Desarrollo de Aplicaciones WEB

CICLO FORMATIVO DE GRADO SUPERIOR

Unidad 5

PL/SQL Fundamentos

Módulo: Bases de Datos

Profesor del módulo: Juan Manuel Fernández Gutiérrez

Materiales registrados (ISBN: 978-84-692-3443-2 Deposito Legal: AS-3564-2009)

FORMACIÓN PROFESIONAL
Principado de Asturias

Introducción

En las unidades anteriores trabajamos con SQL como lenguaje autocontenido, ejecutando sentencias SQL de forma interactiva desde el entorno SQL*Plus. Ahora vamos a ocuparnos de la utilización de SQL dentro de un programa de aplicación escrito en otro lenguaje de programación.

En un programa de aplicación escrito en un lenguaje podemos tener sentencias de SQL intercaladas que se ocupen del acceso a la base de datos. Se dice en este caso que SQL está **embebido** o **inmerso** en el lenguaje de programación, o que el lenguaje SQL es **huésped** del lenguaje de programación que actúa como lenguaje **anfitrión**.

También existen lenguajes de programación que llevan incorporadas las sentencias SQL. No se trata en este caso de que SQL actúe como lenguaje embebido, sino que forma parte del propio lenguaje de programación. Este es el caso del lenguaje PL/SQL desarrollado por Oracle Corporation.

Con el uso de SQL embebido se aprovechan las ventajas del SQL para acceso a los datos de una base y la potencia y flexibilidad de los lenguajes procedimentales.

En principio, cualquier sentencia SQL que pueda funcionar de forma interactiva puede formar parte de un programa de aplicación, aunque es posible que exista alguna diferencia entre el formato de la sentencia interactiva y el de la del programa de aplicación. Igualmente, hay una serie de sentencias que sirven para implementar las particularidades del uso de SQL como lenguaje de programación y no pueden ser utilizadas de forma interactiva.

Objetivos

- Conocer las características del SQL embebido.
- Describir los elementos que intervienen en la ejecución de SQL como lenguaje huésped.
- Codificar bloques anónimos en PL/SQL.
- Utilizar cursores

Contenidos Generales

1. Características del lenguaje PL/SQL.

PL/SQL es la extensión procedimental del lenguaje SQL implementada por Oracle Corporation, que combina la sencillez y flexibilidad de SQL para acceso a la base de datos con las características procedimentales de un lenguaje de programación estructurado como IF ... THEN, WHILE o LOOP. 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. Además, se pueden declarar variables y constantes, definir procedimientos y funciones y capturar y tratar errores en tiempo de ejecución. Así, PL/SQL combina la potencia de SQL para manipular datos con la potencia de proceso de los lenguajes procedimentales.

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) a través de procedimientos suministrados por Oracle.

Los precompiladores de Oracle y la interfaz de programación OCI reconocen y aceptan bloques de código PL/SQL huéspedes de un lenguaje de tercera generación.

El código PL/SQL, como las sentencias SQL, es ejecutado por el S.G.B.D. Oracle, aunque existen algunas herramientas, como Oracle Forms, que disponen de un motor PL/SQL y lo ejecutan localmente.

Los programas PL/SQL están estructurados en bloques, cualquier programa estará formado al menos por un bloque lógico. Cada bloque lógico puede tener anidados cualquier número de sub-bloques.

Existen distintos tipos de programas PL/SQL:

- Bloques anónimos.
- Disparadores.
- Procedimientos y funciones.
- Paquetes, que contienen procedimientos y funciones.

El código PL/SQL puede estar almacenado en la base de datos (procedimientos, funciones, disparadores y paquetes), junto con un programa de aplicación o en ficheros. La ejecución de los bloques PL/SQL puede realizarse interactivamente desde herramientas como

SQL*Plus o Procedure Builder, junto con el programa de aplicación del que forme parte o cuando el S.G.B.D. detecte determinados eventos (disparadores).

2. Estructura de los bloques PL/SQL.

PL/SQL no interpreta un comando cada vez, sino un conjunto de comandos contenidos en un bloque lógico PL/SQL cuya estructura es la siguiente:

```
[DECLARE
declaración de variables, constantes,  excepciones y cursores
BEGIN  [nombre del bloque]
sentencias de SQL,   estructuras …….
[EXCEPTION
    gestión de errores]
END [nombre del bloque];
```

El bloque consta de tres secciones:
DECLARE en ella se declaran las variables, constantes, cursores,subprogramas,….
BEGIN es la sección que contiene las sentencias que se van a ejecutar
EXCEPTION en esta sección se tratan los errores.

Las secciones DECLARE y EXCEPTION son opcionales.

Los bloques pueden contener sub-bloques, es decir, podemos tener bloques anidados. Los anidamientos se pueden realizar en la parte ejecutable y en la de manejo de excepciones, pero no en la declarativa.

```
DECLARE
  Declaraciones;
BEGIN
    DECLARE
      Declaraciones ;
    BEGIN
      Sentencias;
    EXCEPTION
      Excepciones;
    END;
EXCEPTION
  Excepciones;
END;
```

A continuación tenemos un ejemplo de bloque PL/SQL anónimo.

```
DECLARE
                  anos       NUMBER(2);
                  f_alta     DATE;
                  incremento      NUMBER(4,3);
BEGIN
                  SELECT fecha_alta  INTO  f_alta
                         FROM usuarios
                         WHERE num_socio = 122;
                   anos := TRUNC(MONTHS_BETWEEN(SYSDATE, f_alta) / 12);
                   IF anos  >  6 THEN
                          incremento := 1,02;
                         ELSIF  anos  >  4  THEN
                            incremento  :=  1,025;
                                  ELSE
                                  incremento  :=  1,03;
                    END IF;
                    UPDATE usuarios
                      SET cuota_socio  =  cuota_socio  *  incremento
                                WHERE  num_socio  =  122;
                    COMMIT;
END;
```

Con este programa actualizamos la cuota del socio 122. El incremento de la cuota se realiza analizando la antigüedad del socio.

El programa tiene parte declarativa, en la que se definen dos variables, y parte ejecutable.

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

La ejecución del bloque comienza siempre en la primera sentencia detrás de BEGIN y termina al alcanzar la palabra END o al ejecutar la sentencia RETURN, lo que ocurra antes.

3. Variables y constantes.

PL/SQL permite declarar variables y constantes y utilizarlas en cualquier parte de una sentencia SQL o procedimental en que pueda usarse una expresión. Las variables pueden ser de cualquiera de los tipos vistos en SQL o de otros tipos específicos que veremos. Al declarar una variable se le puede asignar valor. PL/SQL permite definir tipos complejos basados en estructuras de tablas y registros.

Ejemplos.

```
importe   NUMBER (9,2);
nombre    VARCHAR2(20);
iva       NUMBER(4,2):=16.00;
casado    BOOLEAN  := FALSE;
```

La palabra reservada **DEFAULT** puede usarse para inicializar una variable a la vez que se define, en lugar del operador de asignación.

```
casado   BOOLEAN  DEFAULT TRUE;
```

La asignación de valores a una variable en la parte ejecutable de un bloque se puede realizar a través de del operador **:=** , pero también con la cláusula INTO de las sentencias SELECT y con FETCH.

Ejemplos.

```
total_factura := base + iva  - descuento

SELECT base INTO importe FROM  facturas WHERE  numero = 1;
```

Se puede utilizar cualquier expresión PL/SQL valida para asignar valor a una variable.

Podemos utilizar el atributo %TYPE para dar a una variable el tipo de dato de otra variable o de una columna de la base de datos.

```
saldo    importe%TYPE;
```

Con la definición anterior, ***saldo*** es del mismo tipo que ***importe***.

En este otro ejemplo, declaramos la variable *total_factura* con el mismo tipo de dato definido para la columna *base_imponible* de la tabla *facturas.*

```
total_factura    facturas.base_imponible%TYPE;
```

Si una columna tiene la restricción NOT NULL la variable definida de ese tipo no asume dicha restricción, pudiendo por lo tanto ponerla a NULL.

Se puede declarar una variable inicializada como NOT NULL

```
comision  NUMBER(5,2)  NOT  NULL  := 2;
```

Esta variable no aceptaría valores nulos

Se pueden declarar constantes utilizando la palabra reservada CONSTANT. Se deben inicialiazar en el momento de la declaración.

El valor de una constante, una vez determinada, puede utilizarse en cualquier lugar del bloque pero no modificarse. La declaración de constantes facilita la modificación de programas que contienen datos constantes.

```
descuento  CONSTANT  NUMBER(2)  := 10;
```

En el ejemplo anterior si no declaramos la constante *descuento* un cambio en el porcentaje de descuento obligaría a cambiarlo en todas las expresiones donde apareciese.

La declaración de una constante es igual al de una variable salvo que hay que utilizar CONSTANT e inicializarla.

Cualquier variable o constante debe ser declarada antes de ser referenciada en otra sentencia, incluso en otra sentencia declarativa.

```
            saldo    importe%TYPE;                    -- Error
```

```
importe  NUMBER(9,2);
```

El nombre de una variable debe estar formado por una letra seguida, opcionalmente, de más letras, números, signos de dólar, subrayados y signos #. No hay distinción entre mayúsculas y minúsculas

Caracteres permitidos.

PL/SQL utiliza el siguiente conjunto de caracteres para la confección de programas:

- Los caracteres en mayúscula y minúscula de la 'A' a la 'Z'
- Los números del 0 al 9
- Tabulados, espacio y retorno de carro.
- Y los símbolos: () + - * / <> = ¡ ~ ; : . ' @ % , " # $ ^ & _ | { } ¿ [ ]

4. Tipos de datos y expresiones

BINARY_INTEGER Subtipos NATURAL POSITIVE	Almacena enteros con signo en un rango de : –2147483647 a 214783647 Podemos usar los subtipos cuando deseamos restringir la variable a positiva 0 .. 2147483647 1 .. 2147483647
NUMBER [precisión, escala] Subtipos DEC DECIMAL DOUBLE PRECISION FLOAT INTEGER INT NUMERIC REAL SMALLINT	Almacena números (enteros y fraccionarios) en coma fija o flotante en un rango de 1.0E-129 a 9.99. Los subtipos tienen las mismas características que NUMBER y existen por compatibilidad con otras bases de datos y normas ANSI/ISO.
CHAR [máxima longitud] Subtipos CHARACTER	Almacena cadenas de longitud fija. Una columna de tipo CHAR admite 255 caracteres, sin embargo una variable del tipo CHAR puede tener una longitud de hasta 32767 bytes. Los subtipos existen por compatibilidad con otros sistemas y normas ANSI/ISO.

VARCHAR2[máxima longitud]	Almacena cadenas de longitud variable, de una longitud de 1 a 32767 bytes. Las columnas definidas con este tipo pueden almacenar hasta 2000 caracteres en la versión que estamos manejando (V7.3); en versiones posteriores tanto los rangos como las capacidades de almacenamiento pueden variar, por lo tanto invitamos a los lectores que estas cifras las tomen como referencia y usen los manuales de la

Subtipos VARCHAR STRING	versión que manejen para establecer los valores reales. Los subtipos solo existen por compatibilidad.
BFILE	Almacena la referencia a un archivo del sistema que contiene datos físicos.
CLOB	Almacena cadenas de caracteres de longitud variable, hasta 4GB
BLOB	Almacena un objeto binario, hasta 4 GB (Se puede utilizar para guardar imágenes, gráficos, etc)
RAW[máxima longitud]	Mismas características que CHAR pero para almacenar información en binario. En este caso el juego de caracteres no tiene importancia.
LONG	Cadena de caracteres de longitud variable máximo 3 GB
LONG RAW	Mismas características que LONG almacenando la información en binario..
BOOLEAN	Sólo admite valores TRUE, FALSE y NULL.
DATE	Permite almacenar fechas.
ROWID	Se utiliza para definir variables que contendrán valores correspondientes a la pseudocolumna ROWID, o sea, los valores binarios que identifican físicamente cada fila de datos de una base.
TABLE y RECORD	Tipos compuestos que definen estructuras de datos complejas. Los veremos mas adelante

Expresiones

Una expresión está formada por un conjunto de operandos unidos por operadores. Un operando puede ser una variable, constante, literal o llamada a función que contribuye a la expresión con un valor. Los valores de los distintos operandos se combinan de acuerdo con los operadores dando lugar a un valor único. Las operaciones en una expresión se hacen según el orden de precedencia de los operadores.

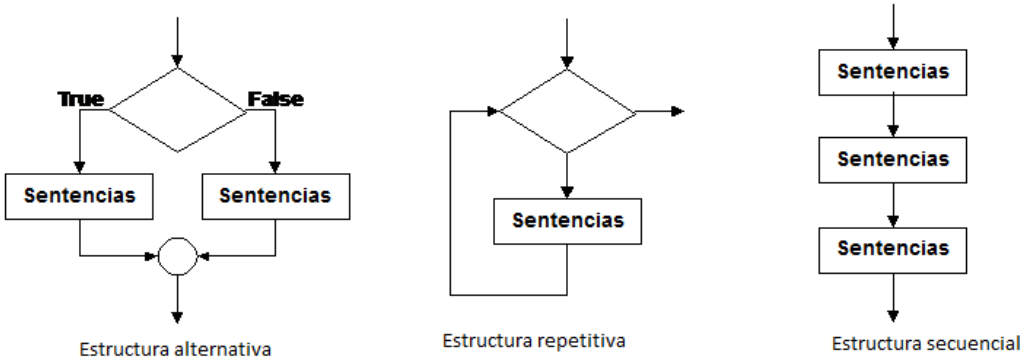
Orden	Operador	Operación
1º	**, NOT	Potencia, negación lógica
2º	+, -	Identidad, negación
3º	*, /	Multiplicación, división
4º	+, -, ||	Suma, resta, concatenación
5º	=, !=, <>, ~=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparación
6º	AND	Conjunción
7º	OR	Inclusión

Como en todos los lenguajes, primero se evalúan las subexpresiones encerradas en paréntesis.

Cuando en una expresión hay operadores con la misma precedencia, por ejemplo, * y ***/*** , PL/SQL los aplica en un orden indeterminado. Por tanto, se deben usar paréntesis para evitar ambigüedades.

5. Estructuras de control

PL/SQL dispone de sentencias que permiten construir las tres estructuras de control de la programación estructurada: ***alternativa, repetitiva*** y ***secuencial.***



5.1. Sentencia IF

La sentencia alternativa IF permite que se ejecuten unas sentencias u otras según sea el resultado de evaluar una condición.

Sintaxis

```
IF condición THEN
                sentencias;
END IF;
```

Si la condición se evalúa como TRUE se ejecuta el grupo de sentencias. Si se evalúa como FALSE o NULL no se ejecutaría. Los operadores utilizados en la condición son los mismos que en SQL.

Ejemplo.

```
IF cuota < 50000 THEN
```

```
                        cuota := cuota * 1.10;
    END IF;
```

Sintaxis con la opción ELSE.

```
IF   condición              THEN
                        Sentencias-1;
             ELSE
                        Sentencias-2;
END IF;
```

Si la condición es verdadera ejecuta el grupo de **Sentencias-1**, si es falsa o NULL el grupo de **Sentencias-2**.

Ejemplo.

```
IF cuota < 50000 THEN
                                cuota := cuota * 1.10;
   ELSE
                                cuota := cuota * 1.05;
END IF;
```

Sintaxis con ELSIF

Se puede realizar un anidamiento de IF a través de la opción ELSIF

```
IF   condición-1 THEN
                                Sentencias-1;
  ELSIF condición-2  THEN
                                Sentencias-2;
  ELSE
                                Sentencias-3;
END IF;
```

Si la primera condición se cumple ejecuta el grupo de **Sentencias-1**, si no se cumple pregunta por la segunda condición, si se cumple ejecuta el grupo de **Sentencias-2** y si no se cumple ejecuta el grupo de **Sentencias-3**.

```
IF cuota > 50000 THEN
                                cuota := cuota * 1.05;
   ELSIF  cuota  > 30000 THEN
                                cuota := cuota * 1.10;
      ELSE
                                cuota := cuota * 1.15;
END IF;
```

5.2. Sentencia CASE

La sentencia CASE permite realizar una ejecución condicional, la diferencia con la IF es que se adapta mejor a las condiciones que implican varias opciones diferentes.

Sintaxis

```
CASE  elemento
        WHEN  valor1        THEN  sentencias;
        WHEN  valor2        THEN  sentencias;
        ……..
        [ELSE  sentencias]
END CASE;
```

Ejemplo

```
DECLARE
        localidad  number := 10;
        nombre varchar2(30);
BEGIN
        CASE localidad
             WHEN 10 THEN nombre := 'Avilés';
             WHEN 20 THEN nombre := 'Oviedo';
             WHEN 30 THEN nombre := 'Santoña';
             WHEN 40 THEN nombre := 'Solares';
             ELSE nombre := 'desconocida';
        END CASE;
END;
```

En el siguiente ejemplo se evalúa la condición en cada cláusula WHEN

```
DECLARE
        localidad  number := 10;
        comunidad varchar2(30);
```

```
BEGIN
       CASE
            WHEN localidad IN (10,20) THEN comunidad := 'Asturias';
            WHEN localidad IN (30,40) THEN comunidad := 'Cantabria';
       END CASE;
END;
```

5.3. Sentencia NULL

NULL es una sentencia ejecutable que no realiza ninguna acción. Se utiliza cuando es necesario que exista una sentencia ejecutable en alguna cláusula que así lo exija, por ejemplo en la IF-THEN-ELSE

```
IF condición  THEN       NULL;
              ELSE       Sentencias;
END IF;
```

En el ejemplo que sigue la última sentencia tiene por objeto que la ejecución del bloque no termine con una excepción no tratada. Cuando se produzcan excepciones distintas a las tratadas por las tres primeras sentencias WHEN, serán tratadas por la última, aunque dicho tratamiento consiste en no hacer nada especial.

```
EXCEPTION
                        WHEN condición THEN …….
                        WHEN condición THEN ……..
                        WHEN condición THEN …….
                        WHEN  OTHERS  THEN  NULL;
END;
```

En algunos casos, la sentencia NULL mejora la legibilidad del programa.

5.4. Bucles.

Los bucles permiten ejecutar de forma iterativa un grupo de sentencias. En los siguientes apartados estudiaremos cada una de las sentencias que podemos utilizar para construir un bucle.

- ***Sentencia LOOP***

El bucle más simple se corresponde con la sintaxis de la sentencia LOOP que a continuación se describe.

```
LOOP
        Sentencias;
END  LOOP;
```

El formato anterior constituye un bucle infinito, dado que no hay ninguna sentencia vinculada a LOOP que provoque una salida.

Sintaxis con EXIT

```
LOOP
      Sentencias-1;
   EXIT;
     Sentencias-2;
END LOOP;
```

EXIT provoca una salida incondicional de un bucle. Se podría colocar dentro de alguna sentencia condicional, por ejemplo en una IF, para hacer que la salida del bucle dependa de una condición.

```
LOOP
   Sentencias-1;
   IF condición  THEN
      Sentencias-2;
     EXIT;
  END IF;
  Sentencias-3;
END LOOP;
```

Ejemplo.

```
contador := 5;
LOOP
```

```
    IF contador > 10  THEN   EXIT;
    END IF;
    contador := contador + 1;
END LOOP;
```

Sintaxis con EXIT WHEN

Con el siguiente formato sale del bucle cuando la condición es verdadera.

LOOP

Sentencias-1;

EXIT WHEN condicion;

Sentencias-2;

END LOOP;

```
contador := 5;
LOOP
   EXIT WHEN contador > 10;
  contador := contador + 1;
END LOOP;
```

Formato con etiquetas

Los bucles pueden identificarse con una etiqueta, o sea con un identificador encerrado entre ángulos inmediatamente antes del bucle.

<<proceso-1>>

LOOP

.....

.....

END LOOP proceso-1;

La etiqueta **proceso_1** es opcional

La sentencia EXIT admite un nombre de bucle.

<<proceso-1>>

LOOP

.....

```
<<proceso-2>>
   LOOP
      .............
      EXIT proceso-1  WHEN  .....;
  END LOOP;
END LOOP;
```

La sentencia EXIT de este ejemplo daría lugar a la finalización tanto del bucle **proceso-2**, el interno, como **proceso-1**, el bucle externo.

Los nombres de las etiquetas siguen las mismas reglas de formación que los de las variables.

- ***Sentencia WHILE***

WHILE constituye un bucle cuyas sentencias se ejecutan mientras la condición asociada a la sentencia sea cierta. Se sale del bucle cuando el resultado de la evaluación sea falso o nulo. (en todas las condiciones el valor nulo se traduce por falso)

Formato de la sentencia WHILE

```
WHILE  condición  LOOP
   Sentencias;
END LOOP;
```

Alguna de las sentencias tiene que afectar a la condición para que en algún momento pase a falsa, de lo contrario el bucle es infinito.

```
contador := 0
WHILE contador  < 10   LOOP
      ........
      contador :=  contador + 1;
END LOOP;
```

- ***Sentencia FOR***

Esta sentencia constituye un bucle controlado por un índice que progresa en un rango establecido. Una de las diferencias con respecto a la sentencia WHILE es que se conoce el numero de iteraciones. Las instrucciones que están dentro del bucle se ejecutan tantas veces como cambia el índice de valor.

Sintaxis de la sentencia FOR

```
FOR indice IN [REVERSE]  valor_inicial    ..   valor_final  LOOP
        Sentencias;
END LOOP;
```

El índice no tiene que declararse explícitamente, es de tipo NUMBER y toma valores enteros. No se puede modificar su valor pero si se puede consultar, por lo que puede aparecer en expresiones. Fuera del bucle no se puede referenciar. Por defecto el índice se incrementa desde el límite inferior hasta llegar al superior. Con la opción REVERSE por cada iteración el índice decrece desde el rango superior hasta el inferior.

```
FOR  i  IN 1..4  LOOP
Sentencias;  -- las ejecuta cuatro veces, tomando i los valores 1,2,3,4
END LOOP;
```

En el siguiente ejemplo el índice es referenciado en una expresión.

```
FOR  i  IN  1 .. 50  LOOP
     IF  MOD (i, 2)  THEN  sentencias;
     END IF;
END LOOP;
```

Con la opción REVERSE

```
FOR  i IN REVERSE 1..4 LOOP
 Sentencias;  -- las ejecuta cuatro veces, tomando i los valores 4,3,2,1
END LOOP;
```

El rango puede ser dinámico, estableciéndose los límites en tiempo de ejecución.

```
SELECT COUNT(*)  INTO  tope  FROM  actividades;
FOR  I  IN  3 ..  TOPE  LOOP
     .........
END LOOP;
```

Si el límite superior es menor que el inferior, el bucle no se ejecuta. En el caso anterior si tope es menor de 3 no iteraría ninguna vez. Si es 3, se ejecutaría una vez.

Con la sentencia **EXIT - WHEN** se puede salir del bucle antes de llegar al límite del rango.

```
 FOR  i  IN 1..4  LOOP
          Sentencias-1;
EXIT WHEN  condición   -- si se cumple la condición salta a ejecutar
           Sentencias-2;              -- Sentencias-3
END LOOP;
Sentencias-3;
```

En un programa PL/SQL los comentarios irán precedidos de dos símbolos guión (--) o bien encerrados entre los símbolos ***/*** y *****/***

- ***Sentencia GOTO***

PL/SQL dispone de una sentencia GOTO, aunque cualquier programa puede ser escrito utilizando solo las sentencias de control anteriores. Su uso no es aconsejable en general por oscurecer la lógica del programa, pero en ciertos casos muy concretos resulta ser una solución adecuada por simplificar el código.

La sentencia GOTO provoca una bifurcación incondicional. Hace que el control pase al punto del programa marcado por una etiqueta. Esta etiqueta tiene que preceder a una sentencia ejecutable o a un bloque PL/SQL.

BEGIN
sentencias-1;
GOTO etiqueta-1
sentencias-2;
<<etiqueta-1>>
sentencias-3;
END;

```
BEGIN
 SELECT COUNT(*)  INTO  tope  FROM  actividades;
 FOR  i  IN  3 ..  tope  LOOP;
    sentencias;
     IF condición  THEN
```

```
                GOTO   salir ;
          END IF;
     <<salir>>    -- es ilegal por no seguirle una sentencia
                        --ejecutable o NULL;
       END LOOP;
     END;
```

El programa del ejemplo anterior se puede arreglar colocando una sentencia NULL detrás de la etiqueta.

```
     BEGIN
          SELECT COUNT(*)  INTO  tope  FROM  actividades;
          FOR  i  IN  3 ..  tope  LOOP
                        sentencias;
                        IF condición  THEN
                              GOTO   salir ;
                        END IF;
           <<salir>>
             NULL;
             END LOOP;
     END;
```

Se puede bifurcar a etiquetas que estén en un bloque mas externo, es decir, etiquetas que estén en un bloque que encierra al bloque en el que se encuentra la sentencia GOTO.

```
     BEGIN
     ......
     <<etiqueta-1>>;
     ......
       BEGIN

       Sentencias;
       ......
       GOTO   etiqueta-1 ;
       ......
       Sentencias;
```

```
    END;
END;
```

Por último, la sentencia GOTO tienen una serie de restricciones, que en general coinciden con las que tiene esta misma sentencia en los lenguajes de tercera generación

- Un GOTO no puede bifurcar al interior de una IF, un bucle LOOP ni a un sub-bloque
- Dentro de una IF, no puede bifurcar de una cláusula THEN a una ELSE o viceversa.
- No puede bifurcar de un bloque externo a uno interno.
- No puede bifurcar fuera de un subprograma.
- No puede bifurcar desde la sección de excepciones a la sección de ejecución

ACTIVIDADES

Actualizar la comisión de los empleados que no tengan jefe, teniendo en cuenta el departamento al que pertenecen.

- Los que pertenecen al departamento 10 la comisión será de un 5% del salario.
- Los que pertenecen al departamento 20 la comisión será de un 7% del salario.
- Los que pertenecen al departamento 30 la comisión será de un 9% del salario.
- Los que no pertenecen a ninguno de los anteriores la comisión será de un 2%.

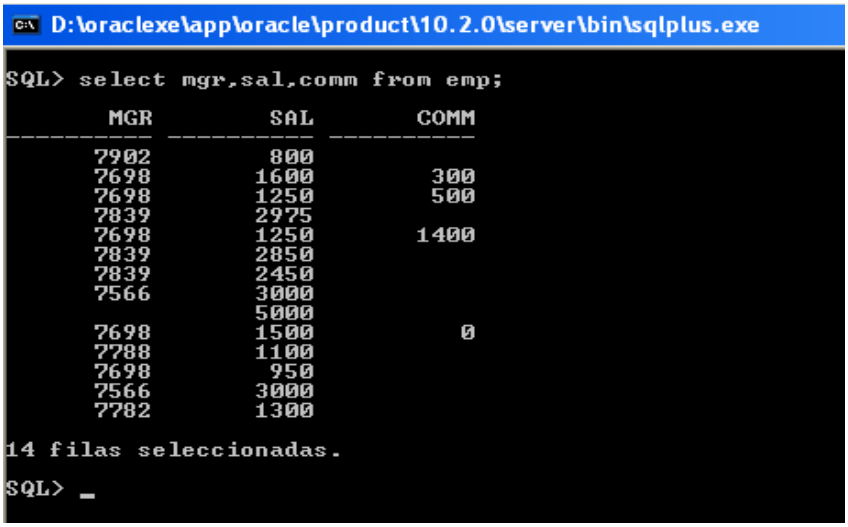
SOLUCION

```
DECLARE
departamento emp.deptno%TYPE;
incremento number(3,2);

BEGIN
SELECT deptno INTO departamento FROM emp WHERE mgr IS NULL;

IF    departamento=10 THEN  incremento:=0.05;

ELSIF departamento=20 THEN  incremento:=0.07;

ELSIF departamento=30 THEN  incremento:=0.09;

ELSE   incremento:=0.02;

END IF;

UPDATE emp SET comm = sal * incremento      WHERE mgr IS  NULL;

-- COMMIT;
```

```
    END;
    /
```

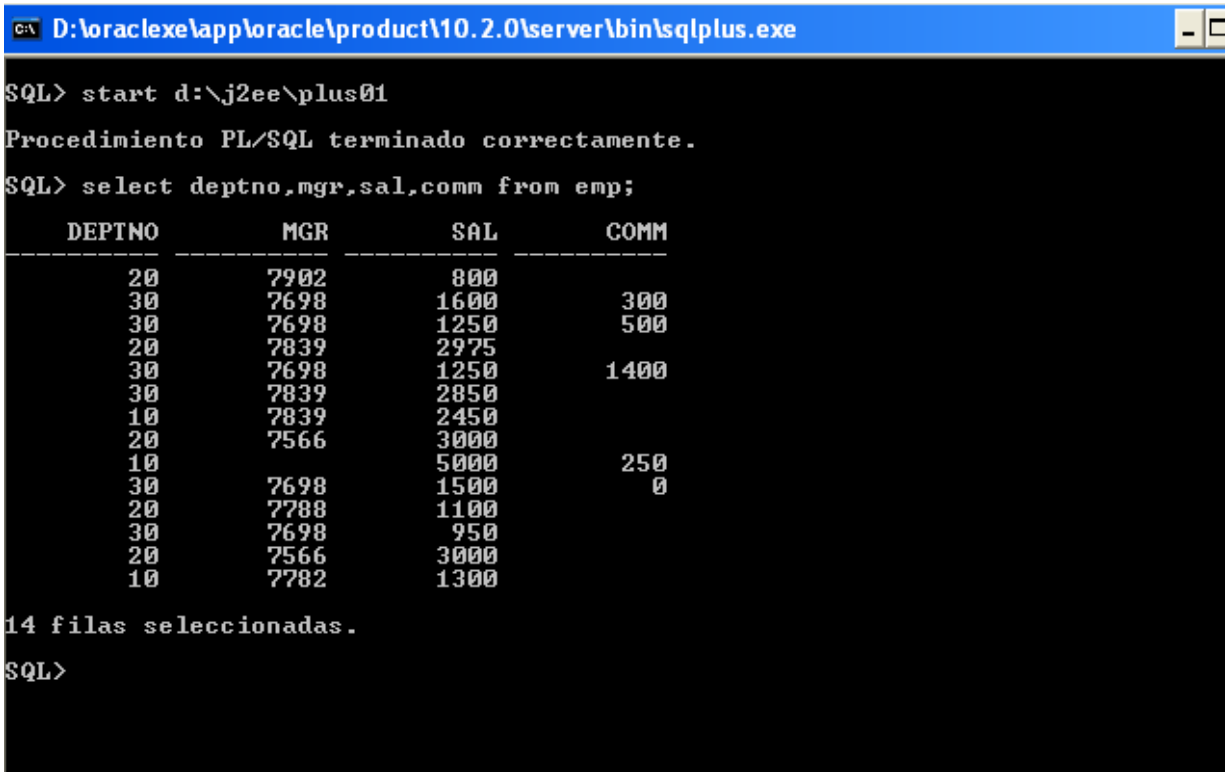
La transacción no se valida, el COMMIT está comentado. La barra (/) es para que lo ejecute. También se puede omitir aquí y ponerla después de la start.

Si queréis comprobar este ejemplo, lo copiáis en un fichero de texto lo guardáis con extensión SQL y lo ejecutáis desde el entorno SqlPlus, como se muestra en las siguientes ventanas.



1º. Consultamos *emp* para ver cuál es el salario, la comisión y el contenido de mgr

2º. Se ejecuta el bloque PL/SQL con start. A continuación se repite la consulta incluyendo el departamento para comprobar que actualizó correctamente.



Se ve que el empleado del departamento 10 que no tiene jefe (mgr) pasa a tener una comisión de 250 que es el 5% de su salario.

6. Cursores

PL/SQL maneja un área de trabajo, también llamada área de contexto, que utiliza para ejecutar las sentencias SQL y almacenar la información que proporciona una consulta. Con los cursores podemos acceder a la información contenida en esta área de contexto.

Existen dos tipos de cursores: los explícitos, que son los definidos por el programador para acceder a las filas de una consulta, y los implícitos que son los generados por Oracle cuando se ejecuta una sentencia SQL no asociada a un cursor explícito.

Cuando se realiza una consulta el número de filas devueltas por la SELECT puede ser cero, una o más de una. Si el número de filas devueltas es más de una y necesitamos procesarlas para obtener una información, necesitamos utilizar un cursor explícito, de lo contrario se produce el error TOO_MANY_ROWS. En definitiva, utilizar cursores es como trabajar sobre un fichero virtual en memoria, en el que las filas son los registros (utilizando

conceptos de los lenguajes de tercera generación) y vamos tratando uno a uno. Es aconsejable trabajar con cursores explícitos.

Sintaxis del Cursor

CURSOR nombre_cursor [(variable [IN] tipo [{:= | DEFAULT} value] ,……)]
IS SELECT ……

La parte opcional de la declaración del cursor son los parámetros.

Para trabajar con cursores hay que realizar los siguientes pasos: declarar el cursor, abrirlo, acceder a cada una de las filas, procesarlas y cerrar el cursor.

Declarar el Cursor

```
DECLARE
CURSOR  c1  IS  SELECT descripcion, cuota FROM actividades  WHERE cuota  >  5000;
BEGIN
     ........
END;
```

El nombre del cursor **c1** es un identificador, no es una variable, se utiliza para identificar la consulta, no se puede utilizar en expresiones. Las reglas para nombrar los cursores son las mismas que las descritas para nombrar las variables

En la declaración de un cursor se pueden incorporar parámetros, que pueden ser útiles para restringir la selección de filas. Un parámetro puede aparecer en cualquier parte de la sentencia SELECT donde pueda ir una constante.

Ejemplo

```
CURSOR c2  (cuota_minima  NUMBER ) IS
SELECT descripcion, cuota FROM actividades WHERE cuota > cuota_minima;
```

Los valores a los parámetros se pasan cuando se abre el cursor.

.

Abrir el Cursor

Al abrir el cursor se ejecuta la consulta asociada a la declaración del mismo y el conjunto de filas cargadas en el cursor, en adelante las denominaremos registros para distinguirlas de las filas de la tabla, quedan disponibles para un posterior acceso registro a registro. Un puntero señala el primer registro.

OPEN c1;

Con el siguiente OPEN pasaríamos el valor 5000 al parámetro *cuota_minima* al abrir el cursor declarado en el ejemplo anterior.

OPEN c2 (5000);

Acceder a las filas

La sentencia **FETCH** lee el registro marcado por el puntero poniendo el valor de sus columnas en las variables que siguen a INTO. La lectura de un registro hace que el puntero pase a señalar el registro siguiente, con lo que el acceso se realiza secuencialmente en el orden en que la consulta recuperó las filas.

Sintaxis de la sentencia FETCH

FETCH cursor INTO variable [, variable, …….]

Ejemplo.

```
FETCH  c1  INTO  denominacion, importe;
```

Debe de existir una correspondencia entre columnas de la consulta y variables en la FETCH en cuanto a número y tipo.

Las variables *denominacion* e *importe* de la FETCH anterior se deben declarar y son del mismo tipo que *descripcion* y *cuota*.

Cerrar el Cursor.

Cuando se cierra el cursor los registros dejan de estar accesibles y se liberan los recursos asociados.

CLOSE nombre del cursor;

Para el ejemplo que estamos planteando sería.

```
CLOSE   c1;
```

6.1. Atributos de cursor

Cada cursor tiene asociados cuatro atributos predefinidos. Cuando se abre un cursor estos atributos devuelven información sobre la ejecución de la sentencia SQL asociada al cursor.

%NOTFOUND

Devuelve TRUE si el último FETCH ejecutado no encontró más registros. En caso contrario devuelve FALSE. Después de abrir el cursor y antes del primer FETCH devuelve NULL.

Sintaxis

nombre_cursor%NOTFOUND

Ejemplo.

```
DECLARE
CURSOR  c1  IS  SELECT descripcion FROM actividades  WHERE cuota  >  1000;
nombre actividades.descripcion%TYPE;
BEGIN
    OPEN C1;
     LOOP
       FETCH  c1 INTO nombre;
       EXIT WHEN C1%NOTFOUND;
        …… mientras haya registros en el cursor se ejecutan estas sentencias
     END LOOP;
     CLOSE c1;
END;
```

Si se hace referencia a un cursor no abierto se produce la excepción predefinida INVALID_CURSOR

%FOUND

Es el atributo cuya función es justamente la contraria a la anterior, devuelve TRUE si el FETCH ha encontrado el registro y FALSE si no lo encontró.

Sintaxis:

nombre_cursor%FOUND

Ejemplo.

```
……….
LOOP
  FETCH  c1 INTO nombre;
  IF C1%FOUND THEN
      Si existe el registro ejecuta estas  sentencias;
    ELSE
      EXIT;  --cuando no encuentre más registros sale del bucle
  END IF;
END LOOP;
```

%ROWCOUNT.

Devuelve el número de registros accedidos por FETCH hasta el momento.

Sintaxis:

Nombre_cursor%ROWCOUNT

Ejemplo.

```
LOOP
    FETCH  c1 INTO nombre;
    IF c1%ROWCOUNT  > 3  THEN
        Ejecuta las sentencias que están dentro del if cuando haya
        accedido  al 4 registro del cursor;
    END IF;
    ……..
END LOOP;
```

Si el cursor no está abierto se produce la excepción INVALID_CURSOR al hacer referencia a este atributo.

%ISOPEN

Devuelve TRUE si el cursor está abierto y FALSE si está cerrado.

Sintaxis:

Nombre_cursor%ISOPEN

Ejemplo.

```
IF c1%ISOPEN  THEN
      FETCH  c1 INTO nombre;
   ELSE
       OPEN c1;
END IF;
```

Estos atributos permiten también obtener información sobre la última sentencia SQL ejecutada que no tuviese cursor explícito asociado.

Si una sentencia no tiene explícitamente asociado un cursor Oracle le asocia uno. Este cursor no es manipulable desde el programa pero si permite el uso de sus atributos.

Las sentencias SQL de las que podemos tener información a través de estos atributos son: SELECT ... INTO, INSERT, UPDATE, DELETE. Los atributos son los descritos para los cursores explícitos anteponiéndoles el prefijo SQL. Veamos su descripción:

SQL%NOTFOUND

Devuelve TRUE cuando la última sentencia SELECT no recuperó ninguna fila, o cuando INSERT, DELETE, UPDATE no afectó a ninguna fila.

```
INSERT INTO actividades VALUES ('111','BALOMMANO',4000);
IF SQL%NOTFOUND  THEN
     Sentencias    --no insertó. Estas sentencias controlarían el error.
END IF;
```

Si una sentencia SELECT ... INTO no devuelve ninguna fila PL/SQL activa la excepción predefinida NO_DATA_FOUND.

```
SELECT descripcion INTO act FROM actividades WHERE codigo_actividad = 999;
```

/* si no existe la actividad para el código 999, se activaría NO_DATA_FOUND pasándose control a la parte de excepciones del bloque. */

```
IF SQL%NOTFOUND THEN
        sentencias     -- esta acción nunca se realizaría.
END IF
```

SQL%FOUND

Devuelve TRUE en el caso contrario al anterior, SELECT devuelve alguna fila o INSERT, DELETE, UPDATE afecta a alguna fila

```
INSERT INTO actividades VALUES ('111','BALOMMANO',4000);
IF SQL%FOUND THEN sentencias   --sí insertó
END IF;
```

SQL%ROWCOUNT

Devuelve el número de filas afectadas por INSERT, DELETE, UPDATE, y las filas devueltas por una SELECT. Si no hay filas afectadas o devueltas, su valor es cero.

```
DELETE FROM actividades  WHERE cuota  > 4000;
IF SQL%ROWCOUNT  >  5   THEN
    sentencias         --sí se borraron más de 5
END IF;
```

SQL%ISOPEN

Siempre devuelve FALSE, ya que Oracle cierra automáticamente el cursor cuando termina de ejecutar una sentencia SQL.

6.2. Ejemplos

Para comprobar los siguientes ejemplos conectarse como *rasty*. Algunos se hacen desde el entorno SQL y otros desde la ventana del sistema.

```
DECLARE
 numemp NUMBER(4);
 comi   NUMBER(7,2);
 CURSOR c1 IS SELECT empno, comm FROM emp;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO numemp, comi;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Número EMPLEADO: ' || numemp || '  COMISION: '|| comi);
  END LOOP;
  CLOSE c1;
END;
```

Resultados Explicar Describir SQL Guardado Historial

```
Número EMPLEADO: 7369  COMISION:
Número EMPLEADO: 7499  COMISION: 300
Número EMPLEADO: 7521  COMISION: 500
Número EMPLEADO: 7566  COMISION:
Número EMPLEADO: 7654  COMISION: 1400
Número EMPLEADO: 7698  COMISION:
Número EMPLEADO: 7782  COMISION:
Número EMPLEADO: 7788  COMISION:
Número EMPLEADO: 7839  COMISION:
Número EMPLEADO: 7844  COMISION: 0
Número EMPLEADO: 7876  COMISION:
Número EMPLEADO: 7900  COMISION:
Número EMPLEADO: 7902  COMISION:
Número EMPLEADO: 7934  COMISION:
```

Nota: dbms_output.put_line es un paquete que permite la salida por pantalla, se verá mas adelante

```
DECLARE
 numemp NUMBER(4);
 comi   NUMBER(7,2);
 CURSOR c1 IS SELECT empno, comm FROM emp;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO numemp, comi;
    IF   c1%FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Número EMPLEADO: ' || numemp || '  COMISION: '|| comi);
    ELSE EXIT;
    END IF;
  END LOOP;
  CLOSE c1;
END;
```

Resultados Explicar Describir SQL Guardado Historial

```
Número EMPLEADO: 7369  COMISION:
Número EMPLEADO: 7499  COMISION: 300
Número EMPLEADO: 7521  COMISION: 500
Número EMPLEADO: 7566  COMISION:
Número EMPLEADO: 7654  COMISION: 1400
Número EMPLEADO: 7698  COMISION:
Número EMPLEADO: 7782  COMISION:
Número EMPLEADO: 7788  COMISION:
Número EMPLEADO: 7839  COMISION:
Número EMPLEADO: 7844  COMISION: 0
Número EMPLEADO: 7876  COMISION:
Número EMPLEADO: 7900  COMISION:
Número EMPLEADO: 7902  COMISION:
Número EMPLEADO: 7934  COMISION:
```

```
DECLARE
 numemp NUMBER(4);
 comi   NUMBER(7,2);
 CURSOR c1 IS SELECT empno, comm FROM emp;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO numemp, comi;
    IF   c1%ROWCOUNT < 5 THEN
         DBMS_OUTPUT.PUT_LINE('Número EMPLEADO: ' || numemp || '  COMISION: '|| comi);
    ELSE EXIT;
    END IF;
  END LOOP;
  CLOSE c1;
END;
```


```
Número EMPLEADO: 7369  COMISION:
Número EMPLEADO: 7499  COMISION: 300
Número EMPLEADO: 7521  COMISION: 500
Número EMPLEADO: 7566  COMISION:
```

6.3. La sentencia FOR para manejo de cursores

El manejo de cursores explícitos se simplifica en la mayoría de los casos mediante el uso de un formato de la sentencia FOR disponible para esta función.

FOR nombre_de_registro IN nombre_de_cursor

La sentencia abre el cursor y recupera el primer registro en su primera ejecución. En sucesivas ejecuciones va recuperando cada uno de los registros del cursor. Cuando ya no hay más registros cierra el cursor y se termina la ejecución de la sentencia.

Los valores de los registros son accesibles a través de los campos del registro (nombres de las columnas declaradas en el cursor). El nombre del registro no hay que declararlo y es del tipo **nombre_cursor%ROWTYPE**. El registro no puede ser referenciado fuera del bucle for.

Ejemplo.

```
DECLARE
........
CURSOR  c1  IS  SELECT  codigo_actividad, descripción FROM actividades;
BEGIN
FOR  reg  IN  c1  LOOP   -- abre el cursor y accede a cada una  de los registros
     Sentencias;        -- Por cada registro accedido ejecuta estas  sentencias
     Para hacer referencia a codigo_actividad y descripcion se  utiliza la
     notación reg.codigo_actividad y reg.descripcion
     ......
END LOOP;
       -- cuando termina con todos los registros cierra el cursor
END;
```

Este formato de la sentencia FOR admite el paso de valores a parámetros del cursor, igual que ocurría con la sentencia OPEN.

```
DECLARE
CURSOR  c1  (tope  NUMBER) IS  SELECT  codigo_actividad, descripcion
                 FROM actividades WHERE cuota > tope;
BEGIN
  FOR  reg  IN  c1(5000)  LOOP  -- selecciona las filas con cuota superior a 5000
     ….   Sentencias;
   END LOOP;
END;
```

Cuando existen columnas del cursor resultado de evaluar expresiones, estas se deben renombrar con un alias para permitir su referencia.

```
 CURSOR c1 IS
  SELECT denominacion, pvp_tarifa - pvp_coste  ganancia,  existencias
                           FROM    almacen;
```

En el ejemplo anterior para hacer referencia a la expresión (*pvp_tarifa - pvp_coste)* utilizaremos el alias *ganancia*

```
                    IF reg.ganancia > 20000  THEN ……
                     END IF
```

6.4. Modificación y eliminación de filas de una tabla usando cursores.

Como se vio en la unidad anterior, las sentencias UPDATE y DELETE permiten modificar y eliminar, respectivamente las filas de una tabla que verifican la condición especificada en su cláusula WHERE.

Cuando se trabaja en PL/SQL con cursores, estas sentencias admiten también el siguiente formato de la cláusula WHERE:

WHERE CURRENT OF nombre_cursor

```
DELETE FROM actividades   WHERE CURRENT OF  c1;
UPDATE actividades SET  ........   WHERE CURRENT OF c1;
```

Al poner esta cláusula, la fila modificada o eliminada se corresponde con la accedida del cursor especificado.

Para poder usar CURRENT OF es imprescindible haber utilizado la cláusula FOR UPDATE en la sentencia SELECT que define el CURSOR. La cláusula FOR UPDATE hace que al ejecutar la SELECT, con OPEN CURSOR, las filas de la tabla leídas queden bloqueadas. Estos bloqueos son necesarios para evitar los problemas de concurrencia citados en unidades anteriores.

CURSOR c1 IS SELECT cuota FROM actividades FOR UPDATE;

Veamos varios ejemplos:

```
a)
DECLARE
      CURSOR c1 IS SELECT cuota FROM actividades FOR UPDATE;
      nueva_cuota  NUMBER(7);
BEGIN
      OPEN c1;
      LOOP
       FETCH  c1 INTO nueva_cuota;
       EXIT WHEN c1%NOTFOUND;
       UPDATE actividades SET cuota = nueva_cuota * 1.20 WHERE CURRENT OF c1;
      END LOOP;
```

```
      CLOSE c1;
      --COMMIT; lo comento para que no valide la transacción
END;
```

```
b)

DECLARE
      CURSOR c1 IS SELECT * FROM actividades FOR UPDATE;
BEGIN
  FOR reg IN c1 LOOP
      UPDATE actividades SET cuota = reg.cuota * 1.2  WHERE CURRENT OF c1;
  END LOOP;
     COMMIT; en este caso validaría la transacción
END;
```

```
     c)

     DECLARE
            CURSOR c1(nombre CHAR) IS  SELECT  *  FROM actividades
            WHERE codigo_actividad = nombre FOR UPDATE;
      registro c1%ROWTYPE;
      BEGIN
      OPEN c1('act1');
      LOOP
            FETCH c1 INTO registro;
            EXIT WHEN c1%NOTFOUND;
            DELETE FROM actividades WHERE CURRENT OF c1;
       END LOOP;
       CLOSE c1;
       END;
```

Ejemplo

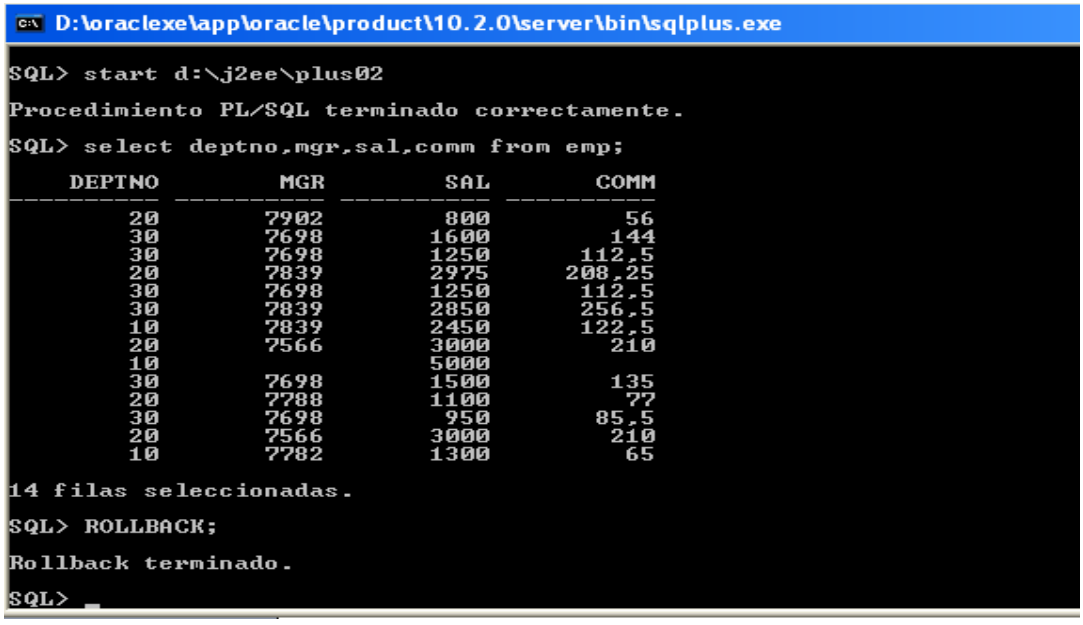
Actualizar la comisión de los empleados que tengan jefe, teniendo en cuenta el departamento al que pertenecen.

- Los que pertenecen al departamento 10 la comisión será de un 5% del salario.
- Los que pertenecen al departamento 20 la comisión será de un 7% del salario.
- Los que pertenecen al departamento 30 la comisión será de un 9% del salario.
- Los que no pertenecen a ninguno de los anteriores la comisión será de un 2%.

En este caso hay que utilizar un cursor dado que hay muchos empleados que tienen jefe.

Solución con FETCH

```
DECLARE
departamento emp.deptno%TYPE;
incremento number(3,2);
CURSOR c1 IS SELECT deptno FROM emp WHERE mgr IS NOT NULL FOR
                                                        UPDATE;
BEGIN
OPEN c1;
LOOP
FETCH c1 INTO departamento;
                      EXIT WHEN c1%NOTFOUND;

                    IF departamento=10    THEN       incremento:=0.05;

                     ELSIF departamento=20 THEN       incremento:=0.07;

                     ELSIF departamento=30 THEN       incremento:=0.09;

                   ELSE   incremento:=0.02;

                  END IF;
UPDATE emp SET comm = sal * incremento           WHERE CURRENT OF C1;
END LOOP;
CLOSE C1;
END;
/
```



En la captura se puede ver el resultado de la ejecución del bloque. Al final se hace *rollback* para deshacer la transacción y que las tablas queden como estaban.

Solución con FOR

```
DECLARE
CURSOR c1 IS SELECT deptno FROM emp WHERE mgr IS NOT NULL FOR
                                                        UPDATE;
incremento number(3,2);
BEGIN
FOR reg IN c1 LOOP
                IF     reg.deptno=10   THEN   incremento:=0.05;

                  ELSIF reg.deptno=20  THEN   incremento:=0.07;

                  ELSIF reg.deptno =30 THEN   incremento:=0.09;

                   ELSE incremento:=0.02;

                END IF;

UPDATE emp SET comm = sal * incremento     WHERE CURRENT OF C1;
END LOOP;
END;
-------------------------------------------------
```

En este caso no hace falta declarar la variable departamento, la notación *reg.deptno* hace referencia al departamento del registro donde está el puntero del cursor que se corresponderá con la fila de la tabla accedida.

Otro ejemplo

Actualizar la columna **CATEGORÍA** de la tabla **EMP** con una cadena de asteriscos teniendo en cuenta que se escribe un asterisco por cada mil unidades de sueldo. Si la tabla no tiene la columna, modificar la estructura para añadirla.

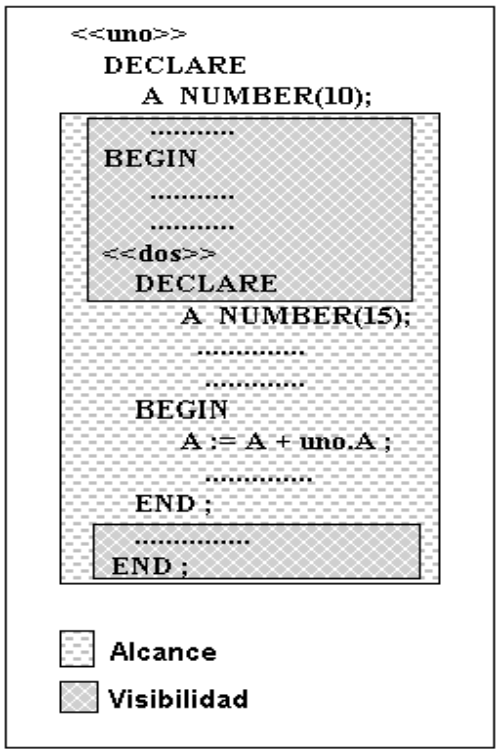
Solución

```
DECLARE
 num NUMBER:=0;
 CURSOR C1 IS SELECT sal FROM emp FOR UPDATE;
BEGIN
 FOR REG IN C1 LOOP
       num:=trunc(REG.sal/1000);
       IF num>=1 THEN
             UPDATE emp
                   SET CATEGORIA=lpad('*',num,'*')
```

```
                        WHERE CURRENT OF C1;
            END IF;
     END LOOP;

    END;
```

7. Alcance y visibilidad de identificadores

Cuando hablamos de identificadores nos estamos refiriendo a los nombres de los distintos objetos de un programa, es decir, a los nombres de las variables, constantes, excepciones, cursores, subprograms y paquetes. También tratamos de las etiquetas que pueden identificar a los bloques PL/SQL o designar determinados puntos de un programa.



Alcance y visibilidad de la variable A **del bloque** *uno*

Las referencias a un identificador se resuelven según su alcance y visibilidad. El **alcance** de un identificador es la región del programa desde la que se puede referenciar. Un identificador es **visible** en aquella parte del programa en que se puede referenciar con un nombre sin calificar.

Los identificadores declarados en un bloque son locales a ese bloque y globales a todos sus sub-bloques. Si un identificador global es redeclarado en un sub-bloque, ambos identificadores mantienen su alcance, pero dentro del sub-bloque, a partir del punto en que se declara, solo es visible el local.

En el ejemplo, se ve el alcance y la visibilidad de la variable **A** declarada en el bloque externo. Para referirse a esta variable en el bloque ***dos***, hay que calificarla (**uno.A**). El alcance y la visibilidad de la **A** del bloque ***dos*** coincidirían con la parte del bloque que va desde el punto donde se declara hasta el final.

Desde un bloque no se pueden referenciar identificadores declarados en sus sub-bloques, ni en otros bloques que no lo contengan.

A continuación tenemos un ejemplo de alcance y visibilidad de identificadores.

```
DECLARE
  a  CHAR;
  b  REAL;
BEGIN
  -- Aquí se puede utilizar: a (CHAR) y b
  DECLARE
        a  INTEGER;
        c  REAL;
  BEGIN
        /* Aquí se pueden referenciar: a (INTEGER), b y c.
         * a (CHAR) tiene alcance, pero no se puede referenciar
         * porque no hay forma de calificarlo.
         */
  END;
  DECLARE
        d  REAL;
  BEGIN
        -- Identificadores referenciables: a  (CHAR), b y d.
  END;
        -- Identificadores referenciables: a (CHAR) y b.
END;
```

9. Tipos de datos compuestos

PL/SQL dispone de dos tipos de datos compuestos: las tablas y los registros (**TABLE** y **RECORD)**

TABLAS

Este tipo de datos corresponde al concepto que tenemos de tablas en otros lenguajes, que nada tienen que ver con las tablas de la base de datos. En este caso las tablas (arrays o vectores) constan de dos columnas y N filas, el numero de filas no tiene porque estar predeterminado ya que crece dinámicamente. Una columna es el índice y es del tipo BINARY_INTEGER, la otra es la que almacena los valores y puede ser de cualquier tipo. La sintaxis para declarar los tipos de tabla es la siguiente:

```
TYPE  tipo_tabla IS  TABLE  OF
                { tipo_dato   |  variable%TYPE  |  tabla.columna%TYPE } [NOT
                                                NULL]   INDEX  BY  BINARY_INTEGER;
```

Para trabajar con estas tablas primero se declara un tipo de tabla y luego se declara una o más variables de ese tipo.

Índices	valores

- Los índices no se cargan sólo se referencian y pueden tomar valores en el rango permitido por el tipo BINARY_INTEGER (-2^{31} -1 a 2^{31} –1 en la versión 2.3).
- Los valores pueden ser de cualquier tipo y son los que se cargan.

Supongamos que deseamos definir una tabla en memoria para cargar los *nombres de las actividades.*

```
DECLARE
 TYPE tabla_actividades IS TABLE OF
           actividades.descripcion%TYPE
           INDEX BY BINARY_INTEGER
```

Declaramos dos variables del tipo TABLA_ACTIVIDADES.

```
 tabla1          tabla_actividades;
 tabla2          tabla_actividades;
```

asignamos un valor a los elementos 1 y 100

```
BEGIN
 tabla1(1)       := 'gimnasia';
 tabla2(100)     := 'judo';
```

también podemos hacer una asignación de tabla a tabla

```
 tabla1 := tabla2
```

y podemos utilizar variables para referenciar el índice, siempre que se declaren del tipo BINARY_INTEGER

```
 tabla1(i) := 'baloncesto';
```

Atributos de las tablas

Podemos utilizar los siguientes atributos para el manejo de los elementos de una tabla

EXISTS(n) Devuelve TRUE si existe el elemento **n** de la tabla

```
IF tabla1.EXISTS(i) THEN ....
```

COUNT Devuelve el número de elemento de una tabla

```
IF tabla1.COUNT = 100 THEN ....
```

FIRST y **LAST** Primer y último elemento de la tabla

```
i  BINARY_INTEGER  :=  tabla1.FIRST;
```

PRIOR(n) y **NEXT(n)** Devuelve el anterior y siguiente elemento respecto a **n** respectivamente

DELETE (m, n) Borra los elementos desde **m** hasta **n**, con un solo parámetro borra el elemento del parámetro y sin ellos borra toda la tabla.

```
tabla1.DELETE(10, 15);   -- borra desde el 10 hasta el 15
```

REGISTROS

El tipo registro coincide con el usado en otros lenguajes, es una estructura formada por varios campos de distintos tipos. De igual forma que en las tablas, primero se debe definir un tipo de registro y luego una variable del tipo registro.

El registro se define declarando cada uno de los campos que lo constituye o bien dándole la estructura de una fila o parte de una fila de una tabla de la base de datos mediante: el atributo **%ROWTYPE**. La sintaxis para declarar los tipos de registro es la siguiente

```
TYPE  tipo_registro   IS  RECORD
                    (campo1  {tipo_dato  |  variable%TYPE  |
                              tabla.columna%TYPE     |
                          cursor%ROWTYPE        |
                              tabla%ROWTYPE} [NOT NULL],
                     campo2  {tipo_dato  |  variable%TYPE  |
                              tabla.columna%TYPE       |
                           cursor%ROWTYPE        |
                              tabla%ROWTYPE} [NOT NULL],
              ......................
              );
```

En el siguiente ejemplo se hace una declaración de registro de forma explícita.

```
DECLARE
 TYPE reg_activ  IS RECORD
      (codigo          VARCHAR2(7) NOT NULL := 'ACTI1',
       denominacion    actividades.descripcion%TYPE,
```

```
            cuotas          actividades.cuota%TYPE
                );
     var_activ  reg_activ;     -- definición de la variable
    BEGIN
     SELECT codigo, denominacion, cuotas INTO var_activ
                               FROM actividades
                               WHERE .............
     var_activ.cuotas := 3000;
```

Para referenciar un campo se sigue la notación *variable_registro.nombre_campo,* como se ve en la última línea del ejemplo anterior.

En el siguiente ejemplo se define *var_activ* como variable registro con los campos iguales a las columnas de la tabla *actividades,* usando el atributo %ROWTYPE

```
    DECLARE
     var_activ  actividades%ROWTYPE;
    BEGIN
     SELECT  *  INTO  var_activ   FROM actividades WHERE …….
     var_activ.cuota  :=  3000;      -- cuota   es una columna de la tabla actividades
```

También podríamos haber definido la variable registro con la misma estructura de las filas recuperadas por la consulta de un cursor.

```
DECLARE
     DECLARE CURSOR acti IS
          SELECT codigo_actividad, cuota
               FROM actividades;
     var_acti acti%ROWTYPE;
     ......................
    BEGIN
     ..................
     FETCH acti INTO var_activ;
     ..................
     IF var_activ.cuota > 3000 THEN  ...
               ......................
               ......................
    END;
```

En una declaración se puede usar un tipo del tipo registro previamente declarado; con lo que se pueden definir registros con subcampos a múltiples niveles.

```
    DECLARE
     TYPE reg_cuentas    IS RECORD
```

```
      (cod_banco     NUMBER(8) ,
      dig_ctrl       NUMBER(2),
      cta            NUMBER(10);
      );
 TYPE  reg_clientes  IS RECORD
      (cod_cli       VARCHAR2(7),
       cuenta        reg_cuentas;        -- registro anidado
                         );
 var_cuentas         reg_cuentas;
 var_clientes        reg_clientes;
BEGIN


  var_cuentas.cod_banco := 00220088;

  var_clientes.cuenta.cod_banco:= 00220088;
```