Referencia PL/SQL

1. Introducción al Procedural Language / SQL (PL/SQL)

El lenguaje PL/SQL es una "extensión" procedural del lenguaje SQL creada por Oracle para trabajar con la base de datos. La sintaxis del lenguaje está basada en ADA. Soporta todas las instrucciones DML de SQL (e.g. `SELECT`, `INSERT`, `UPDATE` y `DELETE`), instrucciones para el control de transacciones, control de cursores y añade estructuras de control condicional y bucles. Para utilizar el resto de instrucciones SQL (e.g. `CREATE TABLE`, etc.) es necesario utilizar *SQL dinámico* y paquetes de funciones proporcionados en el servidor como `DBMS_SQL`.

El concepto básico de PL/SQL y unidad mínima de programación es el bloque. De este modo, PL/SQL es un lenguaje estructurado en bloques. Los tipos de bloque soportados por PL/SQL son:

- Bloque anónimos.
- Subprogramas (bloques nombrados): procedimientos, funciones y paquetes.
- Disparadores.

Algunas características generales de PL/SQL:

- El lenguaje PL/SQL no es sensible a mayúsculas y minúsculas.
- Es necesario declarar los identificadores (e.g. variables, procedimientos, etc.) antes de que sean utilizados.

Ejemplo de bloque PL/SQL:

```
DECLARE
   Xnombre clientes.nombre%TYPE;
   Xarticulo ventas.art%TYPE;
BEGIN
    SELECT nombre, art INTO xnombre, xarticulo
      FROM clientes, ventas
      WHERE clientes.codigo = ventas.codigo AND
            ventas.fecha = sysdate-7;
    DBMS_OUTPUT.PUT_LINE ( 'Solo una venta de '|| xarticulo
                           ||' a: '||xnombre);
EXCEPTION
    WHEN NO_DATA_FOUND THEN   -- El SELECT no devolvió valores
       DBMS_OUTPUT.PUT_LINE ('No hay ventas hace una semana');
    WHEN TOO_MANY_ROWS THEN   -- SELECT devolvió más de un valor
       DBMS_OUTPUT.PUT_LINE ('Más de una venta hace una semana');
    WHEN OTHERS THEN
       DBMS_OUTPUT.PUT_LINE ('Error inesperado');
END;
/
```

1.1. Compilación

Todos los bloques PL/SQL deben incluir al final una fila con el carácter `'/'` para compilar y ejecutar. Si no se introduce el carácter `'/'` el editor SQL*Plus no identifica que se haya finalizado el bloque PL/SQL y por tanto no lo compila ni ejecuta.

Si no hay errores de compilación o ejecución se mostrará en la consola de SQL*Plus alguno de los siguientes mensajes:

```
PL/SQL procedure successfully completed.
ó
Procedimiento PL/SQL terminado correctamente.
```

En el caso de que el proceso de compilación falle SQL*Plus muestra los errores de compilación, incluyendo información acerca del problema y la posición del mismo. En el caso particular de subprogramas almacenados (e.g. `CREATE …`) los errores no se muestran directamente. Para examinar los errores de compilación y ejecución de un programa almacenado se utiliza la se utiliza la orden:

```
SHOW ERRORS;
```

1.2. Salida por consola e interacción con el usuario

Es posible mostrar datos por la consola de SQL*Plus desde un programa PL/SQL utilizando los procedimientos y funciones que se incluyen en el paquete `DBMS_OUTPUT` (ver sección 4.5.2). Para poder utilizar la consola es necesario activar el "eco" del buffer ejecutando la siguiente orden:

```
SET SERVEROUTPUT ON [FORMAT WRAPPED] [SIZE<tamaño>]
```

Aunque es posible trabajar con SQL*Plus, iSQL*Plus y SQL*Worksheet indistintamente para crear bloques PL/SQL, para interactuar con el usuario es recomendable utilizar SQL*Plus.

A través de SQL*Plus se ofrecen distintos mecanismos para poder interactuar con el usuario, habitualmente para pedirle información por teclado. Las instrucciones más relevantes para interactuar con el usuario son:

- `ACCEPT`. Permite leer un dato desde la entrada y almacenarlo en una variable.
- `PAUSE`. Muestra un texto y espera a que el usuario pulse INTRO.

Existen otros mecanismos adicionales para interactuar con el usuario como la sustitución de variables (ver *Writing Interactive Commands* en [7]) y las variables vinculadas (ver *Using Bind Variables* en [7]).

1.3. Referencias del apartado

Ver *PL/SQL Overview* (capítulo 1 y capítulo 14) en [1].
Ver *Overview of PL/SQL* en [2].
Ver *SHOW ERROS* en [4].
Ver *SET SERVEROUTPUT* en [4].
Ver *SQL*Plus Basics* en [4].
Ver *Using Scripts in SQL*Plus* en [4].
Ver *ACCEPT, PROMPT* y *PAUSE* en [4].

2. Bloques anónimos

Los bloques PL/SQL presentan una estructura específica compuesta de tres partes bien diferenciadas:

- `DECLARE…BEGIN`. En esta sección se declaran todas las constantes y variables que se van a utilizar en el bloque PL/SQL. Esta sección es opcional.
- `BEGIN…EXCEPTION/END`. Esta sección incluyen todas las instrucciones a ejecutar. Esta sección es obligatoria.
- `EXCEPTION…END`. La sección de excepciones en donde se definen los manejadores de errores que soportará el bloque PL/SQL. Esta sección es opcional.

De las anteriores, únicamente la sección de ejecución es obligatoria, que quedaría delimitada entre las cláusulas `BEGIN` y `END`. El esquema general de un bloque PL/SQL queda del siguiente modo:

```
[<<nombre del bloque>>]
[DECLARE]
  Declaración de:
    Variables.
    Constantes.
    Cursores.
    Variables de excepción para control de errores.
    Procedimientos / Funciones.
    ...
BEGIN
  Instrucciones PL/SQL
  Bloques PL/SQL anidados
  NULL; -- sentencia que no hace nada
[EXCEPTION]
  Control y tratamiento de errores.
  Punto al que se transfiere el control siempre que exista un
  problema.
END [nombre del bloque];
/
```

El anidamiento de bloques PL/SQL es una práctica habitual para llevar a cabo la gestión de errores (gestión de excepciones) dentro del código PL/SQL.

Un ejemplo simple de bloque anónimo PL/SQL es el siguiente:

```
DECLARE
  nombre_variable  VARCHAR2(5);
  nombre_excepcion EXCEPTION;
BEGIN
  SELECT 1
  INTO nombre_variable
  FROM DUAL;
EXCEPTION
  WHEN nombre_excepcion THEN
    NULL; -- Esta instrucción no hace nada
END;
/
```

2.1. Identificadores

Los identificadores, como en todo lenguaje de programación, se utilizan para nombrar a los objetos que se emplean en un programa: variables, constantes, nombres de procedimientos y funciones, etc. Las restricciones para los identificadores son:

- Deben empezar por letra. El resto del identificador puede contener caracteres alfanuméricos y caracteres especiales como '$', '#' o '_'.
- Pueden tener una longitud máxima 30 caracteres.

Al igual que otros lenguajes procedimentales, los identificadores de los objetos tienen un alcance y una visibilidad (ver *Scope and Visibility of PL/SQL Identifiers* en [4]).

2.2. Comentarios

En un bloque PL/SQL se admiten dos tipos de comentarios:

- Comentarios de una línea. Deben comenzar por `'--'`. Es posible añadir un comentario de línea al final de una instrucción.
- Comentarios de más de una línea. El comentario debe incluirse dentro de `'/* */'`.

2.3. Declaraciones de variables y constantes

La sintaxis para declara una variable es la siguiente:

```
<nombre de variable> <tipo> [NOT NULL] [:= valor inicial] ;
```

Es necesario tener en cuenta que:

- Las variables se inicializan por defecto al valor `NULL`.
- Cualquier variable declarada como **`NOT NULL`** tiene que ser inicializada con un valor.
- La inicialización puede incluir cualquier expresión legal de PL/SQL que devuelva un valor del tipo de la variable que se está declarando.
- No se pueden declarar varias variables en la misma sentencia.

La sintaxis para declarar una constante es similar al a declaración de una variable. En este caso es necesario definir el valor de la constante:

```
<nombre de constante> CONSTANT <tipo> := <valor> ;
```

2.3.1. Tipos de Datos

Al declarar el tipo de una variable o constante se puede utilizar los siguientes tipos de datos (ver *PL/SQL Datatypes* en [4]):

- Tipos de datos SQL: `INT`, `CHAR`, `VARCHAR`, etc.
- Tipos de datos nativos Oracle: `NCHAR`, etc.
- `BOOLEAN`. Puede tomar los valores `TRUE`, `FALSE`, `NULL`.
- Compuestos: arrays, cursores, "fila de tabla", construidos (colecciones y registros).
- Subtipos: `VARCHAR(3)`, `NUMBER(6,2)`, etc.
- `<id>%TYPE` y `<id>%ROWTYPE`.

Los pseudo-tipos `%TYPE` y `%ROWTYPE` permiten definir el tipo de una variable o constante dependiendo del tipo de otro elemento.

- `<id>%TYPE` permite definir el tipo de una variable a partir del tipo de otra variable o de una columna de una tabla de la BBDD cuyo identificador es `<id>`.
- `<id>%ROWTYPE` permite definir el tipo de una variable de tipo registro donde los tipos de los componentes del registro se toman de las columnas de una tabla de la BBDD.

```
v1 INT;
v2 v1%TYPE;                          -- Se toma el tipo de la variable v1
vNombre EMPLEADOS.nombre%TYPE; -- Se toma el tipo de la columna nombre
                                        de la tabla EMPLEADOS
```

2.4. Expresiones

Las expresiones PL/SQL constan de operandos y operadores. Los operandos de una expresión PL/SQL pueden ser valores literales (e.g. `2`, `'Hola'`, `DATE '2009-04-23'`, etc.), variables, constantes y llamadas a funciones.

Los operadores que se pueden incluir en las expresiones PL/SQL son similares a los operadores que se podían incluir en las expresiones SQL:

- Operadores aritméticos: `+`, `-`, `*`, `/`.
- Operadores lógicos: `AND`, `OR` y `NOT`.
- Operador de concatenación de cadenas: `'||'`.
- Operadores de comparación: `=`, `<>`, `!=`, `<`, `>`, `<=`, `>=`, `IS NULL`, `BETWEEN`, `LIKE`, `IN`.

Las funciones que se pueden incluir en las expresiones PL/SQL también son similares a las funciones que se podían incluir en SQL a excepción de las funciones de agregación (e.g. `COUNT`, `SUM`, etc.) solo pueden ser utilizadas dentro de una sentencia `SELECT`.

2.5. Instrucciones

2.5.1. Asignación

Existen dos mecanismos de asignación: utilizando el operador de asignación (:=) y utilizando la instrucción SELECT INTO. Ejemplo:

```
res := suma_tot / elems; -- Ejemplo de asignación

total_A INT;             -- Declaración de variable previa
...
SELECT COUNT(*) INTO total_A FROM TABLA1 WHERE NOM LIKE 'A';
```

2.5.2. Condicional

PL/SQL dispone de algunas estructuras de control que permiten controlar el flujo del programa.

```
IF condición THEN
  sentencia1; ... ; sentenciaN;
[ELSIF condición2 THEN
  sentencia1; ....; sentenciaN;]
[ELSE
  sentencia1; ... ; sentenciaN;]
END IF;
```

2.5.3. Bucles

El bucle LOOP ejecuta las instrucciones que contiene hasta que ejecuta: una instrucción EXIT o EXIT WHEN condición (y se cumple la condición).

```
LOOP
  sentencia1; ... ; sentenciaN;
  [ IF condici... THEN
      EXIT;
    END IF;
    [ EXIT WHEN condicion ]
END LOOP;
```

Bucles FOR:

```
FOR <contador> IN [REVERSE] <inicio>..<fin> LOOP
  sentencia1; ... ; sentenciaN;
END LOOP;
```

Bucles WHILE:

```
WHILE <condicion> LOOP
  sentencia1; ... ; sentenciaN;
END LOOP;
```

2.6. Control de errores: Excepciones

En PL/SQL la detección y gestión de condiciones de error se gestiona mediante el mecanismo de excepciones. Oracle incluye un conjunto de excepciones que identifican los errores en tiempo de ejecución de las instrucciones PL/SQL, pero también es posible definir excepciones propias.

Cuando ocurre un error durante la ejecución de una instrucción PL/SQL Oracle marca esta condición de error con una excepción, deteniendo la ejecución normal del bloque PL/SQL y transfiriendo el control a la sección EXCEPTION…END del bloque PL/SQL donde se incluirán los diferentes manejadores de excepciones. Si esta sección

gestiona adecuadamente la excepción, la ejecución del bloque termina, en otro caso la excepción se propaga transfiriendo el control al bloque PL/SQL que incluya al bloque que lanzó la excepción. Esta propagación de la excepción termina cuando algún bloque gestiona adecuadamente la excepción o en su defecto gestionada por el servidor (enviando a la consola los mensajes de error correspondientes).

Las excepciones predefinidas son lanzadas/levantadas automáticamente por el sistema, no obstante, es posible lanzar las excepciones definidas por el usuario con la instrucción `RAISE <id>`, donde `<id>` es el identificador de la excepción de usuario a lanzar.

Por ejemplo la instrucción `SELECT...INTO` lanza varias excepciones para marcar condiciones de error:

- Lanza `TOO_MANY_ROWS` si la instrucción devuelve más de una fila resultante.
- Lanza `NO_DATA_FOUND` si la instrucción no devuelve ninguna fila.

2.6.1. Gestión de Excepciones

Cuando se lanza una excepción el flujo de ejecución normal se transfiere a la sección `EXCEPTION…END` del bloque PL/SQL donde se ha lanzado la excepción. Esta sección contiene los manejadores de excepciones, donde su estructura es:

```
EXCEPTION
  WHEN Nombre_excepción1 THEN                          -- Manejador
    instrucción1; ...; instrucciónN;
  WHEN Nombre_excepción2 OR Nombre_excepción3 THEN -- Otro manejador
    instrucción1; ...; instrucciónN;
  ...
  WHEN OTHERS THEN  -- Manejador por defecto (es opcional). Si no
                       existe un manejador más específico se ejecutan
                       las instrucciones de este manejador.
    instrucción1; ...; instrucciónN;
END;
```

Una vez que las instrucciones del manejador son ejecutadas, la ejecución del bloque finaliza. Como posible instrucción dentro del manejador de excepciones se puede utilizar la instrucción `RAISE` (sin parámetro) para relanzar la excepción que se está actualmente gestionando.

El manejador `OTHERS` es el manejador por defecto, de modo que si no existe ningún manejador más específico para una excepción, ésta será tratada en este manejador.

Finalmente es posible gestionar varias excepciones a la vez en el mismo manejador de excepciones utilizando el operador **OR** entre los nombres de las excepciones.

Nótese que las excepciones no solo se pueden lanzar en el cuerpo de bloque PL/SQL sino que pueden ser lanzadas como resultado de una mala inicialización en una declaración o como resultado de la ejecución de alguna instrucción dentro de un manejador de excepciones. En ambos casos la excepción se propaga al bloque PL/SQL que contenga al bloque que ha lanzado la excepción.

Debido al comportamiento para la gestión de excepciones, cuando se terminan de ejecutar las instrucciones del manejador de excepciones correspondiente la ejecución del bloque termina. Si deseamos continuar con la ejecución del bloque podemos utilizar la capacidad de PL/SQL de permitir el anidamiento de bloques, de modo que simularemos los bloques `try-catch` que podemos encontrar en otros lenguajes de programación.

Ejemplo:

```
DECLARE
  resultado INTEGER := 1;
```

```
BEGIN
  BEGIN
    SELECT 1 / 0 INTO resultado
    FROM DUAL;
  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      resultado := -1;
  END;
  IF resultado > 0 THEN
  ELSE
    DBMS_OUTPUT.PUT_LINE('Error');
  END IF;
END;
/
```

Algunas excepciones predefinidas de Oracle son:

- NO_DATA_FOUND, TOO_MANY_ROWS. Son lanzadas por problemas relacionados con los cursores implícitos (Sección 3.1).
- INVALID_NUMBER, VALUE_ERROR, ZERO_DIVIDE. Son lanzadas por problemas relacionados con la evaluación de operaciones numéricas.
- CURSOR_ALREADY_OPEN, INVALID_CURSOR. Son lanzadas por problemas relativos a la gestión de cursores explícitos.

2.6.2. Excepciones de usuario

Es posible utilizar excepciones de usuario dentro de un bloque PL/SQL. Para utilizar una excepción de usuario es necesario: definir la excepción y lanzarla. La definición de una excepción se realiza dentro de la sección DECLARE del bloque PL/SQL y para lanzar la excepción se utiliza la instrucción RAISE.

```
DECLARE
  excepcion_propia EXCEPTION;
BEGIN
  RAISE excepcion_propia;
EXCEPTION
  WHEN excepcion_propia THEN
    DBMS_OUTPUT.PUT_LINE('Excepcion capturada');
END;
/
```

Las excepciones siguen las mismas reglas de visibilidad que el resto de identificadores de objetos de PL/SQL, es decir, son visibles en el propio bloque y en los bloques anidados. Cuando se lanza una excepción desde un bloque PL/SQL y ésta no es gestionada, la excepción se propaga al bloque superior o en su defecto al sistema (que mostrará un mensaje de error genérico).

Nótese que, ya que la excepción definida por el usuario sólo tiene ámbito local al bloque donde se lanza, el único manejador que se puede utilizar es el manejador OTHERS. Por ejemplo dado el siguiente subprograma almacenado:

```
CREATE PROCEDURE problemas1 ( ) AS
  x EXCEPTION;
BEGIN
  RAISE x;
END problemas1;
```

Cualquier bloque PL/SQL que llame a este procedimiento almacenado y quiera gestionar la excepción de aplicación debe hacerlo a través del manejador OTHERS.

```
BEGIN
  problemas1();
EXCEPTION
  WHEN OTHERS THEN
    instrucción1; ...; instrucciónN;
END;
```

Existe otro mecanismo para definir excepciones usuario utilizando el procedimiento RAISE_APPLICATION_ERROR. La ventaja de utilizar este procedimiento es que la aplicación puede asignar un código de error ORA-XXXXX y un mensaje descriptivo. El procedimiento RAISE_APPLICATION_ERROR(num_error, mensaje) tiene los siguientes parámetros:

- num_error. Código de error que el desarrollador asigna a la excepción de usuario. Este código debe estar dentro del rango -20000 .. -20999.
- mensaje. Mensaje de texto asignado a la excepción. El tamaño máximo del mensaje es de 2048 bytes.

El uso de RAISE_APPLICATION_ERROR es particularmente útil en los subprogramas almacenados, ya que permite definir un conjunto de errores de aplicación descriptivos. Por ejemplo, se puede definir una excepción para especificar que se ha violado una dependencia funcional particular:

```
CREATE PROCEDURE compruebaDFs ( ) AS
BEGIN
  RAISE_APPLICATION_ERROR(-20101, 'La DF1 A->B se ha violado');
END problemas2;
```

También es posible gestionar las excepciones de aplicación lanzadas con RAISE_APPLICATION_ERROR utilizando el manejador OTHERS, sin embargo, también es posible asociar el código de error particular que ha definido el desarrollador con una declaración de excepción utilizando la directiva PRAGMA INIT_EXCEPTION. A continuación se muestra un bloque PL/SQL que captura la excepción del anterior procedimiento almacenado:

```
DECLARE
  df1_violada EXCEPTION;
  /* Mapea el código de error que devuelve RAISE_APPLICAITON_ERROR
     a una excepción definida por el usuario
  */
  PRAGMA EXCEPTION_INIT(df1_violada, -20101);
BEGIN
  compruebaDFs();
EXCEPTION
  WHEN df1_violada THEN
    instrucción1; ...; instrucciónN;
END;
```

2.7. Referencias del Apartado

Ver *PL/SQL Datatypes* en [4].

Ver *Selecting Datatypes* en [2].

Ver *Native Datatypes* en [1].
Ver *Handling PL/SQL Errors* en [4].

3. Cursores

Los cursores representan una referencia a una zona de la memoria de trabajo de Oracle utilizada para almacenar toda la información necesaria para ejecutar una instrucción SQL (ver *Cursors* en [1]). Existen dos tipos de cursores: implícitos y explícitos. Los cursores implícitos son cursores que utiliza implícitamente Oracle para ejecutar las instrucciones de manipulación de datos incluidas las consultas que devuelven una única fila. Los cursores explícitos son necesarios para consultas en las que se devuelva más de una fila.

3.1. Cursores implícitos

El caso más habitual de cursor implícito es el que cursor que se crea automáticamente al utilizar la instrucción `SELECT…INTO`. La estructura de esta instrucción es la siguiente:

```
SELECT columna1, …, columnaN
INTO variable1, …, variableN
FROM <tablas/Vistas>
WHERE
  <Predicados>
```

Esta instrucción es una consulta `SELECT` habitual, donde el resultado de la consulta se almacena en un conjunto de variables que hayan sido declaradas en el bloque PL/SQL.

Hay que asegurarse que la consulta devuelve una y solo una fila (mínimo una fila y máximo una fila). Si no se cumple esta restricción el cursor implícito lanzará una excepción en cada caso:

- `NO_DATA_FOUND`. Se lanza si la consulta devuelve un resultado vacío.
- `TOO_MANY_ROWS`. Se lanza si la consulta devuelve más de una fila.

3.1.1. Atributos de los cursores implícitos

Los atributos de los cursores implícitos devuelven información relativa a la ejecución de una instrucción SQL `INSERT`, `DELETE`, `UPDATE` o `SELECT…INTO`. Para los cursores implícitos los valores siempre hacen referencia a la última instrucción que se haya ejecutado. Se pueden acceder a los atributos de un cursor implícito utilizando la sintaxis `SQL%<atributo>`, ejemplo:

```
DELETE FROM EMPLEADOS WHERE emp_id = mi_empleado;
IF SQL%FOUND THEN -- La operación de borrado ha tenido éxito
...
END IF;
```

Los atributos que están disponibles para los cursores implícitos tienen valor una vez que se ha ejecutado la instrucción SQL correspondiente, hasta ese momento su valor es `NULL`. Los atributos disponibles y su significado es el siguiente:

- `%FOUND`. Tras ejecutar una instrucción `INSERT`, `DELETE` o `UPDATE` que afecte a una o más instrucciones o una `SELECT…INTO` que devuelva una o más filas `%FOUND` devuelve `TRUE`. En otro caso devuelve `FALSE`.
- `%ISOPEN`. Devuelve `FALSE` siempre para cursores implícitos.
- `%NOTFOUND`. Es el opuesto de `%FOUND`.

- %ROWCOUNT. Devuelve el número de filas afectadas por una instrucción INSERT, DELETE o UPDATE. Si SELECT…INTO devuelve más de una fila se lanza la excepción TOO_MANY_ROWS y %ROWCOUNT devuelve 1.

3.2. Cursores explícitos

Los cursores explícitos son utilizados para procesar el resultado de consultas que devuelven un número indefinido de filas (i.e. 0, 1 o ≥ 1). Los pasos necesarios para utilizar cursores explícitos son:

1. Declarar la variable cursor e inicialización con una consulta (puede estar parametrizada), declaración de variables donde se almacenarán los resultados del cursor.
2. Abrir el cursor (OPEN).
3. Lectura de las filas una a una y procesamiento de las mismas (FETCH). Habitualmente se implementa como un bucle sobre el cursor.
4. Cierre del cursor (CLOSE).

Ejemplo: Dada la tabla MOTOS(MARCA,MODELO,CC,AÑO,EXISTENCIAS), muestra por pantalla la marca y modelo de las motocicletas de al menos 500 cc".

```
DECLARE
  CURSOR cmotos IS
    SELECT marca, modelo
    FROM motos
    WHERE cc>=500;
  -- Variables auxiliares para el cursor
  Xmarca   motos.marca%TYPE;
  Xmodelo  motos.modelo%TYPE;
BEGIN
  OPEN cmotos;
  FETCH cmotos INTO xmarca, xmodelo;
  WHILE cmotos%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE(xmarca || ' ' || xmodelo);
    FETCH cmotos INTO xmarca, xmodelo;
  END LOOP;
  CLOSE cmotos;
END;
/
/* Cuerpo alternativo con bucle LOOP
BEGIN
  OPEN cmotos;
  LOOP
    FETCH cmotos INTO xmarca, xmodelo;
    EXIT WHEN cmotos%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(xmarca || ' ' || xmodelo);
  END LOOP;
  CLOSE cmotos;
END;
*/
```

Existe una versión especial de bucle FOR denominada "Cursor FOR LOOP" para iterar sobre un cursor explícito:

```
-- Cursor FOR LOOP
DECLARE
  CURSOR cmotos IS
    SELECT marca, modelo
    FROM motos
    WHERE cc>=500;
  -- No necesitamos declarar variables de almacenaje.
```

```
BEGIN
  FOR motos_reg IN cmotos LOOP
    DBMS_OUTPUT.PUT_LINE(motos_reg.marca || ' ' || motos_reg.modelo);
  END LOOP;
  -- No hace falta ni abrir ni cerrar el cursor.
END;
/
```

Para declarar las variables donde se almacenarán los valores que se obtendrán del cursor es habitual hacer uso de de `%TYPE` y `%ROWTYPE` para no tener que especificar explícitamente el tipo de la variable, de modo que:

- `%TYPE` se utiliza para obtener el tipo de una columna concreta de alguna de las tablas/vistas que está involucrada en la consulta que forma parte del cursor.
- `%ROWTYPE` se utiliza para obtener el tipo de una fila completa de alguna de las tablas/vistas que está involucrada en la consulta que forma parte del cursor. Habitualmente se utiliza si el cursor se inicializa con una consulta `SELECT *`.

Como se ha mencionado, un cursor puede parametrizarse, de modo que el parámetro formal puede formar parte de la consulta que inicializa el cursor. Los parámetros actuales para el cursor se especifican a la hora de abrir el cursor o el cursor puede utilizar el valor por defecto que se haya especificado.

```
DECLARE
  /* El cursor tiene un parámetro cilindrada que puede tener un valor
     por defecto. */
  CURSOR cmotos (cilindrada INTEGER DEFAULT 125) IS
    SELECT marca, modelo
    FROM motos
    WHERE cc>=cilindrada;
BEGIN
  /* Cuando se abre el cursor se puede dar un valor específico al
     parámetro o dejar que se utilice el valor por defecto. */
  OPEN cmotos(500);
  FETCH cmotos INTO xmarca, xmodelo;
  WHILE cmotos%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE(xmarca || ' ' || xmodelo);
    FETCH cmotos INTO xmarca, xmodelo;
  END LOOP;
  CLOSE cmotos;
END;
/
/* Cuerpo alternativo con bucles Cursor FOR LOOP
BEGIN
  -- El parámetro actual se puede especificar en el IN del bucle
  FOR motos_reg IN cmotos(500) LOOP
    DBMS_OUTPUT.PUT_LINE(motos_reg.marca || ' ' || motos_reg.modelo);
  END LOOP;
END;
*/
```

3.2.1. Atributos de los cursores explícitos

Al igual que con los cursores implícitos, los cursores explícitos tienen una serie de atributos asignados como los atributos `%FOUND` y `%NOTFOUND` utilizados en el apartado anterior. Los atributos disponibles en un cursor explícito y su significado es el siguiente:

- %FOUND. Desde que el cursor es abierto hasta que se obtiene la primera fila el valor de %FOUND es NULL. Tras obtener una fila %FOUND devuelve TRUE si realmente se ha podido obtener una fila FALSE en otro caso.
- %ISOPEN. Devuelve TRUE si el cursor se ha abierto y FALSE en otro caso.
- %NOTFOUND. Es el opuesto de %FOUND.
- %ROWCOUNT. Antes de que se obtenga alguna fila %ROWCOUNT devuelve el valor 0, después de que se haya obtenido alguna fila devuelve el número de filas que se han obtenido hasta el momento (es decir el número de FETCH exitosas hasta el momento).

3.3. Referencias del Apartado

Ver *Cursors* en [1].
Ver *Using Cursors within Applications* en [2].
Ver *Cursors* en [4].
Ver *Managing Cursors*, *Using Cursors FOR Loops* y *Using Cursors Variables* en [4].

4. Procedimientos, funciones y paquetes

En el lenguaje PL/SQL es posible definir subprogramas (procedimientos y funciones) similares a los subprogramas en los lenguajes. Adicionalmente es posible crear agrupaciones de procedimientos, funciones y variables que son denominadas paquetes. De manera simple, los subprogramas pueden verse como bloques PL/SQL con nombre que pueden incluir adicionalmente una serie de parámetros de entrada y salida.

4.1. Procedimientos

La sintaxis genérica para definir un procedimiento es:

```
PROCEDURE <nombre> [(<parámetrol>, ... <parámetroN>)] IS
   [<declaraciones de variables, constantes, ..>]
BEGIN
   sentencias;
[EXCEPTION
   manejadores de excepciones;]
END [<nombre>];
```

Los parámetros de un procedimiento se especifican indicando su nombre, modo y tipo. El nombre del parámetro sigue las mismas reglas que los identificadores, el modo especifica si el parámetro es de entrada (IN, valor por defecto), salida (OUT) o entrada/salida (IN OUT) y finalmente el tipo especifica el tipo de datos del parámetro (NOTA: no puede ser un subtipo). La sintaxis general para definir los parámetros formales de un procedimiento queda del siguiente modo:

```
<nombre> [IN | OUT | IN OUT] <tipo>
```

4.2. Funciones

La sintaxis genérica para definir una función queda es:

```
FUNCTION <nombre> [(<parámetrol>, ... <parámetroN>)] RETURN <tipo> IS
   [<declaraciones de variables, constantes, ..>]
BEGIN
   sentenci
   RETURN v      -- La función devuelve el valor de la variable v.
[EXCEPTION
   manejadores de excepciones;]
```

```
END [<nombre>];
```

Al igual que los procedimientos, las funciones pueden especificar parámetros formales. Aunque el lenguaje PL/SQL permite especificar los parámetros de una función con los modos `OUT` o `IN OUT`, siguiendo las buenas prácticas de programación de los lenguajes procedimientales, una función no utiliza los modos `OUT` o `IN OUT` para sus parámetros formales.

Adicionalmente si se desea que la función pueda ser utilizada en una instrucción SQL es necesario que la implementación de la función siga las siguientes reglas para evitar los efectos laterales:

- Si la función se utiliza en una sentencia `SELECT` la función no debe modificar ninguna tabla.
- Si la función es llamada desde una sentencia `INSERT`, `UPDATE`, o `DELETE` la función no debe modificar ninguna de las tablas que se estén modificando con la sentencia.
- Si la función es llamada desde cualquier sentencia, la función no puede incluir ninguna sentencia de control de transacción, gestión de sesión, sentencia DCL o sentencia DDL.

4.3. Uso de procedimientos y funciones

Al igual que las variables, es necesario definir los procedimientos y funciones antes de poder utilizarlos. Los procedimientos y funciones pueden definirse al final de la sección de declaraciones de un bloque PL/SQL, de modo que estos procedimientos y funciones pueden ser utilizados en el cuerpo del bloque.

Una vez declarados, los procedimientos pueden se llamados desde el cuerpo del bloque PL/SQL correspondiente. Además las funciones también pueden ser llamadas o bien desde el cuerpo de un bloque PL/SQL o como parte de una sentencia SQL.

```
DECLARE
  ...
  variable ...;
  PROCEDURE procedimiento (...) IS
  BEGIN
  ...
  END;
  FUNCTION funcion (...) RETURN ... IS
  BEGIN
  ...
  END;
BEGIN
  procedimiento(...);
  ...
   variable := funcion(...);
  ...
END;
```

4.4. Subprogramas almacenados

Los procedimientos y funciones definidos en los apartados anteriores residen habitualmente en ficheros de texto (ficheros .sql). Sin embargo, es posible crear procedimientos y funciones que estén almacenados en el servidor.

Los procedimientos y funciones almacenados tiene la ventaja de que pueden ser llamados desde las aplicaciones cliente, por ejemplo, utilizando el lenguaje Java, sin necesidad de conocer la implementación del procedimiento o función.

Para crear un procedimiento o función almacenado, simplemente hay que anteponer CREATE a la definición del procedimiento o función. Si añadimos OR REPLACE se reemplaza el procedimiento o función almacenada que existiera en el servidor.

```
CREATE [OR REPLACE] PROCEDURE ...
CREATE [OR REPLACE] FUNCTION ...
```

Para poder crear un procedimiento o función almacenada es necesario que el usuario tenga asignado el privilegio de sistema CREATE PROCEDURE.

También es posible recompilar un subprograma almacenado utilizando la sentencia ALTER:

```
ALTER FUNCTION <nombre> COMPILE;
ALTER PROCEDURE <nombre> COMPILE;
ALTER PACKAGE <nombre> COMPILE ↲
  SPECIFICATION; -- Recompila la declaración;
  BODY;          -- Recompila la implementación del paquete
  PACKAGE;       -- Recompila la declaración y la implementación
```

Finalmente para eliminar un subprograma almacenado se utiliza la sentencia DROP con

```
DROP FUNCTION <nombre>;
DROP PROCEDURE <nombre>;
DROP PACKAGE [BODY] <nombre>;  -- Si se especifica BODY solo se
                                  elimina la implementación.
```

Para evitar problemas con la creación de funciones y procedimientos almacenados es recomendable primero probar los procedimientos y funciones en un bloque PL/SQL anónimo (siguiendo la estructura descrita en 4.3). Además otro problema de crear procedimientos y funciones almacenadas es que no se muestran directamente los errores de compilación, sino que es necesario utilizar la instrucción SHOW ERRORS.

4.5. Paquetes

De manera similar al concepto de módulo en los lenguajes procedimentales, es posible definir paquetes en PL/SQL que permiten agrupar variables, procedimientos y funciones PL/SQL.

Los paquetes son subprogramas almacenados, de modo que no se puede no se pueden definir en la sección de declaraciones de un bloque PL/SQL para probarlos. Por tanto, es recomendable probar los procedimientos y funciones del paquete por separado antes de crear el paquete.

La creación de un paquete se divide en dos partes:

- Especificación. La especificación representa la sección de declaraciones del paquete. Esta sección incluirá la declaración de variables del paquete y la declaración (solo cabeceras) de procedimientos y funciones del paquete.
- Implementación. Se especifican los bloques PL/SQL para implementar los procedimientos y funciones. Adicionalmente también se pueden declarar variables que solo serán accesibles desde la implementación del paquete.

La estructura genérica para la declaración de un paquete es:

```
CREATE [OR REPLACE] PACKAGE <nombre> IS
   /* Declaraciones públicas:
         - variables, constantes, excepciones, cursores.
         - cabeceras de procedimientos y funciones.*/
END <nombre>;
/
```

La estructura genérica para la implementación de un paquete es:

```
CREATE [OR REPLACE] PACKAGE BODY <nombre> IS
   -- Declaraciones privadas
   <Implementación de procedimientos y funciones públicas>
[BEGIN
   <código de inicialización>
   -- Se ejecuta la 1ª vez que se invoca el paquete]
END <nombre>;
/
```

Una vez que se ha creado un paquete, para acceder a los elementos del paquete simplemente hay que anteponer el nombre del paquete al del objeto que se desea acceder.

```
BEGIN
  nombre_paquete.procedimiento(...);
END;
/
```

Un ejemplo mínimo de declaración de paquete es el siguiente:

```
CREATE OR REPLACE PACKAGE paqueteSimple IS
  constante CONSTANT INTEGER := 0;
  PROCEDURE holaMundo;
END paqueteSimple;
/
```

Finalmente la implementación del paquete mínimo es la siguiente:

```
CREATE OR REPLACE PACKAGE BODY paqueteSimple IS
  var_interna INTEGER;
  PROCEDURE holaMundo IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Hola mundo');
  END holaMundo;
BEGIN
  var_interna := 0;
  DBMS_OUTPUT.PUT_LINE('Inicializando paquete');
END paqueteSimple;
/
```

4.5.1. Paquetes preinstalados en Oracle

Oracle incorpora una extensa librería de funcionalidades preinstaladas en forma de paquetes (ver [5] para documentarse de todos los paquetes). Algunos de los paquetes incorporados en el servidor para el desarrollo de aplicaciones son:

- `DBMS_OUTPUT`. Permite gestionar el buffer para intercambio de información con procedimientos, funciones o el terminal del intérprete.
- `DBMS_MVIEW`. Permite gestionar las vistas materializadas, por ejemplo, para poder refrescar una vista materializada.
- `DBMS_PIPE`. Permite la comunicación entre sesiones. Una sesión es una conexión de una aplicación cliente (como SQL*Plus) al servidor de base de datos para un usuario específico.
- `DBMS_ALERT`. Permite para enviar notificar alertas a los usuarios.
- `DBMS_LOCK` y `DBMS_TRANSACTION`. Permiten para gestionar bloqueos y transacciones.

Algunos paquetes incorporados en el servidor para la gestión del servidor son:

- `DBMS_SESSION`. Permite gestionar las sesiones al DBA.
- `DBMS_SYSTEM`. Permite establecer eventos utilizados para la depuración.

- DBMS_SPACE y DBMS_SHARED_POOL. Permite obtener información del espacio y reservar recursos en el pool compartido.
- DBMS_JOB. Permite planificar trabajos en el servidor.

4.5.2. Paquete DBMS_OUTPUT

Este paquete incluye procedimientos y funciones para gestionar el buffer de intercambio de información de procedimientos y funciones con la consola del intérprete. Las funciones más relevantes que se incluyen dentro del paquete son:

- PUT. Coloca una cadena de caracteres en el buffer.
- PUT_LINE. Coloca una cadena en el buffer como una nueva línea
- NEW_LINE. Inserta un salto de línea en buffer (después de un PUT)
- GET_LINE. Obtiene una línea del buffer.
- GET_LINES. Obtiene varias líneas del buffer.

Para activar el "eco" del buffer (salida por consola) en SQL*Plus (SQL*Worksheet y Developer incluidos), se debe introducir la orden:

```
SET SERVEROUTPUT ON [FORMAT WRAPPED] [SIZE <tamaño>]
-- Ejemplo SET SERVEROUTPUT ON SIZE 10000
```

La instrucción SET SERVEROUTPUT ON activa el eco de la sesión durante la sesión actual. Es recomendable que esta instrucción sea la primera instrucción de nuestro programa PL/SQL.

4.6. Referencias del apartado

Ver *Using Procedures and Packages* en [2].
Ver *PL/SQL Subprograms* en [4].
Ver *PL/SQL Packages* en [4].

5. Transacciones

Una transacción es una unidad de trabajo que incluye una o más instrucciones SQL que son ejecutadas por un único usuario. En Oracle, y según el estándar SQL 92, una transacción comienza con la primera instrucción SQL de usuario ejecutable. Una transacción termina cuando es confirmada o rechazada explícitamente por el usuario.

Las transacciones permiten a los usuarios garantizar modificaciones consistentes sobre los datos siempre que las instrucciones SQL de la transacción estén agrupadas lógicamente. Una transacción debe contener las instrucciones SQL necesarias para mantener la consistencia de la BBDD, de modo que los datos de tablas referenciadas están en un estado consistente antes y después de una transacción.

El lenguaje PL/SQL permite controlar la transacción activa mediante las instrucciones:

- COMMIT. Confirma la transacción activa y por tanto se realizan todas las modificaciones pendientes en la BBDD.
- ROLLBACK [<id_savepoint>]. Rechaza la transacción activa y por tanto se deshacen los cambios realizados en la transacción. Si se especifica <id_savepoint> se rechanzan los cambios hasta el punto marcado en la transacción con identificador <id_savepoint>.
- SAVEPOINT <id_savepoint>. Permite marcar un punto de la transacción con el identificador <id_savepoint>.

Además también es controlar el comportamiento de la transacción con el comando SET TRANSACTION, algunos parámetros para este comando son:

- `READ ONLY`. Configura la transacción en modo sólo lectura (no se permiten modificaciones de la BBDD) y se consigue el nivel de aislamiento lectura repetible.
- `ISOLATION LEVEL (SERIALIZABLE | READ COMMITED)`. Configura la transacción en los niveles de aislamiento serializable o lectura comprometidas respectivamente.

```
-- Finaliza la transacción previa
COMMIT;
-- Configura la transacción como solo lectura
SET TRANSACTION READ ONLY;
SELECT ... FROM Empleados WHERE ...
SELECT ... FROM Departamento WHERE ...
/*Obtenemos los mismos resultados que la consulta anterior ya que
las transacciones de solo lectura obtienen el nivel de aislamiento
de lectura repetible */
SELECT ... FROM Empleados WHERE ...
COMMIT;  -- Finaliza la transacción aunque no se modifique la BBDD
```

5.1. Referencias del apartado

Ver *User Processes Overview* en [1].
Ver *Transactions Overview* en [1].
Ver *Data Concurrency and Consistency* en [1].
Ver *Transaction Control Statements* en [1].
Ver *Control of Transactions* en [1].
Ver *Transaction Management* en [1].
Ver *Data Concurrency and Consistency* en [1].
Ver *How Oracle Processes SQL Statements* en [2].
Ver *Transaction Control* en [4].
Ver *Overview of Transaction Processing in PL/SQL* en [4].
Ver *Retrying a Transaction* en [4].

6. Disparadores

Los disparadores son bloques PL/SQL almacenados que son ejecutados automáticamente cuando una tabla es modificada a través de las instrucciones DML `INSERT`, `DELETE` y `UPDATE`. Aunque no se describe en esta sección, Oracle permite crear disparadores para gestionar eventos relacionados con acciones de usuario sobre la BBDD (ejecución de instrucciones DDL) o incluso eventos del servidor (arranque o parada del servidor). Los disparadores pueden ser utilizados para múltiples finalidades, algunas de ellas son:

- Gestión de atributos derivados y redundancias en tablas.
- Hacer respetar restricciones complejas de seguridad.
- Gestión de la integridad de los datos de la tabla, por ejemplo, hacer cumplir las DF asociadas a una tabla.
- Proporcionar un mecanismo de auditoría de las tablas.
- Actualizaciones en cascada (Oracle no tiene `ON UPDATE CASCADE` para las `FOREIGN KEY`).

Aunque los disparadores son muy versátiles, hay que utilizar solo cuando sean necesarios ya que el uso excesivo de disparadores puede dificultar el mantenimiento y gestión. Nótese que cuando se ejecuta un disparador, las instrucciones SQL incluidas en el bloque PL/SQL del mismo disparador pueden disparar la ejecución de otro disparador distinto.

Un disparador está compuesto de tres partes básicas:

- Declaración de la instrucción o instrucciones que generarán un evento que dispare el disparador. Adicionalmente se especifica el instante en el que se ejecutará el disparador.
- Una restricción. Es una expresión Booleana que debe ser cierta para que se ejecute el disparador. La restricción habitualmente está asociada a un disparador a nivel de fila.
- Una acción. Al igual que los subprogramas almacenados, la acción de un disparador puede contener instrucciones PL/SQL incluyendo la llamada a procedimientos y funciones almacenadas.

Ejemplo de disparador:

```
/**
 * Disparador que vigila el stock del almacen.
 * Si el stock está por debajo del nivel mínimo se añade un nuevo
 * pedido (si no se ha solicitado ya).
 */
CREATE OR REPLACE TRIGGER vigilar_almacen
/**
 * Evento de disparo del disparador.
 * El disparador se ejecuta después de modificarse la columna 'stock'
 * de la tabla 'almacen' cuando se ha ejecutado una instrucción
 * UPDATE.
 */
AFTER UPDATE OF stock ON almacen
/**
 * Restricción de aplicación del disparador.
 * El disparador sólo se ejecuta si el stock resultante del UPDATE
 * está por debajo del mínimo.
 */
FOR EACH ROW
WHEN (new.stock < new.stock_minimo)
-- Disparador a nivel de fila
/**
  * Acción del disparador.
  * Si los disparadores son del tipo fila (FOR EACH ROW) el bloque
  * PL/SQL puede acceder a los valores de las filas que son procesadas
  * por el disparador.
  */
DECLARE
  pedido_realizado NUMBER;
BEGIN
  SELECT count(*) INTO pedido_realizado FROM pedidos
  WHERE codigo_producto = :NEW.codigo_producto;
  IF pedido_realizado = 0 THEN
    INSERT INTO pedidos
    VALUES (:NEW.codigo_producto, :NEW.pedido_minimo, SYSDATE);
  END IF;
END;
```

6.1. Tipos de disparadores

Oracle permite la creación de los siguientes tipos de disparadores:

- Disparadores a nivel de fila (`FOR EACH ROW`).
- Disparadores a nivel de instrucción.
- Disparadores `INSTEAD OF`.

6.1.1. Disparadores a nivel de fila (`FOR EACH ROW`)

Los disparadores a nivel de fila (declarados `FOR EACH ROW`) ejecutan la acción asociada al disparador un número de veces igual al número de filas afectadas por la instrucción que ha generado el evento de disparo. Por ejemplo, una instrucción `UPDATE` puede modificar múltiples filas de una tabla, de modo que la acción del disparador se ejecuta una vez por cada una de las filas afectadas por la instrucción `UPDATE`.

Los disparadores a nivel de fila son útiles cuando el bloque PL/SQL asociado a la acción del disparador depende de los valores proporcionados por la instrucción que genera el evento, por ejemplo, para generar valores para atributos derivados.

6.1.2. Disparadores a nivel de instrucción

Los disparadores a nivel de instrucción ejecutan la acción asociada al disparador una vez por instrucción independientemente del número de filas afectadas por la instrucción.

Los disparadores a nivel de instrucción son útiles en aquellos casos en los que el bloque PL/SQL asociado no dependa de los datos que proporciona la instrucción que ha generado el evento, en particular este tipo de disparadores pueden ser utilizados para:

- Realizar tareas de auditoría.
- Realizar comprobaciones complejas de seguridad.

6.1.3. Disparadores `INSTEAD OF`

Los disparadores `INSTEAD OF` ofrecen un mecanismo transparente para modificar vistas utilizando instrucciones DML `INSERT`, `UPDATE` y `DELETE`. Estos disparadores son denominados "en vez de" ya que Oracle ejecuta la acción asociada al disparador en vez de la instrucción que ha generado el evento.

6.2. Diseño, creación y gestión de disparadores

Es recomendable tener en cuenta los siguientes criterios a la hora de definir disparadores:

- Los disparadores deben ser utilizados para garantizar que siempre que se ejecute una instrucción se ejecuten las acciones asociadas al disparador.
- Los disparadores deben utilizarse solo si Oracle no ofrece otro mecanismo para realizar la misma operación. Por ejemplo, un disparador no debería de crearse para realizar tareas que se pueden llevar a cabo automáticamente como la gestión de la integridad referencial si se puede utilizar la restricción `FOREIGN KEY`.
- Limitar el tamaño del disparador. Si la extensión del disparador es > 60 líneas, es recomendable crear un procedimiento almacenado y llamarlo desde la acción del disparador.

Para crear un disparador es necesario que el usuario tenga asignado el permiso `CREATE TRIGGER`.

La sintaxis genérica para crear un disparador es la siguiente:

```
CREATE [OR REPLACE] TRIGGER <nombre>
/**
 * Controla cuando se ejecuta la acción del disparador:
 * - BEFORE: la acción se ejecuta antes que la propia instrucción.
 * - AFTER: la acción se ejecuta después de la propia instrucción.
 */
  { BEFORE | AFTER }
/**
 * Instrucción o instrucciones que provocan la ejecución del
 * disparador.
 * - En el caso de la instrucción UPDATE se puedes especificar
 *   una lista con las columnas específicas monitorizar.
 */
  { DELETE | INSERT | UPDATE [OF column [,column] ...] }
    [OR
     { DELETE | INSERT | UPDATE [OF column [,column] ...] }
    ] ...
/**
 * Tabla a la que se asocia el disparador
 */
  ON nombre_tabla
/**
 * Especificación de disparador a nivel de fila.
 * En este caso tiene sentido especificar una condición que debe
 * debe ser cierta para que la acción del disparador se ejecute.
 * - La condición pueden hacer referencia a las variables :OLD y :NEW.
 * - La condición no puede contener subconsultas.
 * - La condición se evalúa por cada fila afectada.
 */
[FOR EACH ROW [WHEN (<condición>)]]
DECLARE
  ...
BEGIN
-- Código asociado a la acción del disparador
END;
```

En ocasiones, por ejemplo, al cargar datos con SQL*Loader, puede ser necesario desactivar algún disparador concreto o desactivar los disparadores asociados a una tabla:

```
ALTER TRIGGER <nombre_disparador> {DISABLE | ENABLE};
ALTER TABLE <nombre_tabla> {ENABLE | DISABLE} ALL TRIGGERS;
```

Es posible consultar los disparadores que ha creado un usuario a través de la vista `USER_TRIGGERS` del diccionario de datos, por ejemplo:

```
SELECT trigger_type, table_name, triggering_event
FROM USER_TRIGGERS
WHERE trigger_name = '<nombre del disparador>';
```

Al igual que los subprogramas almacenados, también es posible eliminar un disparador:

```
DROP TRIGGER <nombre>;
```

6.2.1. Implementación de la acción de un disparador

El bloque PL/SQL asociado a la acción de un disparador tiene algunas restricciones y extensiones respecto a un bloque PL/SQL normal. Las restricciones asociadas a la implementación de un disparador son:

- No es posible acceder a la tabla asociada al disparador: error de tablas mutantes (ORA-04091). Esta restricción aplica a disparadores del tipo `FOR EACH ROW` y a disparadores de instrucción ejecutados como resultado de un modificador `DELETE CASCADE`.
- No se admiten sentencias de control de transacciones (`COMMIT`, etc.). Además, los procedimientos que sean llamados desde el bloque PL/SQL tampoco pueden utilizar instrucciones de control de transacción.
- No es posible modificar información de los campos de la clave primaria.
- No es posible declarar variables de tipo LONG.

Adicionalmente puesto que un mismo disparador puede asociarse a varios eventos producidos por diferentes instrucciones SQL, es posible diferenciar la fuente del evento en el cuerpo del disparador utilizando los predicados `INSERTING`, `UPDATING` y `DELETING`, ejemplo:

```
IF INSERTING THEN
  op := 'I';
ELSIF UPDATING THEN
  op := 'U';
ELSE
  op := 'D';
END IF;
```

Las variables :NEW y :OLD

Los disparadores a nivel de fila pueden hacer uso de un par de variables `:NEW` y `:OLD` en el bloque PL/SQL asociado que permiten acceder a los valores nuevos y viejos de las filas que ha sido afectadas como resultado de la ejecución de la instrucción.

Las dos variables permiten acceder a todas las columnas de las filas afectadas (como si tuvieran tipo `tabla_asociada%ROWTYPE`). Sin embargo, dependiendo de la instrucción que haya provocado la ejecución del disparador, alguna de las variables puede no tener sentido:

- Instrucción `INSERT`. La variable `:NEW` contiene los nuevos valores para cada fila afectada. `:OLD` contiene valores NULL.
- Instrucción `UPDATE`. La variable `:NEW` contiene los valores nuevos para cada fila afectada. La variable `:OLD` contiene los valores viejos para cada fila afectada.
- La instrucción `DELETE`. La variable `:NEW` contiene valores `NULL`. La variable `:OLD` contiene los viejos valores de cada fila afectada.

Condiciones de error y excepciones en el cuerpo de un disparador

Si una excepción predefinida o definida por el usuario es lanzada durante la ejecución de la acción de un disparador, todos los efectos de las instrucciones ejecutadas como parte del disparador y la instrucción que ha provocado la ejecución del disparador son rechazadas. Por tanto aunque no se puedan utilizar instrucciones de control de transacciones, el lanzamiento de una excepción provoca que la transacción activa sea rechazada.

Los disparadores son habitualmente utilizados como mecanismo para gestionar la integridad de los datos de un modo no declarativo ya que pueden ser configurados para ejecutar automáticamente un bloque PL/SQL cuando se ejecuta una instrucción que modifica la BBDD como `INSERT`, `UPDATE` o `DELETE`.

Las excepciones de usuario presentadas en la sección 2.6.2 son particularmente útiles en los disparadores para controlar la transacción en la que se ejecuta el disparador.

Aunque no se permiten utilizar las instrucciones COMMIT o ROLLBACK en los disparadores, si en el cuerpo del disparador se lanza una excepción la transacción activa es abortada.

6.2.2. Creación de disparadores INSTEAD OF

Los disparadores INSTEAD OF son un caso particular de disparadores a nivel de fila. Algunas restricciones adicionales para los disparadores INSTEAD OF son:

- Solo pueden ser aplicados sobre vistas.
- No se pueden especificar las opciones BEFORE y AFTER en la declaración del disparador.
- Si la vista tiene asociada una restricción CHECK, ésta no se comprueba cuando las inserciones o modificaciones se realicen en el disparador. La acción del disparador es la encargada de realizar la comprobación.

```
CREATE [OR REPLACE] TRIGGER <nombre>
  INSTEAD OF { DELETE | INSERT | UPDATE }
    [OR
     { DELETE | INSERT | UPDATE }
    ] ...
  ON nombre_vista
FOR EACH ROW
DECLARE
  ...
BEGIN
-- Código asociado a la acción del disparador
END;
```

6.3. Ejecución de disparadores

Una única instrucción SQL puede potencialmente disparar hasta cuatro tipos de disparadores:

- Disparador a nivel de fila BEFORE
- Disparador a nivel de instrucción BEFORE.
- Disparador a nivel de fila AFTER.
- Disparador a nivel de instrucción AFTER.

Adicionalmente la instrucción que ha provocado la ejecución del disparador o las instrucciones ejecutadas como parte de la acción del disparador pueden causar la comprobación de una o más restricciones de integridad. Además, como se ha mencionado antes, las instrucciones de la acción de un disparador pueden provocar que se ejecuten otros disparadores. Algunas consideraciones generales que es necesario tener en cuenta cuando se trabaja con disparares son:

- Al igual que con las instrucciones SQL, no se puede suponer el orden en el que se procesan las filas en un disparador a nivel de fila.
- No se puede suponer el orden de ejecución de los disparadores asociados al mismo elemento de datos e instrucción.

El modelo de ejecución utilizado para mantener el orden correcto de ejecución de múltiples disparadores y comprobaciones de restricciones de integridad para una instrucción y elementos de datos es el siguiente:

1. Se ejecutan todos los disparadores a nivel de instrucción del tipo BEFORE asociados a la instrucción.
2. Se itera sobre todas las filas afectadas por la instrucción SQL.
3. Se ejecutan todos los disparadores a nivel fila del tipo BEFORE asociados a la instrucción.

1. Bloquea, modifica la fila y lleva a cabo las comprobaciones de restricciones de integridad (los bloqueos no son liberados hasta el final de la transacción).
4. Se ejecutan todos los disparadores a nivel de fila del tipo `AFTER` asociados a la instrucción.
5. Se completa la comprobación de todas las restricciones de integridad pospuestas (marcadas como `DEFERRABLE`).
6. Se ejecutan todos los disparadores a nivel de instrucción del tipo `AFTER` asociados a la instrucción.

6.4. Referencias del Apartado

Ver *Triggers* en [1].
Ver *Using Triggers* en [2].
Ver *How Oracle Enforces Data Integrity* en [1].
Ver *Deferred Constraint Checking* [1].

7. Referencias

[1] Oracle9i Database Concepts, Release 2 (9.2). Disponible en: librero, oracle.
[2] Application Developer's Guide – Fundamentals Release 2 (9.2).Disponible en: librero, oracle.
[3] Oracle9i SQL Reference, Release 2 (9.2). Disponible en: librero, oracle.
[4] PL/SQL User's Guide and Reference, Release 2 (9.2). Disponible en: librero, oracle.
[5] Oracle9i Supplied PL/SQL Packages and Types Reference, Release 2 (9.2). Disponible en: librero, oracle.
[6] SQL*Plus Getting Started, Release 9.0.1 for Windows. Disponible en: librero oracle.
[7] SQL*Plus User's Guide and Reference, Release 9.2. Disponible en: librero, oracle.
[8] Oracle9*i* Database Reference, Release 2 (9.2). Disponible en: librero, oracle.