

# PROGRAMACIÓN EN ORACLE

## PARTE III



# CURSORES

- **Definición Cursor:** Objeto que hace referencia a un conjunto de datos obtenidos de una consulta.
- **Sintaxis de cursores:**

```
-- DECLARAR  un nuevo cursor en la sección DECLARE  
CURSOR cursor_name IS SELECT_statement;  
-- OPEN: Abre el cursor para acceder a sus filas asociadas  
OPEN cursor_name;  
-- FETCH: Extrae la siguiente fila de un cursor  
FETCH cursor_name INTO variable list;  
-- CLOSE: Cierra el cursor  
CLOSE cursor_name ;
```

# CURSORES

- PL/SQL utiliza cursores para gestionar las instrucciones SELECT. Un cursor es un conjunto de registros devuelto por una instrucción SQL. Técnicamente los cursores son fragmentos de memoria que son 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 usa cuando la consulta devuelve un único registro.
- **Cursores explícitos.** Son los cursores que son declarados y controlados por el programador. Se utilizan cuando la consulta devuelve un conjunto de registros. Ocasionalmente también se utilizan 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 y debe nombrarse de acuerdo a los mismos convenios que cualquier otra variable. Los cursores implícitos no necesitan declaración.

# CURSORES

## Atributos de cursores:

- **%ISOPEN** : True si el cursor esta abierto (entre open y close),
- **%FOUND** :
  - NULL antes de ejecutar.
  - TRUE si uno o mas registros fueron inserted, merged, updated, o deleted o si solo 1 registro fue seleccionado.
  - FALSE si ningún registro fue seleccionado, merged, updated, inserted, o deleted.
- **%NOTFOUND**:
  - NULL antes de ejecutar
  - TRUE si ningún registro fue seleccionado, merged, updated, inserted, o deleted.
  - FALSE si uno o mas registros fueron inserted, merged, updated, deleted o seleccionado.
- **%ROWCOUNT** Cantidad de registros afectados por el cursor.

# EJEMPLO CURSOR EXPLÍCITO1

/\* VISUALIZAR Nombre, Oficio, Salario y Comisión de los empleados del departamento 10 \*/

**DECLARE**

**CURSOR** cursorEmple **IS** SELECT NOMBRE, OFICIO, SALARIO, COMISION FROM EMPLE WHERE  
DEPT\_NO=10 ORDER BY 1;

VNOMBRE EMPLE.NOMBRE%TYPE;

VOFICIO EMPLE.OFICIO%TYPE;

VSALARIO EMPLE.SALARIO%TYPE;

VCOMISION EMPLE.COMISION%TYPE;

**BEGIN**

**OPEN** cursorEmple;

**FETCH** cursorEmple **INTO** VNOMBRE,VOFICIO,VSALARIO,VCOMISION;

**WHILE** cursorEmple%FOUND

**LOOP**

DBMS\_OUTPUT.PUT\_LINE(VNOMBRE || ' ' || VOFICIO || ' ' || VSALARIO || ' ' || NVL(VCOMISION,0));

-- Para evitar errores de formato se convierte el valor numérico a carácter

DBMS\_OUTPUT.PUT\_LINE('EL SUELDO DEL EMPLEADO ES ' || **TO\_CHAR**(VSALARIO+NVL(VCOMISION,0)));

**FETCH** cursorEmple **INTO** VNOMBRE,VOFICIO,VSALARIO,VCOMISION;

**END LOOP;**

**CLOSE** cursorEmple;

**END;**

/

# EJEMPLO CURSOR EXPLÍCITO2

```
/* visualizar TODOS los datos de los empleados del departamento 10 */
set serveroutput on;
DECLARE
CURSOR cursorEmple IS SELECT * FROM EMPLE WHERE DEPT_NO=10 ORDER BY 1;
VREG emple%rowtype;
BEGIN
OPEN cursorEmple;
FETCH cursorEmple INTO VREG;
WHILE cursorEmple%FOUND LOOP
DBMS_OUTPUT.PUT_LINE(VREG.NOMBRE||' '||VREG.OFICIO||' '||VREG.SALARIO||'
    '||NVL(VREG.COMISION,0));
FETCH cursorEmple INTO VREG;
END LOOP;
DBMS_OUTPUT.PUT_LINE('SE HAN PROCESADO '||cursorEmple%ROWCOUNT||'
    Registros');
CLOSE cursorEmple;
END;
/
```

# EJEMPLO CURSOR EXPLÍCITO3

/\* visualizar los datos (NOMBRE,OFICIO,SALARIO Y COMISIÒN) de los empleados del departamento 10, utilizando un registro \*/

**DECLARE**

CURSOR cursorEmple IS SELECT NOMBRE,OFICIO,SALARIO,COMISION FROM EMPLE WHERE  
DEPT\_NO=10 ORDER BY 1;

**type REmple IS RECORD** ( VNOMBRE EMPLE.NOMBRE%TYPE, VOFICIO EMPLE.OFICIO%TYPE,  
VSALARIO EMPLE.SALARIO%TYPE, VCOMISION EMPLE.COMISION%TYPE );

VREG REmple;

**BEGIN**

OPEN cursorEmple;

FETCH cursorEmple INTO VREG;

WHILE cursorEmple%FOUND LOOP

DBMS\_OUTPUT.PUT\_LINE(VREG.VNOMBRE||' '||VREG.VOFICIO||' '||VREG.VSALARIO||'  
'||NVL(VREG.VCOMISION,0));

FETCH cursorEmple INTO VREG;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('SE HAN PROCESADO '||cursorEmple%ROWCOUNT||' Registros');

CLOSE cursorEmple;

END;

/

# CURSORES CON ESTRUCTURA FOR

- La utilización de cursores implica:
  - Declarar el cursor.
  - Declarar una variable que recogerá los datos del cursor.
  - Abrir el Cursor.
  - Recuperar con FETCH una a una las filas extraídas, introduciendo los datos en la variable, procesándolos y comprobando también si se han recuperado todos los datos o no.
  - Cerrar el Cursor.
- La estructura de cursores FOR ... LOOP simplifica estas tareas realizándolas todas, excepto la declaración del cursor, de forma implícita.



# CURSORES CON ESTRUCTURA FOR

- 1.- Se declara el cursor en la sección declarativa:

`CURSOR nombreCursor IS sentenciaSelect;`

- 2.- Se procesa el cursor utilizando el siguiente formato:

`FOR nombreVarReg IN nombreCursor LOOP`

`...`

`END LOOP;`

Donde nombreVarReg será creada automáticamente por el bucle para recoger los datos del cursor.

Al entrar al bucle se abre el cursor de manera automática.

Se declara implícitamente la variable nombreVarReg de tipo nombreVarReg%rowtype y se ejecuta un FETCH implícito, cuyo resultado quedará en nombreVarReg.

A continuación, se realizarán las sentencias hasta llegar al END LOOP; que hará volver al FOR ... LOOP ejecutando el siguiente FETCH implícito...

# Ejemplo de cursor explícito con FOR

```
/* Visualizar el apellido, oficio y comisión de los  
empleados cuya comisión supera 500 euros*/  
set serveroutput on;  
DECLARE  
CURSOR cursorEmple IS SELECT apellido, oficio, comision  
FROM emple WHERE comision >500;  
BEGIN  
FOR VREG IN cursorEmple LOOP  
    DBMS_OUTPUT.PUT_LINE(VREG.APELLIDO || '  
    ' || VREG.OFICIO || ' ' || NVL(VREG.COMISION,0));  
END LOOP;  
END;
```

# Ejemplo de cursor implícito con FOR

```
DECLARE
remple emple%ROWTYPE;
begin
for var in (select * from emple) loop
dbms_output.put_line(var.nombre);
end loop;
end;
/
```

# Ejemplo de cursor explícito con FOR

```
DECLARE
CURSOR c_emp IS /*CURSOR*/
select nombre, salario from emple;
BEGIN
FOR fila IN c_emp LOOP /*no es necesario definir la
    variable fila, será de tipo %ROW */
dbms_output.put_line(fila.nombre || ' tiene un salario de
    ' || fila.salario);
END LOOP;
END;
/
```

# Actividades VI: Cursores

- **EJ1:** Visualizar el apellido y la fecha de alta de todos los empleados ordenados por fecha de alta. Utilizando un bucle FOR.
- **EJ2:** Utilizando un bucle WHILE.
- **EJ3:** Mostrar el nombre y localización de todos los centros en orden.
- **EJ4:** Visualizar el nombre, apellido y nombre del departamento de todos los empleados.

# ALIAS Y CAMPOS CALCULADOS EN UN CURSOR

- Si en un cursor necesitamos incluir un campo calculado, función de agrupamiento o función almacenada, es necesario darle un alias, con el que se tratará ese valor a partir de que el cursor sea abierto. Los alias dentro de PL/SQL no precisan las comillas dobles del SQL estándar.

```
DECLARE
```

```
CURSOR C1 IS SELECT COUNT(*)  numemple, DEPT_NO from emple  
                group by DEPT_NO;
```

```
BEGIN
```

```
FOR v in C1 LOOP
```

```
    DBMS_OUTPUT.PUT_LINE ('EL DEPARTAMENTO ' || v.DEPT_NO || '  
    TIENE ' || v.numemple || ' EMPLEADOS');
```

```
END LOOP;
```

```
END;
```

# Cursor con dos tablas

- Si un cursor debe seleccionar campos de distintas tablas, que coinciden en el nombre, también será necesario utilizar un alias, por ejemplo, en la tabla EMPLE y en la tabla DEPART existe un campo nombre, si quisiéramos obtener los dos con un mismo cursor tendríamos que poner alias a los campos :

```
DECLARE
```

```
CURSOR C1 IS SELECT depart.dnombre nombreD, emple.nombre emplenom  
from depart, emple where emple.dept_no=depart.dept_no;
```

```
BEGIN
```

```
FOR v in C1 LOOP
```

```
    DBMS_OUTPUT.PUT_LINE ('EL empleado ' || v.emplenom || ' trabaja en ' ||  
        v.nombreD );
```

```
END LOOP;
```

```
END;
```

```
/
```

# USO DE CURSORES PARA ACTUALIZAR FILAS

- Si utilizamos un cursor para actualizar determinadas filas, es necesario declararlo de una forma específica incluyendo al final del select la sentencia **FOR UPDATE**.

**CURSOR nombreCursor <declaraciónDelCursor> FOR UPDATE**

- FOR UPDATE indica que las filas seleccionadas por el cursor van a ser actualizadas o borradas. Todas las filas seleccionadas serán bloqueadas al abrirse el cursor y serán desbloqueadas al terminar las actualizaciones (al ejecutar COMMIT implícita o explícitamente)
- Una vez declarado un cursor FOR UPDATE, se incluirá el especificador **CURRENT of <nombrecursor>** en la cláusula **WHERE** para actualizar (UPDATE) o borrar (DELETE) la última fila recuperada con la orden FETCH.

**{UPDATE | DELETE} .... WHERE CURRENT OF nombreCursor**



# Ejemplo CURSOR FOR UPDATE

```
DECLARE
CURSOR C1 IS SELECT * FROM EMPLE FOR UPDATE;
V EMPLE%ROWTYPE;
BEGIN
OPEN C1;
FETCH C1 INTO V;
WHILE (C1%FOUND) LOOP
DBMS_OUTPUT.PUT_LINE (CHR(10)||V.NOMBRE ||','|| V.APELLIDO );
IF V.OFICIO = 'ANALISTA' THEN
DBMS_OUTPUT.PUT_LINE ('ANALISTA, LE SUBIMOS EL SUELDO');
UPDATE EMPLE SET SALARIO=SALARIO+SALARIO*0.1 WHERE CURRENT OF C1;
END IF;
FETCH C1 INTO V;
END LOOP;
CLOSE C1;
END;
/
```

# CURSOR FOR UPDATE

- Si la columna del cursor hace referencia a múltiples tablas, se deberá usar **FOR UPDATE OF nombreDeColumna**, con lo que únicamente se bloquearán las filas correspondientes de la tabla que tenga la columna especificada.

```
CURSOR nombreCursor <declaraciónDelCursor> FOR UPDATE  
OF nombreColumna
```

Ej. CURSOR Cemple IS SELECT oficio, salario FROM  
emple, depart WHERE  
emple.dep\_no=depart.dept\_no **FOR UPDATE OF  
salario;**

# CURSOR FOR UPDATE

- La utilización de la cláusula FOR UPDATE, en ocasiones puede ser problemática ya que:
  - Se bloquean todas las filas de la SELECT, no solo la que se está actualizando en un momento dado.
  - Si se ejecuta un COMMIT, después ya no se puede ejecutar FETCH. Es decir, hay que esperar a que estén todas las filas actualizadas para confirmar los cambios.
  - Se puede utilizar un identificador de fila (**ROWID**) como condición de selección para actualizar filas, esto se hace en la definición del cursor:

CURSOR nombreCursor IS SELECT col1, col2, .., ROWID;

Al ejecutar FETCH se guardará el número de la fila en una variable o en un campo de la variable cursor. Después ese número se utilizará en la cláusula WHERE de la actualización:

{UPDATE|DELETE} ... WHERE ROWID = <VariablequeguardaROWID>

# CURSOR FOR UPDATE (ROWID)

```
set serveroutput on;
DECLARE
cursor cEmple is SELECT NOMBRE, SALARIO, ROWID FROM EMPLE WHERE DEPT_NO=10;
VRegEmple cEmple%ROWTYPE;
id ROWID; -- Tipo ROWID
BEGIN
OPEN cEmple;
dbms_output.put_line('Cursor abierto');
FETCH cEmple INTO VRegEmple;
WHILE cEmple%FOUND LOOP
id:=VregEmple.ROWID;
dbms_output.put_line('Empleado ' || VRegEmple.nombre || ' se le va a actualizar el
    sueldo' || VRegEmple.salario || ' en UN 5%');
UPDATE EMPLE SET SALARIO=SALARIO+SALARIO*5/100 WHERE ROWID=id;
FETCH cEmple INTO VRegEmple;
END LOOP;
CLOSE cEmple;
END;
/
```

# ACTIVIDADES VII: Ampliación Cursores

- **EJ1:** Hacer un procedimiento para subir el salario a todos los empleados del departamento indicado en la llamada. La subida será el porcentaje también indicado en la llamada.
- **EJ2:** Repetir el ejercicio anterior, pero se subirá el sueldo a todos los empleados cuyo departamento esté en la tabla depart.
- **EJ3:** Crear una función **fverempleoficio** al que le paso un oficio y me muestra el nombre, apellidos, oficio, nombre departamento, nombre del centro de los empleados que tienen ese oficio y devuelve el número de empleados de ese oficio.
- **EJ4:** Crear un procedimiento pcalcularsueldosdep al que le paso un número de departamento y me va sumando los sueldos por un lado y las comisiones por otra y cuántas comisiones nulas hay. Luego muestra un listado con el total de dinero en sueldos, el total en comisiones, cuántos empleados tienen comisión y cuántos no, y el total de sueldos y comisiones.

# Triggers o disparadores de BD

- Son bloques PL/SQL almacenados que se ejecutan o disparan automáticamente cuando se producen ciertos eventos.
- Hay tres tipos:
  - **Disparadores de tablas.** Se disparan cuando se produce un determinado suceso o evento de manipulación que afecta a la tabla (insertado, borrado o actualización de filas).
  - **Disparadores de sustitución.** Asociados a vistas. Se disparan cuando se intenta ejecutar un comando de manipulación que afecta a la vista.
  - **Disparadores del sistema.** Se disparan cuando se ocurre un evento del sistema (arranque o parada de la BD, entrada o salida de un usuario, etc.) o una instrucción de definición de datos (creación, modificación o eliminación de una tabla u objeto).

# Disparadores de Tablas

- Un trigger es un bloque PL/SQL asociado a una tabla, que se ejecuta como consecuencia de una determinada instrucción SQL (una operación DML: INSERT, UPDATE o DELETE) sobre dicha tabla.
- Un disparador es un código que se lanza cada vez que se ha modificado o se va a modificar el contenido de una tabla. Puede ejecutarse a nivel de la consulta, o a nivel de cada línea afectada por la consulta. Puede hacerlo antes o después de la consulta, y solo por ciertos tipos de consulta (insert/update/delete), y eventualmente solo cuando cierto/s campo/s están afectado/s.

# Disparadores o Triggers de tablas

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>;
```



# Disparadores o Triggers de tablas

- Los triggers pueden definirse para las operaciones INSERT, UPDATE o DELETE, y pueden ejecutarse antes o después de la operación. El modificador BEFORE|AFTER indica que el trigger se ejecutará antes o después de ejecutarse la sentencia SQL definida por DELETE INSERT UPDATE. Si incluimos el modificador **OF** el trigger solo se ejecutará cuando la sentencia SQL afecte a los campos incluidos en la lista.
- El alcance de los disparadores puede ser de fila o de orden. El modificador FOR EACH ROW indica que el trigger se disparará cada vez que se realizan operaciones sobre una fila de la tabla. Si se acompaña del modificador WHEN, se establece una restricción; el trigger solo actuará, sobre las filas que satisfagan la restricción.

# Disparadores o Triggers de tablas

- El cuerpo de un trigger es un bloque PL/SQL. Cualquier orden que sea legal en un bloque PL/SQL, es legal en el cuerpo de un disparador, con las siguientes restricciones:
  - Un disparador no puede emitir ninguna orden de control de transacciones: **COMMIT**, **ROLLBACK** o **SAVEPOINT**. El disparador se activa como parte de la ejecución de la orden que provocó el disparo, y forma parte de la misma transacción que dicha orden. Cuando la orden que provoca el disparo es confirmada o cancelada, se confirma o cancela también el trabajo realizado por el disparador.
  - Por razones idénticas, ningún procedimiento o función llamado por el disparador puede emitir órdenes de control de transacciones.
  - El cuerpo del disparador no puede contener ninguna declaración de variables LONG o LONG RAW
  - No se pueden utilizar comandos DDL
  - No puede contener instrucciones que consulten o modifiquen tablas mutantes (tabla que está siendo modificada por una sentencia UPDATE, DELETE o Insert en una misma sesión).

# Disparadores o Triggers de tablas

- **INSERT, DELETE, UPDATE:** Define qué tipo de orden DML provoca la activación del disparador.
- **BEFORE , AFTER** Define si el disparador se activa antes o después de que se ejecute la orden.
- **FOR EACH ROW** Los disparadores con nivel de fila se activan una vez por cada fila afectada por la orden que provocó el disparo. Los disparadores con nivel de orden se activan sólo una vez, antes o después de la orden, independientemente del número de filas afectadas por ella. Se puede incluir la cláusula **FOR EACH STATEMENT**, aunque no es necesario, se asume por omisión. Los disparadores con nivel de fila se identifican por la cláusula **FOR EACH ROW** en la definición del disparador.

# Orden de ejecución de los disparadores de tablas

- Una misma tabla puede tener varios disparadores, El orden en que se comprueban los disparadores será el siguiente:
- Antes de comenzar a ejecutar la orden que produce el disparo, se ejecutarán los disparadores BEFORE ... FOR EACH STATEMENT.
- Para cada fila afectada por la orden:
  - 1.- Se ejecutan los disparadores BEFORE ... FOR EACH ROW
  - 2.- Se ejecuta la actualización de la fila (INSERT, UPDATE o DELETE). En este momento se bloquea la fila hasta que la transacción se confirma.
  - 3.- Se ejecutan los disparadores AFTER ... FOR EACH ROW
- Una vez realizada la actualización se ejecutarán los disparadores AFTER ... FOR EACH STATEMENT.

# Disparadores o Triggers de tablas

- 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, con la salvedad de que 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 con nivel de fila.

La siguiente tabla muestra los valores de OLD y NEW.

ACCION SQL	OLD	NEW
INSERT	No definido; todos los campos toman 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 toman el valor NULL.

# Ejemplo Trigger subida de salario

1/ Crearemos una tabla TMODEmple (desde la interfaz gráfica o desde la línea de comando) con un campo VARCHAR2(100) para almacenar los mensajes que queremos guardar cuando se produzca la actualización del sueldo de un empleado.

2/ Código del trigger:

```
CREATE OR REPLACE TRIGGER SUBIDASALARIO
AFTER UPDATE OF SALARIO
ON EMPL
FOR EACH ROW
BEGIN
INSERT INTO TmodEmple VALUES ('SUBIDA SALARIO EMPLEADO
'||:OLD.EMP_NO);
END;
/
```

# Ejemplo Trigger subida de salario

3/ Visualizar el salario del empleado 7369 para luego poder comprobar la actualización.

```
select salario from emple where emp_no=7369;
```

4/ Actualizar el salario de ese empleado en un 10%:

```
update emple set salario=salario+salario*0.10  
where emp_no=7369;
```

5/ Consultar los registros de la tabla TMODEmple y ver que se ha añadido un nuevo mensaje.



# La restricción del trigger de tablas

- La cláusula WHEN seguida de una condición restringe la ejecución del trigger al cumplimiento de la condición especificada. Esta condición tiene algunas limitaciones:
- Sólo se puede utilizar con triggers a nivel de fila (FOR EACH ROW)
- Se trata de una condición SQL, no PL/SQL
- No puede incluir una consulta a la misma tabla o a otras tablas o vistas



# Ejemplo Triggers borrado empleado

- Trigger que se disparará cada vez que se elimine un empleado, guardando su número de empleado, apellido y departamento en la tabla TBorraEmple que se creará previamente con un solo campo VARCHAR2(100).

```
CREATE OR REPLACE TRIGGER BORRAEMPLE
before DELETE
ON EMPL
FOR EACH ROW
BEGIN
INSERT INTO TBORRAEmple VALUES ('BORRADO EMPLEADO
'||:OLD.EMP_NO||' '||:old.APELLIDO||' DEPARTAMENTO
'||:OLD.DEPT_NO);
END;
/
```

# Ejemplo Triggers borrado empleado

- Si quisiéramos incluir una restricción para que sólo se ejecute el disparador cuando el empleado borrado sea el PRESIDENTE, lo indicaremos insertando una cláusula WHEN antes del cuerpo del trigger:

create or replace TRIGGER **BORRAEMPLESoloPresidente**

before DELETE

ON EMPLE

FOR EACH ROW **WHEN (OLD.OFICIO = 'PRESIDENTE')**

BEGIN

INSERT INTO TBORRAEmple VALUES ('BORRADO EMPLEADO '||:OLD.EMP\_NO||'  
'||:old.APELLIDO||' DEPARTAMENTO '||:OLD.DEPT\_NO);

END;

/

- Probar el trigger:

SELECT \* FROM EMPLE WHERE OFICIO LIKE 'PRESIDENTE';

DELETE FROM EMPLE WHERE OFICIO LIKE 'PRESIDENTE';

ROLLBACK; -- restablecer el registro borrado

# Ejemplo Triggers borrado empleado

```
SQL>select NOMBRE,APELLIDO from emple  
      where emp_no=7366;
```

```
SQL>DELETE FROM emple where  
      emp_no=7366;
```

-- Comprobaremos que se ha creado un nuevo  
registro en la tabla TBORRAEMPLE.

```
SQL>ROLLBACK;
```

# Trigger de tablas

- Cuando se dispara un trigger, éste forma parte de la operación de actualización que lo disparó, de forma que si el trigger falla, Oracle dará por fallida la actualización completa. Aunque el fallo se produzca a nivel de una sola fila, Oracle hará ROLLBACK de toda la actualización.
- Se puede asociar varios triggers a una tabla o utilizar un solo trigger con múltiples eventos de disparo.

# Múltiples eventos de disparo y predicados condicionales

- Un mismo trigger puede ser disparado por distintas operaciones o eventos de disparo. Para indicarlo se utilizará OR.
- Ej. BEFORE DELETE OR UPDATE ON Emple ...
- En estos casos, Oracle permite la utilización de predicados condicionales que devolverán un valor true o false para cada una de las posibles operaciones:
  - **INSERTING** Devuelve TRUE si la orden de disparo es INSERT; FALSE en otro caso.
  - **UPDATING** TRUE si la orden de disparo es UPDATE; FALSE en otro caso.
  - **DELETING** TRUE si la orden de disparo es DELETE; FALSE en otro caso.
  - **UPDATING (nombreColumna)** TRUE si la orden de disparo es UPDATE y la columna especificada ha sido actualizada; FALSE en otro caso.

# Ejemplo de UNION DE INSERT, DELETE Y UPDATE EN UN SOLO TRIGGER

```
CREATE OR REPLACE TRIGGER tdepart BEFORE DELETE OR INSERT OR UPDATE ON
  DEPART
FOR EACH ROW
declare
BEGIN
IF INSERTING THEN
DBMS_OUTPUT.PUT_LINE('Incidencia, nuevo departamento');
END IF;
IF DELETING THEN
DBMS_OUTPUT.PUT_LINE('Incidencia, borrado de departamento');
END IF;
IF UPDATING THEN
  DBMS_OUTPUT.PUT_LINE('Incidencia, actualizado de departamento');
END IF;
END;
/
```

# Ejemplo de UNION DE INSERT, DELETE Y UPDATE EN UN SOLO TRIGGER

- Comprobación:

```
Sql> insert into depart values (90,'prueba',1,7566,1000,10,11);
```

1 filas insertadas.

Incidencia, nuevo departamento

```
Sql> update depart set dnombre='PRUEBAS' where dept_no=90;
```

1 filas actualizadas.

Incidencia, actualizado de departamento

R	DEPT_NO	R	DNOMBRE	R	NUMCE	R	DIREC	R	TDIR	R	PRESU	R	DEPDE
1	90		PRUEBAS		1		7566 A				1000		10
2	10		CONTABILIDAD		2		7782 F				-1233,8		30
3	20		INVESTIGACION		1		7566 P				12346,897		(null)
4	30		VENTAS		2		7698 P				30000		40
5	40		PRODUCCION		3		(null) (null)				(null)		(null)

```
Sql> delete from depart where dept_no=90;
```

confirmadas.

1 filas eliminado

# ACTIVIDADES VIII

**EJ1.-** Escribe un disparador que inserte en la tabla *auditaremp* (*col1* (*VARCHAR (200)* ) cualquier cambio que supere el 5% del salario del empleado indicando la fecha y hora, el empleado, y el salario anterior y posterior.

**EJ2.-** Escribe un disparador de base de datos que permita auditar las operaciones de inserción o borrado de datos que se realicen en la tabla EMPLE según las siguientes especificaciones.

- Se creará desde SQL\*Plus la tabla auditaremp con la columna col1 VARCHAR2(200).
- Cuando se produzca cualquier manipulación, se insertará un fila en dicha tabla que contendrá: fecha y hora, número del empleado, apellido y la operación de actualización INSERCIÓN o BORRADO.

**EJ3.-** Escribe un trigger que permita auditar las modificaciones en la tabla EMPLEADOS insertado en la tabla auditaremp los siguientes datos: fecha y hora, número de empleado, apellido , la operación de actualización MODIFICACIÓN y el valor anterior y el valor nuevo de cada columna modificada (sólo en las columnas modificadas).



# DISPARADORES DE SUSTITUCIÓN

- Son disparadores asociados a **vistas** que arrancan al ejecutarse una sentencia de actualización sobre la vista a la que están asociados. Se ejecutan en lugar de (INSTEAD OF) la orden de manipulación que produce el disparo del trigger.

```
CREATE [OR REPLACE] TRIGGER <nombre_trigger>
INSTEAD OF
{DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]
[OR {DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]...}]}
ON <nombre_VISTA>
[FOR EACH ROW]
<CUERPO DEL trigger>
```

# DISPARADORES DE SUSTITUCIÓN

- Se ejecutan siempre a nivel de fila
- La sintaxis consiste en que se sustituye el AFTER/BEFORE por un INSTEAD OF.
- No se puede especificar una restricción de disparo mediante la cláusula WHEN, pero se puede utilizar una estructura alternativa dentro del bloque PL/SQL.
- Vamos a crear una vista que muestre el número de empleado, nombre, apellido y nombre del departamento

CREATE VIEW VEMPLE AS

```
SELECT EMP_NO, NOMBRE, APELLIDO, DNOMBRE FROM EMPLE,  
       DEPART WHERE EMPLE.DEPT_NO=DEPART.DEPT_NO;
```

- Una manipulación como insert , delete o update en esta vista da error:  
update vemple set dnombre ='PRODUCCION' where apellido='GIL';

# DISPARADORES DE SUSTITUCIÓN

/\* Creación de una vista que visualice el número de empleado, apellido, oficio, nombre del departamento y localización de todos los empleados \*/

```
create or replace VIEW VEMPLEDEP AS SELECT  
    EMP_NO, APELLIDO, OFICIO, DNOMBRE, CIUDAD  
FROM EMPL E, DEPART D, CENTROS C WHERE  
    E.DEPT_NO=D.DEPT_NO AND  
    D.NUMCE=C.NUMCE;  
SELECT * FROM VEMPLEDEP;
```

# DISPARADORES DE SUSTITUCIÓN

- Todas las operaciones DML sobre la vista darían error:

Sql> DELETE VEMPLEDEP WHERE EMP\_NO=7799;

Sql> INSERT INTO VEMPLEDEP VALUES  
(7999,'TORRES','PROFE','CONTABILIDAD','SEVILLA');

```
SQL> UPDATE VEMPLEDEP SET DNOMBRE='CONTABILIDAD' WHERE APELLIDO='TORRES';  
UPDATE VEMPLEDEP SET DNOMBRE='CONTABILIDAD' WHERE APELLIDO='TORRES'  
*  
ERROR at line 1:  
ORA-01779: cannot modify a column which maps to a non key-preserved table
```

# DISPARADORES DE SUSTITUCIÓN

- Para facilitar estas operaciones de manipulación crearemos el siguiente disparador:  
CREATE OR REPLACE TRIGGER **GEMPLE** INSTEAD OF DELETE OR INSERT OR UPDATE  
ON VEMPLEDEP FOR EACH ROW  
DECLARE  
VDept DEPART.DEPT\_NO%TYPE;  
BEGIN  
IF DELETING THEN -- SI SE PRETENDE BORRAR UNA FILA  
DELETE FROM EMPLE WHERE EMP\_NO=:OLD.EMP\_NO;  
ELSIF INSERTING THEN  
SELECT DEPT\_NO INTO VDept FROM DEPART,CENTROS WHERE DEPART.DNOMBRE=:NEW.DNOMBRE AND  
CIUDAD=:NEW.CIUDAD;  
INSERT INTO EMPLE (EMP\_NO, APELLIDO,OFICIO,DEPT\_NO) VALUES  
(:NEW.EMP\_NO,:NEW.APELLIDO,:NEW.OFICIO,VDept);  
ELSIF UPDATING ('DNOMBRE') THEN -- SI SE TRATA DE ACTUALIZAR LA COLUMNA DNOMBRE  
SELECT DEPT\_NO INTO VDept FROM DEPART WHERE DNOMBRE=:NEW.DNOMBRE;  
UPDATE EMPLE SET DEPT\_NO=VDept WHERE EMP\_NO=:OLD.EMP\_NO;  
  
ELSIF UPDATING ('OFICIO') THEN -- SI SE TRATA DE ACTUALIZAR LA COLUMNA OFICIO  
UPDATE EMPLE SET OFICIO=:NEW.OFICIO WHERE EMP\_NO=:OLD.EMP\_NO;  
ELSE  
RAISE\_APPLICATION\_ERROR(-20500, 'ERROR EN LA ACTUALIZACIO');  
END IF;  
END;  
/

# DISPARADORES DE SUSTITUCIÓN

- Una vez creada la vista VEMPLEDEP, las operaciones anteriores podrían realizarse:

```
Sql>INSERT INTO VEMPLEDEP VALUES  
    (7999,'TORRES','PROFE','CONTABILIDAD','SEVILLA');
```

```
Sql>UPDATE VEMPLEDEP SET  
    DNOMBRE='CONTABILIDAD' WHERE  
    APELLIDO='TORRES';
```

```
Sql> DELETE VEMPLEDEP WHERE EMP_NO=7999;
```

# ACTIVIDADES IX

- **EJ1:** Se quiere insertar un nuevo empleado como DIRECTOR, pero a través de un disparador se comprobará antes de que ocurra, si ya hay un DIRECTOR en ese departamento.
- **EJ2:** 4.- Suponiendo que se dispone de la siguiente vista:

```
CREATE VIEW VDEPART AS SELECT DEPART.DEPT_NO, DNOMBRE,  
    NUMCE, COUNT(EMP_NO) TOTAL_EMPL FROM EMPLE E,DEPART E  
    WHERE E.DEPT_NO=D.DEPT_NO GROUP BY  
    D.DEPT_NO,DNOMBRE,NUMCE;
```

Construye un disparador que permita realizar actualizaciones en la tabla DEPART, a partir de la vista VDEPART. Se contemplarán las siguientes operaciones:

- Insertar departamento.
- Borrar departamento.
- Modificar el nombre de un departamento.

# ACTIVIDADES IX

create or replace trigger ModVistaEmple **INSTEAD OF**

DELETE OR update OR INSERT ON VDEPART FOR EACH ROW

**DECLARE**

CAD VARCHAR2(100);

**BEGIN**

cad:=to\_char(current\_date,'dd/mm/yy hh24:mi');

IF DELETING THEN

DELETE FROM DEPART WHERE DEPT\_NO= :OLD.DEPT\_NO;

CAD:=cad || 'BORRANDO DEPARTAMENTO ' || :OLD.DEPT\_NO;

ELSIF UPDATING('DNOMBRE') THEN

UPDATE DEPART SET DNOMBRE= :NEW.DNOMBRE WHERE DEPT\_NO= :OLD.DEPT\_NO;

CAD:=cad || 'ACTUALIZANDO DEPARTAMENTO ' || :OLD.DEPT\_NO;

ELSIF INSERTING THEN

INSERT INTO DEPART (DEPT\_NO,DNOMBRE,NUMCE) VALUES  
(:NEW.DEPT\_NO,:NEW.DNOMBRE,:NEW.NUMCE);

CAD:=cad || 'INSERTANDO DEPARTAMENTO ' || :NEW.DEPT\_NO;

ELSE

RAISE\_APPLICATION\_ERROR (20001,'ERROR, EN LA ACTUALIZACION');

END IF;

INSERT INTO AUDITAREMPLE3 VALUES(CAD);

**END;**



# ACTIVIDADES IX

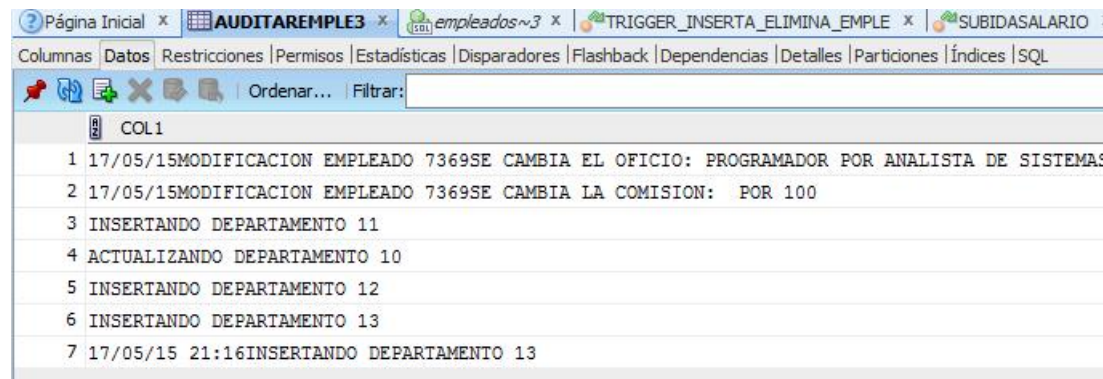
- PRUEBAS:

SQL>SELECT \* FROM VDEPART;

SQL>DELETE FROM VDEPART WHERE DEPT\_NO=10;

SQL>INSERT INTO VDEPART (DEPT\_NO,  
DNOMBRE,NUMCE) VALUES (13,'electricidad',7);

SQL>UPDATE VDEPART SET DNOMBRE='INFORMATICA'  
WHERE DEPT\_NO=10;



The screenshot shows a database audit trail window with the following tabs: 'Página Inicial', 'AUDITAREMPLE3', 'empleados~3', 'TRIGGER\_INSERTA\_ELIMINA\_EMPLE', and 'SUBIDASALARIO'. The 'AUDITAREMPLE3' tab is active, displaying a table with columns 'Columnas', 'Datos', 'Restricciones', 'Permisos', 'Estadísticas', 'Disparadores', 'Flashback', 'Dependencias', 'Detalles', 'Particiones', 'Índices', and 'SQL'. The table contains 7 rows of audit data:

	COL1
1	17/05/15MODIFICACION EMPLEADO 7369SE CAMBIA EL OFICIO: PROGRAMADOR POR ANALISTA DE SISTEMAS
2	17/05/15MODIFICACION EMPLEADO 7369SE CAMBIA LA COMISION: POR 100
3	INSERTANDO DEPARTAMENTO 11
4	ACTUALIZANDO DEPARTAMENTO 10
5	INSERTANDO DEPARTAMENTO 12
6	INSERTANDO DEPARTAMENTO 13
7	17/05/15 21:16INSERTANDO DEPARTAMENTO 13

# TRANSACCIONES

- Una transacción es un conjunto de sentencias agrupadas que pueden ser confirmadas con el comando COMMIT o pueden ser desechadas con el comando ROLLBACK
- Se pueden poner en el programa SAVEPOINTS de forma que al hacer un rollback decimos exactamente desde donde con ROLLBACK TO.
- Al final de la transacción si no ha habido ningún error se hace un COMMIT.
- Al igual que en mysql, la variable AUTOCOMMIT puede desactivarse o activarse :     set autocommit off|on;  
Para ver su estado: SHOW AUTOCOMMIT;

# Ejemplo Transacción

```
DROP TABLE PRUEBA;  
CREATE TABLE PRUEBA(TEXTO VARCHAR2(30));  
create or replace PROCEDURE PRUEBATRANS (NFILAS NUMBER) IS  
BEGIN  
SAVEPOINT NINGUNA;  
INSERT INTO PRUEBA VALUES ('UNA FILA');  
SAVEPOINT UNA;  
INSERT INTO PRUEBA VALUES ('DOS FILAS');  
SAVEPOINT DOS;  
IF NFILAS=1 THEN ROLLBACK TO UNA;  
ELSIF NFILAS=2 THEN ROLLBACK TO DOS;  
ELSE DBMS_OUTPUT.PUT_LINE('NO SE GRABARA NINGUNA FILA');  
    ROLLBACK TO NINGUNA;  
END IF;  
COMMIT;  
EXCEPTION  
    WHEN OTHERS THEN ROLLBACK;  
END;
```

-

# Creación de atributos autoincrementados en Oracle

- Para simular esta funcionalidad en Oracle debemos crear una secuencia, la cual poseerá las siguientes características básicas:
  - nombre
  - valor de inicio
  - valor de fin (valor máximo)
  - incremento.

```
SQL>CREATE SEQUENCE seq_usuarios_idusuario --nombre de la secuencia  
  
START WITH 1 --la secuencia empieza por 1  
  
INCREMENT BY 1 --se incrementa de uno en uno  
  
NOMAXVALUE; --no tiene valor maximo
```

# Creación de atributos autoincrementados en Oracle

- Con la secuencia creada, podemos obtener su valor actual llamando a la función **nextval** indicando nombreDeSecuencia.nextval.

```
SQL>SELECT seq_usuarios_idusuario.nextval FROM dual;
```

- Con este select obtenemos un nuevo valor de la secuencia. Una vez ésta da un valor, no lo volverá a proporcionar mas. Si queremos consultar el valor por el que se encuentra actualmente la secuencia, lo podemos hacer mediante currval.

```
SQL>SELECT seq_usuarios_idusuario.currval FROM dual;
```

# Creación de atributos autoincrementados en Oracle

- Ya tenemos la secuencia que genera números de forma continua, ahora necesitamos algún método para añadir estos valores a la clave primaria de la tabla creada anteriormente. Se puede usar nextval en un insert de forma:

```
SQL>INSERT INTO usuarios(dni, nombre) VALUES  
      (seq_usuarios_idusuario.nextval, '12345678', 'Pepe', ...);
```

Pero es mejor buscar una forma automática de hacer esta inserción. Para ello, usaremos un disparador o trigger.

# Utilización de un disparador para generar el auto\_incremento

```
CREATE TRIGGER trig_usuarios_seq BEFORE INSERT ON usuarios FOR  
EACH ROW
```

```
BEGIN
```

```
SELECT seq_usuarios_idusuario.nextval INTO :new.idusuario FROM  
dual;
```

```
END ;
```

El cual, antes de realizar cualquier inserción en la tabla usuarios, asignará el nextval de nuestra secuencia al nuevo valor del campo id, con lo cual, posteriormente, se realizará el insert con el id generado automáticamente. De esta forma, en el insert anterior, podemos obviar la inserción de la clave primaria sin obtener ningún error.

- INSERT INTO usuarios(dni, nombre,apellidos) VALUES('12345678b', 'Luis','Perez');

# Bibliografía

- <http://elbauldelprogramador.com/plsql-declaracion-de-variables/>
- <http://www.techonthenet.com/oracle>
- <http://devjoker.com/contenidos/catss/52/Procedimientos-almacenados-en-PLSQL.aspx>
- <https://blogdeaitor.wordpress.com/2013/04/02/creacion-de-atributos-autoincrement-en-oracle/>