TEMA

Tema 4. Cursores y disparadores

# Administración de sistemas informáticos en red

## Administración de sistemas gestores de bases de datos

Autor: Joan Pou

Linkia FP

# Tema 4: Cursores y disparadores

## ¿Qué aprenderás?

- Aplicar un lenguaje de programación de bases de datos.
- Crear funciones y procedimientos almacenados.
- Crear triggers.
- Crear y gestionar cursores.

## ¿Sabías que…?

- PL/SQL: Es un lenguaje de programación.
- Los triggers son procedimientos que se ejecutan automáticamente.
- Los cursores nos permiten manipular los datos de las tablas.

# 4. Cursores y disparadores

## 4.1. Cursores

Un cursor es un conjunto de registros devuelto por una instrucción SQL. PL/SQL utiliza cursores para gestionar las instrucciones SELECT. Los cursores son fragmentos de memoria reservados para procesar los resultados de una consulta SELECT.

Podemos distinguir dos tipos de cursores:

- **Cursores implícitos.** Este tipo de cursores se utiliza para operaciones SELECT INTO. Se usan cuando la consulta devuelve un único registro. También se usan en las operaciones insert, update o delete.
- **Cursores explícitos**. Son los cursores que son declarados y controlados por el programador. Se utilizan cuando la consulta devuelve un conjunto de registros. También se pueden utilizar en consultas que devuelven un único registro por razones de eficiencia. Son más rápidos.

Un cursor se define como cualquier otra variable de PL/SQL. Los cursores implícitos no necesitan declaración.

Atributos de los cursores:

- **SQL%ISOPEN**: True si el cursor está abierto.
- **SQL%FOUND**: True si una instrucción INSERT, UPDATE o DELETE afecta a uno o más registros. Y también si un SELECT devuelve uno o más registros.
- **SQL%NOTFOUND**: El contrario del anterior. TRUE si una instrucción INSERT, UPDATE o DELETE no afecta a ningún registro o un SELECT que no devuelve ninguna fila.
- **SQL%ROWCOUNT**: Cantidad de registros afectados por el cursor.

### 4.1.1. Cursores implícitos

Se utilizan para realizar consultas que devuelven un único registro o para operaciones de INSERT, DELETE o UPDATE. Debemos tener en cuenta:

- Si utilizamos un SELECT debe existir también la palabra clave INTO.
- Las variables que reciben los datos devueltos por el cursos deben ser del mismo tipo que las columnas de la tabla.

- Los cursores implícitos solo pueden devolver una única fila, en caso contrario se producirá una excepción.

Las excepciones asociadas a los cursores implícitos son:

- **NO_DATA_FOUND**: Cuando el resultado de un select no devuelve datos.
- **TOO_MANY_ROWS**: Cuando el resultado del select devuelve más de una fila.

Ejemplo 1:

Cursor implícito con la instrucción SELECT INTO. Devuelve el salario de un trabajador. La variable nomina es del mismo tipo que la columna salario de la tabla de trabajadores.

```
SET SERVEROUTPUT ON;
DECLARE
    nomina trabajadores.salario%TYPE;
    BEGIN
            SELECT salario INTO nomina FROM trabajadores WHERE dni='11112222A';
            dbms_output.put_line('El trabajador tiene un salario de '||nomina||' €');
    END;
    /
```

```
El trabajador tiene un salario de 60000 €

Procedimiento PL/SQL terminado correctamente.
```

Ejemplo 2:

Cursor implícito de actualización. Se utilizan los atributos de los cursores para saber el resultado del número de filas actualizadas.

```
SET SERVEROUTPUT ON;
DECLARE
        numfilas number(2);
BEGIN
  UPDATE trabajadores  SET salario = salario + 500;
  IF SQL%NOTFOUND THEN
        dbms_output.put_line('No hay trabajadores');
  ELSIF SQL%FOUND THEN
         numfilas := SQL%ROWCOUNT;
         dbms_output.put_line( numfilas || ' trabajadores actualizados ');
  END IF;
END;
/
```

```
9 trabajadores actualizados

Procedimiento PL/SQL terminado correctamente.
```

### 4.1.2. Cursores explícitos

Los cursores explícitos se utilizan para realizar consultas SELECT que pueden devolver ninguna o más de una fila. Son más rápidos y eficientes que los implícitos.

Para trabajar con un cursor explicito necesitamos realizar 4 operaciones básicas:

- Declarar el cursor en la zona de declaraciones con la siguiente sintaxis:

**CURSOR nombre_cursor IS instrucción_SELECT**
**También debemos declarar los parámetros que pueda tener el cursor:**
**CURSOR nombre_cursor(param1 tipo1, ..., paramN tipoN) IS instrucción_SELECT**

- Abrir el cursor con la instrucción OPEN. La instrucción OPEN ejecuta la setencia SELECT y el resultado se almacena en la memoria interna reservada al cursor.

```
OPEN nombre_cursor;
    o bien si tiene parámetros
OPEN nombre_cursor(valor1, valor2, ..., valorN);
```

- Leer los datos del cursor con la instrucción FETCH. Cada FETCH recupera una fila de la consulta y el cursor avanza automáticamente a la siguiente fila.

**FETCH nombre_cursor INTO lista_variables|variable_registro.**

Si es una lista de variables cada una obtendrá la columna correspondiente de la cláusula SELECT, por tanto deberán ser del mismo tipo que las columnas. Si se utiliza una única variable que obtendrá toda la información del registro la declararemos de la siguiente forma:

**Variable nombrecursor%ROWTYPE;**

- Cerrar el cursor y liberar los recursos con la instrucción CLOSE.

**CLOSE nombre_cursor**

Para recorrer los datos que obtiene el cursor deberemos realizar un bucle. Una primera forma de obtener los datos es utilizar un bucle LOOP con una sentencia EXIT condicionada que nos permita salir del bucle al final del cursor.

La sintaxis seria la siguiente:

```
OPEN nombre_cursor;
LOOP
        FETCH nombre_cursor INTO lista_variables;
        EXIT WHEN nombre_cursor%NOTFOUND;
        /* Procesamiento de los registros recuperados */
END LOOP;
CLOSE nombre_cursor;
```

Ejemplo 1:

Cursor explícito que recupera el nombre, el dni y el salario de los trabajadores.

```
DECLARE
        v_dni trabajadores.dni%type;
        v_nombre trabajadores.nombre%type;
        v_salario trabajadores.salario%type;
-- definición del cursor
        CURSOR c_trabajadores IS
        SELECT dni, nombre, salario FROM trabajadores;
BEGIN
-- abrimos el cursor espacio de memoria
        OPEN c_trabajadores;
-- bucle hasta que no queden más registros
  LOOP
        FETCH c_trabajadores into v_dni, v_nombre, v_salario;
-- salimos del bucle cuando no hay datos
        EXIT WHEN c_trabajadores%notfound;
        dbms_output.put_line(v_dni || ' ' || v_nombre || ' ' || v_salario);
  END LOOP;
-- cerramos el cursor
        CLOSE c_trabajadores;
END;
/
```

Muchas veces la cláusula del SELECT deberá seleccionar las filas de acuerdo con una condición. En estos casos podemos pasar parámetros al cursor. Los parámetros se declaran en la definición del cursor con la siguiente sintaxis:

```
CURSOR nombre_cursor(param1 tipo1, ..., paramN tipoN) IS  instrucción_SELECT
```

Ejemplo 2:

En el siguiente ejemplo pasamos un valor al cursor, el salario mínimo a partir del cual listamos a los usuarios.

```
DECLARE
        v_dni trabajadores.dni%type;
        v_nombre trabajadores.nombre%type;
        v_salario trabajadores.salario%type;
-- definición del cursor
        CURSOR c_trabajadores (p_salario NUMBER) IS
        SELECT dni, nombre, salario FROM trabajadores WHERE salario>p_salario;
BEGIN
-- abrimos el cursor espacio de memoria
        OPEN c_trabajadores(30000);
-- bucle hasta que no queden más registros
  LOOP
        FETCH c_trabajadores into v_dni, v_nombre, v_salario;
-- salimos del bucle cuando no hay datos
        EXIT WHEN c_trabajadores%notfound;
        dbms_output.put_line(v_dni || ' ' || v_nombre || ' ' || v_salario);
  END LOOP;
-- cerramos el cursor
        CLOSE c_trabajadores;
END;
/
```

En el ejemplo el parámetro aparece tres veces. En la definición del cursor, en la instrucción SELECT formando parte del WHERE y en la apertura del cursor donde le pasamos el valor.

```
11112222A Rojo Iglesias, Marta 60500
11112233A Perez Carrillo, Iván 48500
11112244B Torres Marqués, Fernando 55500
11112255B Rubio Sánchez, María 36500
22112222A Roca Benítez, Elena 35500
33112222A Nualart Vives, Carlos 30500
22334455C Perez Cano, Isabel 36500

Procedimiento PL/SQL terminado correctamente.
```

En los cursores explícitos también podemos utilizar variables del tipo %ROWTYPE en vez de utilizar una lista de variables que representa a cada una de las columnas de la tabla.

De esta forma simplificamos el cursor y con una sola variable podemos acceder a todos los datos del registro.

Ejemplo 3:

En este ejemplo observamos que la variable registro del tipo trabajadores%ROWTYPE nos sirve para almacenar y acceder a todos los datos de un trabajador.

```
DECLARE
        registro trabajadores %ROWTYPE;
        CURSOR c_trabajadores IS SELECT * FROM trabajadores;
BEGIN
        OPEN c_trabajadores;
LOOP
        FETCH c_trabajadores INTO registro;
        EXIT WHEN c_trabajadores%notfound;
        dbms_output.put_line(registro.dni || ' ' || registro.nombre || ' ' ||
registro.salario);
  END LOOP;
        CLOSE c_trabajadores;
END;
/
```

### 4.1.3. Cursores FOR..LOOP

En los cursores vistos hasta ahora teníamos que realizar una siempre un mínimo de 4 operaciones con el cursor: definir, abrir, recuperar los datos (bucle) y cerrarlo.

Podemos usar un bucle FOR..LOOP para trabajar con los cursores, en este tipo de bucle se ejecutan de forma implícita las operaciones de OPEN, FECTH y CLOSE simplificando significativamente su utilización.

```
DECLARE
        CURSOR c_trabajadores IS SELECT * FROM trabajadores;
BEGIN
        --Estructura FOR..LOOP
        FOR registro IN c_trabajadores LOOP
                dbms_output.put_line(registro.dni || ' ' || registro.nombre || ' ' ||
registro.salario);
        END LOOP;
END;
/
```

La variable registro se declara de forma implícita en el bucle y por tanto es local al mismo, debemos tener en cuenta, por tanto, que al salir del bucle la variable registro no estará disponible.


Vídeo: Cursores

## 4.2. Triggers (Disparadores)

Los disparadores o triggers de base de datos son bloques PL/SQL almacenados que se ejecutan automáticamente cuando se producen ciertos eventos.

Hay tres tipos de disparadores de bases de datos:

- **Disparadores de tablas**. Asociados a una tabla. Se disparan cuando se produce un determinado suceso que afecta a la tabla.
- **Disparadores de sustitución**. Asociados a vistas. Se disparan cuando se intenta ejecutar que afecta a la vista.
- **Disparadores del sistema**. Se disparan cuando ocurre un evento del sistema (arranque o parada de la base de datos, entrada o salida de un usuario, etcétera).

Los triggers de tablas son un bloque PL/SQL asociado a una tabla, que se ejecuta como consecuencia de una determinada operación DML (INSERT, UPDATE o DELETE) sobre dicha tabla.

La sintaxis para crear un trigger es la siguiente:

```
CREATE [OR REPLACE] TRIGGER <nombre_trigger>
{BEFORE|AFTER}
{DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]
[OR {DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]...]}
ON <nombre_tabla>
[FOR EACH ROW [WHEN (<condicion>)]] DECLARE
-- variables locales
BEGIN
-- Sentencias
[EXCEPTION]
-- Sentencias control de excepcion
END <nombre_trigger>;
```

Debemos tener en cuenta:

- El uso de REPLACE permite sobreescribir un trigger existente.
- Los triggers pueden definirse para las operaciones INSERT, UPDATE o DELETE.
- La cláusula BEFORE|AFTER indica si el trigger se ejecutará antes o después de la instrucción SQL definida (INSERT, DELETE o UPDATE).
- La cláusula OF nos permite ejecutar el trigger solo cuando la instrucción SQL afecte a la lista de campos incluidos en esta cláusula.
- Los triggers pueden actuar a nivel de fila o de orden. Si indicamos FOR EACH ROW el trigger se activa una vez por cada fila afectada por la instrucción SQL que provocó el disparo. Opcionalmente podemos incluir un WHEN para indicar una restricción a nivel de fila.

### 4.2.1. Utilización de las variables OLD y NEW

Dentro del ámbito de un trigger disponemos de las variables OLD y NEW. Estas variables se utilizan del mismo modo que cualquier otra variable PL/SQL, pero no es necesario declararlas, son de tipo %ROWTYPE y contienen una copia del registro antes (OLD) y después (NEW) de la acción SQL (INSERT, UPDATE, DELETE) que ha ejecutado el trigger.

Utilizando esta variable podemos acceder a los datos que se están insertando, actualizando o borrando.

Los registros OLD y NEW son sólo válidos dentro de los disparadores a nivel de fila. La siguiente tabla muestra los valores de OLD y NEW.

| ACCION SQL | OLD | NEW |
|---|---|---|
| INSERT | No definido; todos los campos tienen valor NULL. | Valores que serán insertados cuando se complete la orden. |
| UPDATE | Valores originales de la fila, antes de la actualización. | Nuevos valores que serán escritos cuando se complete la orden. |
| DELETE | Valores, antes del borrado de la fila. | No definidos; todos los campos tienen el valor NULL. |

Ejemplo:

El siguiente trigger se ejecutará cada vez que se modifique el salario de un trabajador. Nos mostrará por pantalla su salario antiguo, el nuevo y la diferencia entre ellos.

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE TRIGGER cambiosalario
BEFORE DELETE OR INSERT OR UPDATE ON trabajadores
FOR EACH ROW
DECLARE
  diferencia number;
BEGIN
  diferencia := :NEW.salario  - :OLD.salario;
  dbms_output.put_line('Salario antiguo: ' || :OLD.salario);
  dbms_output.put_line('Nuevo Salario: ' || :NEW.salario);
  dbms_output.put_line('Diferencia: ' || diferencia);
END;
/
```

Podemos observar el comportamiento del trigger realizando diversas operaciones sobre la tabla trabajadores.

- Modificamos el salario de un trabajador:

  **update trabajadores set salario=salario+500 where dni='11112222A';**

```
Salario antiguo: 60500
Nuevo Salario: 61000
Diferencia: 500


1 fila actualizadas.
```

- Borramos un trabajador.

  **delete from trabajadores where dni='11112222A';**

```
Salario antiguo: 61000
Nuevo Salario:
Diferencia:


1 fila eliminado
```

En ambos casos podemos comprobar el comportamiento de las variables OLD y NEW, en el caso del update nos muestran los dos valores, en el caso del delete solo muestra los valores anteriores a la ejecución del trigger.

Ejemplo 2:

El siguiente trigger registra a modo de log las modificaciones en el salario de un trabajador. En este caso guarda, el nombre del trabajador, su salario antiguo y nuevo, la fecha de modificación del mismo y el usuario que ha realizado el cambio. Para estos dos últimos valores se consultan las variables del sistema SYSDATE y USER.

La tabla modisalario almacenará el log del trigger.

```
CREATE TABLE  modisalario (
 nombre  VARCHAR2(50),
 salario_ant decimal (8,2),
 salario_nuevo decimal (8,2),
 fecha date,
 usuario VARCHAR2(50));
```

Creación del trigger modisalario.

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE TRIGGER guadarsalario
BEFORE UPDATE ON trabajadores
FOR EACH ROW
DECLARE

BEGIN
  INSERT INTO modisalario VALUES (:OLD.nombre, :OLD.salario,
:NEW.salario, SYSDATE, USER);
END;
/
```

Comprobación del mismo. Actualizamos el salario de un trabajador.

```
update trabajadores set salario=salario+1000 where dni='22334455C';
```

```
1 fila actualizadas.
```

Consultar el contenido la tabla modisalario.

```
select * from modisalario;
```

| | NOMBRE | SALARIO_ANT | SALARIO_NUEVO | FECHA | USUARIO |
|---|---|---|---|---|---|
| 1 | Perez Cano, Isabel | 36500 | 37500 | 11/09/17 | JOAN |

### 4.2.2. Activar, desactivar, compilar y eliminar triggers

Los triggers pueden estar activados o desactivados. Cuando se crea el trigger por defecto está activado.

- Para desactivar un trigger utilizamos la orden:

  **ALTER TRIGGER nombre_trigger DISABLE.**

- Para volver a activar un trigger utilizamos la orden:

  **ALTER TRIGGER nombre_trigger DISABLE.**

- Para eliminar un trigger utilizamos la orden:

  **DROP TRIGGER nombretrigger.**


Vídeo: Triggers

## Recursos y enlaces

- Lenguaje PL/SQL

- Triggers

## Conceptos clave

- **PL/SQL**: Lenguaje de programación de base de datos.
- **Bloque anónimo**: Código que se ejecuta y no se almacena en la base de datos.
- **Cursor**: Conjunto de registros devueltos por una instrucción SQL.
- **Trigger**: Bloque de código que se ejecuta automáticamente tras un evento.
- **Excepciones:** Permiten controlar los errores en tiempo de ejecución.

# Test de autoevaluación

Los cursores explícitos se utilizan básicamente para realizar operaciones del tipo:

a) INSERT
b) DELETE
c) UPDATE
d) SELECT

Para leer los datos de un cursor utilizamos la orden:

a) READ
b) FETCH
c) SELECT
d) OPEN

¿Cuál de los siguientes no es un tipo de disparador?

a) Sistema
b) Tabla
c) Vista
d) Sustitución

Los disparadores se pueden ejecutar al realizar una operación de:

a) INSERT
b) SELECT
c) DELETE
d) UPDATE

# Ponlo en práctica

## Actividad 1

Utilizando la tabla de vehículos. Crea un cursor que nos visualice los datos de una determinada marca de vehículos.

# SOLUCIONARIOS

## Test de autoevaluación

Los cursores explícitos se utilizan básicamente para realizar operaciones del tipo:

a) INSERT
b) DELETE
c) UPDATE
**d) SELECT**

Para leer los datos de un cursor utilizamos la orden:

a) READ
**b) FETCH**
c) SELECT
d) OPEN

¿Cuál de los siguientes no es un tipo de disparador?

a) Sistema
b) Tabla
**c) Vista**
d) Sustitución

Los disparadores se pueden ejecutar al realizar una operación de:

**a) INSERT**
b) SELECT
**c) DELETE**
**d) UPDATE**

# Ponlo en práctica

## Actividad 1

Utilizando la tabla de vehículos. Crea un cursor que nos visualice los datos de una determinada marca de vehículos.

**SOLUCIÓN:**

```
DECLARE
        registro tvehiculos %ROWTYPE;
        CURSOR c_vehiculo IS SELECT * FROM tvehiculos where marca='Seat';
BEGIN
        OPEN c_vehiculo;
LOOP
        FETCH c_vehiculo INTO registro;
        EXIT WHEN c_vehiculo%notfound;
        dbms_output.put_line(registro.matricula || ' ' || registro.marca || ' ' || registro.modelo || ' ' ||
registro.kilometros || ' ' || registro.precio);
  END LOOP;
        CLOSE c_vehiculo;
END;
/
```

Estructura de datos utilizada en los ejemplos. Tabla trabajadores y empresas con sus datos.

```
create tablespace empresa datafile 'c:\oracle\empresa.dbf' size 100M;

CREATE TABLE  trabajadores (
 dni  VARCHAR(9) PRIMARY KEY ,
 nombre  VARCHAR(50),
 ciudad VARCHAR(40),
 antiguedad   int,
 salario decimal (8,2),
 departamento int not null);


INSERT INTO trabajadores VALUES ('11112222A','Rojo Iglesias, Marta', 'Barcelona', 12,
60000, 2);
INSERT INTO trabajadores VALUES ('11112233A','Perez Carrillo, Iván','Bilbao',5, 48000, 2);
INSERT INTO trabajadores VALUES ('11112244B','Torres Marqués, Fernando','Madrid',11,
55000, 2);
INSERT INTO trabajadores VALUES ('11112255B','Rubio Sánchez, María','Sevilla',4, 36000,
3);
INSERT INTO trabajadores VALUES ('11112266C','Llamas Rocasolano,
Isabel','Barcelona',13, 28000, 3);
INSERT INTO trabajadores VALUES ('11112222C','Gomez Corachán, Manuel','Madrid',15,
22000, 1);
INSERT INTO trabajadores VALUES ('22112222A','Roca Benítez, Elena','Sevilla',6, 35000,
4);
INSERT INTO trabajadores VALUES ('33112222A','Nualart Vives, Carlos', 'Barcelona',10,
30000, 4);
INSERT INTO trabajadores VALUES ('22334455C','Perez Cano, Isabel', 'Madrid',2, 36000,
3);

SELECT * FROM trabajadores;



CREATE TABLE departamento (
numdep int not null,
nombredep varchar(20) not null unique,
primary key (numdep));

-- insertamos datos
INSERT INTO departamento VALUES ('1','Contabilidad');
INSERT INTO departamento VALUES ('2','Recursos Humanos');
INSERT INTO departamento VALUES ('3','Informática');
INSERT INTO departamento VALUES ('4','Comercial');
INSERT INTO departamento VALUES ('5','Facturación');
```