UD6. PL/SQL

Índice

Introducción

Junto al lenguaje universal de gestión de bases de datos relacionales, SQL, los diferentes sistemas gestores de bases de datos pueden disponer de lenguajes que complementen a este y que estén orientados a la creación de programas que posibiliten la automatización de los procesos de administración de una base de datos. Es el propio SGBD el encargado de mantener los datos que soporta, sus actualizaciones y sus cambios, asi como sus controles de integridad y de coherencia. Algunos ejemplos de estos lenguajes que posibilitan la creación de estos mecanismos son PL/SQL para bases de datos Oracle, T-SQL para SQL Server o PL/PGSQL para Postgres.

PL/SQL es un lenguaje de programación que se usa para bases de datos Oracle. Está integrado junto al motor del SGBD, lo que hace que la ejecución de programas escritos en este lenguaje sea rápida y eficiente. PL/SQL supone un complemento muy potente junto con SQL, ya que combina la potencia del lenguaje SQL con estructuras procedimentales de un lenguaje procedimental, además de proveer nuevas estructuras que permiten implementar de una forma cómoda mecanismos de acceso a bases de datos implementadas con el SGBD Oracle.

El PL/SQL

Los programas PL/SQL están compuestos por instrucciones y comandos de SQL. No todos los comandos de SQL se usan del mismo modo que cuando se ejecutan en una terminal, sino que pueden variar para ser integrados con las instrucciones del lenguaje que los embebe. Los programas están compuestos por instrucciones que se ejecutan una detrás de otra. Excepto la primera y la última instrucción todas se ejecutan dependiendo de lo ocurrido con su instrucción anterior, lo que provocará una situación diferente para la instrucción siguiente. Evidentemente, en este conjunto de instrucciones pueden existir saltos para ejecutar una sección u otra dependiendo de diferentes casos y existirán estructuras que permitan ejecutar un conjunto de instrucciones más de una vez. Estas nuevas estructuras, denominadas estructuras de control, son las que les dan personalidad a estos lenguajes de programación orientados a procedimientos. Las estructuras de control pueden ser bucles que repiten unas acciones o condiciones que bifurcan la ejecución secuencial del programa.

Introducción al lenguaje

lenguaje PL/SQL está compuesto por técnicas propias de la programación donde se pueden embeber los comandos del sublenguaje de consultas SQL, pero no se pueden usar aquellos propios para la definición del esquema de la base de datos ni los de control, es decir, los comandos asociados al DDL y al DCL. Los programas elaborados con PL/SQL se ejecutan en el servidor y están almacenados junto a los datos y no están orientados a la entrada por teclado y salida por pantalla, sino, más bien, leen datos de la base de datos e introducen o controlan la entrada de nuevos datos. La mayoria de los procedimientos que se elaboran están relacionados con el control de inserciones,

modificaciones y eliminaciones de registros en las tablas. Tanto los tipos de datos como los operadores estudiados en SQL son totalmente válidos en programas PL.

Para introducir comentarios entre los comandos del lenguaje se usan los caracteres **--** cuando el comentario ocupa una linea. Cuando ocupa más de una linea, el comentario se delimita con los caracteres **/*** al principio y ***/** al final. Cualquier procedimiento PL/SQL tiene tres bloques principales, **DECLARE, BEGIN** y **EXCEPTION**, más la etiqueta **END**. A continuación se detalla cada uno de ellos:

Bloque **DECLARE**: este bloque es opcional y se usa para declarar todas las variables y constantes que se utilizan, los cursores declarados y las excepciones definidas por el usuario.

Bloque **BEGIN**: este bloque es obligatorio, siempre debe aparecer declarado y es el espacio más importante del programa, ya que contiene todos los comandos SQL junto con las sentencias de control propias de PL. En definitiva, es el bloque principal del procedimiento.

Bloque **EXCEPTION**: en este bloque se define qué hacer en caso de que, durante la ejecución del bloque BEGIN, ocurra algún error. Es un bloque opcional.

Etiqueta **END**: es obligatoria y finaliza con un punto y coma. De forma opcional, al igual que después de la etiqueta BEGIN, se puede indicar el nombre del bloque.

El siguiente ejemplo muestra un programa PL/SQL sin ningún comando especial, solo contiene comentarios y las etiquetas que definen diferentes bloques:

```
--Este es un comentario que ocupa una linea
/*Aqui comienza una serie de líneas
que corresponden a comentarios del programador.
Es bueno comentar el código, asi en un futuro todo será más claro.
Este comentario ocupa una serie de lineas*/

DECLARE
        --Aqui se declaran las constantes y las variables
BEGIN
        --Aqui están todos los comandos del programa PL/SOL
EXCEPTION
        --Aquí vamos a capturar las excepciones
END;
/   --Este carácter se usa Si el programa está almacenado en un fichero o luego viene más código
```

Para ejecutar un procedimiento almacenado se usa el comando EXECUTE de la forma:

```
SQL>EXECUTE nombreProcedimiento
```

Variables

Cuando se leen datos, ya sea por teclado o accediendo a un fichero, estos deben llevarse a la memoria principal para poder trabajar con ellos. Para poder referirse a cada uno de los datos que están en memoria y que han sido leidos recientemente, se les asigna un nombre que representa su valor. En realidad, es un espacio de memoria que permite el almacenamiento de un valor. Para introducir el dato en ese espacio es necesario, además de etiquetarlo con un nombre, indicar qué espacio puede ocupar. Esto es lo que se conoce como variable y tipo de datos de una variable.

Estos identificadores no pueden tener más de 30 caracteres y tampoco tener el nombre de un comando, palabra reservada o variable del sistema o alguna variable definida por el usuario. Los tipos de datos son los mismos que se usan en SQL, aunque ahora hay unos nuevos que permiten más potencialidad.

La sintaxis de las variables es la siguiente:

nombreVariable [CONSTANT] tipo [NOT NULL] [{DEFAULT | := } valor]

En PL/SQL se declara una variable por linea y no se puede declarar más de una, aunque tengan el mismo tipo. Igual ocurre cuando se asignan valores.

Una vez que se ha indicado el tipo de la variable y se le ha dado un nombre, esta puede ser inicializada si se conoce el valor inicial que toma. Generalmente, una variable está destinada a tomar un valor que no se conoce y que puede variar a lo largo de la ejecución de un programa. Pero de otras se conoce su valor inicial, por lo que pueden ser inicializadas y tomarán nuevos valores a lo largo de la ejecución. En tal caso se usarán los caracteres de asignación **:=** seguidos de su valor inicial.

Una variable declarada como NOT NULL siempre tiene que ser inicializada. Si el valor para una variable es constante, es que se conoce su valor inicial al comienzo de la ejecución de un programa y no varia a lo largo de su ejecución. Se usa para ello la opción CONSTANT. Cuando se declara una constante no se usan las opciones NOT NULL y DEFAULT.

Cuando se usan los caracteres := para asignar a una variable un valor, este puede ser cualquier expresión válida para el lenguaje. Esta expresión puede ser un valor explicito, otra variable o un operando con uno o dos argumentos. Los operadores pueden ser numéricos, cuando los argumentos son números; alfanuméricos o de caracteres, cuando son cadenas; y lógicos, cuando los argumentos devuelven un valor cierto (true) o falso (false). Un operador de carácter muy usado es ||, que permite concatenar dos cadenas devolviendo la unión de ambas.

```
v_precio NUMBER(6,2);
v_numRegistros NUMBER:=1;
v_Dirección VARCHAR2(60) DEFAULT ‘Desconocida’;
v_NumMaxAsig CONSTANT NUMBER:= 15;
v_sueldoCorrecto BOOLEAN := v_sueldo>2000 and v_sueldo<5000;
v_sueldo number := v_suedoBase * 1.2;
v_usuario := ‘COD_’ || v_login;
```

Tipos de datos

Los tipos de datos más comunes en PL/SQL son

Tipos de datos numéricos	BINARY_INTEGER, DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INT, INTEGER, NATURAL, NATURALN, NUMBER, NUMERIC, PLS_INTEGER, POSITIVE, POSITIVEN, REAL, SIGNTYPE, SMALLINT
Tipos de datos de cadenas	CHAR, CHARACTER, LONG, LONG RAW, NCHAR, NVARCHAR2, RAW, ROWID, STRINGM UROWID, VARCHAR, VARCHAR2
Tipos de datos lógicos	BOOLEAN
Tipos de datos de fecha	DATE, INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH, TIMESTAMP, TIMESTAMP WITH LOCAL TIME ZONE, TIMESTAMP WITH TIME ZONE
Tipos de datos compuestos	RECORD, TABLE, VARRAY
Tipos de datos grandes	BFILE, BLOB, CLOB, NCLOB

Select... into

Las consultas en PL/SQL se utilizan con el fin de almacenar su resultado en una estructura para poder utilizar los datos de interés. Por esta razón, toda consulta debe tener la cláusula INTO donde se indica la variable donde se almacena el resultado. La variable debe ser capaz de almacenar la cantidad de datos que devuelve la consulta. Cuando se usa una variable simple, la consulta solo puede devolver un valor, y se requiere garantizar que esto es asi; en caso contrario devolveria una excepción. La siguiente consulta almacena en la variable v_Cantidad el número de registros que tiene la tabla Venta:

```
SELECT COUNT(*) INTO v_Cantidad FROM Venta;
```

Como se ha indicado antes, esta consulta almacena la cantidad de registros de la tabla Venta en la variable v_Cantidad. Esto se hace através de la cláusula INTO. Toda consulta en PL/SQL debe tener esta cláusula.

Sin embargo, la siguiente consulta puede fallar porque no se garantiza la existencia de un único registro, aunque parezca improbable:

```
SELECT dni INTO v_dni FROM Profesor WHERE Nombre='David';
```

Existen variables de sustitución donde se usa el simbolo &, por ejemplo se quiere consultar el número de profesores que pertenecen a un departamento cuyo código se solicita por teclado. La consulta seria la siguiente:

```
SELECT COUNT(*) INTO v_NumProf FROM Profesor WHERE CodDep=&departamento;
```

Se recomienda almacenar en una variable el valor que se pide por teclado, como se indica en el siguiente ejemplo:

```
v_dep:=&codigo_Departamento;
SELECT COUNT(*) INTO v_NumPrOf FROM Profesor WHERE CodDep=v_dep;
```

El atributo %TYPE es muy interesante, ya que permite obtener el tipo de una columna o de una variable definida con anterioridad. Esto supone una caracteristica muy útil que permite la flexibilidad en cuanto a cambios en la definición de columnas, ya que se obtendría el tipo de datos según el momento de ejecución del programa. En el siguiente ejemplo se muestra un programa PL/SQL en el que se declaran varios tipos de datos. Se usa también el atributo %TYPE para obtener el tipo de dato de las columnas CF y la sigla de la tabla Ciclo.

Para imprimir datos por pantalla usamos los procedimientos **put** y **put_line** al que le pasamos la cadena de caracteres que queremos imprimir. Estos procedimientos se encuentran en el paquete **dbms_output** y para usarlos debemos redirigir la salida con **set serveroutput on**

```
SET SERVEROUTPUT ON
DECLARE
        vNumHorasTotal PLS_INTEGER;
        --La variable vCodigoCiclo será del mismo tipo que la columna CF de la tabla Ciclo
        vCodigoCiclo ciclo.CF%TYPE;
        -- La variable vSiglaCiclo será del mismo tipo que la columna Sigla de la tabla Ciclo
        vSiglaCiclo ciclo.sigla%TYPE;
BEGIN
        dbms_output.put.line('Esto lo muestra por pantalla y luego retorno carro');
END;
```

PRÁCTICA PROPUESTA

Consultas y variables

Para la base de datos "Tienda de informática" crea un bloque pl/sql que imprima por pantalla el número de ventas y compras realizadas. La salida debe tener este formato:
"se han realizado hasta el momento numero ventas y numero2 compras"
Siendo numero y numero2 el número de ventas y compras que has obtenido de la base de datos.

PRÁCTICA PROPUESTA

Consultas y variables

Para la base de datos HR crea un bloque pl/sql que imprima por pantalla el nombre y apellido del trabajador cuyo ID escribes por teclado. Comprueba que funciona y luego pruébalo con un ID que no existe y mira qué sucede.

PRÁCTICA PROPUESTA

Consultas y variables

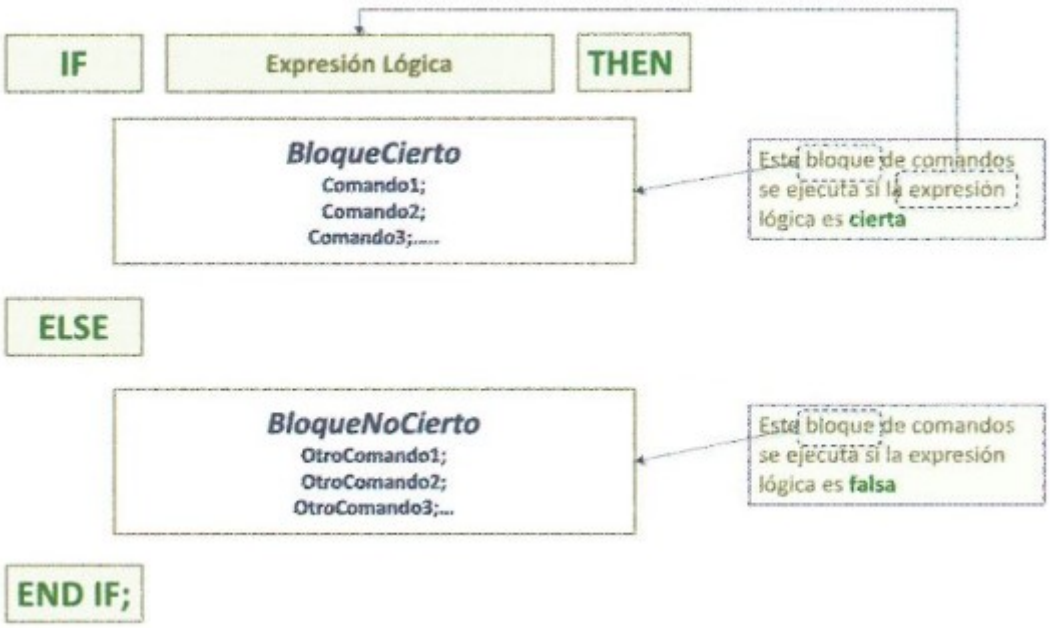
Haz un bloque PL/SQL que te pregunte por la fecha de nacimiento y devuelva el día de la semana en que naciste (luns...domingo). Para que te pregunte por una variable hay que poner antes del nombre && como por ejemplo: v_data_nacemento DATE := &&s_data_nacemento;
Para escribir por pantalla usaremos la función DBMS_OUTPUT.PUT_LINE(cadena) donde cadena es el mensaje que queremos que se muestre por pantalla. Usa la función TO_CHAR, para eliminar espacios puedes usar RTRIM (que elimina los espacios delante de la palabra). Usa el comando SET VERIFY ON/OFF para que si/no se muestren las sustituciones.

Estructuras de control

Las estructuras de control permiten controlar la secuencia de ejecución de los comandos del cuerpo de un programa. Cuando se ejecutan las instrucciones de un programa o los comandos de un script, se comienza por el principio hasta el final. Todo programa debe permitir el fin de su ejecución. Este orden puede ir cambiando de una ejecución a otra dependiendo de los valores que se estén procesando. Asi, la ejecución no es siempre la misma, existiendo casos en los que ciertos comandos no se ejecutan y otras veces si.

if-then-else

Permiten ejecutar un subbloque de comandos atendiendo a si se cumple una condición o no.

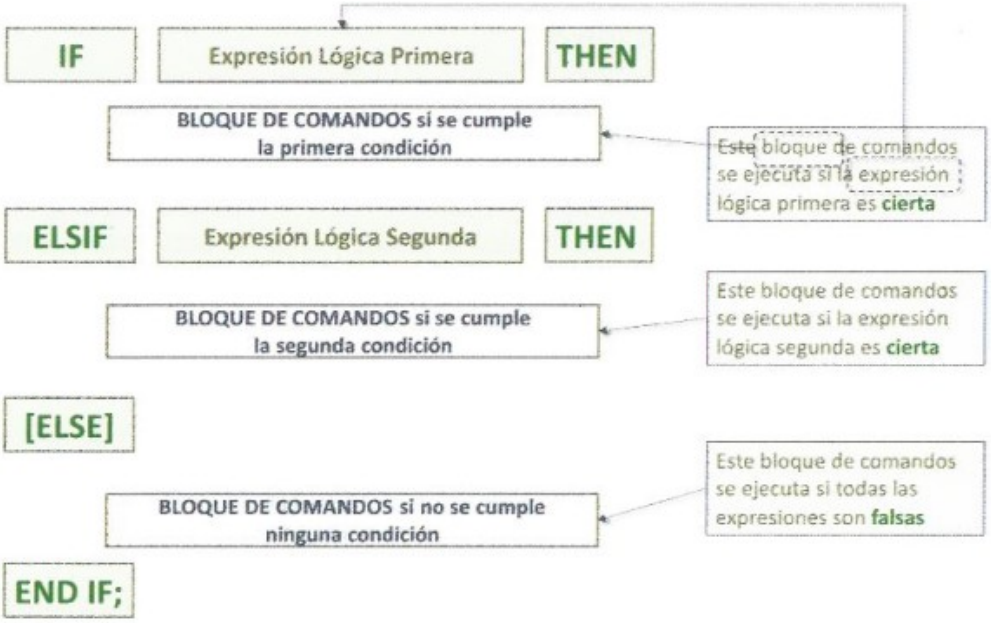
La 

condición siempre está relacionada con una expresión lógica o booleana que devuelve el valor TRUE o el valor FALSE. En caso de que la condición sea cierta, se ejecutan las órdenes del bloque llamado BloqueCierto. En caso de que sea falsa, se ejecutan las del bloque BloqueNoCierto. En un subbloque de este tipo pueden existir tantas órdenes como se desee, e incluso una nueva orden IF. Cuando se ejecuta cualquier bloque de un IF, el flujo de la ejecución salta al siguiente comando después de la etiqueta END IF.

El primer bloque está definido entre las etiquetas THEN y ELSE; el segundo entre ELSE y END IF, siendo este opcional. El comando IF debe verse como un único comando con su punto y coma final, es decir:

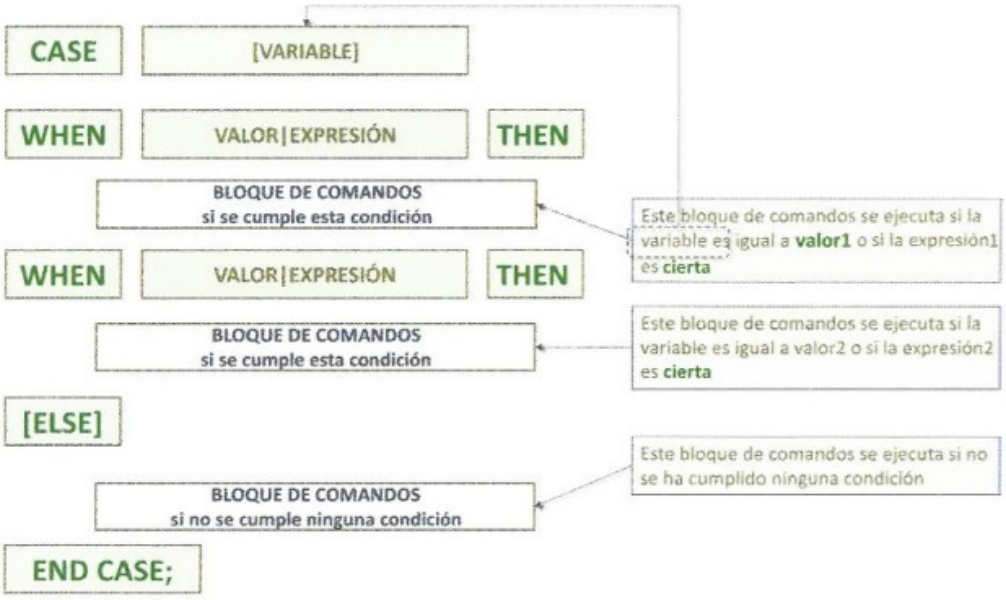
IF condición THEN BloqueSiCierto; ELSE BloqueSiFalso; END IF ;

Si en el BloqueNoCierto el primer comando que se requiere indicar es otro IF, este se puede anidar usando la etiqueta ELSIF o poner otro IF en el bloque ELSE.



Cuando se evalúa la expresión lógica primera, si no es cierta, salta a la etiqueta ELSIF para evaluar la expresión lógica segunda. Si de nuevo no es cierta, salta hasta la siguiente etiqueta ELSIF siguiendo este proceso hasta que alguna de ellas es cierta o hasta que se llega a la etiqueta opcional ELSE, que obliga a ejecutar el bloque de código llamado “BLOQUE DE COMANDOS si no se cumple ninguna condición”. Recuérdese que cuando se ejecuta cualquier bloque de un IF, el flujo de la ejecución salta al siguiente comando después de la etiqueta END IF.

El comando CASE permite construir estructuras selectivas múltiples. Estas estructuras evalúan múltiples expresiones



Si al lado del case ponemos el nombre de una variable en los When pondremos valores

Si al lado del case no ponemos nada en el When pondremos expresiones que sean condiciones.

PRÁCTICA PROPUESTA

Consultas y variables

Realiza un programa PL/SQL que pida dos fechas por teclado y que muestre cuál es la más cercana a la actual.

Bucles : LOOPS

Los bucles permiten ejecutar un bloque de código un número de veces. A este bloque de código se le llama bucle, y cada una de las vueltas se denomina iteración. A veces, el número de iteraciones es conocido por el programador; otras, porque depende de una condición, no se conoce. Por ejemplo, si se pide a un usuario que introduzca un número hasta que este sea negativo, el programador no puede prever cuántas veces el usuario introducirá números positivos.

Los comandos para realizar estructuras repetitivas son LOOP, FOR y WHILE.

Loop

El Loop se llama bucle básico y debemos salirnos del bucle con exit para que no se convierta en un bucle infinito.

Para los bucles

simples se usa la siguiente sintaxis:

```
LOOP
        [bloque];
        EXIT WHEN condición;
        [bloque;]
END LOOP;
```

El bloque de código que se repite está comprendido entre las etiquetas LOOP y END LOOP. La orden EXIT WHEN condición permite finalizar las iteraciones en el momento que la condición no sea cierta, saltando a la orden que sigue a la etiqueta END LOOP. También se puede implementar usando IF condición THEN EXIT;. Esta orden está orientada a ejecuciones donde no se conoce el número de veces que se necesita iterar, aunque también se usan acompañadas de un contador que finaliza el bucle llegado a un valor conocido.

Fijémonos como el EXIT WHEN puede ir al principio, en medio del código o al final del código, todo depende de si queremos ejecutar algo antes de que se evalúe la expresión que nos dice si tenemos que salirnos del bucle o no.

Con el bucle básico o simple se pueden hacer todos los bucles que se te puedan ocurrir, sin embargo hay otras sintaxis para los tipos de bucles más comunes. De esta forma es más fácil hacerlos pues se parecen más al lenguaje natural y es más difícil equivocarse pues todos llevan incorporada la condición de salida y no tenemos que ponerla nosotros.

While

El funcionamiento es análogo, diferenciándose en que con LOOP la orden de fin de bucle está incorporada y se evalúa al comienzo del bucle. Esto supone que, antes de ejecutar los comandos internos del bucle, la condición se evalúa pudiendo ocurrir que no se ejecute ni una sola vez. Si no ponemos condición en el while tendremos un error de sintaxis.


```
WHILE condicion LOOP
        Bloque While;
END LOOP;
```

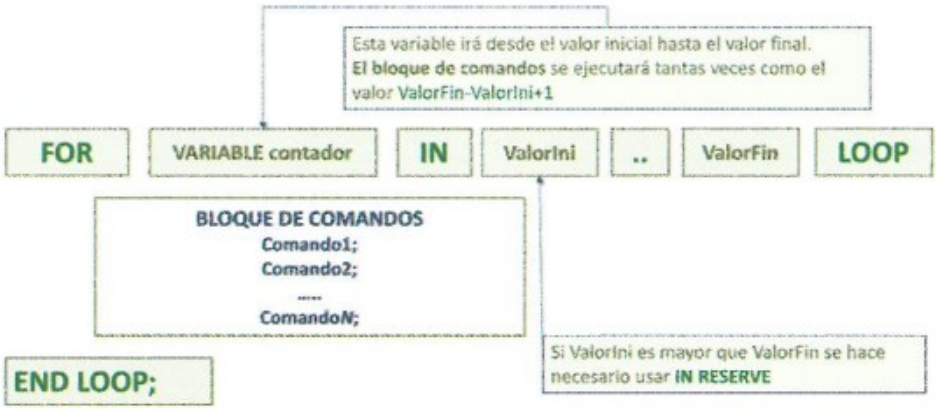
Ojo, es muy importante que dentro del while en algún momento la condición sea falsa para que salgamos del bucle, si no haremos un bucle infinito.

For

Otro comando para construir bucles con valores numéricos es FOR. Esta orden se usa cuando se conoce el número de veces que se requiere repetir el bucle.

```
FOR contador IN [REVERSE] número1..número2 LOOP
        BloqueFor;
END LOOP;
```

La variable que se emplea en la estructura FOR se puede no declarar en la sección DECLARE.



PRÁCTICA PROPUESTA

Bucles

Realiza un programa PL/SQL que muestre la tabla de multiplicar de un número solicitado por teclado. Plantea la solución usando los tres tipos de bucles.

Subprogramas

La idea es bien sencilla: se determina con claridad un conjunto de comandos que puede ser utilizado en cualquier parte de un programa y se encapsula mediante un subbloque especial llamado subprograma. En vez de escribir de nuevo todo el código, el código del subbloque se reutiliza simplemente invocándolo por su nombre.

Estos subprogramas también pueden crearse para que sean externos a un programa con el fin de que puedan reutilizarse con el simple hecho de invocarlos. Para invocar un subprograma solo es necesario utilizar el nombre de este en el código de un programa, como si de un comando se tratase. Este tipo de subprogramas se almacenan en la base de datos. Cualquier subprograma está compuesto por dos partes principales:

- **Especificación**: aqui se indican el nombre del subprograma y la definición de parámetros de entrada y salida.
- **Cuerpo**: bloque PL/SQL asociado al subprograma y que tiene las mismas partes que un programa completo, es decir, bloque DECLARE, bloque BEGIN, bloque EXCEPTION y etiqueta END.

Los subprogramas pueden aceptar parámetros de entrada y pueden devolver valores.
En esta entrada y esta salida de valores radica la potencia de este tipo de estructuras de programación.

Existen dos tipos de subprogramas:

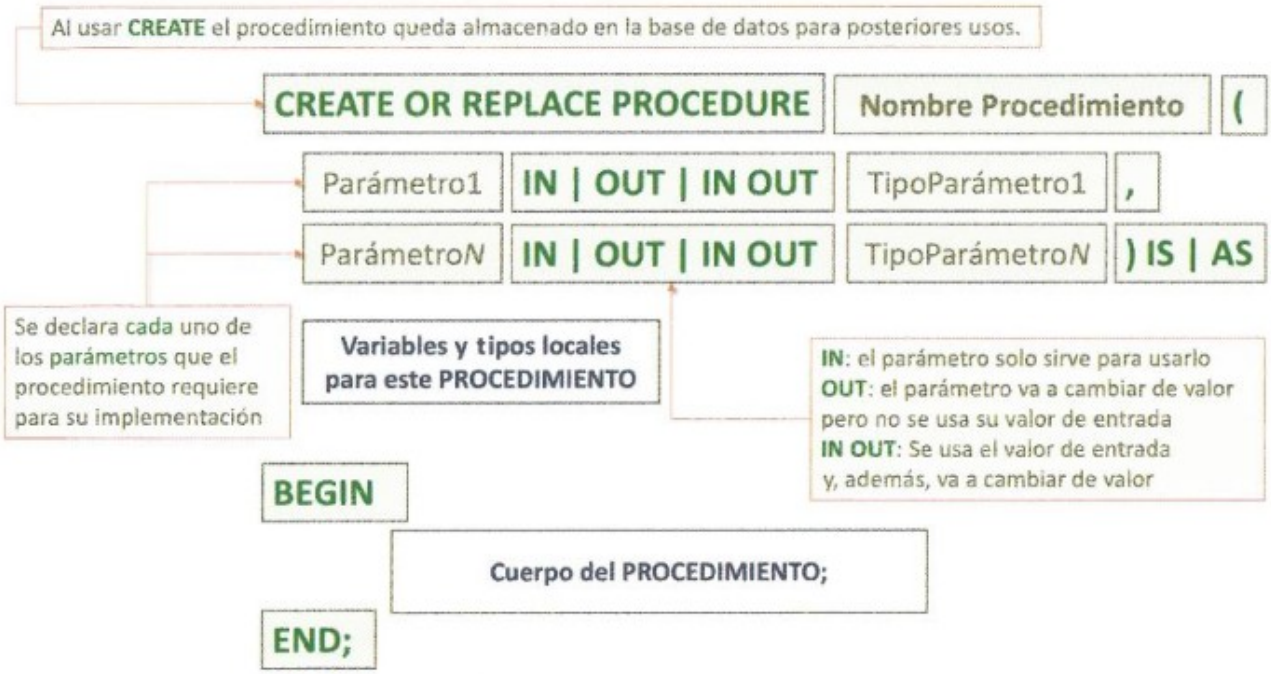
- **Funciones** : Siempre devuelven un valor, su llamada se sustituye por el valor de retorno. Se usan para calculos sencillos en los que hay que devolver un valor (sólo uno)
- **Procedimientos** : Se usan para “hacer algo” pueden devolver varios valores pero en variables que también se pasan al procedimiento para que deje ahí los valores a devolver.

Procedimientos

Un procedimiento engloba un conjunto de comandos con un nombre con el fin de que el código que incluye se pueda ejecutar desde cualquier lugar del mismo programa o desde otro, simplemente ¡nvocándolo por su nombre. Estos procedimientos, al igual que las funciones, pueden ser privados para un programa o públicos para cualquier programa que lo importe desde un paquete de procedimientos.

Para declarar un procedimiento se usa una sintaxis muy parecida a la de un bloque principal.

El nombre del procedimiento es obligatorio y es el que se usa para invocarlo desde otras partes del cuerpo del programa principal entre BEGIN y END. EI cuerpo del procedimiento también está limitado por las etiquetas BEGIN y END, y se le puede añadir el nombre del procedimiento a la etiqueta END con el fin de aclarar a qué bloque pertenece ese END. Los parámetros son opcionales. Cuando un procedimiento no tiene parámetro se denomina procedimiento desnudo. Estos están diseñados para evitar tener que escribir el mismo código cada vez que se necesita.



Los parámetros

La verdadera potencialidad de los procedimientos y las funciones es la capacidad que tienen de aceptar valores del programa principal. Estos valores que están almacenados en variables pueden ser pasados al procedimiento porque este necesita de ellos para cumplir con el objetivo que el diseño del procedimiento persigue. El objetivo es que el código que se repite para diferentes datos se recoja en un bloque que puede invocarse desde cualquier lugar. Los valores que necesita el subprograma se pasan como argumento. Por ejemplo, el procedimiento ¡mprimirA/argarCadena imprime a razón de un carácter por linea cualquier cadena que se le pasa como argumento. Asi, este procedimiento se podria usar en cualquier programa y tantas veces como se necesite. El siguiente ejemplo usa este procedimiento definido por el programador:

```
SET SERVEROUTPUT ON
PROCEDURE imprimirAlargarCadena(p_cadena IN varchar2) IS
BEGIN
        FOR i IN 1..LENGTH(p_cadena) LOOP
                DBMS_OUTPUT.PUT (SUBSTR(p_cadena, i, 1) II' '));
        END LOOP;
        DBMS_OUTPUT.PUTLINE();
END imprimirAlargarCadena;

DECLARE
BEGIN
        imprimirAlargarCadena('Hola');
        imprimirAlargarCadena(' mundo);
END;
```

Cuando el control de flujo de la ejecución del bloque llega al procedimiento imprimirAlargarCadena, salta al lugar donde está implementado, lo ejecuta e imprime la cadena alargada. Cuando termina, vuelve al flujo del programa principal. El procedimiento se usa tantas veces como se quiera.

Para indicar si un parámetro solo se requiere por la información que lleva, se usa la opción IN. Si solo es para cambiar su valor, se declara como OUT, y si es para ambas acciones, se emplea IN OUT. Al parámetro OUT sólo se le puede pasar una variable.

Los parámetros IN no se pueden modificar en el transcurso del programa.

Parámetro [IN | OUT | IN OUT] tipoDato [{:= | DEFAULT} valor]

La declaración es muy parecida a la que se hace en el bloque DECLARE. Estos parámetros solo tendrán validez dentro del código del subprograma y se inicializan cuando se invocan.
Para llamar a un procedimiento pasando valores a los parámetros podemos usar:

- Notación posicional. Especificamos todos los parámetros en el mismo orden en elq que están declarados en el procedimiento.
- Notación nominal. Especificamos el nombre de cada parámetro junto con su valor separando con una flacha (=>). Lo bueno de esta notación es que si los parámetros cambian nuestra llamada sigue funcionando.
- Notación mixta. Especificamos los primeros parámetros con la notación posicional y luego con la nominal. Usamos esta notación para procedimientos que tinen parámetros obligatorios (los primeros) y opcionales (los últimos)

```
DECLARE
        vOrganizacion CONSTANT VARCHAR2(30):='adeide';
        vCorreo empleado.correo%TYPE;
        vAp1 empleado.prApellido%TYPE;
        vAp2 empleado.sgApellido%TYPE;
PROCEDURE crearCorreo(
        parte1 IN empleado.prApellido%TYPE,
        parte2 IN empleado.sgApellido%TYPE,
        correo OUT empleado.correo%TYPE)
IS
BEGIN
        Correo:=parte1 || '.' || parte2 || '@' || vOrganizacion || '.com';
END p_CrearCorreo;
BEGIN
        crearCorreo('&prApellido', '&sgApellido', vCorreo);
        DBMS.PUT_LINE('El correo de ' || prApellido || ' ' || sgApellido || ' es ' || vCorreo);
END;
```

En el siguiente ejemplo se declara un procedimiento que toma dos valores de entrada y construye el correo corporativo almacenándolo en el tercer argumento. Por tanto, los dos primeros son parámetros de entrada, ya que solo se necesita su contenido. Sin embargo, el tercer parámetro es de salida, pues va a cambiar su contenido después de la ejecución del procedimiento.

Para invocar un procedimiento podemos
- Llamarlo en un bloque begin.. end
- usar execute y el nombre del procedimiento.

Por ejemplo estos dos casos son equivalentes:

SQL> execute doble(1,2);
SQL> begin doble(1,2); end; /

PRÁCTICA PROPUESTA

Procedimientos

Para la base de datos HR crea un procedimiento pl/sql que imprima por pantalla el nombre y apellido del trabajador cuyo ID pasas por parámetro.

PRÁCTICA PROPUESTA

Procedimientos

Modifica el procedimiento anterior para que deje en una variable de salida el sueldo del trabajador además de imprimir el nombre por pantalla.

PRÁCTICA PROPUESTA

Procedimientos

Crea un procedimiento PL/SQL que imprima los 5 primeros trabajadores llamando al procedimiento anterior y luego imprima la suma de sus salarios con el formato: "suma de salarios : " suma

PRÁCTICA PROPUESTA

Procedimientos

Con la base de datos Academia realiza un procedimiento que dado un DNI y un estado, cambie del alumno con dicho DNI su columna B que hace referencia a si el alumno es bilingüe o no.

Funciones

Las funciones son muy similares a los procedimientos, con la salvedad de que devuelven un valor tras la ejecución del código que contienen. Para ello, se usa RETURN.

Esta palabra reservada aparece dos veces en el código de una función. La primera para declarar el tipo de datos del valor que se devuelve, y la segunda, junto a una expresión, para devolver el valor deseado. Cuando el flujo de la ejecución encuentra el segundo RETURN vuelve al programa que invocó la función con el nuevo valor.

El comando RETURN se debe colocar al final de la función. Cuando el valor que se debe retornar depende de diferentes situaciones, no es recomendable usar diferentes comandos RETURN, sino asignar cada valor a una variable para cada situación y luego retornar esa variable.

El valor que devuelve la función se sustituye por la llamada a la función en el mismo lugar donde se encuentra. Es decir, si se ha implementado una función que devuelve el triple de un número y se invoca en num:=triple(5); es como haber puesto num:=15; pues 15 es el valor que retorna la función.

Así pues la función nunca se invoca sola o con execute, al devolver un valor hay que hacer algo con el. Así pues podemos:

- Invocar a la función asignando su valor de retorno a una variable
  ej. Num:=triple(5);
- Invocar a la función en un select
  eje. Select nombre,triple(salario) from empleados;
- Invocar a la función en la llamada a otra función o procedimiento (pasando su valor por parámetro)
  ej. muestraSalario(triple(5));

Visibilidad

Dependiendo del lugar donde se declara cualquier identificador en un programa PL/SQL, este será visible solo para el bloque PL/SQL, para el subprograma en cuestión o para toda la base de datos.

Un identificador declarado en la sección DECLARE es visible solo en el bloque PL/SQL en cuestión. Los subprogramas invocados desde este no conocen su existencia, a menos que se pase como argumento su valor al parámetro declarado en la definición del subprograma.

Estos son los identificadores locales. Con relación a esto, los subprogramas también pueden ser locales o globales. Los globales se almacenan en la base de datos y son accesibles desde cualquier bloque PL/SQL. Los locales se declaran en la sección declarativa de un bloque PL/SQL y, por tanto, solo son accesibles desde este bloque. Así, las reglas de ámbito y visibilidad de los subprogramas son exactamente iguales al resto de los identificadores.

Ahora bien, los subprogramas se declaran al final de la sección DECLARE de cualquier bloque.

Tratamiento del resultado de una consulta

Los valores que devuelven las consultas pueden ser almacenados en variables para realizar operaciones con ellos. En PL/SQL, para volcar el resultado de una consulta, se usa SELECT INTO precedido de las columnas que se han proyectado y seguido de las variables en las que almacenar los valores; se debe tener en cuenta el número de datos que la consulta devuelve. Esto es importante, ya que la estructura de datos que se usa después de la palabra INTO debe ser capaz de almacenar el conjunto de valores que devuelve la consulta.

La consulta devuelve un único valor simple

En estos casos, la variable que se usará después de la palabra INTO será una variable simple del tipo de datos capaz de guardar el valor que devuelve la consulta.

Supóngase que se quiere conocer el número de alumnos que se han matriculado en el curso correspondiente al año actual. La consulta siempre devuelve un único valor, por lo que en la estructura que se usará después de la palabra INTO es suficiente una variable de tipo numérico, como se muestre en el ejemplo siguiente:

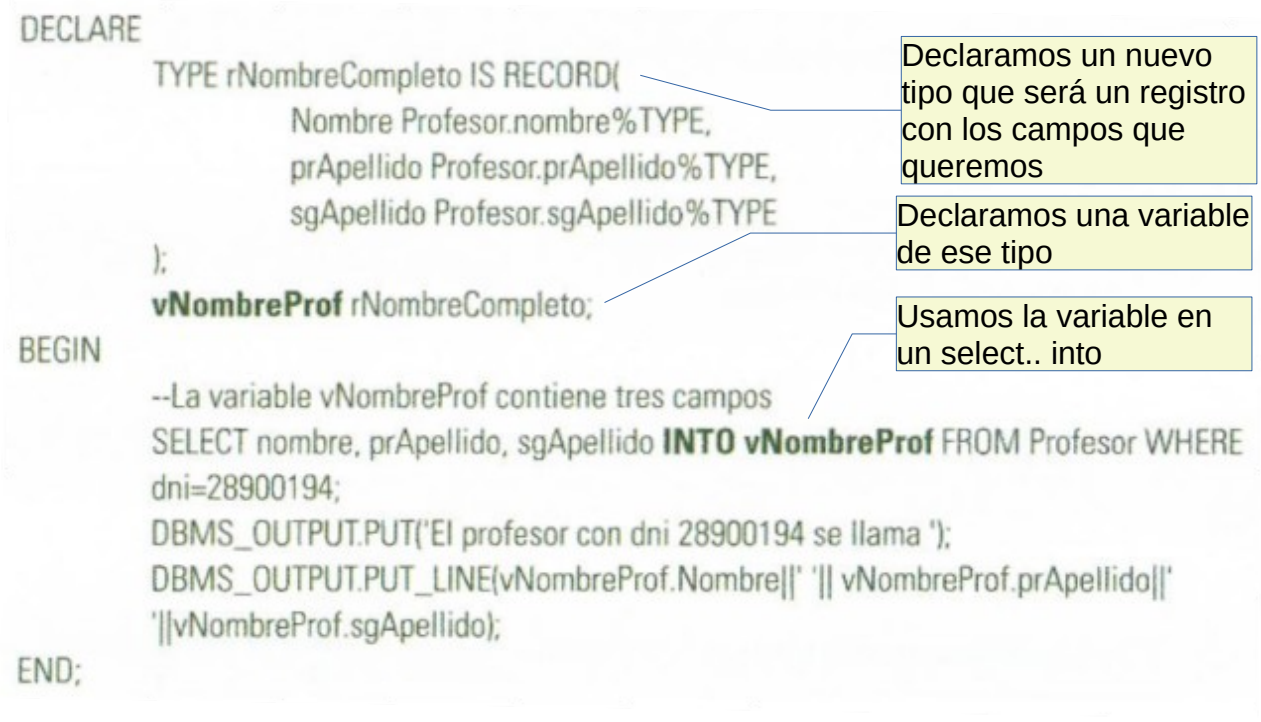
Las funciones de agregación siempre devuelven un valor y solo devuelven uno.

```
SET SERVEROUTPUT ON
DECLARE
        vNumAlumno PLS_INTEGER;
BEGIN
        SELECT COUNT(*) INTO vNumAlumno
        FROM Matricula
        WHERE curso=TO_NUMBER(TO_CHAR(SYSDATE,'YYYY'));
        DBMS_OUTPUT.PUT_LINE('En el año actual se han matriculado '||vNumAlumno||' alumnos.');
END;
```

La consulta devuelve un valor compuesto (una fila con varios campos)

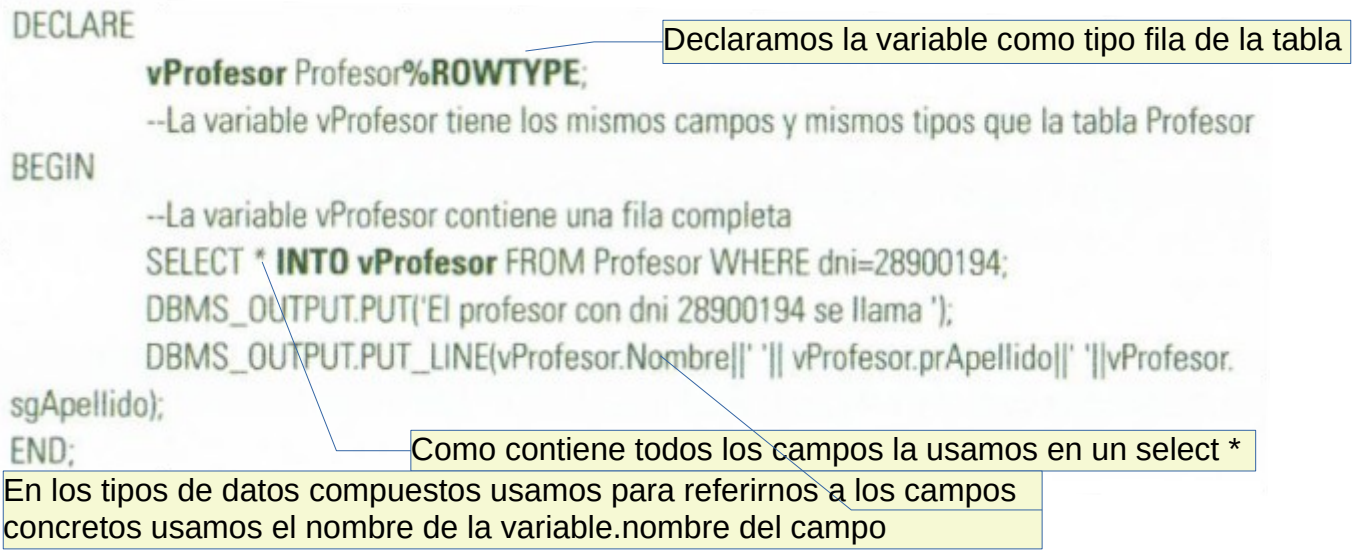
Si se sabe que la consulta que se va a ejecutar siempre devuelve una única fila, la variable que se necesita es la que tenga tantos campos como los que devuelve la consulta o tantas variables como campos seleccionemos.. Para ello, se usa el tipo de dato denominado RECORD (registro). Si un RECORD tiene todos los campos de la tabla no es necesario crearlo. Para ello, se usa el operador **%ROWTYPE**. Para usarlo ponemos al declarar la variable Tabla%ROWTYPE

La sintaxis para declarar una variable de tipo RECORD con campos es la siguiente



Ojo, un select... into sólo puede devolver un valor y siempre tiene que devolver uno, si no es así dará error.

El siguiente ejemplo utiliza el operador ROWTYPE con el fin de crear una variable que tiene todos los campos de la tabla:



PRÁCTICA PROPUESTA

Tienda de informática

Realiza un programa que muestre por pantalla la media del precio de venta de los productos de tipo C. La salida por pantalla será "La media del precio de venta de productos de tipo C es " *media*.

Elabora un programa que muestre por pantalla el nombre comercial de producto que cuesta más caro con el formato El producto más caro es NombreComercíal.

Lleva a cabo un programa que muestre por pantalla el DNI del cliente Elena García Sánchez con el formato El dni de Elena García Sánchez es dni. ¿Qué ocurre con la consulta de este programa?

Diseña un programa que muestre por pantalla cuántas compras se le ha hecho al proveedor 5 con el formato Al proveedor NombreProv se le ha hecho cantidad compras.

PRÁCTICA PROPUESTA

Academia

Realiza una una función llamada apruebaTodo que, dado el DNI de un alumno, devuelva TRUE si todas las asignaturas en las que se ha matriculado tiene una nota igual o superior a Cinco.
Realiza una función que devuelva el número de horas que un alumno tiene suspensas.
Realiza un procedimiento al que pasándole un DNI muestre si el alumno pasa de curso o no y si titula o no.

La consulta devuelve más de una fila. CURSORES

Cuando una consulta devuelve más de una fila, se recomienda el uso de los cursores. Los cursores permiten almacenar en una variable de tipo registro toda la descripción de una tabla, que se llamará registro completo, toda una fila para su tratamiento. Una vez que se han hecho las operaciones oportunas con ese registro, el cursor salta a la siguiente fila volcándola en la misma variable.

Otra alternativa es almacenar en una tabla todas las filas que devuelve una consulta. Esto requiere mucha más memoria principal que un cursor, por lo que se debe usar solo cuando sea necesario tener en memoria todas esas filas.

La sintaxis para declarar una tabla es la siguiente **TYPE tNombreTabla IS TABLE OF tipoDatos INDEX BY BINARY_INTEGER;**. Para trabajar con las tablas, es muy habitual usarla salida de una consulta SELECT que devuelve más de una fila y se almacene en una colección.

Para ello, se utiliza **BULK COLLECT** con la sintaxis SELECT columnas BULK COLLECT INTO colección FROM restoConsulta

Un cursor es una herramienta muy potente y permite acceder al proceso de ejecución de un comando SQL. A través de un cursor se puede, por ejemplo, recorrer las filas de una tabla, o las filas de una consulta y ejecutar órdenes nuevas para cada una de ellas.

El procesamiento de un cursor pasa por cuatro fases, que son:

- **declaración** : **cursor <nombre> is select...**
- **apertura** : **open <nombre>;**
- **almacenamiento** : **fetch <nombre> into <variable>;**
- **cierre** : **close nombre;**

El cursor declara un registro (RECORD) implícito para cada fila del select así pues estos dos códigos son equivalentes:

```
DECLARE
   TYPE rNombreCompleto IS RECORD(
      Nombre Profesor.nombre%TYPE,
      prApellido Profesor.prApellido%TYPE,
      sgApellido Profesor.sgApellido%TYPE
   );
   vNombreProf rNombreCompleto;

BEGIN
   SELECT nombre, prApellido, sgApellido INTO vNombreProf FROM
Profesor WHERE dni=28900194;
   DBMS_OUTPUT.PUT('El profesor con dni 28900194 se llama ');
   DBMS_OUTPUT.PUT_LINE(vNombreProf.Nombrell' 'll
vNombreProfprApellido ll' '|l vNombreProfsgApellido);
END;
```

```
DECLARE
   cursor c_profe is SELECT nombre, prApellido, sgApellido  FROM
Profesor WHERE dni=28900194;
   v_profe c_profe%ROWTYPE;

BEGIN
   open c_profe;
   fetch c_profe into v_profe;
   DBMS_OUTPUT.PUT('El profesor con dni 28900194 se llama ');
   DBMS_OUTPUT.PUT_LINE(v_profe.Nombrell' 'll v_profe.prApelido ll' '|l
v_profe.sgApellido);
   close c_profe;
END;
```

Atributos de cursor

Cada vez que se ejecuta una sentencia se llenan una serie de atributos de cursor ya que Oracle usa cursores implícitos. Así pues:

SQL%ROWCOUNT Contiene el número de filas que se devolvieron en la consulta anterior.

SQL%FOUND y **SQL%NOTFOUND** será verdadero o falso si alguna fila fue insertada, borrada, actualizada o seleccionada en la sentencia anterior.

Si en lugar de SQL ponemos el nombre de un cursor, nos dará el valor verdadero o falso en cada fetch.

Pero el cursor tiene mucha potencia para tratar selects que devuelven más de una fila y procesarlas con un bucle.

Cursores parametrizados

Los cursores parametrizados permiten trabajar con diferentes argumentos del mismo modo que los procedimientos y su sintaxis es análoga a la de estos.

CURSOR nombreCursor<parámetro1 tipo1, parametro2 tipo2....) IS Consulta;

Los argumentos se pasan cuando se abre el cursor, es decir, en el comando OPEN o al usar un FOR. Se usan sobretodo para reutilizar los cursores o en bucles anidados.

Bucle simple

```
DECLARE
   cursor c_profe is SELECT * FROM Profesor;
   v_profe c_profe%ROWTYPE;
BEGIN
   open c_profe;
   loop
      fetch c_profe into v_profe;
      exit when c_profe%NOTFOUND;
      DBMS_OUTPUT.PUT('El profesor con dni 28900194 se llama ');
      DBMS_OUTPUT.PUT_LINE(v_profe.Nombrell' 'll v_profe.prApelido ll' '|l v_profe.sgApellido);
   end loop;
   close c_profe;
END;
```

PRÁCTICA PROPUESTA

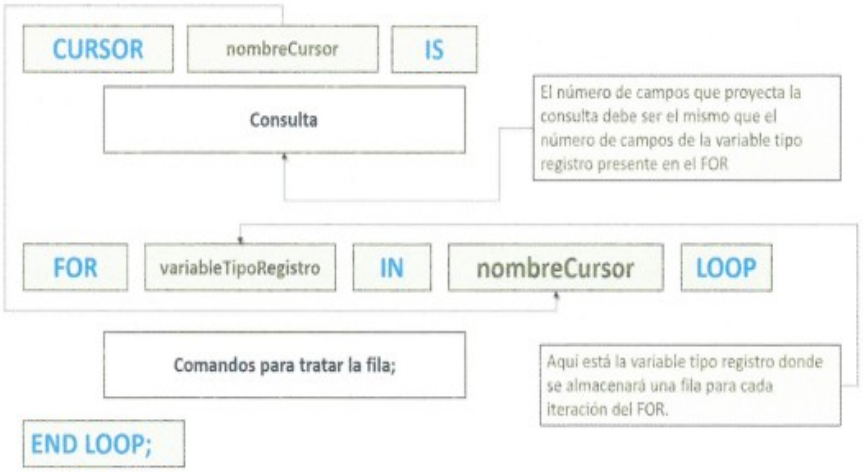
Cursores

Crea un procedimiento en HR que imprima todos los nombres de los trabajadores del departamento_id pasado por parámetro. Hazlo con bucle simple y bucle While usando %FOUND en lugar de %NOTFOUND, recuerda que estos atributos tienen valor luego del fetch.

FOR IN CURSOR

Viendo los ejemplos anteriores parece que es bastante farragoso estar abriendo cursores, asignar cada fila a una variable, cerrando... Además puede suceder que queden cursores abiertos. Para evitarlo podemos usar un FOR especial que:

a) No hay que declarar la variable que contendrá cada tupla del cursor
b) No hay que abrir, hacer fetch ni cerrar el cursor



Cuando usamos el bucle FOR no hace falta que declaremos la variable que usaremos.

Si queremos declarar la variable lo haremos como %ROWTYPE del cursor.

```
DECLARE
   cursor c_profe is SELECT * FROM Profesor;
BEGIN
   for v_profe in c_profe loop
      DBMS_OUTPUT.PUT('El profesor con dni 28900194 se llama ‘);
      DBMS_OUTPUT.PUT_LINE(v_profe.Nombrell' ‘ll v_profe.prApelido ll‘ ’|l v_profe.sgApellido);
   end loop;
 END;
```

Ojo, el bucle for recorre todo el cursor. Si necesito salir del bucle antes hacer exit when

FOR in Query

Podemos usar directamente la select en el for en lugar de declarar el cursor. Esto aunque parece más fácil es de lectura más engorrosa pues no se saben las selects que se hacen en el código a no ser que lo leamos todo.

```
BEGIN
   for v_profe in (select * from profesores) loop
      DBMS_OUTPUT.PUT('El profesor con dni 28900194 se llama ‘);
      DBMS_OUTPUT.PUT_LINE(v_profe.Nombrell' ‘ll v_profe.prApelido ll‘ ’|l v_profe.sgApellido);
   end loop;
 END;
```

PRÁCTICA PROPUESTA

Cursores II

Con la tienda de informática
a) Muestra la cantidad de productos que hay de cada tipo, donde las posibilidades son C, P, F, S y D. El formato de salida será Hay cantidad productos del tipo tipo.
b) Otro que muestre el nombre y apellidos de los clientes que son de Sevilla, de Madrid, de Barcelona, de Valencia y de Zaragoza, con el formato El cliente Nombre PRAPELLIDO SGAPELLIDO es de Ciudad. A continuación, muestra cuántos hay de cada ciudad de esas.
c) Otro que muestre el número de productos diferentes para cada venta realizada por un empleado que no sea de la misma ciudad que el cliente. El formato de salida será Para la venta con código código se ha vendido cantidad productos. El cliente es de ciudadcliente y el vendedor es de ciudadVendedor.

PRÁCTICA PROPUESTA

EJERCICIOS DE REPASO

Con la base de datos de HR
a) Haz un procedimiento que **calcule la media de los salarios de un departamento pasado por parámetro.** Si la media del salario de ese departamento es menor que la media total de todos los salarios debe mostrar “Departamento con salarios baijos” y “Departamento sobrepagado” en caso contrario.

b)Usando la función MOD(a,b) que devuelve el resto de la división entre a y b. Escribe un procedimiento PL/SQL que diga si un número es par o impar. Hazlo con un CASE y con un SEARCHED CASE.

c) Haz un procedimiento PS/SQL que vaya contando en cada paso de un bucle simple y mostrando la cuenta por pantalla. Debe contar hasta llegar al número máximo que el usuario debe pasarle como parámetro.

d) Haz una función PL/SQL que devuelva la suma de los n primeros números naturales (n es pasado por parámetro) usando un bucle simple y luego otro usando un WHILE, por último un tercero usando FOR

e) Haz unha función que calcule el factorial de un número. Haz luego un procedimiento PL/SQL que usando FOR anidados calcule el factorial de los n primeros números (n pasado por parámetro). Haz otro procedimiento que haga lo mismo pero usando la primera función que hiciste.

f) Usando la BBDD de HR haz un bloque PL/SQL que devuelva los 5 empleados que más cobran. Hazlo de tres maneras:
a) Usando cursores con un LOOP
b) Usando cursores con un FOR in CURSOR
c) Usando sólo FOR... in SELECT

La consulta devuelve más de una fila. COLECCIONES.

Cuando una consulta devuelve más de una fila, hemos visto que se puede declarar un cursor y recorrerlo, pero también se pueden almacenar todas ellas en memoria, si se prevé que es posible y que aporta ventajas. Volcar todas las filas a una estructura en memoria principal requiere de mucho espacio, asi que este tipo de acción se hará cuando realmente sea imprescindible tener todas las filas a la vez. Estos datos se pueden almacenar en tablas usando el tipo de datos Table o en listas usando el tipo Varray.

Colecciones de tipo TABLE

El tipo de datos table permite crear una pila de elementos del mismo tipo. La sintaxis para definir este tipo es:

TYPE nombre IS TABLE OF tipoElementos INDEX BY {BINARY_INTEGER | PLS_INTGER | VARCHAR2};

Los elementos pueden ser desde un tipo numérico hasta un registro con n valores. El índice, con INDEX BY, que acelera los accesos, generalmente es de tipo numérico, BINARY INTEGER o PLS_INTEGER, pero también se puede usar alguno de tipo cadena. EI siguiente ejemplo es la definición del tipo de datos t_dniProf compuesto por una pila de elementos de tipo NUMBER(8):

TYPE t_dniProf IS TABLE OF NUMBER(8) INDEX BY BINARY_INTEGER;

El siguiente ejemplo es equivalente, pero usa el tipo de datos declarado en la tabla:

TYPE t_dniProf IS TABLE OF Profesor.dni%TYPE INDEX BY BINARY_INTEGER;

Pero también se puede declarar una lista de valores del tipo registro, por ejemplo

TYPE t_Profesor IS TABLE OF rProfesor;

O incluso del tipo registro fila de la tabla Alumno:

TYPE t_AIumno IS TABLE OF Alumno%ROWTYPE;

Las variables de tipo colección tienen asociados unos atributos que permiten realizar ciertas operaciones o conocer ciertos valores, y son los siguientes:

- count: devuelve el número de elementos de la colección.
- delete: borra todos los elementos de la colección.
- delete(*posición*): elimina el elemento de la posición que se indica entre paréntesis.
- delete(*posic_ini, posic_fin*): elimina los elementos que se encuentran entre la posición *posic_ini* y la posición *posic_fin*.
- exists(*posición*): devuelve TRUE si en la posición entre paréntesis hay un elemento. En caso contrario devuelve FALSE.
- first: devuelve la posición donde se encuentra el primer elemento de la colección.
- last: devuelve la posición donde se encuentra el último elemento de la colección.
- next(*posición*): devuelve la posición del elemento siguiente a la posición que se indica entre paréntesis que contiene un elemento no nulo de la colección.
- prior(*posición*): funcional igual que next, pero esta vez devuelve la posición del elemento anterior.

Para volcar el contenido de una consulta a una tabla se usa la cláusula BULK COLLECT en el comando SELECT. La sintaxis de este SELECT es la siguiente:

SELECT columnas BULK COLLECT INTO colección FROM restoConsulta;

La variable con la que trabaja BULK siempre debe ser de tipo colección y se usa tanto en consultas SELECT como en comandos FETCH de los cursores. La sintaxis para la asignación de valores es variable(posición):=valor.

El siguiente ejemplo es un programa PL/SQL en el que se define el tipo de datos trAsignatura que contiene elementos de tipo registro con las columnas Nombre y NH de la tabla Asignatura. Se declara una variable de este tipo denominada vtAsignatura (variable de la tabla Asignatura). En esta se va a almacenar las asignaturas y número de horas que son bilingües y más tarde se mostrará cuál de estas en la que más horas tiene.

```
DECLARE
  TYPE rAsigBil IS RECORD(
    Nombre VARCHAR2(50),
    NH NUMBER(3,0),
    B CHAR(1)
  );
  TYPE trAsignatura IS TABLE OF rAsigBil INDEX BY BINARY_INTEGER;
  vtAsignatura trAsignatura;
  vrAsigMaxHora rAsigBil;

BEGIN
  vrAsigMaxHora.NH:=0;

  SELECT Nombre, NH, B BULK COLLECT INTO vtAsignatura FROM Asignatura WHERE B='S';
  IF vtASignatura.LAST>0 THEN
  DBMS_OUTPUT.PUT_LINE('De las asignaturas binlingües:');
  FOR i IN 1..vtAsignatura.LAST LOOP
    IF vtAsignatura(i).NH>vrASigMaxHora.NH THEN
      vrAsigMaxHora.NH:=vtASignatura(i).NH;
      vrAsigMaxHora.nombre:=vtASignatura(i).nombre;
    END IF;
    DBMS_OUTPUT.PUT_LINE(vtAsignatura(i).Nombre||' con '||vtAsignatura(i).NH||' horas');

  END LOOP;
  DBMS_OUTPUT.PUT_LINE('La de mayor hora es '||vrAsigMaxHora.nombre||' con '||vrAsigMaxHora.
  NH||' horas.');
  ELSE
  DBMS_OUTPUT.PUT_LINE('La tabla Asignatura está vacía.');
  END IF;
END;
```

Colecciones tipo VARRAY

Las colecciones que usan el tipo de datos vArray tienen muchas similitudes con las que usan Table. En los tipos vArray su primer elemento siempre comienza en la posición 1 y su longitud máxima se declara al definir el tipo. En este tipo de colecciones no se emplean algunos tipos de datos como Boolean, nChar, nVarchar, nClob, ref cursor, table y vArray. La sintaxis es parecida a la de IS TABLE OF, pero en esta se indica el tamaño máximo:

TYPE nombre IS VARRAY(tamaño) OF tipoElementos;

Estas estructuras deben estar inicializadas antes de usarlas, ya sea en la sección DECLARE o en un bloque. Se pueden incluir valores null al final de la colección no null con el objetivo de otorgar valores posteriormente. La colección no null, es decir, aquella que contiene valores inicializados distintos de null, no puede nunca sobrepasar el limite definido en la declaración de la colección. Además de los atributos indicados para las colecciones de tipo tabla, el tipo vArray tiene estas otros:

extend: añade elemento null final de colección no null.
extend(número): añade tantos null como indique número entre paréntesis.
extend(copia,pasición): añade copias del elemento posición final de colección no null.
limit: devuelve número máximo de elementos con que se declaró.

Para inicializar los elementos de un vArray se usa su método constructor. El siguiente es un ejemplo del empleo de este modo:

```
TYPE tNumDiaMeS IS VARRAY(2) OF BINARY_INTEGER;
vTabla tNumDiaMes := tNumDiaMes(31,28,31,30,31,30,31,31,30,31,30,31);
```

El ejemplo siguiente usa el método extend para añadir un elemento nuevo a la variable de tipo varray. Primero se declara el tipo tCiudad como un array de 52 cadenas de longitud máxima de 50 caracteres. El array vCiudades contendrá cada una de las ciudades contenidas en una supuesta tabla Ciudad(P-idCiudad, ciudad) con el nombre de 52 ciudades. Para inicializar el array se usa el constructor con el comando vCiudades:=tCiudad().

```
DECLARE
        TYPE tCiudad IS vArray(52) OF VARCHAR2(50);
        vCiudades tCiudad;
        vContador NUMBER:=0;
        vNumCiudades NUMBER;
BEGIN
        vCiudades:=tCiudad();
        SELECT COUNT(*) INTO vNumCiudades FROM Ciudad;
        FOR i IN 1..vNumCiudades LOOP
                SELECT ciudad INTO vCiudad FROM Ciudad WHERE idCiudad=i;
                vContador=vContador+1;
                vCiudades.extend;
                vCiudades(vContador):=vCiudad;
                DBMS_OUTPUT.PUT_LINE('Se ha cargado la ciudad '||vCiudad||' en la posición '||i);
        END LOOP;
END;
```

Excepciones

La gestión de excepciones es un proceso muy importante en la programación. Cuando en un programa se pretenden realizar acciones sobre la base de datos, estas pueden estar sujetas a numerosos errores. El motor de PL/SQL puede comunicar los errores que son provocados a lo largo de Ia traducción de los comandos, atendiendo a la entrada y salida de los valores procesados. Estos errores tienen un número único para cada uno de ellos y se pueden tomar decisiones alternativas con el fin de evitar que el programa sea abortado. Esto se consigue con el bloque excepción de cualquier programa o subprograma.

Los errores serán procesados en este bloque, de tal modo que se llevarán acciones en consecuencia del error. Este mecanismo procesamiento de errores permite aislar Ia gestión de errores del programa en su conjunto, consiguiendo aclarar su lógica. Cuando no se usan excepciones, se hace necesario controlar los posibles errores comando a comando, con el fin de que el programa no aborte sin que estos sean procesados.

Los errores pueden ser tanto de tipo SQL, como errores procedimentales relacionados con los procesos del sublenguaje PL, y se generan en cualquier parte de un bloque. Los errores procesables de dicho bloque se resolverán en el propio bloque de excepciones de él y nunca de otro.

Declaración del bloque de excepciones

En cada programa o subprograma, se declara la sección de excepciones que las resuelven, ya sean definidas por el usuario o preestablecidas por el propio lenguaje. Estas últimas son errores asociados a los comandos SQL y PL. La sintaxis de la sección de excepciones es la siguiente:

```
EXCEPTION
        WHEN nombreExcepcion1 THEN
                TratramientoErrorTipo1;
        [WHEN nombreExcepcion2 THEN
                TratamientoErrorTipo2;]*
        [WHEN OTHER THEN
                TratamientoRestoErrores;
```

Se podrán usar tantas cláusulas WHEN como sean necesarias. La opción WHEN OTHERS THEN hace referencia al resto de errores posibles, unificando su tratamiento en esta subsección.

Las excepciones internas están establecidas por Oracle y se disparan automáticamente, saltando Ia ejecución del programa a la sección EXCEPTION. Una vez procesado que se ha tratado el error, la ejecución del programa finaliza. Es por ello que esta sección se de clara al final del programa. Los errores más comunes son los que se definen en Ia tabla siguiente.

El programador puede definir sus propias excepciones. Para ello se declaran en la sección DECLARE usando la palabra EXCEPTION. En la sección del bloque principal se disparan con RAISE y se tratan en la sección EXCEPTION igual que las internas.

En esta tabla, se pueden observar los códigos que Oracle les ha asignado tanto para procesos SQL como procesos Oracle y el nombre de la excepción.

Excepción	Descripción	Código
ACCESS_INTO_NULL	El objeto al que se quiere acceder no está inicializado	-6530
CASE_NOT_FOUND	El valor del *case* no está en la enumeración	-6592
COLLECTION_IS_NULL	La colección está vacía, por lo que no se puede acceder a ella	-6531
CURSOR_ALREADY_OPEN	El cursor se intenta abrir y ya está abierto	-6511
DUP_VAL_ON_INDEX	El valor que se pretende insertar crea duplicados en las columnas UNIQUE	-1
INVALID_CURSOR	Acción no permitida sobre un cursor	-1001
INVALID_NUMBER	Error al transformar cadenas a números	-1722
LOGIN_DENIED	Error de autenticación, ya sea por el nombre del usuario o por su contraseña	-1017
NO_DATA_FOUND	Una consulta SELECT INTO no devuelve ninguna fila	+100 o -1403
NOT_LOGGED_ON	No se está conectado aún a la base de datos	-1012
PROGRAM_ERROR	Error interno	-6501
ROWTYPE_MISMATCH	La variable del cursor no es del mismo tipo que el usado en el programa	-6504
STORAGE_ERROR	Errores de acceso a memoria	-6500
SUBSCRIPT_BEYOND_COUNT	Se intenta acceder a una posición de una tabla anidada o una colección que está fuera de su tamaño actual	-6533
SUBSCRIPT_OUTSIDE_LIMIT	Se intenta acceder a una posición de una tabla anidada o una colección fuera del rango declarado	-6532
TIMEOUT_ON_RESOURCE	Tiempo de espera para un recurso agotado	-51
TOO_MANY_ROWS	La consulta SELECT INTO ha devuelto más de una fila	-1422
VALUE_ERROR	Error con valores numéricos	-6502
ZERO_DIVIDE	Se intenta dividir entre 0	-1476

WHEN OTHERS

A través de las funciones SQLCODE y SQLERRM, Error Message, se pueden obtener el código y el mensaje de un error. El mejor lugar para usarlas es en la opción WHEN OTHERS de la sección EXCEPTION. Los valores devueltos por estas funciones pueden almacenarse en variables para su posterior uso en comandos sqlplus.

PASO DE EXCEPCIONES PROPIAS

Las excepciones no capturadas en un programa son pasadas al programa que lo llamó, si este no lo captura se le pasará al programa que llamó a este y así sucesivamente.

Esto es válido para excepciones internas, ya que tienen nombre. Pero no es así con las propias ya que sólo tienen nombre en el programa en el que son declaradas. Para lanzar excepciones entre subprogramas hay que usar RAISE_APPLICATION_ERROR y lanzar una excepción con un error ORA (Oracle deja los números ORA entre 20000 y 20999 para nuestros propios errores) La sintaxis será:

raise_application_error(error_ora, mensaje de error);

ejemplo : raise_application_error(-20001,'El usuario no existe');

Para capturarlas luego en el programa llamante hay que asociar a ese error ORA un nombre.

Se puede asociar una excepción al código de error mediante una directiva PRAGMA EXCEPTION_INIT. La sintaxis de esta directiva de tipo pragma es

PRAGMA EXCEPTION_INIT(nombre, numeroError);.

En la sección DECLARE, se declara la excepción de usuario y la directiva EXCEPTION_INIT. El siguiente ejemplo aclara este uso:

```
DECLARE
        mierror EXCEPTION;
        PRAGMA EXCEPTION_INIT(mierror, -20001);
BEGIN
EXCEPTION
        WHEN mierror then
```

Podemos encontrar todos los errores ora aquí:

http://www.oracle.com/pls/db92/db92.error_search?prefill=ORA-

Por ejemplo unas muy usadas son las -2291, -2292 y -1400 que se refieren a foreign keys y nulos respectivamente.

Depuración de procedimientos

La siguiente instrucción mostrará los errores de compilación de los procedimientos:

SHOW ERRORS [[PROCEDURE] <nome>];

Paquetes

Un paquete es una estructura que guarda definiciones de variables, procedementos y funciones de forma similar a un objeto. El paquete es tratado como un todo y por tanto no se pueden dar permisos a un procedimiento específico de un paquete si no al paquete completo.

```
CREATE [OR REPLACE] PACKAGE [schema.]package_name
{AS | IS}
        public_variable_declarations |
        public_exception_declarations |
        public_cursor_declarations |
        function_declarations |
        procedure_declarations
END [package_name];

CREATE [OR REPLACE] PACKAGE BODY [schema.]package_name
{AS | IS}
        private_variable_declarations |
        private_exception_declarations |
        private_cursor_declarations |
function_specifications | procedure_specifications
END [package_name];
```

Para ejecutar un procedimiento o función que esté en un paquete tenemos simplemente que utilizar el operador punto. Es decir, el nombre real del procedimiento será paquete.procedimiento(...)

PRÁCTICA PROPUESTA

Excepciones

Usando la base de datos de HR. Haz un procedimiento PL/SQL que devuelva el nombre y apellido de un empleado con el id pasado por parámetro. Hazlo de dos formas:
a) Con un select... into con excepciones
b) Con un cursor sin excepciones

Fallo transaccional

Cada vez que realizamos una sentencia DML Oracle pone un savepoint implícito antes de la sentencia de tal forma que si falla se produce un rollback hasta ese punto. Cuando se lanza una excepción y no es capturada y manejada dentro del procedimiento, Oracle hará un rollback completo y perderemos todas las transacciones hasta el último commit.

```
Create or replace procedure insire(id in number) as
begin
    insert into dept(deptno,dname) values(id,'un');
    insert into dept(deptno,dname) values(id,'outro');
exception when others then
    return;
end;
```

```
Create or replace procedure insire(id in number) as
begin
    insert into dept(deptno,dname) values(id,'un');
    insert into dept(deptno,dname) values(id,'outro');
end;
```

Viendo los dos procedimientos anteriores, que tendríamos que hacerle al segundo para que se comporte como el primero?

Transacciones autónomas

Cuando un procedimiento llama a otro y este último hace un commit está, no solo confirmando las transacciones del segundo procedimiento, sinó también las del primero. A veces sólo queremos que se confirmen las transacciones hechas en el subprocedimiento.

Las transaccións autónomas nos permten ejecutar un subprocedemento con independencia transaccional del procedemento principal. Para identificar un procedemento como autónomo usaremos PRAGMA AUTONOMOUS_TRANSACTION; en el bloque de definiciones.

Dados los dos procedimientos siguientes, serías capaz de explicar que sucede? Y si no tuviese la directiva pragma?

```
Create or replace procedure insire2(id in number) as
Pragma autonomous_transaction;
Begin
        Insert into dept(deptno,dname) values(id+1,'outro');
        Commit;
End;
```

```
Create or replace procedure insire(id in number) as
begin
        insert into dept(deptno,dname) values(id,'un');
        insire2(id);
        insert into dept(deptno,dname) values(id,'un');
end;
```

Triggers

Al igual que un procedimiento almacenado, un disparador es una unidad PL/SQL con nombre que se almacena en la base de datos y se puede invocar repetidamente. A diferencia de un procedimiento almacenado, puede habilitar y deshabilitar un desencadenador, pero no puedes invocarlo explícitamente.

Mientras un disparador está habilitado, la base de datos lo invoca automáticamente (es decir, el disparador se activa) cada vez que ocurre el evento desencadenante. Mientras un disparador está desactivado, no se dispara.

Programación de un disparadores

Las secciones de un disparador son su bloque de declaración DECLARE, el bloque ejecutable y el de manejo de excepciones para ese disparador. El objetivo de un disparador versa sobre el propio mantenimiento de la integridad, las auditorias sobre los cambios de la base de datos y la comunicación con otros programas sobre eventos ocurridos. La sintaxis de un disparador es la siguiente:

```
CREATE [OR REPLACE] TRIGGER disparador
[BEFORE|AFTER] evento ON tabla
[FOR EACH ROW|WHEN condición]]
bloque disparador;
```

Los triggers: No devuelven ningún valor ni admiten parámetros de entrada
No pueden facer sentencias contra la tabla que hace saltar el trigger de tipo FOR EACH ROW.
Las Foreign keys se retrasan hasta que se compruebe el trigger. No pueden tener sentencias DDL ni commit o rollback.

Para programar un disparador se debe tener en cuenta qué operación es la que se quiere controlar y cuando, si antes de llevarla a cabo o después. Las operaciones que se pueden testear son INSERT, DELETE y UPDATE y los tiempos se controlan con **AFTER** y **BEFORE**, como se muestra en el siguiente ejemplo:

```
AFTER INSERT ON Asignatura
```

También se puede definir un trigger que salte ante varios eventos de la forma siguiente:

```
AFTER INSERT OR UPDATE OR DELETE ON Asignatura
```

Si en lugar de querer controlar la situación después de la operación DML se desea controlar qué hacer antes, se usa **BEFORE**

Cuando se ejecuta un comando DML, este tiene una fase antes de su ejecución. Puede conllevar n acciones para un conjunto de filas y pasar a un estado final al finalizar. Los disparadores se pueden programar para cada una de estas fases: la acción anterior a la ejecución del comando, acciones para las filas, y la acción posterior a la ejecución del comando.

Cuando el evento es **UPDATE** podemos poner **OF columna** para indicar que el trigger salte sólo cuando actualizamos esa columna y no cualquier otra de la tabla

```
CREATE OR REPLACE TRIGGER CambioHorasAsignaturas
    BEFORE UPDATE OF NH ON Asignatura
    FOR EACH ROW
```

El orden de activación de los disparadores es, en primer lugar, la acción anterior (BEFORE) a la ejecución del comando. A continuación, para cada fila, se ejecuta su acción BEFORE, luego el comando de la fila y, más tarde, Ia acción posterior (AFTER) para la fila. Finalmente se ejecuta Ia acción AFTER para el comando.

Para los disparadores de control de filas se usa Ia cláusula **FOR EACH ROW;** el siguiente ejemplo seria la declaración de un trigger para controlar si se ejecuta Ia eliminación de cualquier fila en Ia tabla Asignatura:

Al programar un disparador para las filas de un comando INSERT, UPDATE o DELETE se ejecuta para cada fila. Para controlar el contenido que se pretende modificar, se usan dos registros virtuales llamados **:old y :new**. El primero, :old, contiene el valor actual sin ejecutar Ia acción asociada al disparador, y el segundo, :new, contiene el valor que correspondería tras la acción del comando.

```
CREATE OR REPLACE TRIGGER HistorialBorradoAsignatura
        BEFORE DELETE ON Asignatura
        FOR EACH ROW
BEGIN
        INSERT INTO HistorialAsignatura VALUES(:old.codAsig,:old.nombre,:old.NH,:old.B,:old.
CodCF);
END;
```

Si el trigger no es FOR EACH ROW, es decir, salta antes o despues del comando (**FOR EACH STATEMENT**) no existen los registros NEW y OLD. Tampoco existe :NEW en los DELETE ni :OLD en los INSERT

Control de más de un evento

Si en un mismo trigger se están programando acciones para más de una operación DML, es decir, INSERT, UPDATE o DELETE, se emplea el operador OR para indicarlas en la declaración. En estos casos, en el cuerpo del disparador se pueden usar predicados condicionales para la ejecución de diferentes acciones dependiendo del tipo de operación.

Estos predicados son INSERTING, UPDATING, UPDATING(columna) y DELETING. Estos predicados valdrán TRUE en caso de que la operación sea de inserción (INSERTING), de actualización (UPDATING) o de borrado (DELETING). El siguiente disparador muestra la acción del disparador antes de llevarla a cabo:

```
CREATE OR REPLACE TRIGGER BorrarEnAsignatura
        BEFORE INSERT OR UPDATE OR DELETE ON Asignatura
        FOR EACH STATEMENT
DECLARE
        vtexto VARCHAR2(50);
BEGIN
        IF INSERTING THEN
                vtexto:='insertar';
        ELSIF UPDATING THEN
                vtexto:='actualizar';
        ELSIF UPDATING('NH') THEN
                vtexto:= 'actualizar el número de horas en los ';
        ELSIF DELETING THEN
                vtexto:='borrar';
        ELSE
                vtexto:='hacer algo con los;
        END;
        DBMS_OUTPUT.PUT_LINE('Se procede a ' || vtexto || ' registro(s) en la tabla Asignatura');
END;
```

La cláusula WHEN se utiliza para disparadores que usan FOR EACH ROW, esto es, disparadores de filas. Si se quiere que solo se ejecute el disparador para aquellas filas que cumplan cierta condición, se hace uso de WHEN. Por ejemplo, el siguiente disparador salta para aquellas asignaturas cuyo número de horas es mayor que 250:

```
CREATE OR REPLACE TRIGGER AsignaturasGrandes
        BEFORE INSERT OR UPDATE ON Asignatura
        FOR EACH ROW WHEN (:new.nh>250)
DECLARE
BEGIN
        DBMS_OUTPUT.PUT_LINE('La asignatura ' || :new.nombre || ' tiene más de 250 horas.');
END;
```

Instead of

Los **triggers instead of** (en lugar de) sirven para redefinir totalmente la sentencia que lo lanza. No se pueden usar máis que en vistas. Su uso es principalmente permitir la inserción, borrado o actualización de las tablas base independentemente de las reglas que rigen la actualización de las vistas.

Estos triggers siempre son de nivel de fila, pueden leer los valores de OLD y NEW pero no pueden cambiarlos.

Triggers de eventos

Los triggers vistos hasta ahora saltan con sentencias DML sobre tablas o vistas. Hay unos triggers especiales que saltan ante eventos del sistema (shutdown, startup, logon, etc.)

Ejemplos

```
create or replace trigger_logon after logon on database
begin
        insert into auditoria values(audit_seq.nextval, ‘logon’, USER);
end;

create or replace trigger_drop after drop on pepe.Schema
begin
        insert into auditoria values(audit_seq.nextval, ‘drop’, USER);
end;
```

ACTIVAR/DESACTIVAR TRIGGERS

ALTER TRIGGER nome_trigger ENABLE/DISABLE;
ALTER TABLE nome_taboa ENABLE/DISABLE ALL TRIGGERS;
DROP TRIGGER nombre_trigger;

Usos de triggers

Generar automáticamente valores de columnas virtuales -> asigna valores al :NEW
Registrar eventos -> actualiza otras tablas
Recopilar estadísticas sobre el acceso a las tablas -> Actualiza otras tablas
Modificar los datos de la tabla cuando se emiten declaraciones DML contra vistas
Hacer cumplir la integridad referencial cuando las tablas secundarias y principales se encuentran en diferentes nodos de una base de datos distribuida
Evitar operaciones DML en una tabla después del horario comercial habitual
Prevenir transacciones no válidas -> Lanza excepción para evitar el evento
Aplicar reglas complejas de integridad empresarial o referencial que no puedas definir con restricciones.

Tanto los triggers como las restricciones pueden limitar la entrada de datos, pero difieren significativamente. Un disparador siempre se aplica solo a datos nuevos. Por ejemplo, un disparador puede impedir que una instrucción DML inserte un NULL en una columna de base de datos, pero la columna puede contener NULL que se insertaron en la columna antes de que se definiera el disparador o mientras el disparador estaba deshabilitado.

Triggers compuestos

Un trigger compuest puede activarse en varios puntos de tiempo. Cada sección de punto de sincronización tiene su propia parte ejecutable y una parte opcional de manejo de excepciones, pero todas estas partes pueden acceder a un estado PL/SQL común. El estado común se establece cuando comienza el evento desencadenante y se destruye cuando se completa el evento desencadenante, incluso si el evento provoca un error.

Cuando se define un trigger a nivel de fila (FOR EACH ROW) con eventos BEFORE o AFTER y dentro del cuerpo del trigger se referencia (de cualquier forma, incluso con un select) a la misma tabla que hace saltar el trigger, Oracle presenta error ORA-04091 diciendo que la tabla está mutando (cambiando).

La única excepción es un trigger BEFORE INSERT si la inserción se produce fila a fila.
Este error no se produce en triggers que saltan a nivel de sentencia.

Punto de sincronización	Sección
Antes de que se ejecute la declaración desencadenante	BEFORE STATEMENT
Después de que se ejecute la declaración desencadenante	AFTER STATEMENT
Antes de cada fila a la que afecta la declaración desencadenante	BEFORE EACH ROW
Después de cada fila a la que afecta la declaración desencadenante	AFTER EACH ROW

```
CREATE OR REPLACE TRIGGER compound_trigger_name
FOR [INSERT | DELETE] UPDATE [OF column] ON table
COMPOUND TRIGGER
  -- Declarative Section (optional)
  -- Variables declared here have firing-statement duration.
   --Executed before DML statement
BEFORE STATEMENT IS
BEGIN
     NULL;
END BEFORE STATEMENT;
   --Executed before each row change- :NEW, :OLD are available
BEFORE EACH ROW IS
BEGIN
     NULL;
END BEFORE EACH ROW;
   --Executed aftereach row change- :NEW, :OLD are available
AFTER EACH ROW IS
BEGIN
     NULL;
END AFTER EACH ROW;
   --Executed after DML statement
AFTER STATEMENT IS
BEGIN
     NULL;
END AFTER STATEMENT;
END compound_trigger_name;
```

Esto podemos hacerlo o bien guardando los valores a los que queremos acceder de la tabla en el before statement y luego recorrer cada fila y mirar la colección o bien guardar los :new / :old en un before statement y luego recorrerlo en el after statement.
Podemos usar para guardar los datos una tabla temporal o bien atributos generales.

Ejemplo: Trigger que impida que el salario de un trabajador supere en un 10% la media de salarios del departamento.

```
CREATE OR REPLACE TRIGGER Check_Employee_Salary_Raise
  FOR UPDATE OF Salary ON Employees
COMPOUND TRIGGER
        Ten_Percent CONSTANT NUMBER := 0.1;
        TYPE Salaries_t  IS TABLE OF Employees.Salary%TYPE;
        Avg_Salaries Salaries_t;
        TYPE Department_IDs_t  IS TABLE OF Employees.Department_ID%TYPE;
        Department_IDs  Department_IDs_t;

  TYPE Department_Salaries_t  IS TABLE OF Employees.Salary%TYPE INDEX BY PLS_INTEGER;
  Department_Avg_Salaries Department_Salaries_t;

  BEFORE STATEMENT IS
  BEGIN
        SELECT  AVG(e.Salary), NVL(e.Department_ID,-1) BULK COLLECT INTO Avg_Salaries, Department_IDs
                FROM   Employees e  GROUP BY e.Department_ID;
        FOR j IN 1..Department_IDs.COUNT() LOOP
                Department_Avg_Salaries(Department_IDs(j)) := Avg_Salaries(j);
        END LOOP;
  END BEFORE STATEMENT;

  AFTER EACH ROW IS
  BEGIN
        IF :NEW.Salary - :Old.Salary > Ten_Percent*Department_Avg_Salaries(:NEW.Department_ID) THEN
                Raise_Application_Error(-20000, 'Raise too big');
        END IF;
  END AFTER EACH ROW;
END Check_Employee_Salary_Raise;
```

También se podía haber creado usando una tabla temporal para guardar los datos

CREATE GLOBAL TEMPORARY TABLE <nombre>(...

PRÁCTICA PROPUESTA

TRIGGERS

Para la tienda de informática:
a) crear un trigger que, al insertar un producto, le descuente automaticamente el 10% al precio.
b) Crear un trigger que impida el borrado de un producto del que tengamos unidades.
c) Crea una tabla llamada productos_backup y realiza un trigger que al borrar un producto lo inserte automaticamente en productos_backup por si acaso.
d) Realizar un trigger que evite que de un proveedor tengamos más de 3 contactos.

PRÁCTICA PROPUESTA

TRIGGERS

Para la Academia:
a) crear un trigger que evite que un profesor perteneciente a un departamento sea jefe de otro al que no pertenece.
b) Crea una tabla auditoriaMatricula Auditoria(P-CodAuditMatr, F-CodLinMatr—>LineaMatrícula, fecha, nombreUsuario, valorAnterior,ValorNuevo) donde guardes información sobre una nota que es cambiada. En la tabla se almacenará la linea de matricula que se ha modificado, la fecha, el usuario que ha hecho la modificación, el valor que tenia y el valor nuevo.
c) Crea un trigger que impida que un profesor sea también alumno.

PRÁCTICA PROPUESTA

AMPLIACIÓN

Para la base de datos de HR, crear un procedimiento que muestre qué salarios no están entre el mínimo y el máximo para el trabajo que realiza el empleado. El procedimiento debe recorrer todos los empleados y mostrar por pantalla los que están mal pagados.

Para la base de datos de la Tienda de informática, muestra por pantalla, para cada empleado, el número de productos portátiles que ha vendido, con el formato:
El empleado PRAPELLIDO SGAPELLIDO, Nombre ha vendido un total de *número* portátiles.

Crea un trigger para la base de datos «Tienda de informática» para que solo se permita la actualización de la columna fechaHora de la tabla Compra si la fecha está en el mismo mes; de lo contrario mantendrá su valor.

Programa dos disparadores con el fin de controlar que los valores para los campos DNI de las tablas Cliente y Empleado de la base de datos «Tienda de informática» sean excluyentes, es decir, si existe un cliente con cierto DNI, en la tabla no se puede incluir un registro en la tabla empleado con ese mismo DNI, y viceversa.

Crea una función que devuelva el número de productos vendidos por un empleado, que se le pasa como argumento de entrada.

Programa un disparador que limite el sueldo de los empleados. Para valores mayores de 5000, el sueldo máximo será de 5000 €.

Programa un disparador que limite el número de productos diferentes en una venta. Ese número será 15.