



TEMA 8

PL/SQL. CURSORES Y DISPARADORES

Bases de Datos
CFGS DAW

Autor: Raquel Torres
Revisado por: Pau Miñana Climent
2020/2021

Licencia



Reconocimiento – NoComercial – CompartirIgual (by-nc-sa): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:



Importante



Atención




Interesante

ÍNDICE

1. CURSORES, REGISTROS Y VECTORES.....	4
1.1 Cursores implícitos.....	4
1.2 Cursores explícitos.....	4
1.3 Atributos.....	6
1.4 Registros.....	7
1.5 Bucle <i>FOR</i> de cursor.....	10
1.6 Vectores/Colecciones.....	12
1.7 Consideraciones sobre el uso de cursores y vectores.....	14
2. TRIGGERS.....	15
2.1 Sintaxis.....	15
2.2 Orden de ejecución de los Triggers.....	16
2.3 Restricciones en la utilización de Triggers.....	16
2.4 Creando Triggers I.....	16
2.5 Creando Triggers II.....	19
2.6 Gestionando los triggers.....	20


UD8 PL/SQL. CURSORES Y DISPARADORES

1. CURSORES, REGISTROS Y VECTORES

 Los **cursores** son áreas de memoria que almacenan datos extraídos de la base de datos.

Hay dos tipos los cursores implícitos y los explícitos.


1.1 Cursores implícitos

 Los **cursores implícitos** sirven para todas aquellas consultas que **devuelven únicamente una fila**.

No es necesario declararlos pues se crean de forma implícita (son los que hemos utilizado hasta ahora).


Repetimos, solamente los podemos utilizar en instrucciones *SELECT* donde se devuelva **una única fila**, si no devuelve ninguna o devuelve más de una se producirá un error. Estos errores se controlarán en la zona *EXCEPTION* mediante las excepciones *NO_DATA_FOUND* y *TOO_MANY_ROWS*.

1.2 Cursores explícitos

 Los **cursores explícitos** deben ser **declarados por el programador** y se emplearán en todas las consultas que puedan **devolver más de una fila** en el resultado.

La sintaxis para la declaración de un cursor se colocará dentro del apartado *DECLARE* de la siguiente forma:

```
CURSOR nombre_cursor IS sentencia_select_del_cursor;
```

 La sentencia *SELECT* del cursor explícito **NO** lleva ***INTO*** para pasar el resultado a las variables del procedimiento o función.

Por ejemplo:

```
DECLARE  
    CURSOR herramientas IS  
        SELECT Nombre, PrecioVenta  
        FROM PRODUCTOS  
        WHERE GAMA='Herramientas';
```

La declaración del cursor aún no ha realizado ninguna operación; para que se ejecute la consulta se tiene que abrir el cursor. En este caso, a diferencia de los cursores implícitos, si no se devuelve ninguna fila no se producirá una excepción.

Para abrir el cursor bastará con ejecutar una instrucción *OPEN* seguida del nombre del cursor de esta manera:

```
OPEN herramientas;
```

Una vez abierto el cursor procederemos a recorrer las filas que se han obtenido como resultado de la consulta que tiene asociada. Para ello emplearemos la instrucción *FETCH*. La sintaxis será:

```
FETCH nombre_cursor INTO [var1,var2,var3,... ] | registro;
```

Cada vez que hacemos un *FETCH* recibiremos una fila y se guardarán los datos en las variables o en el registro que indiquemos. La asignación de valores se realizará posicionalmente, la primera columna a la primera variable, la segunda columna a la segunda variable, etc. Por ello debe haber el mismo número de variables que el número de columnas que se devuelve en cada fila.

Es labor del programador comprobar si la consulta ha devuelto filas o si el procesamiento de las filas ya ha concluido.

Por último, una vez procesadas las filas que nos interesen, debemos cerrar el cursor. Para ello emplearemos la instrucción *CLOSE* seguida del nombre del cursor de la siguiente forma:

```
CLOSE herramientas;
```

Por supuesto, una vez cerrados no se podrán recuperar filas del cursor a no ser que volvamos a abrirlo. También hay que tener en cuenta que los gestores de bases de datos

suelen tener un parámetro (aquí 'open_cursors', en *v\$parameter*) para indicar cuántos cursores pueden estar abiertos de forma simultánea.

Si en algún momento tenemos cursores que a veces funcionan y otras veces no, puede ser que dicho parámetro deba ser incrementado. También podría aparecer un error como *SQL ORA-01000* (número máximo de cursores abiertos excedido).

Para conocer el número máximo de cursores podemos hacer:

```
select value from v$parameter where name='open_cursors';
```

Y para actualizar el número máximo de cursores a, por ejemplo, 1000:

```
alter system set open_cursors = 1000;
```

1.3 Atributos

Los cursores nos proporcionan un conjunto de atributos que nos permitirán controlar el funcionamiento de los cursores. Los atributos que vamos a utilizar son:

- **%ISOPEN**, es de tipo booleano y nos permite comprobar si un cursor está abierto. El atributo tendrá el valor *TRUE* si el cursor está abierto y *FALSE* en caso contrario.
- **%NOTFOUND**, también de tipo booleano y nos permite comprobar si la última recuperación devuelve fila (*FALSE*) o no (*TRUE*). Si aún no se ha hecho ningún *FETCH* devuelve *FALSE*.
- **%FOUND**, de tipo booleano, devolverá *TRUE* si la recuperación más reciente devuelve fila y *FALSE* si no lo hace. También devuelve *FALSE* antes de hacer *FETCH*.
- **%ROWCOUNT** número de filas devueltas hasta ese momento.

Veamos cómo utilizarlos con un ejemplo (*p123.sql*)

1.3.1 Ejemplo 1

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR herramientas IS
        SELECT Nombre, PrecioVenta
        FROM PRODUCTOS
        WHERE GAMA='Herramientas';
    mi_nombre varchar2(70);
    mi_precioventa number(10,2);
BEGIN
    OPEN herramientas;
    IF herramientas%ISOPEN THEN
```

```
        FETCH herramientas INTO mi_nombre, mi_precioventa;  
        WHILE herramientas%FOUND LOOP  
            DBMS_OUTPUT.PUT_LINE('Fila: '||herramientas  
                %ROWCOUNT||'- '||mi_nombre||'- '||mi_precioventa);  
            FETCH herramientas INTO mi_nombre, mi_precioventa;  
        END LOOP;  
        CLOSE herramientas;  
    END IF;  
END;  
/
```


Hemos declarado el cursor y dos variables que van a recoger los resultados de las filas que devuelve. Después abrimos el cursor y comprobamos que se ha abierto correctamente. Si está abierto solicitamos la primera fila con un *FETCH* y si la ha encontrado entramos en el *WHILE* (si no hubiese filas en el resultado no entraría en el bucle). Una vez dentro del bucle mostramos los datos recogidos y pedimos la siguiente fila con otro *FETCH*. Realizaremos el bucle hasta que se hayan procesado todas las filas del cursor. Posteriormente cerramos el cursor y terminará la ejecución del script.

El resultado será:

```
SQL> @ c:\src\pl23.sql  
Fila: 1-Sierra de Poda 400MM-14  
Fila: 2-Pala-14  
Fila: 3-Rastrillo de Jardín-12  
Fila: 4-Azadón-12  
  
PL/SQL procedure successfully completed.
```

1.4 Registros

Los registros son elementos bastante utilizados en PL/SQL.

 Un **registro** es un grupo de variables relacionadas bajo un mismo nombre, cada una de las cuales tiene su propio nombre y su tipo de dato.

La forma de declarar un registro es la siguiente:

```
TYPE direccion IS RECORD  
(calle varchar2(30),  
numero int,  
localidad varchar2(30),  
codpostal varchar2(5));
```

Una vez declarado el tipo de registro podemos declarar variables de dicho tipo.

```
mi_direccion direccion;
```

Donde *mi_direccion* es la variable y *direccion* es el tipo de dato definido como registro. Para poder acceder a los campos del registro que hemos creado bastará con utilizar el operador punto de la siguiente forma:

```
mi_direccion.calle  
mi_direccion.numero  
mi_direccion.localidad  
mi_direccion.codpostal
```

Podremos asignar valores a los campos:

```
mi_direccion.calle := 'Avda. la Constitución';  
mi_direccion.numero := 20;  
mi_direccion.localidad := 'Madrid';  
mi_direccion.codpostal := '28003';
```

Aunque la forma más sencilla de crear un registro es utilizando los atributos de tipo (%ROWTYPE) creando un registro con los mismos campos que la fila que devolverá un cursor de la siguiente forma:

```
DECLARE  
CURSOR pagos_clientes IS  
    SELECT codigocliente, sum(cantidad) as total_pagado  
    FROM pagos group by codigocliente  
    ORDER BY codigocliente;  
reg_pagos pagos_clientes%ROWTYPE;
```

Veamos cómo utilizarlos con un script (*pl24.sql*)

1.4.1 Ejemplo 2

```
SET SERVEROUTPUT ON
```



```

DECLARE
    CURSOR pagos_clientes IS
        SELECT codigocliente, sum(cantidad) as total_pagado
        FROM pagos group by codigocliente
        ORDER BY codigocliente;
    reg_pagos pagos_clientes%ROWTYPE;
BEGIN
    OPEN pagos_clientes;
    IF pagos_clientes%ISOPEN THEN
        FETCH pagos_clientes INTO reg_pagos;
        WHILE pagos_clientes%FOUND LOOP
            DBMS_OUTPUT.PUT_LINE('Fila: '||pagos_clientes
                %ROWCOUNT||'- Cliente:'||reg_pagos.codigocliente||'-
                Total:'||reg_pagos.total_pagado);
            FETCH pagos_clientes INTO reg_pagos;
        END LOOP;
        CLOSE pagos_clientes;
    END IF;
END;
/

```

El resultado será:

```

SQL> @ c:\src\pl24.sql
Fila: 1- Cliente:1- Total:4000
Fila: 2- Cliente:3- Total:10926
Fila: 3- Cliente:4- Total:81849
Fila: 4- Cliente:5- Total:23794
Fila: 5- Cliente:7- Total:2390
Fila: 6- Cliente:9- Total:929
Fila: 7- Cliente:13- Total:2246
Fila: 8- Cliente:14- Total:4160
Fila: 9- Cliente:15- Total:12081
Fila: 10- Cliente:16- Total:4399
Fila: 11- Cliente:19- Total:232
Fila: 12- Cliente:23- Total:272
Fila: 13- Cliente:26- Total:18846
Fila: 14- Cliente:27- Total:10972
Fila: 15- Cliente:28- Total:8489
Fila: 16- Cliente:30- Total:7863
Fila: 17- Cliente:35- Total:3321
Fila: 18- Cliente:38- Total:1171

PL/SQL procedure successfully completed.

```

1.4.2 Ejemplo 3

Los registros también se pueden utilizar con cursores implícitos. El uso más habitual es asignar el atributo de tipo de una tabla. Veamos un ejemplo sencillo (*pl24a.sql*)

```
SET SERVEROUTPUT
ON DECLARE
    Reg_cliente Clientes%ROWTYPE;
BEGIN
    SELECT * INTO Reg_cliente
    FROM clientes WHERE codigocliente=6;
    DBMS_OUTPUT.PUT_LINE(Reg_cliente.nombrecliente||' - '||
    Reg_cliente.telefono);
END;
/
```

El resultado será:

```
SQL> @ c:\src\pl24a.sql
Lasas S.A. - 34916540145
PL/SQL procedure successfully completed.
```

1.5 Bucle *FOR* de cursor

El bucle *FOR* funciona de una forma especial con los cursores explícitos. Al crear una instrucción *FOR* con un cursor, el sistema realiza de forma implícita la apertura del cursor (*OPEN*), la recuperación de las filas una a una (*FETCH*) y el cierre del cursor (*CLOSE*). Además la variable índice del *FOR* ahora actuará como un registro con el atributo de tipo de la fila recuperada en el cursor. Es decir, lo hace prácticamente todo de forma automática; únicamente debemos procesar las líneas.

La sintaxis será:

```
FOR nombre_registro IN nombre_cursor LOOP
    instrucción_1;
    instrucción_2;
    ....
END LOOP;
```

Veamos un ejemplo (*pl25.sql*)

1.5.1 Ejemplo 4

```
SET SERVEROUTPUT ON
DECLARE
    Total_Pedidos NUMBER(10,2):=0;
```

```

Reg_p10p detallepedidos%ROWTYPE; -- funciona incluso sin esta declaración
CURSOR primeros_10_pedidos IS
    SELECT Codigopedido, sum(cantidad * preciounidad) as total
    FROM detallepedidos
    WHERE codigopedido BETWEEN 1 and 10
    GROUP BY codigopedido
    ORDER BY codigopedido;

BEGIN
    FOR Reg_p10p IN primeros_10_pedidos LOOP
        DBMS_OUTPUT.PUT_LINE('Pedido: ' || Reg_p10p.codigopedido || '
        Total: ' || Reg_p10p.total);

        Total_pedidos := Total_pedidos + Reg_p10p.total;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('El total de los pedidos es: ' || Total_Pedidos);
END;
/

```

El resultado será:

```

SQL> @ c:\src\pl25.sql
Pedido: 1 Total: 1567
Pedido: 2 Total: 7113
Pedido: 3 Total: 10850
Pedido: 4 Total: 2624
Pedido: 8 Total: 1065
Pedido: 9 Total: 2535
Pedido: 10 Total: 2920
El total de los pedidos es: 28674

PL/SQL procedure successfully completed.

```

Si los valores no se van a utilizar fuera del bucle, incluso cabe la posibilidad de no declarar ni el cursor ni el registro y poner directamente la consulta entre paréntesis en el IN:


```

DECLARE
    Total_Pedidos NUMBER(10,2):=0;
BEGIN
    FOR Reg_p10p IN (SELECT Codigopedido, sum(cantidad * preciounidad) as total
    FROM detallepedidos WHERE codigopedido BETWEEN 1 and 10
    GROUP BY codigopedido ORDER BY codigopedido) LOOP
        DBMS_OUTPUT.PUT_LINE('Pedido: ' || Reg_p10p.codigopedido || '
        Total: ' || Reg_p10p.total);

        Total_pedidos := Total_pedidos + Reg_p10p.total;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('El total de los pedidos es: ' || Total_Pedidos);
END;
/

```

1.6 Vectores/Colecciones

 Los vectores o colecciones permiten almacenar una cantidad fija de elementos del mismo tipo a los que se puede acceder directamente con el índice del lugar que ocupan en esta estructura.

Para poder declarar un vector primero se crea el tipo con:

```
TYPE nombre_tipo_vector IS VARRAY(n) OF tipo_datos
```

Donde *n* es el número máximo de elementos del vector y *tipo_datos* el tipo de datos que el vector puede almacenar. Después de esto se puede proceder a declarar el vector como cualquier otra variable.

```
nombre_vector nombre_tipo_vector;
```

Se pueden dar valores al vector de varios modos:

- Directamente a todo el vector: *nombre_vector:=nombre_tipo_vector(valor1,valor2,...);*
Para controlar la longitud del vector, con *:= nombre_tipo_vector()* se puede inicializar un vector vacío e ir añadiendo elementos (con valor NULL) uno a uno con *nombre_vector.extend* o varios de golpe con *nombre_vector.extend(cantidad)*
- A cada elemento individualmente: *nombre_vector(indice):=valor;*
Para esto el elemento con ese índice debe existir previamente.

Por ejemplo, damos valor directamente a un vector de 5 números y creamos un vector de nombres al que añadimos 3 elementos vacíos y rellenamos el último por índice.

1.6.1 Ejemplo 5


```
DECLARE  
    TYPE vector_enteros IS VARRAY(5) OF INT;  
    numeros vector_enteros;  
    TYPE vector_nombres IS VARRAY(5) OF empleados.nombre%TYPE;  
    nombres vector_nombres:=vector_nombres();  
BEGIN  
    numeros:=vector_enteros(2,3,4,5,1);  
    nombres.extend(3);  
    nombres(3):='Maria';  
    DBMS_OUTPUT.PUT_LINE('Primer numero del vector: '||numeros(1));  
    DBMS_OUTPUT.PUT_LINE('Nombre: '||nombres(3));  
END;  
/
```

Da como resultado:

```
Primer numero del vector: 2  
Nombre: Maria  
  
PL/SQL procedure successfully completed.
```

Si se intenta extender o dar valor más allá de los elementos máximos determinados en el VARRAY se producirá un error.

Obviamente, se pueden añadir los resultados de una consulta a un vector. Si es una única fila, se puede añadir en una posición determinada con `SELECT ... INTO nombre(indice)` igual que con los cursores implícitos.

 Si se desea almacenar una consulta con varias filas se puede hacer añadiendo `BULK COLLECT` antes del `INTO`:

```
SELECT campos BULK COLLECT INTO nombre_vector FROM ...
```

Por ejemplo vamos a almacenar en un vector nombres todos los nombres de los empleados y en un vector nombres2 el nombre del empleado con código 3 en la primera posición, posteriormente mostramos el tercer elemento del vector nombres para comparar (tener presente que el vector nombres contiene los 31 que devuelve la consulta):

1.6.2 Ejemplo 6

```
DECLARE  
    TYPE vector_nombres IS VARRAY(31) OF empleados.nombre%TYPE;  
    nombres vector_nombres;  
    nombres2 vector_nombres:=vector_nombres();  
BEGIN  
    SELECT nombre BULK COLLECT INTO nombres FROM empleados;  
    nombres2.extend(1);  
    SELECT nombre INTO nombres2(1) FROM empleados WHERE codigoempleado=3;  
    DBMS_OUTPUT.PUT_LINE('Nombre: '||nombres(3));  
    DBMS_OUTPUT.PUT_LINE('Nombre2: '||nombres2(1));  
END;  
/
```

```
Nombre: Alberto  
Nombre2: Alberto  
  
PL/SQL procedure successfully completed.
```

Como habréis podido deducir, las colecciones no se limitan a simples variables y cada posición del índice puede contener un registro si se crea de este modo y se accede de la misma forma que con estos, sólo que especificando el índice:

```
nombre_vector(indice).nombrevariable
```

Vamos a ver un ejemplo en el que almacenamos así todos los datos de los empleados en vez de sólo el nombre y se muestran el nombre y código del último.

1.6.3 Ejemplo 7

```
DECLARE
    TYPE vector_empleados IS VARRAY(31) OF empleados%ROWTYPE;
    emple vector_empleados;
BEGIN
    SELECT * BULK COLLECT INTO emple FROM empleados;
    DBMS_OUTPUT.PUT_LINE(emple(31).codigoempleado || ':' ||
        emple(31).nombre);
END;
/
```

```
31 : Mariko
```


```
PL/SQL procedure successfully completed.
```

1.7 Consideraciones sobre el uso de cursores y vectores

Una vez vistas todas estas posibilidades cabe preguntarse cuando conviene usar cada una de ellas. Lo cierto es que no hay ninguna regla estricta escrita al respecto y la experiencia es la que aporta claridad a la hora de tomar decisiones pero se pueden tener en cuenta varios factores:

- Los cursores implícitos son los que ejecutan sus operaciones más rápido, así que son convenientes siempre que se pueda respecto al uso de vectores o cursores.
- Los vectores nos proporcionan una estructura de datos concreta que el programa interpreta perfectamente, así que resultan útiles para enviar a otro proceso estos fragmentos de información, preferiblemente ya procesada.
- Con los cursores no se “descargan” todos los datos sino una fila cada vez que se hace un FETCH, así pues, aunque funcionen a menos velocidad, si no se van a usar todos los datos pueden resultar más útiles. Por ejemplo, si se envían los resultados a una página donde sólo se van a ver los 20 primeros no se necesitan los, por decir algo, miles que podría contener un vector.

2. TRIGGERS

 Los **disparadores** (o **triggers**) son bloques de código asociados a una tabla que se ejecutan automáticamente como reacción a una operación específica (*INSERT*, *UPDATE* o *DELETE*) sobre dicha tabla.

Los *triggers* pueden ejecutarse antes o después (*BEFORE*, *AFTER*) de que se produzca la operación indicada sobre la tabla.

2.1 Sintaxis

La sintaxis de los *triggers* en Oracle es:

```
CREATE {OR REPLACE} TRIGGER nombre_disparador [BEFORE|AFTER] [DELETE|
INSERT|UPDATE {OF columnas}]
[OR [DELETE|INSERT|UPDATE {OF columnas}]]...
ON tabla
[FOR EACH ROW [WHEN (condicion disparo)]]
[DECLARE]
    -- Declaración de variables locales
BEGIN
    -- Instrucciones de ejecución
[EXCEPTION]
    -- Instrucciones de excepción
END;
```

Como podemos deducir de la sintaxis, se puede crear un disparador o reemplazar uno que ya exista.

Debemos elegir si se debe ejecutar antes o después de que se realice la operación a la que está asociado con las palabras *BEFORE* o *AFTER*.

También debemos indicar la operación sobre la que se va a aplicar *INSERT*, *DELETE* o *UPDATE*. Hay que indicar la tabla sobre la que se realiza la operación: *ON* tabla.

Además debemos indicar si es un disparador a nivel de fila con *FOR EACH ROW*, eso quiere decir que el disparador se activará una vez por cada fila afectada en la operación que provoca el disparo. También existen los *Triggers* a nivel de orden, que se activan sólo una vez después de la ejecución completa de la instrucción.

2.2 Orden de ejecución de los Triggers

En una misma tabla podemos tener asociados varios *Triggers*; si esto ocurre, debemos conocer el orden en el que se ejecutarán. Los disparadores se activarán al comenzar la instrucción SQL y se ejecutarán en el siguiente orden:

- Primero se ejecuta el disparador a nivel de **orden** con el tipo **BEFORE**.
- Después, para cada **fila**, se ejecuta el disparador de tipo **BEFORE** a nivel de **fila**.
- Lo siguiente será ejecutar la **instrucción SQL** con la fila correspondiente.
- Después se ejecuta el disparador de tipo **AFTER** a nivel de **fila**.
- Por último, se ejecuta el disparador a nivel de **orden** con el tipo **AFTER**.

2.3 Restricciones en la utilización de Triggers

El cuerpo del disparador es un bloque PL/SQLs luego cualquier instrucción de las que conocemos hasta ahora la podemos emplear en un *Trigger*. Solamente debemos tener en cuenta algunas restricciones:

- Un disparador no puede incluir control de transacciones ni puede llamar a funciones o procedimientos que puedan incluir un control de este tipo.
- Además, aunque hay alguna excepción, como regla general nunca modificaremos la tabla que se está viendo afectada por la instrucción SQL.

2.4 Creando Triggers I

Para comenzar, vamos a crear un *Trigger* en la BD **teoriaud6**, que nos permita controlar el número de empleados que tiene un departamento. Para ello vamos a modificar la tabla *DEPARTAMENTOS* añadiendo el campo *NumEmp* con valor 0 por defecto:

```
SQL> ALTER TABLE DEPARTAMENTOS ADD NUMEMP INT DEFAULT 0;
Table altered.
```

Después vamos a actualizar los registros de los departamentos incluyendo el número de empleados que hay de cada uno de ellos, de modo que los datos quedan:

```
SQL> UPDATE DEPARTAMENTOS SET NUMEMP=3 WHERE CODDPTO='IT';
1 row updated.
SQL> UPDATE DEPARTAMENTOS SET NUMEMP=1 WHERE CODDPTO='CONT';
1 row updated.
SQL> UPDATE DEPARTAMENTOS SET NUMEMP=1 WHERE CODDPTO='ALM';
1 row updated.
```

```
SQL> SELECT CODDPTO, NOMBRE, NUMEMP FROM DEPARTAMENTOS;
```

CODDPTO	NOMBRE	NUMEMP
ADM	Administración	0
ALM	Almacén	1
CONT	Contabilidad	1
IT	Informática	3

Bien, ahora vamos a crear el *Trigger* de tal forma que cada vez que se realice un *INSERT* en la tabla *empleados* se aumente en uno el número de empleados del departamento al que pertenece.

Fíjate que para hacer esto necesitaremos acceder a los campos que se insertan, pues necesitamos conocer el código del departamento del nuevo empleado. Cuando trabajamos a nivel de fila, para acceder al contenido de los campos, Oracle proporciona las referencias *NEW* y *OLD* para acceder a los nuevos datos (*NEW.Nombre_Campo*) o a los antiguos (*OLD.Nombre_Campo*) respectivamente. Debemos tener en cuenta que:

En un *INSERT* solo tenemos datos *NEW*, y si nuestro *Trigger* es de tipo *BEFORE* podemos modificar esos valores antes de que se almacenen en la tabla.

En un *UPDATE* tenemos datos *NEW* y *OLD*, los primeros son los nuevos datos y los segundos son los existentes en la tabla. Igual que en el caso anterior, si es de tipo *BEFORE* también se podrán modificar los datos de *NEW*.

En un *DELETE* solo tenemos datos tipo *OLD*.

Cuando vamos a utilizar estos campos dentro del cuerpo del *Trigger* será necesario colocar delante de ellos el signo dos puntos (:), excepto en la cláusula *WHEN* que veremos después.

Bien, vamos con nuestro ejemplo, se trata de aumentar de forma automática con un *Trigger* el campo *NUMEMP* del departamento, cuando se inserta un nuevo empleado.

2.4.1 Ejemplo 8

```
CREATE OR REPLACE TRIGGER nuevo_empleado
  AFTER INSERT
  ON EMPLEADOS
  FOR EACH ROW
  BEGIN
    UPDATE DEPARTAMENTOS SET NUMEMP = NUMEMP + 1
    WHERE CODDPTO = :NEW.DPTO;
  END;
/
```

Prestad atención al modo en el que se emplea el código de departamento del nuevo empleado con *:NEW.DPTO*

Si al crear el *Trigger* hubiera errores, podríamos verlos con el comando:

```
Show Errors;
```

Una vez creado el *script* lo ejecutamos y se creará el *Trigger*.

```
SQL> @ c:\src\trigger_01.sql
Trigger created.
```

Para comprobar su funcionamiento tenemos que insertar un nuevo registro en empleados y comprobar si, de forma automática, se aumenta el campo *NUMEMP* del departamento al que pertenece el nuevo empleado.

```
SQL> insert into empleados
  2 values('112233445P','Pedro Gil','Informática',
  3 TO_DATE('10/10/2013','DD/MM/YYYY'),'IT','MAD20',1200);
1 row created.
SQL> SELECT CODDPTO, NOMBRE, NUMEMP FROM DEPARTAMENTOS;
```

CODDPTO	NOMBRE	NUMEMP
ADM	Administración	0
ALM	Almacén	1
CONT	Contabilidad	1
IT	Informática	4

Como podemos observar se ha aumentado en 1 el *NUMEMP* del departamento de Informática al que pertenece el nuevo empleado.

¿Cómo sería un *trigger* que al eliminar un empleado actualizase correctamente el número de empleados del departamento al que pertenece? Crea el *Trigger* indicado y pruébalo.

2.4.2 Ejemplo 9

Ahora vamos a crear otro *Trigger* que al actualizar un empleado, si se cambia el departamento, actualice de forma correcta el campo *NUMEMP* de los departamentos involucrados:

```
CREATE OR REPLACE TRIGGER cambio_en_empleado
  AFTER UPDATE
  ON EMPLEADOS
  FOR EACH ROW
  BEGIN
    IF :NEW.DPTO != :OLD.DPTO THEN
      UPDATE DEPARTAMENTOS SET NUMEMP = NUMEMP - 1
      WHERE CODDPTO = :OLD.DPTO;
      UPDATE DEPARTAMENTOS SET NUMEMP = NUMEMP + 1
      WHERE CODDPTO = :NEW.DPTO;
    END IF;
  END;
```

El resultado será:

```
SQL> UPDATE EMPLEADOS SET DPTO = 'CONT'
  2 WHERE DNI = '112233445P';

1 row updated.

SQL> SELECT CODDPTO, NOMBRE, NUMEMP FROM DEPARTAMENTOS;
```

CODDPTO	NOMBRE	NUMEMP
ADM	Administración	0
ALM	Almacén	1
CONT	Contabilidad	2
IT	Informática	3

2.5 Creando Triggers II

En la creación de *Triggers* podemos añadir una cláusula *WHEN* en el procesamiento de cada fila (solo se puede utilizar con *triggers* a nivel de fila) de tal forma que se evalúe una condición, que implique si se debe ejecutar o no el código del *Trigger*. Su sintaxis sería:

```
FOR EACH ROW WHEN (condición)
```

Recordemos que si queremos emplear los prefijos *NEW* y *OLD* en la condición del *WHEN* no debemos colocar el signo dos puntos (:) delante del prefijo.

Por ejemplo en el trigger anterior se elimina el IF poniendo la condición en el WHEN

2.5.1 Ejemplo 10

```
FOR EACH ROW WHEN (NEW.DPTO != OLD.DPTO)

BEGIN
    UPDATE DEPARTAMENTOS SET NUMEMP = NUMEMP - 1
    WHERE CODDPTO = :OLD.DPTO;
    UPDATE DEPARTAMENTOS SET NUMEMP = NUMEMP + 1
    WHERE CODDPTO = :NEW.DPTO;
END;
```

Dentro de *Triggers* con varias opciones de disparo podemos detectar si se trata de una instrucción SQL de inserción (*INSERTING*)s modificación (*UPDATING*) o borrado (*DELETING*).

```
IF [INSERTING|UPDATING|DELETING] THEN
    INSTRUCCIONES....;
END IF;
```

Además si estamos ante una actualización a nivel de fila podemos comprobar si se está actualizando un determinado campo:

```
IF UPDATING('NOMBRE_CAMPO') THEN
    INSTRUCCIONES ... ;
END IF;
```

2.6 Gestionando los triggers

Para ver los *triggers* que hay en la base de datos podemos emplear la vista *ALL_TRIGGERS* que contiene los siguientes campos:

```
SQL> DESC ALL_TRIGGERS;
```

Name	Null?	Type
OWNER		VARCHAR2(30)
TRIGGER_NAME		VARCHAR2(30)
TRIGGER_TYPE		VARCHAR2(16)
TRIGGERING_EVENT		VARCHAR2(227)
TABLE_OWNER		VARCHAR2(30)
BASE_OBJECT_TYPE		VARCHAR2(16)
TABLE_NAME		VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
REFERENCING_NAMES		VARCHAR2(128)
WHEN_CLAUSE		VARCHAR2(4000)
STATUS		VARCHAR2(8)
DESCRIPTION		VARCHAR2(4000)
ACTION_TYPE		VARCHAR2(11)
TRIGGER_BODY		LONG
CROSSEDITION		VARCHAR2(7)
BEFORE_STATEMENT		VARCHAR2(3)
BEFORE_ROW		VARCHAR2(3)
AFTER_ROW		VARCHAR2(3)
AFTER_STATEMENT		VARCHAR2(3)
INSTEAD_OF_ROW		VARCHAR2(3)
FIRE_ONCE		VARCHAR2(3)
APPLY_SERVER_ONLY		VARCHAR2(3)

Podemos ver cuáles son los disparadores de nuestro usuario con:

```
SQL> SELECT OWNER, TRIGGER_NAME, TRIGGER_TYPE FROM ALL_TRIGGERS
2  WHERE OWNER = 'USUARIO_PRUEBA';
```

OWNER	TRIGGER_NAME	TRIGGER_TYPE
USUARIO_PRUEBA	NUEVO_EMPLEADO	AFTER EACH ROW
USUARIO_PRUEBA	GAMBIO_EN_EMPLEADO	AFTER EACH ROW

También podemos activar o desactivar un *trigger* con:

```
ALTER TRIGGER nombre_disparador {ENABLE | DISABLE}
```

Por ejemplo:

```
SQL> SELECT TRIGGER_NAME,TRIGGER_TYPE,STATUS FROM ALL_TRIGGERS
2  WHERE OWNER = 'USUARIO_PRUEBA';
```

TRIGGER_NAME	TRIGGER_TYPE	STATUS
NUEVO_EMPLEADO	AFTER EACH ROW	ENABLED
CAMBIO_EN_EMPLEADO	AFTER EACH ROW	ENABLED

```
SQL> ALTER TRIGGER NUEVO_EMPLEADO DISABLE;
```

Trigger altered.

```
SQL> SELECT TRIGGER_NAME,TRIGGER_TYPE,STATUS FROM ALL_TRIGGERS
2  WHERE OWNER = 'USUARIO_PRUEBA';
```

TRIGGER_NAME	TRIGGER_TYPE	STATUS
NUEVO_EMPLEADO	AFTER EACH ROW	DISABLED
CAMBIO_EN_EMPLEADO	AFTER EACH ROW	ENABLED

Por último, podemos eliminar un Trigger con:

```
DROP TRIGGER nombre_disparador;
```

```
SQL> DROP TRIGGER NUEVO_EMPLEADO;
```

Trigger dropped.

```
SQL> SELECT TRIGGER_NAME,TRIGGER_TYPE,STATUS FROM ALL_TRIGGERS
2  WHERE OWNER = 'USUARIO_PRUEBA';
```

TRIGGER_NAME	TRIGGER_TYPE	STATUS
CAMBIO_EN_EMPLEADO	AFTER EACH ROW	ENABLED