4

Lenguaje Procedimental como extensión de SQL

Contenido

Objetivos

- ?.
  -

4.1 Introducción

El capítulo 4 de SQL/PSM del estándar ISO/IEC 9075, con PSM la sigla en inglés de **Persistent Stored Modules** o Módulos persistentes almacenados, se ocupa de la extensión procedimental de SQL, que incluye el control de flujo, el manejo de condiciones, las sentencias de señalización de condiciones, los cursores y las variables locales y la forma de asignar valores o expresiones a variables y parámetros. Además, SQL/PSM formaliza la declaración y el mantenimiento de las rutinas persistentes escritas en lenguaje de bases de datos, como los procedimientos almacenados. Esta parte del estándar es completamente opcional, ya que las distintas proveedoras de Bases de Datos tienen sus propios lenguajes de extensión procedimental de SQL: Transactional-SQL o T-SQL de Microsoft, para su SQLServer; Procedural Language SQL o PL/SQL, de Oracle; Procedural SQL o PSQL, para Postgresql. Este trabajo se centrará en Oracle, Microsoft y MySQL.

Antes de que las bases de datos crearan estos lenguajes de extensión de SQL, se utilizaron las llamadas a librerías específicas, denominadas User-Exits —de los lenguajes procedimentales como COBOL, C, ForTran y orientado a objetos como Java—, para que los programas escritos con lenguaje procedimental tuvieran acceso a la lectura y modificación de datos en las tablas almacenadas en RDBMS. A estas llamadas —que significaban porciones de sentencias SQL dentro del código Anfitrión o Host, llamado Embedded SQL, que, traducido, se lo podría denominar “SQL embebido”—. La respuesta a esta forma inicial fue que el motor almacenara el código ya compilado del lado servidor, y que cualquier lenguaje lo pudiera llamar para su ejecución.

4.2 Vista General de PL/SQL

La extensión procedimental de SQL tiene ventajas sobre el SQL Embebido que otros lenguajes de programación ofrecen desde hace mucho tiempo, ya que permite fijar la lógica y las reglas del negocio de las aplicaciones y almacenarlas en un solo lugar, dentro de la base de datos. En general, los lenguajes de extensión son simples y directos y poseen las construcciones comunes de los lenguajes de uso general. PL/SQL es así y agrega cosas que le permiten el tratamiento de los datos, la modularización de los bloques de código en funciones, los procedimientos, los triggers y los paquetes, además de un manejo de errores muy robusto. Las sentencias, que se comunican internamente con la base de datos, se almacenan en la base de datos Oracle.

Se verá, en esta vista general, los detalles básicos de las ventajas del uso de PL/SQL y sus construcciones básicas.

4.3 Uso del PL/SQL para acceder a la Base de datos Oracle

Muchas aplicaciones que usan la arquitectura cliente/servidor tienen una cuestión en común: la dificultad de mantener las reglas de negocio de una aplicación.

Cuando las reglas de negocio están descentralizadas a lo largo de la aplicación, los desarrolladores deben realizar cambios en toda esta extensión e implementar pruebas

de sistemas para determinar si los cambios han sido suficientes. Sin embargo, en situaciones críticas en tiempo, estas pruebas integrales no pueden llevarse adelante, ya que aumentan el riesgo de salidas inesperadas de sistemas. El cambio lógico en el diseño apunta a la centralización de la lógica en la aplicación, que permita una administración más sencilla de los cambios. Esta capa intermedia de la lógica de negocio en la aplicación puede ser diseñada con PL/SQL y con estos beneficios:

- PL/SQL se administra centralmente dentro de la base de datos Oracle. Se manejan el código fuente y los privilegios de ejecución con la misma sintaxis que se usó para manejar otros objetos de base de datos.
- PL/SQL se comunica en forma directa o nativa con otros objetos de la base de datos Oracle.
- PL/SQL es fácil de leer y tiene muchas características para modularizar el manejo de código y de los errores.

4.4 Programas con PL/SQL

Hay varias formas de construir programas con PL/SQL, desde los diferentes tipos de módulo disponibles a los componentes de un bloque PL/SQL hasta las construcciones lógicas que manejan el flujo de los procesos.

Se Verán los componentes del lenguaje y algunos puntos importantes sobre cada una de sus áreas.

4.4.1 Modularidad

Este lenguaje permite al desarrollador crear módulos de programas para mejorar la reusabilidad del software y para esconder la complejidad de la ejecución de una operación específica detrás de un nombre de procedimiento o función. Por ejemplo, en un proceso complejo como el que requiere agregar un registro de un nuevo estudiante y que involucre varias tablas relacionadas de distintas aplicaciones asociadas: la asistencia, la gestión académica y la contable. Los procedimientos almacenados pueden manejar este agregado de estudiantes para cada una de las aplicaciones, haciendo parecer al usuario que es un solo paso el necesario, cargar los datos en la pantalla y que estén involucrados varios procesos separados se ocupan cada porción del proceso de agregar un nuevo estudiante.

4.4.2 Procedimientos, Funciones, Triggers y Paquetes

Los módulos de PL/SQL se dividen en cuatro categorías: Procedimientos, Funciones, Triggers y Paquetes, cuyas definiciones siguen a continuación:

Procedimientos: son un conjunto de sentencias que aceptan y retornan cero o más variables, que se denominarán parámetros.

Funciones: son un conjunto de sentencias que aceptan parámetros y que su trabajo principal es calcular un valor y devolverlo para finalizar su trabajo, no pueden realizar transacciones mientras son ejecutadas.

Triggers: también se conocen por su traducción: “Disparador”. Sin embargo, en este trabajo, se mantendrá la denominación de origen, que se podría definir como **“**un conjunto de sentencias asociadas a una tabla de la base y también a los eventos de sistema”, como, por ejemplo, la conexión de un usuario o el inicio del apagado o del encendido de la base (estos últimos no se estudiarán porque se encuentran fuera del alcance de este trabajo).

Paquetes: son una colección de procedimientos y funciones que tienen dos partes: una especificación de los procedimientos, funciones que están disponibles o que se ponen como ‘públicas’ (además, pueden tener estructuras de datos definidas por el usuario) y, la otra parte del paquete —denominada cuerpo del paquete—, que contiene el código de los procedimientos y de las funciones especificadas.

4.5 Componentes de un Bloque PL/SQL

Existentes tres componentes de cualquier bloque PL/SQL de los presentados anteriormente: la sección de declaración de variables, la sección ejecutable y el manejo de excepciones. La sección de declaración contiene la identificación de todas las variables que se usarán en el bloque de código. Una variable puede ser de un tipo de datos disponible en las base de datos Oracle, como así también de algún tipo exclusivo de PL/SQL.

La sección ejecutable de un bloque comienza con la palabra clave, BEGIN, y, finaliza, con la palabra END, o con la palabra EXCEPTION, que es el inicio del último componente encargado de atender los errores que ocurrieron en la sección ejecutable y de definir cómo se debe manejar. Esta sección es opcional en PL/SQL.

Existen dos tipos de bloques de código: los denominados “bloques con nombre” y los “anónimos”. Se verán dos ejemplos: el primero de un código de PL/SQL con nombre, más precisamente, una función denominada “convertir_moneda”:

```
FUNCTION convertir_moneda
(cantidad                IN NUMBER(12,3),
moneda_original  IN VARCHAR2,
moneda_destino  IN VARCHAR2)  RETURNS NUMBER(12,3)
IS                                          /* ahora comienza la declaración */
conversion               IN NUMBER(12,3);
datos_mal                EXCEPTION;



BEGIN                                  /*inicio de la sección ejecutable de un bloque*/
        IF cantidad > 0 THEN
        . . .;
        ELSE
```

```
        . . .;
        END IF;
EXCEPTION                       /*inicio de la sección de manejo de excepciones */
        WHEN datos_mal THEN
        . . .;
END;
```

La otra clase de bloques se conoce como “bloque sin nombre o anónimo”. En esta clase de bloques, la sección de declaración se inicia, explícitamente, con la palabra DECLARE y contiene las mismas secciones que los bloques con nombre.

```
DECLARE                         /*inicio de la sección de declaración de un blo-
que*/
conversion              IN NUMBER(12,3);
datos_mal               EXCEPTION;
BEGIN                           /*inicio de la sección ejecutable de un bloque*/
        IF cantidad > 0 THEN
        . . .;
        ELSE
        . . .;
        END IF;
EXCEPTION                       /*inicio de la sección de manejo de excepciones */
        WHEN datos_mal THEN
        . . .;
END;
```

4.5.1 Las construcciones lógicas y de control de flujo

Están disponibles las estructuras lógicas como loops o bucles FOR, WHILE y las sentencias IF-THEN-ELSE, la asignación de valores y de expresiones. Existen otras construcciones lógicas que incluyen tablas y registros propios de PL/SQL, diferentes de las tablas y de las filas de la Base de datos. Todos estos ítems permiten al lenguaje soportar reglas de negocios y, también, proveer datos a las aplicaciones que lo soliciten.

4.6 Cursores

Una de las fortalezas de PL/SQL es la habilidad de manejar cursores, que son los controladores de áreas de memoria que almacenan los resultados de una sentencia SQL. Se utilizan para realizar operaciones con CADA fila resultante de una sentencia SELECT, ya que, en el entorno SQL, el resultado de una sentencia SELECT siempre es tratado como un conjunto de FILAS y no a cada una de ellas

como una individualidad. Para el manejo de estas filas en un cursor, es necesario que se utilicen construcciones repetitivas con lógica incluida, como los LOOP y, también, las operaciones de manipulación de los cursores. Esto se profundizará más adelante con ejemplos concretos.

4.7 Manejo de Errores

Los errores se denominan excepciones en PL/SQL y se analizan implícitamente en cualquier lugar del código ejecutable en el que aparecen. Si en algún momento ocurriera un error en el bloque de código, la correspondiente excepción al error se levantaría con la acción RAISE. En este punto, la ejecución en el código se detendrá y el control se transferirá al manejador de excepciones dentro de la sección de correspondiente, que se podrá escribir al final de la sección ejecutable. Esta sección, como es opcional en el bloque, puede no se haya escrito, lo que hará que el error se envíe al entorno que convocó a este bloque, y allí se atenderá o se ignorará, según cómo se haya previsto la condición de error en este lenguaje anfitrión.

4.8 El desarrollo de un bloque PL/SQL

En esta sección, se verá cómo se declaran, se asignan valores y se usan las variables, dentro de los bloques que se han visto anteriormente.

4.8.1 Los tipos de datos de la base de datos

Hay varios tipos de datos que se utilizan en PL/SQL y que corresponden a los tipos de datos que se usan en la base de datos. Éstos son:

NUMBER(tamaño[, precisión]): se usa para almacenar cualquier número.

CHAR(tamaño), VARCHAR2(tamaño): se usan para almacenar una cadena de texto alfanumérico. El tipo CHAR ocupa todo el tamaño y agrega, al contenido significativo, los blancos hasta que cubra el tamaño de la variable. VARCHAR2 almacena sólo el contenido significativo, lo que ahorra espacio.

DATE: se usa para almacenar fechas, horas, minutos y segundos.

LONG: almacena grandes bloques de texto, hasta 2 GigaBytes de largo.

LONG RAW: almacena grandes bloques de datos en formato binario.

ROWID: se usa para almacenar el formato especial de ROWIDs en la base de datos

BLOB, CLOB, NCLOB, BFILE: son tipos de datos para objetos de gran tamaño que, respectivamente, están en formato binario, en formato carácter, con soporte de caracteres nacionales y en archivos con información en formato binario y que, por su tamaño, no se almacenan en tablas, sino en archivos administrados por el motor.

4.8.2 Tipos de datos que son propios del lenguaje PL/SQL

Existen otros tipos de datos no diseñados para almacenar datos en una tabla, sino para manipulación de datos dentro de los bloques PL/SQL. Se clasifican de la siguiente manera:

DEC, DECIMAL, REAL, DOUBLE_PRECISION, INTEGER, INT, SMALLINT, NATURAL, POSITIVE, NUMERIC: son un subconjunto del tipo NUMBER que se usan para declaración de variables.

BINARY_INTEGER, PLS_INTEGER: estos tipos de datos almacenan enteros y las variables definidas así se deben convertir explícitamente a un formato de base de datos para que se almacenen en una columna.

BOOLEAN: almacena un valor TRUE o FALSE o NULL.

TABLE, RECORD: los tipos de datos TABLE pueden almacenar algo similar a un arreglo, mientras un RECORD puede almacenar variables con tipos de datos compuestos.

En general, las variables que almacenarán datos para luego insertarlos en columnas de tablas de bases de datos, se definen con el mismo tipo de datos que las columnas destino y, para no tener que buscar qué tipo de dato tiene cada columna que se afectará, cuando se definen las variables se puede usar el modificador %TYPE en la declaración de la variable; por ejemplo:

```
DECLARE
v_legajo                estudiantes.legajo%TYPE;
```

De esta manera, la variable v_legajo tendrá el tipo de datos declarado en la tabla estudiantes para legajo. Esto ahorra esfuerzo y disminuye la probabilidad de errores en el tiempo de ejecución, con una pequeña sobrecarga en el momento de la compilación.

Existe algo similar con el modificador **%ROWTYPE**, que permite, al desarrollador, crear un tipo de dato compuesto con todas las columnas de la tabla referenciada: por ejemplo, con la tabla estudiantes:

```
DECLARE
r_estudiante                estudiantes%ROWTYPE;
```

Con el ejemplo anterior, se habrá creado un RECORD con las columnas y con sus respectivos tipos de datos idénticos a las columnas de la tabla Estudiantes.

En PL/SQL, también se deben declarar las constantes, según se ve en el ejemplo:

```
DECLARE
pi                      CONSTANT NUMBER(15,14) := 3,14159265358;
```

En el ejemplo, además de declarar como constante al identificador pi, se ha inicializado su valor con el símbolo de asignación ':=' y, el valor exacto, que ya no podrá modificarse durante la ejecución del programa. En el caso de las variables, se pueden usar varios modos de asignar variables, además de ':=' , una función, cuya cláusula RETURN, le asignará el resultado a la variable, como parámetro de salida de un procedimiento y con la cláusula INTO en el SELECT, como se ejemplificará más adelante.

4.9 Interacción con la Base de Datos Oracle

Se verá cómo se usan las sentencias SELECT, INSERT, UPDATE y DELETE en los bloques de programación, cómo se utilizan los tributos de cursores implícitos y cómo se aplica el procesamiento de transacciones en PL/SQL.

Para la interacción de los bloques de programación en PL/SQL, no es necesaria ninguna interfaz, como ODBC, con lo que se gana en simpleza y en *performance*.

La sentencia SELECT agrega una cláusula INTO para que pueda definirse a qué variable asignar cada columna de la lista del SELECT. Como se verá en el siguiente ejemplo de código. Cabe mencionar, que se ha usado un RECORD a imagen y semejanza de la tabla estudiantes, para almacenar una fila que tenga como legajo el nro. 7547. Este conjunto de valores se puede referenciar completamente, o por sus partes, con la notación "record.columna", de la misma manera que en el INSERT. Se asigna un legajo cualquiera a la variable del registro, pero cambiándole el contenido original por el valor 7544 y, luego, se lo usará para elegir que fila se borrará en la última sentencia DELETE.

```
DECLARE
        r_estudiante            estudiantes%ROWTYPE;
        v_apellido              VARCHAR2(30)    := 'Perez';
        v_nombre                VARCHAR2(30)    := 'Juan';
BEGIN
        SELECT *
        INTO r_estudiante
        FROM estudiantes
        WHERE legajo = 7547;

        UPDATE estudiantes
        SET nombre = nombre||', '||v_nombre
        WHERE legajo = r_estudiante.legajo;

INSERT INTO estudiantes (legajo, apellido, nombre, tipo_documento, documento,
id_sexo, calle, calle_nro, id_barrio)
VALUES (r_estudiante.legajo, r_estudiante.apellido, r_estudiante.nombre, r_estudiante.
```

```
tipo_documento, r_estudiante.documento, r_estudiante.id_sexo, r_estudiante.calle,
r_estudiante.calle_nro, r_estudiante.id_barrio);

r_estudiante.legajo := 7544;

DELETE FROM estudiantes
WHERE legajo = r_estudiante.legajo;
END;
```

4.10 El tratamiento de transacciones en PL/SQL

Las mismas opciones del proceso de transacciones de SQL están disponibles en PL/SQL. El inicio de la transacción lo marca la primera sentencia de modificación de datos, y deben ser explícitamente definidos los puntos de confirmación COMMIT o de corrección ROLLBACK y los eventuales puntos de salvaguarda SAVEPOINT, para definir apropiadamente la transacción.

En un programa, la primera sentencia SQL inicia la transacción y, cuando ésta finaliza, la próxima comienza otra automáticamente. Por esta razón, cada sentencia SQL es parte de una transacción. El uso de COMMIT y ROLLBACK asegura que los cambios hechos con sentencias SQL se confirmen o se deshagan en conjuntos consistentes simultáneamente. Con SAVEPOINT se pone el nombre y se marcan las posiciones de las distintas sentencias de una transacción, para poder deshacer sus partes en el caso en que fuera necesario por alguna regla de negocio.

Pero, ¿qué sucede cuando se realizan transacciones desde distintas sesiones de usuarios en forma concurrente? En esta situación, el Motor de Base de datos debe garantizar que no haya interferencias o anomalías con los cambios que se están llevando a cabo y, además, que las sesiones que están haciendo lecturas de las tablas modificadas no puedan leer datos que todavía no han sido confirmados. Esta anomalía se denomina “Lecturas sucias”.

Para tener una primera idea de los niveles de aislamiento, es preciso indicar que se aplica el principio de que los LECTORES no bloquean a los ESCRITORES, y viceversa.

Para esto, el estándar SQL define niveles de aislamiento que previenen ésta y otras anomalías, como la modificación de datos leídos y la grabación destructiva de cambios que no estén correctamente aislados. Para una mejor compresión, podríamos describir estas situaciones de la siguiente manera: dadas dos transacciones T1 y T2 tendrían un conflicto de Escritura-lectura, si la segunda leyera un cambio sin confirmación de la primera (denominada, en el párrafo anterior, “Lectura Sucia” o, también, “Lectura Desfasada”). Si T2 modifica el valor de una fila mientras T1 ha leído el valor anterior a este cambio, se produce una anomalía de lecto-escritura conocida como “lectura no repetible”. Por último, se produce un conflicto de Escritura-escritura si T2 modifica un valor que ha sido alterado por T1 en una transacción que no ha finalizado aún. Por esta razón, a esta anomalía se la denomina “Actualización perdida”, ya que no se llega a confirmar el cambio de T1.

El estado por defecto de todas las transacciones en Oracle garantiza la consistencia de lectura en el nivel de Sentencia, que es el asegura que una consulta verá solamente los cambios confirmados por otras sesiones *antes* de que la sentencia SELECT inicie su procesamiento, más los cambios que se hubieran realizado *antes* en la misma transacción.

Se podrá usar la sentencia SET TRANSACTION para establecer una transacción sólo de lectura (read-only transaction) que provee consistencia de lectura en el nivel de sentencia, de tal modo que se garantiza que una consulta leerá solamente los cambios confirmados *antes* que la sentencia SELECT dentro de la transacción actual haya iniciado. Esta sentencia no tiene argumentos y debe ser la primera sentencia SQL en la transacción de sólo lectura, como se expresa en el ejemplo siguiente:

```
SET TRANSACTION READ ONLY;
```

Además de esta sentencia, es posible determinar cerramientos o bloqueos de las tablas que se tratarán en una transacción, asignando diferentes niveles de aislamiento con la sentencia LOCK TABLE y la indicación de los modos disponibles según el siguiente ejemplo:

```
LOCK TABLE mitabla IN <modo> MODE [NOWAIT];
```

Los distintos modos se detallan a continuación:

- EXCLUSIVE permite consultas en la tabla, pero prohíbe cualquier otra actividad sobre ella.
- SHARE permite acceso concurrente a la tabla afectada, pero prohíbe cualquier actualización a ella.
- ROW SHARE permite acceso concurrente a la tabla afectada, pero impide que otro usuario establezca un LOCK EXCLUSIVE sobre ella.
- ROW EXCLUSIVE es un bloqueo similar al anterior pero, además, impide que otra sesión fije un modo SHARE sobre el objeto. Es el modo que se fija automáticamente cuando se realiza un UPDATE, INSERT o DELETE sobre una tabla.
- SHARE ROW EXCLUSIVE bloquea toda la tabla y permite leer las filas, pero impide las actualizaciones y que otra sesión establezca bloqueos del tipo SHARE.
- SHARE UPDATE —sinónimo de ROW SHARE— ha sido incluido para mantener la compatibilidad con versiones anteriores del motor ORACLE.

Es posible que algunos de estos modos de bloqueos se establezcan en la misma tabla simultáneamente, como los SHARE y otros; como los EXCLUSIVOS, se pueden aplicar solamente una vez en una tabla.

Finalmente, el parámetro NOWAIT indica que el motor debe retornar el control de la sesión que intenta hacer alguna operación o establecer un LOCK sobre una tabla que ya posee un bloqueo de otra sesión. Si no se incluye este parámetro, la sesión esperará hasta que se hayan liberado los bloqueos de la otra sesión.

4.11 Sentencias de control de flujo

Su característica procedimental obliga a PL/SQL a tener sentencias de control de flujo, que ya hemos mencionado anteriormente, y dos categorías de sentencias, como las expresiones condicionales y el LOOP. A continuación, se verán algunos detalles de estas sentencias para controlar la ejecución de un bloque PL/SQL.

Una condición en un programa equivale a la idea de tomar una decisión y, detrás de esta condición, subyace el concepto de valores booleanos de verdadero o falso. Estos valores son el resultado de la ejecución de un tipo de sentencias, denominadas "operaciones de comparación". Algunas comparaciones pueden ser como las siguientes:

- 3 + 5 = 8
- Mi mano tiene 16 dedos
- 4 = 10
- Hoy es jueves

Estas operaciones de comparación se pueden evaluar por su validez, ya que su valor puede resultar Verdadero o Falso. En el primer caso, es cierto que 3+5 es igual a 8, por lo que esta comparación devolverá el valor TRUE, la segunda no es cierta, ya que mi mano tiene 5 dedos y devolvería un valor FALSE. La tercera también devolverá un valor FALSE porque 4 no es igual a 10. Por último, la afirmación 'Hoy es jueves', si se realiza una vez por día durante toda una semana, devolverá seis veces FALSE y una sola TRUE.

El mecanismo de proceso de sentencias condicionales permite estructurar sentencias que se ejecutarán, o no, de acuerdo con los resultados de la evaluación de las condiciones. La sintaxis para las sentencias condicionales es "IF (SI) la comparación es verdadera THEN (ENTONCES) haga lo siguiente". En PL/SQL, se ofrecen dos agregados: el ELSE (SI NO), que permite decir "SI NO es verdadero, haga lo que sigue a esta cláusula ELSE".

Se agrega una cláusula al IF-THEN: el ELSIF, que permite evaluar por distintas condiciones definidas con cada una de sus líneas, que se analizan ordenadas y, si ninguna de ellas es verdadera, sale por el ELSE.

Es importante aclarar, en este punto, que las comparaciones harán entre datos del mismo tipo de dato, y que la comparación con el valor null se realizará con la expresión 'IS null' o por la negativa 'IS NOT null'.

Se analizarán los ejemplos de sintaxis de las comparaciones:

```
DECLARE
        lado_a    number(2) := 20;
        lado_b    number(2) := 30;
        lado_c            number(2) := 40;
        resultado         number(2) := 0;
BEGIN
```

```
        carga_de_datos(parametro_externo, lado_a,lado_b,lado_c);
        IF lado_a > lado_b THEN
                Calcular _ perimetro (lado_a, lado_b,lado_c, resultado);
        ELSIF lado_b > lado_c THEN
                Calcular _ perimetro2 (lado_a, lado_b,lado_c, resultado);
        ELSE
                Resultado=350;
        END IF;
END;
```

El ejemplo crea las variables y las inicializa; cuando inicia el bloque, hay un procedimiento que recibe una variable ya definida en el entorno de ejecución de llamada que cambiará el valor de las tres variables locales. La sentencia condicional evaluará las variables lado_a, lado_b, lado_c y, según su valor, calculará el perímetro y lo asignará a la variable resultado, inicializada en 0 y, en caso de que lado_a sea mayor a lado_b, se calculará el perímetro con el procedimiento y, si lado_a es menor a lado_b, pero lado_b es mayor a lado_c, se calculará con otro procedimiento diferente y, si las dos condiciones anteriores devuelven valores FALSE, el bloque asignará a resultado un valor fijo de 350.

4.11.1 Usando Loops

Otra situación que surge en programación es cuando es necesario ejecutar un conjunto de sentencias repetidamente. Estas repeticiones se pueden controlar de dos maneras: la primera, es repetir el código una determinada cantidad de veces y, la segunda, es repetir el código hasta que alguna condición devuelve TRUE. Hay tres tipos de repetidores o bucles, también denominados LOOPs por la sentencia asociada, por ejemplo:

- **LOOP-EXIT** o bucle simple: repite las sentencias hasta que procesa la sentencia de salida, EXIT.
- **WHILE-LOOP**
- **FOR-LOOP**

Se analizarán, a continuación, ejemplos de estas sentencias:

La sentencia **LOOP-EXIT** está diseñada para determinar si la condición dentro del loop tiene el valor para terminar y procesar la sentencia EXIT, que se debe escribir dentro del grupo de sentencias del loop, que tiene esta desventaja. Se escribirá esta sentencia para evitar que el ciclo se repita indefinidamente, y se asegurará que la condición que permitirá ejecutarlo también se cumpla cuando es deseado que el loop termine. En el ejemplo que sigue, si lado_b nunca alcanza el valor 25, no se detendrá el loop, habrá que cancelar desde afuera la ejecución del bloque.

```
DECLARE
        lado_a    number(2) := 20;
        lado_b    number(2) := 30;
        lado_c            number(2) := 40;
        resultado         number(2) := 0;
BEGIN
        LOOP
                Preguntar_datos(parametro_externo, lado_a,lado_b,lado_c);
                Calcular _ perimetro (lado_a, lado_b,lado_c, resultado);
        IF   lado_b =25 THEN
                EXIT;
        END IF;
        END LOOP;
END;
```

Existe una variante en la sentencia EXIT que es la cláusula WHEN a continuación del EXIT en el que se evalúa la condición y se evita que se construya el IF-THEN, para decidir cuándo se ejecuta la salida. Un ejemplo:

```
DECLARE
        lado_a    number(2) := 20;
        lado_b    number(2) := 30;
        lado_c            number(2) := 40;
        resultado         number(2) := 0;
BEGIN
        LOOP
                Preguntar_datos(parametro_externo, lado_a,lado_b,lado_c);
                Calcular _ perimetro (lado_a, lado_b,lado_c, resultado);
                EXIT WHEN lado_b =25;
        END LOOP;
END;
```

La sentencia **WHILE-LOOP** difiere del loop básico en que evalúa, primero, la condición para salir del bucle cuando devuelve FALSE. Tiene como ventaja que lo procesa siempre y que la evaluación no se puede eludir. Nótese que se debe invertir el sentido de la comparación, porque se desea que, cuando sea distinto a 25, se ejecute el conjunto de sentencias necesarias. Por ejemplo:

```
DECLARE
        lado_a    number(2) := 20;
        lado_b    number(2) := 30;
        lado_c            number(2) := 40;
        resultado         number(2) := 0;
BEGIN
        WHILE lado_b <>25;
        LOOP
                Preguntar_datos(parametro_externo, lado_a,lado_b,lado_c);
                Calcular _ perimetro (lado_a, lado_b,lado_c, resultado);
        END LOOP;
END;
```

La sentencia **FOR-LOOP** difiere del loop básico en que define exactamente la cantidad de veces que el CODIG se ejecuta antes que se salga del mismo ya que crea un contador y un rango entre el cual se valorizará el mismo. Puede definirse que este contador incremente su valor o que lo vaya disminuyendo hasta encontrar el valor inferior del rango. Vemos el ejemplo:

```
DECLARE
        lado_a    number(2) := 20;
        lado_b    number(2) := 30;
        lado_c            number(2) := 40;
        resultado         number(2) := 0;
        conta             number(2) := 0;
BEGIN
        FOR conta IN 1 . . 25 LOOP
                Preguntar_datos(parametro_externo, lado_a,lado_b,lado_c);
                Calcular _ perimetro (lado_a, lado_b,lado_c, resultado);
        END LOOP;
END;
```

En esta sentencia, la variable ‘conta’ inicializada en 0, cuando se procesa el FOR-LOOP, se lo valora en 1 y, por cada vuelta del LOOP, se lo incrementa de 1 en 1 hasta llegar al valor 25 y, en la próxima vuelta, cuando se lo valora en 26, se interrumpe la ejecución y continua la ejecución a partir de la sentencia END LOOP.

4.12 Manejo de Cursores

Ya se definió qué es un cursor; ahora se verá que existen dos tipos, definidos explícita o implícitamente, que pueden tener parámetros y como se combinan

cursores y sentencias de control de loops y la sentencia que integra ambos elementos llamados Loop FOR de cursores.

Se recordará que los cursores son una dirección de memoria en la que se almacenan los resultados de una sentencia SQL. Se utilizan, frecuentemente, en PL/SQL para que procese el conjunto de filas que devuelve un SELECT.

Los cursores explícitos tienen un nombre definido por el desarrollador y agregan un control mayor sobre la ejecución de la sentencia. Se verá, a continuación, una declaración de un cursor explícito:

```
DECLARE
        CURSOR cursor_estudiante        IS
                SELECT * FROM estudiantes;
BEGIN
        . . .
END;
```

Cada vez que se ejecuta una sentencia SQL, se crea un cursor implícito. El desarrollador puede necesitar controlar la operación de una sentencia, aprovechando la definición que tiene PL/SQL asociada con estas direcciones de memoria. Estos atributos son: **%notfound,** que se valúa en TRUE cuando la sentencia SELECT no trae filas; el caso inverso, valúa en TRUE a **%found** y, cuando trae filas, la cantidad se asigna al atributo **%rowcount**; por último, define en TRUE al **%isopen**. A continuación, se Verá un ejemplo de un atributo de un cursor implícito que corresponde al procesamiento del UPDATE: aquí, el desarrollador puede preguntar si el resultado de la búsqueda de las filas que se actualizarán —con el atributo %notfound (en este caso, como es implícito, se le antepone el prefijo SQL)— fue positivo o negativo. Si no hubiera filas con el legajo 7547 se insertará, con ese legajo, el apellido 'Perez' y el documento 16883022;

```
DECLARE
        mi_estudiante     estudiantes.legajo%TYPE := 7547;
        mi_apellido               estudiantes.apellido%TYPE := 'Perez';
        mi_documento    estudiantes.documento%TYPE := 16883022;
BEGIN
        UPDATE estudiante
        SET documento = mi_documento
        WHERE legajo = mi_estudiante;

        IF SQL%NOTFOUND  THEN
                INSERT INTO estudiantes (legajo, apellido, documento)
                VALUES (mi_estudiante, mi_apellido, mi_documento);
        END IF;
    END;
```

Para operar adecuadamente un cursor se necesitan tres pasos: abrir, obtener un valor y cerrarlo. Se usará una tabla denominada “empleados” que cuenta con el número de empleado, el apellido y el sueldo.

```
DECLARE
        mayor_porc_aumento      constant number (10,5)  := 1.2;
        medio_porc_aumento      constant number (10,5)  := 1.1;
        menor_porc_aumento      constant number (10,5)  := 1.05;
        TYPE t_emp IS RECORD (
                t_sueldo empleado.sueldo%TYPE,
                t_nro_emp       empleado.nroempleado%TYPE);
        r_empleado              t_emp;
        CURSOR c_empleado IS
                SELECT nroempleado, sueldo
                FROM empleados;
BEGIN
        OPEN c_empleado;
        LOOP
                FETCH c_empleado INTO r_empleado;
                EXIT WHEN c_empleado%NOTFOUND;
                IF r_empleado. t_nro_emp = 688
                        OR r_empleado. t_nro_emp = 700 THEN
                        UPDATE empleados
                        SET sueldo =  r_empleado.t_sueldo*mayor_porc_aumento
                        WHERE nroempleado = r_empleado.t_nro_emp;
                ELSIF r_empleado.t_nro_emp = 777
                        OR r_empleado.t_nro_emp = 788 THEN
                        UPDATE empleados
                        SET sueldo =  r_empleado.t_sueldo*medio_porc_aumento
                        WHERE nroempleado = r_empleado. t_nro_emp;
                ELSE
                        UPDATE empleados
                        SET sueldo =  r_empleado.t_sueldo*menor_porc_aumento
                        WHERE nroempleado = r_empleado. t_nro_emp;
                END IF;
        END LOOP;
        CLOSE c_empleado;
END;
```

En el ejemplo, se ha realizado un acceso a la base buscando todos los empleados. Con las filas en memoria, en la dirección del cursor, el proceso abre el cursor que consiste en ejecutar el select asociado y se ubica en su primer registro. Con FETCH toma los valores de éste y se los asigna al RECORD r_empleado. A medida que se vaya repitiendo la ejecución de este FETCH, se moverá al siguiente registro del cursor, correspondiente a la siguiente fila recogida por el select, y los datos se cambian en r_empleados. El loop tiene la condición de finalización o salida, que será cuando no encuentre filas usando el atributo del cursor definido (c_empleados%NOTFOUND), que será porque el select no trajo ninguna fila o ya se ha procesado la última. Es importante destacar que el uso de una definición de tipo de dato diseñado por el usuario, en la línea en la que TYPE indica que se deben precisar un tipo de dato compuesto, un vector con dos campos —t_nro_emp y t_sueldo—, con la misma definición de las columnas relacionadas en la tabla empleados.

En el momento de abrir un cursor, se pueden definir parámetros para que, al ejecutar el select asociado, se determinen las condiciones por las que, este select, filtrará las filas haciendo más chico el conjunto de filas para, de esta manera, mejorar la *performance* del proceso.

El lenguaje PL/SQL combina dos construcciones en el denominado LOOP FOR para Cursores que abrevia todos los pasos: de apertura, de recorrido y de cierre. A continuación, se verá un bloque que hace lo mismo pero, al usar el LOOP FOR para Cursores, con menos líneas de código.

```
DECLARE
        mayor_porc_aumento      constant number (10,5)  := 1.2;
        medio_porc_aumento      constant number (10,5)  := 1.1;
        menor_porc_aumento      constant number (10,5)  := 1.05;
        TYPE t_emp IS RECORD (
                t_sueldo empleado.sueldo%TYPE,
                t_nro_emp       empleado.nroempleado%TYPE);
        r_empleado              t_emp;
        CURSOR c_empleado IS
                SELECT nroempleado, sueldo
                FROM empleados;
BEGIN
        FOR r_empleado IN c_empleado LOOP
                IF r_empleado. t_nro_emp = 688
                        OR r_empleado. t_nro_emp = 700 THEN
                        UPDATE empleados
                        SET sueldo =  r_empleado.t_sueldo*mayor_porc_aumento
                        WHERE nroempleado = r_empleado.t_nro_emp;
                ELSIF r_empleado.t_nro_emp = 777
                        OR r_empleado.t_nro_emp = 788 THEN
```

```
                    UPDATE empleados
                    SET sueldo =  r_empleado.t_sueldo*medio_porc_aumento
                    WHERE nroempleado = r_empleado. t_nro_emp;
            ELSE
                    UPDATE empleados
                    SET sueldo =  r_empleado.t_sueldo*menor_porc_aumento
                    WHERE nroempleado = r_empleado. t_nro_emp;
            END IF;
        END LOOP;
END;
```

No es necesario escribir la apertura, asignar los valores al registro, la condición de cierre y el cierre propiamente dicho del cursor, ya que todo se procesa internamente por la sentencia FOR LOOP asociada con un cursor.

4.13 Manejo de errores

En esta sección, se verán los tres tipos básicos de errores, las excepciones comunes y cómo codificar la sección de excepción. Esta posibilidad de manejo de errores es una de las mejores contribuciones a la robustez de las aplicaciones construidas en Oracle. Los errores necesitan ser tratados explícitamente, sin necesidad de usar IF para detectar la situación anómala. En PL/SQL, se usan los manejadores de excepciones que identifican el problema y le asignan una porción de código para resolver la situación o, al menos, para dejar registro de lo sucedido y para poder deshacer la transacción en curso.

Los tres tipos de excepciones son: las **predefinidas, las definidas por el usuario y las internas**. El manejo de excepciones ofrece ventajas en simplicidad y en flexibilidad. Las excepciones predefinidas brindan al desarrollador la posibilidad de revisar los problemas predefinidos; las excepciones definidas por el usuario permiten determinar situaciones que se tratarán como excepción, y tratarla de la misma manera. Las excepciones internas se desarrollarán en los siguientes ejemplos.

Las excepciones predefinidas se usan en conjunto con las situaciones predefinidas que pueden ocurrir en la base; por ejemplo, cuando una sentencia SELECT no devuelve filas —NO_DATA_FOUND—, se ejecuta automática y no es necesario que se ejecute la sentencia RAISE. Cuando este tipo de excepción ocurre, si fuera imprescindible realizar algún cambio, se escribirá el código necesario en la sección de excepciones, en el manejador definido para esta excepción.

Las excepciones definidas por el usuario se pueden usar para aplicar situaciones previstas en las reglas de negocios, para que se traten de la misma manera que las otras excepciones. Éstas últimas no se consideran errores y se podrían escribir bloques que traten estas situaciones de manera consistente en toda la aplicación. Para utilizar estas definiciones, se cumplirán los siguientes pasos: la **declaración de la excepción** con un nombre con la que se la invocará durante el proceso cuando

esta situación se presente; luego, se necesitará **una prueba de aparición de la excepción**, con un código específico que preguntará si las condiciones indican la aparición de esta situación y que ejecutará el comando RAISE. Para que la ejecución se dirija a la próxima etapa. El **manejo de la excepción,** en la sección EXCEPTION, con la cláusula WHEN y el nombre de la excepción con el que se creó y el bloque de código o la llamada a un procedimiento construido para que siempre se resuelva de la misma manera.

A continuación, un bloque con los dos tipos de excepciones:

```
DECLARE
        mi_estudiante           estudiantes.legajo%TYPE := 7547;
        r_estudiante            estudiantes%ROWTYPE;
        documento_null          EXCEPTION; /*declara la Excepción del usuario
*/
BEGIN
        SELECT *
        INTO r_estudiante
        FROM estudiantes
        WHERE legajo = mi_estudiante;
        IF r_estudiante.documento IS NULL THEN
                RAISE documento_null;  /*Levanta la Excepción declarada*/
        END IF;
EXCEPTION
        WHEN NO_DATA_FOUND THEN  /*Excepción predefinida*/
                DBMS_OUTPUT.PUT_LINE('No hay filas');
        WHEN documento_null THEN  /*Excepción definida por el usuario */
                DBMS_OUTPUT.PUT_LINE('La columna Sueldo es nula para el
        estudiante: '||r.estudiante.documento);
END;
```

Cabe aclarar que se ha usado un procedimiento provisto por la base denominado PUT_LINE, que se almacena en el paquete de la base DBMS_OUTPUT, que muestra este mensaje en la consola de usuario o en el área de mensajes de los programas que llamarían a este bloque de datos.

Las excepciones internas asocian un nombre de excepción con un error de la base de datos. El usuario o desarrollador puede extender la lista de excepciones asociadas con errores de la base de datos con el uso de la palabra reservada **pragma,** seguido por **exception_init,** que es una directiva al compilador que permite asociar un numero de error, entre el -20000 y -21000, que es el rango asignado internamente en la Base de datos, a los errores del usuario. En el siguiente uso, se observa cómo se utiliza esta sentencia, basado en el código anterior. Cabe destacar que, cuando se recurre a **exception_init,** se logra que el motor devuelva el código asignado a la excepción al entorno anfitrión de este bloque de programación.

```
DECLARE
        mi_estudiante          estudiantes.legajo%TYPE := 7547;
        r_estudiante           estudiantes%ROWTYPE;
        documento_null         EXCEPTION; /*declara la Excepción del usuario
*/
        PRAGMA EXCEPTION_INIT(documento_null, -20001) /*asigna el numero
de error a
la Excepción del usuario */
```

Excepciones comunes

Existen numerosas excepciones que se pueden usar y que permiten manejar los errores o las situaciones inesperadas en los programas, algunas de las definidas son:

too_many_rows: cuando un select, ubicado donde se espera sólo una fila, devuelve más de una.

no_data_found: cuando un select, un update o delete no encuentran filas en la búsqueda.

invalid_cursor: ocurre cuando se quiere cerrar un cursor que no ha sido abierto.

cursor_already_open: ocurre cuando se intenta abrir un curso que ya está abierto.

dup_val_on_index: ocurre cuando se quiere insertar una fila con clave primaria ya existente.

zero_divide: ocurre cuando se intenta dividir un número por cero.

Las dos primeras son las excepciones más utilizadas; en el ejemplo anterior, se vio el empleo de este tipo de excepciones. Es importante recalcar que no hay que definirlas ni 'levantarlas' con RAISE.

4.13.1 Codificando en la seccion de excepciones

Las excepciones aparecen y se deben tratar con código ejecutable que permita registrar y, si es posible, resolver la situación planteada. En la sección de excepciones, se codificará y se separará en párrafos que se ejecutarán cuando una excepción se detecte. Estos párrafos comienzan con la palabra WHEN y, a continuación, se coloca el nombre de la excepción. Cabe destacar que se pueden tratar excepciones que no tengan un párrafo WHEN asociado, esto se logra si se usa el nombre genérico de excepción, OTHERS. Por lo tanto, esta sería una sección de excepciones muy común:

```
EXCEPTION
        WHEN NO_DATA_FOUND THEN  /*Excepción predefinida*/
                DBMS_OUTPUT.PUT_LINE('No hay filas');
        WHEN OTHERS THEN/*Código ejecutable ante cualquier otro tipo de
Excepción */
                DBMS_OUTPUT.PUT_LINE('Se ha presentado una excepcion');
END;
```

Cabe destacar que una vez ejecutada la sección dedicada a una excepción, el control del programa sale del bloque que lo contiene, esto quiere decir que, si se ingresa a una excepción, no se puede procesar el resto de las excepciones.

4.14 Construyendo procedimientos y funciones con PL/SQL

Se han visto hasta aquí, los elementos del lenguaje y las estructuras de programas o bloques de programación, que se denominan anónimas porque no tienen un nombre asignado; resta conocer cómo se construyen los procedimientos, las funciones, los paquetes y los triggers.

Estas construcciones —denominados, genéricamente, "Programas almacenados"— tienen, como los bloques anónimos, una sección de declaración, codificación y manejo de errores. Dos diferencias significativas con los bloques anónimos: los programas almacenados tienen nombres que los bloques anónimos no poseen, esto permite que se los pueda invocar en cualquier línea de ejecución, simplemente indicando el nombre y los parámetros con que deben ingresarse. La segunda diferencia ha sido nombrada, los bloques anónimos no pueden usar parámetros y los programas almacenados sí lo permiten, lo que les da mayor flexibilidad.

A su vez, existen diferencias entre los procedimientos y las funciones que se tratarán a continuación. Un procedimiento puede aceptar ninguno, uno o más valores como parámetros, pero una función siempre DEBE devolver uno, y nada más que un solo valor.

Antes de ver algunos ejemplos, unas palabras acerca de los aspectos de seguridad. Para crear los procedimientos y las funciones almacenadas, se requiere que el usuario o desarrollador tenga el permiso de creación, que se asigna como sigue:

```
GRANT CREATE PROCEDURE TO miusuario;
```

Y una vez desarrollado el código, cuando se lo crea, se define el esquema en el que se agrupa junto con otros objetos del mismo usuario. Para que otros usuarios puedan ejecutarlo, el dueño del procedimiento o función almacenada les debe dar el permiso de ejecución.

```
GRANT EXECUTE ON miprocedimiento TO miusuario;
```

Los bloques anónimos tienen la desventaja de que, una vez ejecutados, la base de datos no registra nada acerca de éstos; en cambio, cuando se usan los procedimientos almacenados, la base guarda la compilación y el código parseado, lo que define una mayor rapidez en el procesamiento.

Un ejemplo de construcción de un procedimiento almacenado:

```
CREATE OR REPLACE PROCEDURE cambios_estudiantes (p_legajo IN
NUMBER,
                                                 p_apellido
IN VARCHAR2,
```

```
                                                                    p_nombre
IN VARCHAR2) AS
BEGIN

        UPDATE estudiantes
        SET nombre = p_nombre, apellido = p_apellido
        WHERE legajo = p_legajo;


EXCEPTION
        WHEN no_data_found THEN
        DBMS_OUTPUT.PUT_LINE('No existe el empleado con el legajo '|| p_legajo)
END;
```

El uso de parámetros es una de las características principales de estos programas almacenados, ya que permiten una mayor flexibilidad y se declaran junto con el nombre del programa o función. Esto los hace parte del nombre completo, por lo que luego se deben valorizar al ejecutarse desde otro bloque o desde la línea de comandos. A continuación, un ejemplo de cómo se debe ejecutar este procedimiento:

```
BEGIN
cambios_estudiantes(7547, 'Alvarez', 'Juan');
END;
```

El resultado es que, al empleado con legajo 7547, se le habrá cambiado el nombre y el apellido por los valores que se ingresaron en la llamada del procedimiento **cambio_estudiantes.** Cabe destacar que, en este caso, se usan solamente los parámetros de entrada, que podrán modificarse en otra versión para mostrar un ejemplo de parámetros de salida. A continuación:

```
CREATE OR REPLACE PROCEDURE cambios_estudiantes2(p_legajo              IN
NUMBER,
                                                        p_apellido
IN VARCHAR2,
                                                        p_nombre
IN VARCHAR2,                                                    p_resultado
OUT NUMBER ) AS
BEGIN
        UPDATE estudiantes
        SET nombre = p_nombre, apellido = p_apellido
        WHERE legajo = p_legajo;
        p_resultado := 1;
EXCEPTION
```

```
        WHEN no_data_found THEN
        p_resultado := 0;
        DBMS_OUTPUT.PUT_LINE('No existe el empleado con el legajo '|| p_legajo)
        WHEN others THEN
        p_resultado := -1;
END;
```

Este nuevo parámetro cambia el modo de ejecutarlo; ahora, el bloque que lo convocó debe recibir el resultado de la ejecución que serán tres valores: el 1 significa que el proceso ha sido exitoso, el 0 que el proceso fracasó porque no existía el empleado con el legajo ingresado y, con -1, también fracasó pero por otro tipo de problema. Se verá el código que llamará al procedimiento.

```
DECLARE
        s_resultado NUMBER := 0;
BEGIN
cambios_estudiantes2(7547, 'Alvarez', 'Juan', s_resultado);
        IF s_resultado = 1 THEN
                DBMS_OUTPU.PUT_LINE('actualizado');
        ELSIF s_resultado = -1 THEN
                DBMS_OUTPU.PUT_LINE('no existe el estudiante');
        ELSE
                DBMS_OUTPU.PUT_LINE('problemas');
        END IF;
END;
```

Se finalizará este capítulo mostrando una función almacenada y la forma en que se puede ejecutar. Se preverá el ingreso de las dos primeras letras de las monedas y, luego, se buscará en la tabla cotizaciones el registro que tiene previsto las combinaciones de monedas y el coeficiente que se aplicará en la conversión. Si todo resultara exitoso, se devolverá el valor de conversión del cambio entre monedas. Si se ingresara una combinación no prevista, la función saldrá por una excepción y retornará un valor de 0 de conversión y un mensaje.

```
CREATE OR REPLACE FUNCTION convertir_moneda
(cantidad                IN NUMBER(12,3),
moneda_original  IN VARCHAR2,
moneda_destino  IN VARCHAR2)  RETURNS NUMBER(12,3)
IS
        conversion              NUMBER(12,3) := 0;
        coeficiente             NUMBER(6,3);
        datos_mal               EXCEPTION;
```

```
BEGIN                                   /*inicio de la sección ejecutable de un bloque*/
                IF moneda_destino IS NULL OR moneda_original IS NULL THEN
                        RAISE datos_mal;
                END IF;
                SELECT coeficiente_conversión
                INTO coeficiente
                FROM cotizaciones
                WHERE origen = moneda_original
                AND destino = moneda_destino;
                conversion = cantidad * coeficiente;
                RETURN conversion;
EXCEPTION                               /*inicio de la sección de manejo de excepciones */
        WHEN no_data_found THEN
                DBMS_OUTPU.PUT_LINE('Combinación de monedas no prevista');
                RETURN conversion;
        WHEN datos_mal THEN
                DBMS_OUTPU.PUT_LINE('No ingresa datos de moneda');
                RETURN conversion;
END;
```

A continuación, se ilustrará cómo se ejecuta esta función, a la que se le pedirá la conversión de 100 pesos a euros. El resultado sería la cantidad de euros que se comprarían con 100 pesos.

```
DECLARE
        resultado NUMBER(12,3) := 0;
BEGIN
        resultado := convertir_moneda(100,'PE', 'EU');
END;
```

Otra forma de usar la función dentro de un select, sobre una tabla de Oracle, que tiene una sola fila y que se adopta como comodín para superar la restricción de obligatoriedad de la cláusula FROM y de una tabla en ella.

```
DECLARE
        resultado NUMBER(12,3) := 0;
BEGIN
        SELECT convertir_moneda(100,'PE', 'EU')
        INTO resultado
        FROM dual;
    END;
```

4.14.1 Creando Paquetes con PL/SQL

En las bases de datos Oracle, existe una construcción denominada "paquete" —en inglés, Package— que ayuda a los desarrolladores a agrupar todos los bloques PL/SQL que pueden estar relacionados dentro de una aplicación, o por las características de su funcionamiento. Además de que este agrupamiento permite tener ordenado todos los programas en uso, otorga ventajas en los tiempos de ejecución porque, cuando es convocado por primera vez algún componente del paquete, todo el paquete se sube a la memoria; incluso, puede mantenerse permanentemente allí, para mejorar la performance de los procesos.

En los paquetes se pueden almacenar programas, tipos definidos por el usuario, excepciones, variables y hasta definición de cursores y tablas temporarias. Responde a las características del concepto de Encapsulamiento de la orientación a objetos y, para convocar a un programa, se usa la notación de esta modalidad; es decir, para llamar una construcción cualquiera que esté dentro de un paquete se utiliza esta sintaxis, "**nombre_paquete.miprocedimiento".**

Los paquetes tienen dos partes: la especificación y el cuerpo. Esta estructura proviene de lenguajes como ADA y C. Se podría afirmar que la especificación es una unidad de código PL/SQL que nombra los procedimientos, las funciones, las excepciones, los tipos definidos por el usuario, las variables y otras construcciones disponibles para uso público en ese paquete. Se podría pensar en esta declaración como un índice de contenido del paquete, sin sus detalles de construcción. El cuerpo del paquete contiene el código de todos los procedimientos y las funciones que se han nombrado en la especificación. Si la especificación tiene un nombre de procedimiento o de función y el código de este programa no se incluyera en el cuerpo, la compilación del cuerpo fallará con un error hasta que se escriba el código de este programa presente en la especificación. Dada esta condición, también se podría decir que puede haber construcciones dentro del cuerpo que no están declaradas en la especificación, y que a estas construcciones —que se denominan privadas— las utilizan otros programas que existen dentro del cuerpo del paquete. Ejemplo de un paquete que agrupe al código visto hasta ahora.

Inicio de la declaración del paquete "útiles_varios"

```
CREATE OR REPLACE PACKAGE útiles_varios IS
FUNCTION convertir_moneda
(cantidad                    IN NUMBER(12,3),
moneda_original  IN VARCHAR2,
moneda_destino  IN VARCHAR2)  RETURNS NUMBER(12,3);
/* - - - - - - - - - - - - - - - - - -separacion de programas - - - - - - - - - - - - - - - - */
PROCEDURE cambios_estudiantes2(p_legajo          IN NUMBER,
                                                 p_apellido                  IN
VARCHAR2,
                                                 p_nombre                    IN
VARCHAR2,                                                          p_resultado
OUT NUMBER );
END PACKAGE utiles_varios;
```

/

Inicio del cuerpo del paquete:

```
CREATE OR REPLACE PACKAGE BODY útiles_moneda IS
CREATE OR REPLACE FUNCTION convertir_moneda
(cantidad                IN NUMBER(12,3),
moneda_original  IN VARCHAR2,
moneda_destino  IN VARCHAR2)  RETURNS NUMBER(12,3)
IS
        conversion              NUMBER(12,3) := 0;
        coeficiente             NUMBER(6,3);
        datos_mal               EXCEPTION;
BEGIN                           /*inicio de la sección ejecutable de un bloque*/
                IF moneda_destino IS NULL OR moneda_original IS NULL THEN
                        RAISE datos_mal;
                END IF;
                SELECT coeficiente_conversión
                INTO coeficiente
                FROM cotizaciones
                WHERE origen = moneda_original
                AND destino = moneda_destino;
                conversion = cantidad * coeficiente;
                RETURN conversion;
EXCEPTION                       /*inicio de la sección de manejo de excepciones */
        WHEN no_data_found THEN
                DBMS_OUTPU.PUT_LINE('Combinación de monedas no prevista');
                RETURN conversion;
        WHEN datos_mal THEN
                DBMS_OUTPU.PUT_LINE('No ingresa datos de moneda');
                RETURN conversion;
END FUNCTION convertir_moneda;
/* - - - - - - - - - - - - - - - - - -separacion de programas - - - - - - - - - - - - - - - - */
PROCEDURE cambios_estudiantes2(p_legajo                 IN NUMBER,
                                                        p_apellido              IN VARCHAR2,
                                                        p_nombre                IN VARCHAR2,
                                p_resultado             OUT NUMBER ) AS
BEGIN
        UPDATE estudiantes
```

```
        SET nombre = p_nombre, apellido = p_apellido
        WHERE legajo = p_legajo;
        p_resultado := 1;
EXCEPTION
        WHEN no_data_found THEN
        p_resultado := 0;
        DBMS_OUTPUT.PUT_LINE('No existe el empleado con el legajo '|| p_legajo)
        WHEN others THEN
        p_resultado := -1;
END PROCEDURE cambios_estudiantes2;

END PACKAGE útiles_varios;
/
```

De esta manera, se describió la construcción de programación de PL/SQL que permite trabajar mejor las unidades de programación individuales, agrupándolas, simplificando su administración y dándoles mejor *performance*.

El último punto de este capítulo trata sobre las unidades de programación PL/SQL que se asocian a las tablas o a los eventos de sistema y tienen la característica de que no se ejecutan por el motor, sin necesidad de que se las convoque explícitamente por la aplicación o por un usuario determinado Se ejecutan si el evento para el que se definen acontece.

Su utilización se justifica por la dependencia que existe entre un objeto principal y otro derivado dentro de la aplicación y, cuando en aquél existe un cambio, el secundario, lo debe reflejar y se requiere que, implícitamente, se ejecute el código necesario para mantener la integridad de la información. Por ejemplo, esta dependencia marca que la aplicación que se encarga de la gestión de envíos de los productos comercializados, no puede procesar un registro hasta que la aplicación de cobranza no marque este pedido como pagado, y esto no será posible hasta que la aplicación de cobranzas no haya enviado la factura al cliente. Cada una de estas aplicaciones puede ir actualizando el valor de la columna estado en una tabla; sin embargo, si es necesario registrar la cantidad de días que llevará a la factura transformarse en un remito, se requerirá un procesamiento especial. Si se decide que habrá una tabla de historia con el pedido, el estatus inicial y el final, junto con la fecha de cambio, un trigger o disparador de base de datos será muy útil para procesar este trabajo.

4.14.2 Creando Triggers con PL/SQL

Los triggers o disparadores de base de datos se almacenan como objetos compilados en la base de datos y el código fuente de su cuerpo se puede consultar en el diccionario de la base de datos que lo almacena.

Un trigger se ejecuta automáticamente cuando cierto tipo de sentencias se ejecutan. Si existe un trigger para la sentencia UPDATE sobre una tabla determinada, cuando se actualice una de sus filas, se ejecutará, sin necesidad de convocarlo por su nombre, en forma implícita.

El tipo básico de trigger de base de datos es el que se encuentra en el nivel de la sentencia. Éste se ejecutará cada vez que una sentencia que lo active sea ejecutada sobre la tabla a la que el trigger está asociado. Como ejemplo, se puede definir que se quiere monitorear la actividad de borrado de la tabla pagos de estudiantes de tal forma que, cuando se borren cuotas de un estudiante, se guardará en otra tabla la información de la fecha de borrado y la identificación del usuario que realizó la operación. A continuación, se verá el código que realiza esta operación:

```
CREATE OR REPLACE TRIGGER bd_pagos_estudiantes
BEFORE delete ON pagos_estudiantes
BEGIN
        INSERT INTO auditoría_pagos_estud (usuario, fecha_hora_cambio,
                                                    motivo)
        VALUES (user, to_char(sysdate, 'dd/mm/yy hh:mi:ss','borrado de filas de
pagos_estudiantes');
END;
```

El nombre del trigger, elegido por el autor, tiene un prefijo 'BD_', que indica que es "Antes del Borrado" (Before Delete) porque, al ejecutarse automáticamente, un error en su ejecución genera un mensaje que trae el nombre del trigger que ha fallado. La práctica común de nombres ayuda al mantenimiento de la aplicación. Este trigger se ejecutará cuando se ejecute un DELETE sobre la tabla pagos_estudiantes y guardará en la tabla de Auditoría de pagos de estudiantes la acción de borrado, fecha y autor de la acción.

El otro tipo de trigger permitirá tener la información de cada fila borrada en la acción anterior; se trata de los triggers en el de la fila. Para esto, la sintaxis de la creación permite la definición del valor de las columnas antes de cada cambio y el valor después del cambio. A continuación, se verá el ejemplo de un trigger después de la actualización:

```
CREATE OR REPLACE TRIGGER bd_pagos_estudiantes
AFTER update ON pagos_estudiantes
REFERENCING OLD AS old NEW AS new
FOR EACH ROW
BEGIN
        INSERT INTO h_pagos_estudiantes (usuario, fecha_hora_cambio,
fecha_anterior, importe_anterior, fecha_posterior, importe_posterior)
        VALUES (user, to_char(sysdate, 'dd/mm/yy hh:mi:ss',:OLD.fecha,      :OLD.
importe, :NEW.fecha, :NEW.importe);
END;
```

En el código anterior, se observa que la segunda línea especifica el momento de ejecución, que es después de ejecutado; luego se determina el prefijo con el

que se definirán los valores anteriores y posteriores a la acción de las columnas y su contenido, en este caso, diferenciará el anterior del resultante por la operación UPDATE. A continuación, la línea que establecerá que este trigger se ejecutará una vez por cada fila actualizada. En este tipo de triggers, se deben considerar detenidamente la cantidad de proceso que se define, porque puede ser ejecutado miles de veces en una acción muy extendida por las filas de la tabla original.

Se pueden escribir doce triggers diferentes sobre una tabla, un grupo antes, otro después; uno en el nivel de la sentencia para cada acción DML (INSERT, UPDATE y DELETE); es decir: seis diferentes y, luego, el mismo cálculo para los que se encuentran en el nivel de la fila, hacen los seis restantes. Esta cantidad hace nuevamente llamar la atención que un trigger puede disparar otros triggers, imaginemos, definida sobre la tabla h_pagos_estudiantes, lo que constituiría un efecto llamada cascada de triggers. En sí esto no es un problema, pero debe ser diseñado adecuadamente, ya que podría darse que en esta cascada se afecte a la tabla que inicialmente disparó el evento.

Se finaliza, así, el estudio del lenguaje de extensión procedimental de SQL en las bases de datos Oracle, denominado PL/SQL, luego de mostrar las construcciones básicas, las anónimas y con nombre de procedimientos, las funciones, los paquetes y los triggers. Se anima al lector a que practique la creación de todos los objetos para que se asegure el entendimiento de los temas expuestos.