando se borran registros, son los valores de NEW los que se borran.

(5.9.5) IF INSERTING, IF UPDATING e IF DELETING

Son palabras que se utilizan para determinar la instrucción DML que se estaba realizando cuando se lanzó el trigger. Esto se utiliza en triggers que se lanza para varias operaciones (utilizando INSERT OR UPDATE por ejemplo). En ese caso se pueden utilizar sentencias IF seguidas de INSERTING, UPDATING o DELETING; éstas palabras devolverán TRUE si se estaba realizando dicha operación.

```
CREATE OR REPLACE TRIGGER trigger1
BEFORE INSERT OR DELETE OR UPDATE OF campo1 ON tabla
FOR EACH ROW
BEGIN
   IF DELETING THEN
         instrucciones que se ejecutan si el trigger saltó por borrar filas
   ELSIF INSERTING THEN
         instrucciones que se ejecutan si el trigger saltó por insertar filas
   ELSE
         instrucciones que se ejecutan si el trigger saltó por modificar filas
   END IF
END;
```

(5.9.6) triggers de tipo INSTEAD OF

Hay un tipo de trigger especial que se llama INSTEAD OF y que sólo se utiliza con las vistas. Una vista es una consulta SELECT almacenada. En general sólo sirven para mostrar datos, pero podrían ser interesantes para actualizar, por ejemplo en esta declaración de vista:

```
CREATE VIEW
   existenciasCompleta(tipo,modelo,precio,
   almacen,cantidad) AS
SELECT p.tipo, p.modelo, p.precio_venta,
   e.n_almacen, e.cantidad
FROM PIEZAS p, EXISTENCIAS e
WHERE p.tipo=e.tipo AND p.modelo=e.modelo
ORDER BY p.tipo,p.modelo,e.n_almacen;
```

Esta instrucción daría lugar a error

```
INSERT INTO existenciasCompleta
VALUES('ZA',3,4,3,200);
```

Indicando que esa operación no es válida en esa vista (al utilizar dos tablas). Esta situación la puede arreglar un trigger que inserte primero en la tabla de piezas (sólo si no se encuentra ya insertada esa pieza) y luego inserte en existencias.

Eso lo realiza el trigger de tipo INSTEAD OF, que sustituirá el INSERT original por el código indicado por el trigger:

```
CREATE OR REPLACE TRIGGER ins_piezas_exis
INSTEAD OF INSERT
ON existenciascompleta
BEGIN
    INSERT INTO piezas(tipo,modelo,precio_venta)
          VALUES(:NEW.tipo,:NEW.modelo,:NEW.precio);
    INSERT INTO existencias(tipo,modelo,n_almacen,cantidad)
          VALUES(:NEW.tipo,:NEW.modelo,
                :NEW.almacen,:NEW.cantidad);
END;
```

Este trigger permite añadir a esa vista añadiendo los campos necesarios en las tablas relacionadas en la vista. Se podría modificar el trigger para permitir actualizar, eliminar o borrar datos directamente desde la vista y así cualquier desde cualquier acceso a la base de datos se utilizaría esa vista como si fuera una tabla más.

(5.9.7) administración de triggers

eliminar triggers

Sintaxis:

```
DROP TRIGGER nombreTrigger;
```

desactivar un trigger

```
ALTER TRIGGER nombreTrigger DISABLE;
```

activar un trigger

```
ALTER TRIGGER nombreTrigger ENABLE;
```

desactivar o activar todos los triggers de una tabla

Eso permite en una sola instrucción operar con todos los triggers relacionados con una determinada tabla (es decir actúa sobre los triggers que tienen dicha tabla en el apartado ON del trigger). Sintaxis:

```
ALTER TABLE nombreTabla {DISABLE | ENABLE} ALL TRIGGERS;
```

(5.9.8) restricciones de los triggers

- Un trigger no puede utilizar ninguna operación de transacciones (COMMIT, ROLLBACK, SAVEPOINT o SET TRANSACTION)
- No puede llamar a ninguna función o procedimiento que utilice operaciones de transacciones
- No se pueden declarar variables LONG o LONG RAW. Las referencias :NEW y :OLD no pueden hacer referencia a campos de estos tipos de una tabla

- No se pueden modificar (aunque sí leer y hacer referencia) campos de tipo **LOB** de una tabla

(5.9.9) orden de ejecución de los triggers

Puesto que sobre una misma tabla puede haber varios triggers, es necesario conocer en qué orden se ejecutan los mismos. El orden es:

(1) Primero disparadores de tipo **BEFORE** de tipo instrucción

(2) Disparadores de tipo **BEFORE** por cada fila

(3) Se ejecuta la propia orden que desencadenó al trigger.

(4) Disparadores de tipo **AFTER** con nivel de fila.

(5) Disparadores de tipo **AFTER** con nivel de instrucción.

(5.9.10) problemas con las tablas mutantes

Una tabla mutante es una tabla que se está modificando debido a una instrucción DML. En el caso de un trigger es la tabla a la que se refiere el disparador en la cláusula **ON**.

Un disparador no puede leer o modificar una tabla mutante (una tabla que está cambiando) ni tampoco leer o modificar una columna que tiene restricciones asociadas a una tabla mutante (sí se podrían utilizar y modificar el resto de columnas). Es decir, no podemos hacer consultas ni instrucciones DML sobre una tabla sobre la que ya se ha comenzado a modificar, insertar o eliminar datos.

En realidad **sólo los triggers de fila** no pueden acceder a tablas mutantes, los triggers de tabla sí pueden. Por ello la solución al problema suele ser combinar triggers de tabla con triggers de fila (conociendo el orden de ejecución de los triggers).

El truco consiste en que los triggers de tabla almacenan los valores que se desean cambiar dentro de una tabla preparada al efecto o dentro de variables de paquete (accesibles desde cualquier trigger si se crea dentro de dicho paquete o si el paquete es global).

índice de ilustraciones