



## EJEMPLOS TUTORÍAS COLECTIVAS

Bases de Datos  
CFGs DAW



Pau Miñana Climent  
2020/2021

## UD08. EJEMPLOS USADOS EN LAS TUTORÍAS COLECTIVAS

En estos ejemplos se usa la base de datos teoriaud6.

### 1. Trigger para evitar la subida de precios en productosped

Con esta primera versión se lanza una excepción que rompe la operación y por tanto la actualización no se produce. A modo de ejemplo se añade de forma comentada el tratamiento de la excepción, pero en este caso concreto no debe hacerse. Aunque el trigger se detiene, al tratar la excepción el programa considera la salida correcta y procede a realizar la operación update, que es lo que se quería evitar.

```
CREATE OR REPLACE TRIGGER restriccion_v1
BEFORE UPDATE
ON productosped
FOR EACH ROW WHEN (old.precio<new.precio)
DECLARE
    -- precio_subido EXCEPTION;
    --PRAGMA EXCEPTION_INIT (precio_subido,-20000);
BEGIN
    Raise_Application_Error (-20000,'Nunca subimos los precios');
    -- Con la excepción declarada la línea anterior es equivalente a esta:
    -- RAISE precio_subido;
    DBMS_OUTPUT.PUT_LINE('Aquí nunca se llega');

    -- EXCEPTION
    -- WHEN precio_subido THEN
    -- DBMS_OUTPUT.PUT_LINE('Al tratar la excepción el insert/update se realiza');
END;
/

-- Si se ejecuta una actualización de todos los precios se puede ver que no cambia ninguno.
-- En cambio si se descomenta el tratamiento de la excepción los precios sí se actualizan.
UPDATE productosped
SET precio=5;
```

Veamos ahora otra versión sin usar excepciones donde, en vez de evitar la actualización, modificamos los datos de la fila que no debe actualizarse para que no cambie. De este modo ante actualizaciones de varias filas, como la vista anteriormente, no se anula toda la operación sino que sólo se evita que cambien los datos que no cumplen las condiciones deseadas. Esto tiene sus ventajas e inconvenientes; es más versátil, pero si la orden de actualización era errónea, actualizará los datos que sí cumplan las condiciones y puede que no sea lo deseado

Antes de empezar se debe desactivar el trigger anterior para que no interfiera.

```
ALTER TRIGGER restriccion_v1 disable;

CREATE OR REPLACE TRIGGER restriccion_v2
BEFORE UPDATE
ON productosped
FOR EACH ROW WHEN (old.precio<new.precio)
BEGIN
    :new.precio := :old.precio;
    DBMS_OUTPUT.PUT_LINE('El producto '||:new.refeproducto||' no se modifica');
END;
/
```

Se podrá observar que en cualquier caso la operación de actualización afecta a todas las tuplas que indica, pero en las que no cumplan la restricción se actualizará el precio a sí mismo y por tanto no cambia. Esto es relevante pues permite que se actualizen otros datos. Además cabe observar que los triggers no dejan ninguna indicación evidente de su disparo. Si se quita el texto de salida o SERVEROUTPUT no está en ON no se sabe si se ha disparado.

## 2. Trigger para evitar productos por encima de 35 €

En este caso vamos a aplicar un trigger para evitar que se puedan insertar productos por encima de 35€ pero de un modo distinto. Permitimos la inserción y después borramos los productos que superen el precio

```
CREATE OR REPLACE TRIGGER limite_precio
AFTER INSERT
ON productosped
BEGIN
    DELETE FROM productosped
    WHERE precio>35;
END;
/
```

Cuidado con los triggers de este tipo pues son muy poderosos; borrarán cualquier producto por encima de 35 € independientemente de que se inserte en ese momento o el insertado sea otro. Es decir que todo producto por encima de 35 euros desaparece. Esto por un lado evita errores en la BD pero al mismo tiempo puede borrar productos ya existentes a los que se cambie el precio. Se podría combinar con la idea mostrada en el trigger anterior y cuando se inserten elementos (IF INSERTING) hacer el DELETE pero si se actualizan (IF UPDATING) y superan los 35€ anular la subida usando “:new.precio := :old.precio” o usar 2 triggers distintos que hagan esto, uno para la actualización y otro para las nuevas tuplas.

### 3. Trigger para actualizaciones en cascada

La actualización en cascada de las tablas no es una opción ofrecida por Oracle, debido a que se considera que las relaciones entre tablas se harán con campos clave y se considera que una clave no debe cambiar nunca. Aún así, en la práctica es una necesidad que nos podemos acabar encontrando, aunque en teoría suponga una mala praxis, así que se puede implementar un trigger para que al actualizar un campo se cambie también donde aparece como clave foránea. En este caso lo realizamos con refeproducto en productosped, que es clave foránea en productospedido. En caso de haber más tablas implicadas sólo se necesita un UPDATE por cada una.

```
CREATE OR REPLACE TRIGGER productos_cascada
AFTER UPDATE
ON productosped
FOR EACH ROW when (old.refeproducto<>new.refeproducto)
BEGIN
    UPDATE productospedido
    SET refeproducto=:new.refeproducto
    WHERE refeproducto=:old.refeproducto;
    DBMS_OUTPUT.PUT_LINE('Ref. actualizada en los pedidos');
END;
/
```

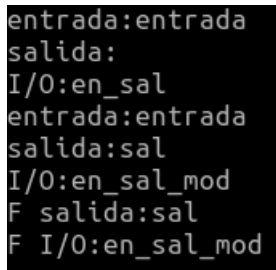
### 4. Introducción a los procedimientos

Este es un simple ejemplo sin demasiada aplicación práctica, más allá de comprender las limitaciones de los distintos tipos de parámetros que se le pueden pasar a un procedimiento o función. Se usa un parámetro IN, otro OUT y otro IN OUT y se analiza su estado y si se pueden modificar en la función.

```
CREATE OR REPLACE PROCEDURE intro (entrada IN varchar2, salida OUT varchar2,
                                   en_sal IN OUT varchar2)
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('entrada:' || entrada);
    DBMS_OUTPUT.PUT_LINE('salida:' || salida);
    DBMS_OUTPUT.PUT_LINE('I/O:' || en_sal);
    -- Esta línea no se puede activar pues no permite modificar parámetros IN
    -- entrada:='Imposible';
    salida:='sal';
    en_sal:='en_sal_mod';
    DBMS_OUTPUT.PUT_LINE('entrada:' || entrada);
    DBMS_OUTPUT.PUT_LINE('salida:' || salida);
    DBMS_OUTPUT.PUT_LINE('I/O:' || en_sal);
END;
/
```

Ahora simplemente creamos unas variables con valores iniciales, las usamos en el procedimiento y volvemos a ver el valor después del mismo.

```
DECLARE
    entrada1 varchar(10):='entrada';
    salida1 varchar(10):='salida';
    en_sal1 varchar(10):='en_sal';
BEGIN
    intro(entrada1,salida1,en_sal1);
    DBMS_OUTPUT.PUT_LINE('F salida:' || salida1);
    DBMS_OUTPUT.PUT_LINE('F I/O:' || en_sal1);
END;
/
```



```
entrada:entrada
salida:
I/O:en_sal
entrada:entrada
salida:sal
I/O:en_sal_mod
F salida:sal
F I/O:en_sal_mod
```

Se puede observar que:

- Las variables tipo IN no pueden modificarse dentro del procedimiento.
- Las variables tipo OUT sí, pero empiezan sin valor alguno aunque originalmente ya se les hubiese dado algún valor.
- Las variables tipo IN OUT conservan su valor al entrar y pueden modificarse.
- Tanto las tipo OUT como las IN OUT conservan las modificaciones después de salir del procedimiento.
- OJO! Puesto que una variable de tipo OUT pierde su valor al entrar al procedimiento, si no se modifica en su interior, a su salida la variable ha perdido el valor original igualmente y pas a valer NULL.

## 5. Procedimiento para mostrar consultas de múltiples t  pulas. Cursores.

Para estos pr  ximos ejemplos se van a usar distintos m  todos para mostrar los empleados de un departamento que se pasa como par  metro. En este primer caso usaremos un cursor.

```
CREATE OR REPLACE PROCEDURE empleados_dep1 (departamento IN varchar2) IS
BEGIN
    FOR empleado IN (SELECT * FROM empleados WHERE dpto=departamento)
    LOOP
        DBMS_OUTPUT.PUT_LINE(empleado.Nombre || ': ' || empleado.Especialidad);
    END LOOP;
END;
/
```

La gran ventaja de este m  todo es su simplicidad, por lo que se usa para mostrar datos o tareas simples en muchas ocasiones. El defecto es que "empleado" s  lo existe durante el bucle, y adem  s con los cursores se tiene que acceder a las tuplas uno a uno y en orden.

## 6. Procedimiento para mostrar consultas de múltiples t  plas. Bucle con registros.

Aprovecho esta opci  n para introducir la funci  n ROWNUM, que devuelve el numero de fila de la tupla dentro de una consulta (no en la tabla original). Por tanto nos permite gestionar las consultas para que s  lo nos devuelvan una fila, y usando un bucle, para poder descargar toda la consulta tupla a tupla. Por ejemplo:

```
SELECT ROWNUM,empleados.* FROM empleados;
```

Nos devuelve todos los datos de los empleados con el n  mero de la fila delante. Si s  lo queremos acceder a una de las t  plas, digamos la segunda, no se puede hacer directamente a  adiendo "WHERE ROWNUM=2", pues el dato se calcula cada vez y nos devolver  a una consulta vac  a. Debe hacerse mediante una subconsulta:

```
SELECT * FROM (SELECT ROWNUM as linea,empleados.* FROM empleados)
WHERE linea=2;
```

Dicho esto, s  lo nos queda saber cuantas filas devuelve una consulta para poder hacer nuestro bucle, y eso se puede averiguar con un COUNT. De ese modo se puede implementar la siguiente versi  n de nuestro procedimiento donde almacenamos cada vuelta del bucle una tupla en el registro del tipo empleados%ROWTYPE que declaramos:

```
CREATE OR REPLACE PROCEDURE empleados_dep2 (departamento IN varchar2) IS
    lin INT;
    a empleados%ROWTYPE;
BEGIN
    SELECT COUNT(*) INTO lin FROM empleados WHERE dpto=departamento;
    IF (lin>0) THEN
        FOR b IN 1 .. lin LOOP
            SELECT dni,nombre,especialidad,fechaalta,dpto,codp INTO a
            FROM (SELECT ROWNUM as linea,empleados.*
                  FROM empleados where dpto=departamento)
            WHERE linea=b;
            DBMS_OUTPUT.PUT_LINE(a.dni||' '||a.nombre||' '||
            a.especialidad||' '||a.fechaalta||' '||a.dpto||' '||a.codp);
        END LOOP;
    END IF;
END;
/
```

Se debe tener en cuenta que a diferencia de lo que pasa con cursores y vectores al almacenar consultas en variables mediante INTO existe una comprobaci  n de datos y se puede dar la excepci  n **NO\_DATA\_FOUND** si nuestra consulta no devuelve datos, as   que habr  a que tratar la excepci  n. En este caso no es aplicable puesto que nos aseguramos que el n  mero de tuplas de la consulta es mayor que 0.

La ventaja que aporta esta versi  n es que el volcado de datos mediante variables es m  s r  pido que mediante cursores, as   que para consultas con muchos miles de resultados, que se repitan mucho o puedan ejecutar muchos usuarios aporta m  s fluidez al sistema.

## 7. Procedimiento para mostrar consultas de múltiples t  pulas. Arrays.

Si necesitamos hacer un tratamiento avanzado de los datos, tener disponibles varias t  pulas a la vez y no ir descargando las tuplas una a una (y una   nica vez a menos que volvamos a empezar desde el principio) los cursores pueden no ser la forma m  s adecuada de trabajar. Para este prop  sito existen los arrays, que permiten acceder directamente a cada uno de los elementos mediante el   ndice. Puesto que los arrays pueden ser casi de cualquier tipo, podemos hacer una array de registros que almacene en cada   ndice una tupla completa de nuestra consulta. A modo de curiosidad se incluye esta versi  n 3 donde se rellena el vector uno a uno con un bucle de forma similar a la anterior para demostrar la versatilidad de los mismos y que se podr  an almacenar distintos resultados o varias consultas que compartiesen los campos, pero lo m  s habitual es hacerlo directamente con BULK COLLECT INTO como en la versi  n 4.

```
CREATE OR REPLACE PROCEDURE empleados_dep3 (departamento IN varchar2) IS
lin INT;
TYPE vec_emp IS VARRAY(5) OF empleados%ROWTYPE;
a vec_emp:=vec_emp();
BEGIN
    SELECT COUNT(*) INTO lin FROM empleados WHERE dpto=departamento;
    a.extend(lin);
    IF (lin>0) THEN
        FOR b IN 1 .. lin LOOP
            SELECT dni,nombre,especialidad,fechaalta,dpto,codp INTO a(b)
            FROM (SELECT ROWNUM as linea,empleados.*
                  FROM empleados
                  WHERE dpto=departamento)
            WHERE linea=b;
            DBMS_OUTPUT.PUT_LINE(a(b).dni||' '||a(b).nombre
            ||' '||a(b).especialidad||' '||a(b).fechaalta
            ||' '||a(b).dpto||' '||a(b).codp);
        END LOOP;
    END IF;
END;
/
```

El proceso es bastante similar al anterior as   que s  lo un par de consideraciones:

- Puesto que hay 5 empleados se declara el m  ximo del TIPO del vector “5”, al estar en el DECLARE estamos algo limitados en este aspecto pero el vector s  lo debe extenderse hasta lo necesario para ahorrar espacio.
- Suele convenir inicializar los vectores en su declaraci  n aunque no se les d   valor todav  a ( *a vec\_emp:=vec\_emp()* )
- Para acceder a los elementos de un array de registros “*nombre(indice).variable*”.

- Con .extend(n) se puede ampliar en “n” el número de elementos del vector hasta el máximo en el tipo (5). N no es el índice, extend(2) no hace un array de 2 elementos sino que lo extiende en 2 más de los que ya tenía.

Pasamos a la versión simplificada del procedimiento, esta vez tanto el extender el array como el volcado de los datos se hacen directamente con la orden BULK COLLECT INTO, el bucle sólo lo usamos para mostrar los datos como en los anteriores casos.

```
CREATE OR REPLACE PROCEDURE empleados_dep4 (departamento IN varchar2) IS
lin INT;
TYPE vec_emp IS VARRAY(5) OF empleados%ROWTYPE;
a vec_emp:=vec_emp();
BEGIN
    SELECT * BULK COLLECT INTO a FROM empleados WHERE dpto=departamento;

    SELECT COUNT(*) INTO lin FROM empleados WHERE dpto=departamento;
    IF (lin>0) THEN
        FOR b IN 1 .. lin LOOP
            DBMS_OUTPUT.PUT_LINE(a(b).dni||' '||a(b).nombre
                                ||' '||a(b).especialidad||' '||a(b).fechaalta
                                ||' '||a(b).dpto||' '||a(b).codp);
        END LOOP;
    END IF;
END;
/
```