

Apuntes Complementarios al Programa Docente

Tema IX: **Control de la Integridad Semántica**

Control Semántico en Bases de Datos Objeto-Relacionales. Triggers

<http://sinbad.dit.upm.es>

OBJETOS RELEVANTES PARA LA SEMÁNTICA DE LA BASE DE DATOS

Los principales objetos de una Base de Datos son los siguientes:

- las tablas,
- las constrains o restricciones,
- las vistas,
- las secuencias,
- los índices,
- los cluster o agrupamientos,
- Los triggers o disparadores,
- los snapshots o instantáneas,
- los DB Links o enlaces a otras bases de datos,
- los procedimientos, las funciones, los paquetes,
- los sinónimos, los usuarios, los perfiles, los privilegios y los roles.

- Como sabemos, **CREATE TABLE** es la sentencia DDL que origina una nueva relación o tabla base con su nombre, atributos (nombres y dominios) y restricciones. Ahora nos fijamos en las **restricciones**:

❖ **CREATE TABLE** <NombreTabla>

- (<NombreA1> <TipoA1> [DEFAULT lit] <RestricA1> ,
- ...
- <RestriccionesTabla>);

```

CREATE TABLE <NombreTabla>
  (<NombreA1> <TipoA1> [DEFAULT lit] <RestricA1>, .....
  <RestriccionesTabla>);

```

Las restricciones (de atributo y/o de tabla) nominadas se definen en DDL con la sintaxis:

CONSTRAINT <nombreR>

Restricciones de Atributos (puede haber varias por cada atributo):

NOT NULL , **PRIMARY KEY** , **UNIQUE** , **DEFAULT** <Valor> , **CHECK** (<Condición>) , **REFERENCES**...

Restricciones de Tabla:

- ✓ **PRIMARY KEY** (<Atributos de la PK>), **CHECK** (<Condición>), **UNIQUE** (<Clave Candidata>),
- ✓ **FOREIGN KEY** (<Clave Externa>) **REFERENCES** <Tabla>(<Atributos>)

[ON DELETE {**CASCADE** | **SET NULL** | **SET DEFAULT**}]

[ON UPDATE {**CASCADE** | **SET NULL** | **SET DEFAULT**}]

Si se borra la clave referenciada (PK), se borran las tuplas que la referencian (FK).
 Si se actualiza la PK, se actualizan las tuplas que la referencian (FK).

Si se borra o actualiza la clave referenciada (PK), se ponen a **NULL** los valores que la referencian (FK).

Si se borra o actualiza la clave referenciada (PK), entonces los valores que la referencian (FK) se ponen a su valor por defecto.

- ✓ Oracle no distingue entre restricciones de Tabla y de Atributo (Columna). Las almacena todas en la vista **USER_CONSTRAINTS** del Diccionario de Datos, como sigue:
 - **CONSTRAINT_NAME**: Nombre de la restricción. Si no lo tiene, Oracle asigna uno con un código.
 - **TABLE_NAME**: Nombre de la tabla con dicha restricción.
 - **CONSTRAINT_TYPE**: Es un carácter (**P** para PRIMARY KEY, **U** para UNIQUE, **R** para la regla de integridad referencial, **C** para una restricción de tipo **CHECK** (o **NOT NULL**) con la condición almacenada en el atributo **SEARCH_CONDITION...**)
 - **STATUS**: Estado de la restricción (**ENABLE** o **DISABLE**).
- ✓ Si la restricción involucra a varios atributos, entonces se considera como restricción de tabla.
- ✓ **NOT NULL** y **DEFAULT** son forzosamente restricciones de atributo.
- ✓ Oracle 8 no implementa las opciones de **ON UPDATE** tampoco la opción **ON DELETE SET DEFAULT**.
 - Por defecto, Oracle no permite borrar una tupla si existe una o varias tuplas que referencian a algún valor de la tupla que se intenta borrar.
 - ❖ Por defecto, las restricciones de tipo **CHECK** sólo se exigen si los atributos afectados tienen valores distintos de **NULL**.

DROP sentencia que borra el Esquema de la BD y el esquema de la Tabla:

● **DROP SCHEMA** <NombreEsquema [CASCADE | RESTRICT]

CASCADE borra el esquema totalmente y

RESTRICT sólo borra si el esquema está vacío de tuplas

● **DROP TABLE** <NombreTabla> [CASCADE CONSTRAINTS | RESTRICT]

CASCADE: Borra la tabla, su contenido y las referencias que haya sobre ella (FK en otras tablas).

RESTRICT: Borra la tabla si no hay referencias sobre ella.

- ✓ En Oracle, **RESTRICT** es la opción por defecto y no existe sintácticamente. Para borrar las restricciones que hacen referencia a la tabla se usa **CASCADE CONSTRAINTS**.

ALTER TABLE sentencia DDL que modifica el esquema de la Tabla:

 **ALTER TABLE** <NombreTabla> <ACCIÓN>

ACCIÓN Añadir una columna:

ADD (<NombreA, Tipo, Restric_de_Columna>);

Oracle, usa **MODIFY** para modificar un atributo en lugar de **ADD**.

ACCIÓN Borrar una columna:

DROP <NombreA> [**CASCADE** | **RESTRICT**];

ACCIÓN Añadir restric. a un Atributo:

ALTER <NombreA> **SET** <RestricA>;

ACCIÓN Borrar restric. a un Atributo:

ALTER <NombreA> **DROP** <TipoRA: DEFAULT...>;

ACCIÓN Borrar restric. a una tabla nominada:

DROP CONSTRAINT <NombreC> **CASCADE**;

ACCIÓN Añadir restric. a una tabla:

ADD (<Restric_de_Tabla>);

ALTER TABLE <NombreTabla> <ACCIÓN> en Oracle 8i

ACCIÓN Añadir una columna: **ADD** (<NombreA >, <Tipo >, [<Restric_de_Columna>]);

ACCIÓN Añadir restric. a una tabla: **ADD** (<Restric_de_Tabla>);

ACCIÓN Modificar una columna: **MODIFY** <NombreA> [<Tipo>] [DEFAULT <expr>] [[NOT] NULL];

ACCIÓN Modificar restricción de una columna:

MODIFY CONSTRAINT <NombreR> <Estado>;

<Estado> (tras el nombre de la restricción) puede ser:

[NOT] DEFERRABLE: Indica si la verificación de la restricción se aplaza al final de la transacción o no se aplaza (por defecto) y se comprueba tras la sentencia DML.

ENABLE [VALIDATE | NOVALIDATE]: Activa la restricción para los nuevos datos (opción por defecto). **VALIDATE** es la opción por defecto y comprueba si la restricción es válida en los datos antiguos (que tenía previamente la tabla).

DISABLE: Desactiva la restricción. Si la restricción no tiene nombre, Oracle asigna uno que se puede consultar en la vista **USER_CONSTRAINTS** del diccionario de datos.

● **ALTER TABLE** <NombreTabla> <ACCIÓN> en Oracle 8i

ACCIÓN Borrar Restricción por Tipo: **DROP {PRIMARY KEY | UNIQUE(<Cols>)} [CASCADE];**

No se pueden borrar las columnas con esas dos restricciones (PK y UNIQUE) si existe una clave externa que referencie sus atributos. Con **CASCADE** se borran todas esas claves externas.

ACCIÓN Borrar Restricción por Nombre: **DROP CONSTRAINT <NombreR>;**

Obligatorio para las restricciones de tipo **CHECK** y **REFERENCES**.

ACCIÓN Borrar Columna: **DROP COLUMN <NombreA> [CASCADE CONSTRAINTS];**

ACCIÓN Renombrar Tabla: **RENAME TO <Nuevo_Nombre_Tabla>;**

```
CREATE TABLE CLIENTE (  
    NIF CHAR(12) NOT NULL,  
    Nombre CHAR(80) NOT NULL,  
    Edad NUMBER(2) CONSTRAINT Edad_Pos CHECK (Edad>0),  
    CONSTRAINT PK_Cliente PRIMARY KEY(NIF));  
  
CREATE TABLE MASCOTA (  
    NIF_Owner CONSTRAINT Sin_Propietario  
        REFERENCES CLIENTE(NIF)  
        ON DELETE CASCADE,  
    Nombre CHAR(30) NOT NULL,  
    Fecha_Nac DATE,  
    Especie CHAR(30) DEFAULT 'Perro' NOT NULL,  
    CONSTRAINT PK_Mascota PRIMARY KEY(NIF_Owner,Nombre));  
  
ALTER TABLE CLIENTE ADD Telefono CHAR(20);  
ALTER TABLE CLIENTE DROP COLUMN Edad CASCADE CONSTRAINTS;  
ALTER TABLE MASCOTA DROP CONSTRAINT Sin_Propietario;  
ALTER TABLE MASCOTA MODIFY Especie NULL;  
ALTER TABLE MASCOTA DROP PRIMARY KEY;  
ALTER TABLE MASCOTA ADD PRIMARY KEY (NIF_Owner,Nombre);
```

Observación: La FK no necesita explicitar su tipo, ya que se copia el tipo del atributo de su PK.

VISTA: Tabla virtual cuyas tuplas se derivan de otras tablas (tablas base u otras vistas previas). No tiene nivel extensional y sus tuplas se calculan en el momento de procesar la consulta, a partir de las tablas base de las que la Vista se deriva. La Vista se usa, como si fuera tabla, y sirve para realizar consultas frecuentes, y también para cuestiones de seguridad.

● **CREATE [OR REPLACE] [[NO] FORCE] VIEW** <NombreV> [(<Lista_Atrbs>)]
AS
(<Subquery>) **[WITH READ ONLY];**

El nombre de la vista, <NombreV>, se crea asociado a la subquery especificada.

La <Lista_Atrbs> son los nombres de los atributos de la vista. Por defecto, toma los nombres de los atributos de la subconsulta. Son necesarios si los atributos son calculados (funciones de grupo...).

OR REPLACE : Permite modificar una vista ya existente sin borrarla.

WITH READ ONLY: Indica que sólo se puede leer en la vista (no permitido: borrar, insertar o actualizar).

FORCE: Fuerza a crear la vista aunque no existan los objetos que se usan en ella (tablas, otras vistas...) o no se tengan los privilegios suficientes. Esas condiciones serán necesarias para usar la vista. La opción contraria es **NO FORCE** y es la opción por defecto.

Ejemplos:

```
CREATE OR REPLACE VIEW SumiNombres
AS (SELECT NombreS, NombreP
    FROM Suministro SP, Suministrador S, Pieza P
    WHERE SP.S#=S.S# AND SP.P#=P.P#);
```

```
CREATE OR REPLACE VIEW Cantidad(NombreS, NumPieza)
AS (SELECT NombreS, COUNT(*)
    FROM Suministros SP, Suministrador S
    WHERE SP.P#=S.P#
    GROUP BY NombreS);
```

Posibles formas de Vista:

- La vista se consulta como la tabla y su estado (*up to date*) depende del que tengan las tablas de las que se deriva.
- La vista es una “instantánea”, “foto” o vista materializada (*materialized view*) de la BD (con **CREATE SNAPSHOT**, o bien con **CREATE MATERIALIZED VIEW**).

Para borrar: **DROP VIEW...** ...ya sabemos que la Vista tiene fuertes limitaciones en las operaciones INSERT, UPDATE y DELETE.

La transacción termina con una de estas dos sentencias: **COMMIT** o **ROLLBACK**.

- **COMMIT**: Si la transacción termina bien, guarda los cambios efectuados a la BD. COMMIT hace visibles los cambios a otras sesiones y libera los bloqueos de la transacción. Antes de ejecutar COMMIT los cambios no son permanentes (son temporales) y sólo pueden ser vistos por la sesión que los hizo.
 - **ROLLBACK**: Si la transacción no termina bien, se deshacen los cambios efectuados a la BD. ROLLBACK también libera los bloqueos establecidos. Es la opción por defecto si una sesión se desconecta de la BD sin haber finalizado la transacción.
 - **SAVEPOINT** <Nombre>: Son puntos de salvaguarda con distinto nombre cada uno que permiten deshacer sólo parte de la transacción.
 - Tras esto, la orden **ROLLBACK TO SAVEPOINT** <Nombre>, deshace todo lo hecho desde el punto <Nombre> y libera los recursos bloqueados establecidos tras ese punto de salvaguarda.
 - La transacción no termina con este tipo de ROLLBACK.
- La transacción empieza con la 1ª sentencia SQL después de conectarse a la BD o bien con la sentencia COMMIT después de terminar la transacción anterior.

TRIGGER es un bloque PL/SQL que se ejecuta implícitamente con algunas sentencias DML: **INSERT**, **DELETE** o **UPDATE**. Por contra, los Procedimientos y Funciones sólo se ejecutan tras una llamada explícita a ellos. El Trigger **NO** admite Argumentos.

Sintaxis:

```
CREATE [OR REPLACE] TRIGGER <NombreT> {BEFORE | AFTER} <Suceso_Disparo>
ON
  <Tabla> [ FOR EACH ROW [ WHEN <Condición_Disparo> ] ]
  <Cuerpo_del_TRIGGER>;
```

Los trigger son de mucha utilidad y, entre otros, sirven para:

- Mantener Restricciones de Integridad complejas. Por ejemplo: Restricciones de Estado (como que el sueldo sólo puede aumentar).
- La Auditoría de una Tabla, registra los cambios efectuados y la identidad del que los llevó a cabo.
- Lanzar cualquier acción cuando se modifica el estado de una tabla.

Elementos del TRIGGER

- Nombre (**NombreT**), que debería identificar su función y la tabla sobre la que se define.
- <Suceso_Disparo> es la sentencia DML que, efectuada sobre <Tabla>, disparará el trigger. Puede haber varios sucesos separados por la palabra **OR**.
- Existen 12 Tipos Básicos de Triggers según sea la:
 - ✓ Sentencia DML que dispara el trigger: **INSERT**, **DELETE** o **UPDATE**.
Si **UPDATE** lleva una lista de atributos, el Trigger sólo se ejecuta si se actualiza algún atributo de la lista: **UPDATE OF** <Lista_Atributos>
 - ✓ Temporización: Puede ser **BEFORE** (anterior) o **AFTER**(posterior) y define si el Trigger se activa antes o después de que se ejecute la operación DML causante del disparo.
 - ✓ Nivel: Puede ser a Nivel de Sentencia o a Nivel de Fila (**FOR EACH ROW**).
 - Nivel de Sentencia (*statement trigger*): Se activan sólo una vez, antes o después de ejecutar la sentencia DML.
 - Nivel de Fila (*row trigger*): Se activa una vez por cada Fila afectada por la sentencia DML (una misma operación DML puede afectar a varias filas).

Ejemplo de definición de trigger:

“Guardar en una tabla de control el usuario que modificó la tabla Empleado y la fecha.”

(NOTA: Este trigger es **AFTER** y a nivel de sentencia)

Codificación del trigger del ejemplo:

```
CREATE OR REPLACE TRIGGER Control_Empleados
    AFTER INSERT OR DELETE OR UPDATE ON Empleado
    BEGIN
        INSERT INTO Ctrl_Empleados (Tabla, Usuario, Fecha)
        VALUES ('Empleado', USER, SYSDATE);
    END Control_Empleados;
```

Este es el
<Cuerpo_del_TRIGGER>

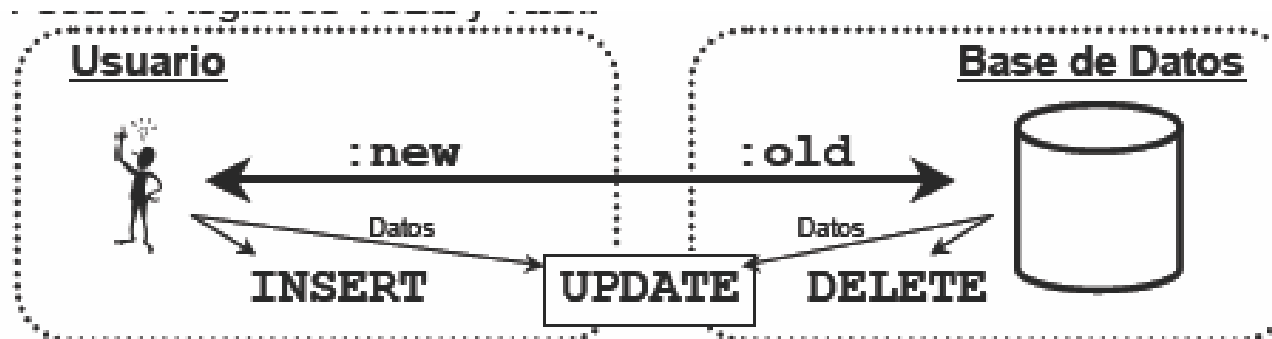
Si la tabla tiene varios tipos de triggers para una misma sentencia DML, entonces el Algoritmo que define el orden de Ejecución es como sigue:

1. Se ejecuta el trigger tipo **BEFORE** a nivel de sentencia, en el caso que exista.
2. Para cada fila afectada por la sentencia: (como la tupla variable ligada a una función lógica, o un cursor posicionado en cada fila afectada)
 - ✓ a) Se ejecuta el trigger **BEFORE** (si existe) a nivel de fila, sólo si dicha fila cumple la condición de la cláusula **WHEN** (si existiera WHEN).
 - ✓ b) Se ejecuta la propia sentencia.
 - ✓ c) Se ejecuta el trigger **AFTER** (si existe) a nivel de fila, sólo si dicha fila cumple la condición de la cláusula **WHEN** (si existiera).
3. Se ejecuta el trigger tipo **AFTER** (si existe) a nivel de sentencia.

- ✓ Los Triggers a nivel de Fila se ejecutan una vez por cada Fila que procesa la sentencia DML disparadora. Para cada Fila Procesada en ese momento, se usan dos Pseudo-Registros de tipo <TablaDisparo>%ROWTYPE. :old y :new

Sentencia DML	:old	:new
INSERT	No Definido: NULL \forall campo.	Valores Nuevos a Insertar.
UPDATE	Valores Originales (antes de la orden).	Valores Actualizados (después).
DELETE	Valores Originales (antes de la orden).	No Definido: NULL \forall campo.

Esquema con el significado de los Pseudo-Registros del Trigger de Fila :old y :new



Ejemplo de definición de trigger:

*T1: "Cada vez que se inserte una nueva pieza en **Pieza**, calcular el número de Pieza (#P)"*

```
CREATE OR REPLACE TRIGGER NuevaPieza
  BEFORE INSERT ON Pieza FOR EACH ROW
  BEGIN
    --Cálculo del nuevo número de la nueva pieza:
    SELECT MAX(P#)+1 INTO :new.P# FROM Pieza;
    IF :new.P# IS NULL THEN
      :new.P# := 1;
    END IF;
  END NuevaPieza;
```

Cada inserción usa el valor del Pseudo-Registro **:new**. Como el trigger pone la clave, una inserción válida sería la siguiente :

```
INSERT INTO Pieza (Nombre, Peso)
VALUES ('Alcayata', 0.5);
```

● **Modificación de los Pseudo-Registros:**

- ✓ **:new** No se puede modificar en un trigger **AFTER** a nivel de fila.
- ✓ **:old** nunca se puede modificar, sólo se puede leer.

Sintaxis: ... **FOR EACH ROW WHEN** <Condición_Disparo>

- La cláusula **WHEN** es siempre opcional y válida sólo en triggers a Nivel de Fila. **WHEN** hace que el cuerpo del trigger se ejecute sólo para las filas que cumplen su Condición de Disparo.
- La Condición_Disparo puede usar los Pseudo-Registros **:old** y **:new**, pero sin escribir los dos puntos (:), que sí son obligatorios en el cuerpo del trigger.

Ejemplo.: El precio alto sólo tendrá 1 decimal (con 2 ó más decimales, redondear el precio). Codificar el trigger:

T2: "Todo valor del atributo Precio mayor que 200 (Precio>200), se redondea a un decimal"

```
CREATE OR REPLACE TRIGGER Redondeo_Precio_Alto
  BEFORE INSERT OR UPDATE OF Precio ON Pieza FOR EACH ROW WHEN (:new.Precio > 200)
BEGIN
  :new.Precio := ROUND(:new.Precio,1);
END Redondeo_Precio_Alto;
```

El mismo trigger sin la cláusula WHEN y usando IF sería ... BEGIN

```
IF :new.Precio > 200 THEN
  :new.Precio := ROUND(:new.Precio,1);
END IF;
END Redondeo_Precio_Alto;
```

Los triggers con: **INSERT**, **DELETE** o **UPDATE** pueden usar tres predicados booleanos por cada sentencia DML disparadora:

INSERTING	vale TRUE	si la orden de disparo es INSERT .
DELETING	vale TRUE	si la orden de disparo es DELETE .
UPDATING	vale TRUE	si la orden de disparo es UPDATE .

Ejemplo:

T3: "Guardar en la tabla de control: la nueva tabla Empleado, usuario que la modificó, sentencia DML de modificación y la fecha."

```
CREATE OR REPLACE TRIGGER Control_Empleados
  AFTER INSERT OR DELETE OR UPDATE ON Empleado
  BEGIN
    IF INSERTING THEN
      INSERT INTO Ctrl_Empleado (Tabla_Empleado, Usuario, Oper., Fecha)
        VALUES ('Empleado', USER, 'INSERT', SYSDATE);
    ELSIF DELETING THEN
      INSERT INTO Ctrl_Empleado (Tabla_Empleado, Usuario, Oper., Fecha)
        VALUES ('Empleado', USER, 'DELETE', SYSDATE);
    ELSE
      INSERT INTO Ctrl_Empleado (Tabla_Empleado, Usuario, Oper., Fecha)
        VALUES ('Empleado', USER, 'UPDATE', SYSDATE);
  END Control_Empleados;
```

Con los Pseudo-Registros **:old** y **:new** también se pueden registrar los valores antiguos y los nuevos (si procede):

El trigger **INSTEAD OF** anula el efecto de la sentencia DML disparadora. Se ejecuta en lugar de esa sentencia DML (ni antes ni después de ella).

INSTEAD OF sólo se define sobre Vistas, y es útil si se intenta modificar una vista no actualizable.

Se activa en lugar de la operación DML que provoca el disparo, o sea, la orden disparadora no se ejecutará nunca. Debe tener Nivel de Fila. Se declara usando **INSTEAD OF** en vez de **BEFORE / AFTER**.

Ejemplo: Sea la siguiente vista:

```
CREATE VIEW Total_por_Suministrador AS
  SELECT #S, MAX(Precio) 'Mayor', MIN(Precio) 'Menor'
  FROM Suministro SP, Pieza P
  WHERE SP.#P = P.#P
  GROUP BY #S;
```

T4: *“Si se lanza el borrado de una tupla desde la vista anterior, entonces borrar un suministrador de la tabla Suministrador”*

```
CREATE OR REPLACE TRIGGER Borrar_en_Total_por_#S
  INSTEAD OF DELETE ON Total_por_Suministrador FOR EACH ROW
BEGIN
  DELETE FROM Suministrador WHERE #S = :old. #S;
END Borrar_en_Total_por_#S;
```

En un bloque PL/SQL, el trigger puede contener cualquier sentencia legal con las siguientes

Restricciones:

- **No puede tener sentencias** de Control de Concurrencia (CC): **COMMIT, ROLLBACK o SAVEPOINT**. El trigger se activa como parte de la sentencia DML que provocó el disparo y, por tanto, forma parte de la misma transacción. Cuando esa transacción es confirmada o abortada, se confirma o aborta también el trabajo realizado por el disparador.
- Cualquier **Subprograma** llamado por el trigger **tampoco puede tener sentencias de CC**.
- **El trigger puede tener Restringido el acceso a ciertas tablas**. Dependiendo del tipo de trigger y de las restricciones que afecten a las tablas, dichas **tablas** pueden ser **mutantes**.
- Diccionario de Datos. Los datos del TRIGGER se almacenan en la vista **USER_TRIGGERS**, cuyas columnas son: **OWNER, TRIGGER_NAME, TRIGGER_TYPE, TABLE_NAME, TRIGGER_BODY...**
- Sentencia de Borrado del Trigger:
DROP TRIGGER <NombreT>;
- Sentencia para Habilitar/Deshabilitar un Trigger, sin necesidad de borrarlo:
ALTER TRIGGER <NombreT> {ENABLE | DISABLE};

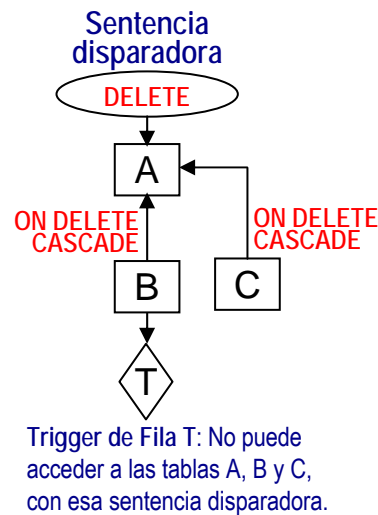
- Tabla de Restricción o Tabla Padre (*constraining table, Parent o PK*): es la tabla a la que referencia una clave externa en una Tabla Hija (*Child o FK*), por la Regla de Integridad Referencial.
- Tabla Mutante (*mutating table*): Una tabla es mutante si:
 1. está modificándose “actualmente” por una sentencia DML (**INSERT**, **DELETE** o **UPDATE**).
 2. está siendo leída por Oracle para forzar que cumpla la regla de integridad referencial.
 3. está siendo actualizada para cumplir una restricción con **ON DELETE CASCADE**.

Ej.: Si la tabla SP (Suministro) tiene la cláusula **ON DELETE CASCADE**, entonces borrar una Pieza implica borrar todos sus ventas en SP.

- ✓ Pieza (y Suministrador) son Tablas Padres (son PK) de SP (es FK). Entonces, si hay que borrar una pieza suministrada, Pieza y SP son ambas tablas mutantes. Si se borra una pieza no suministrada, sólo será mutante la tabla Pieza.
- ✓ La Tabla Mutante casi siempre tiene definido un Trigger de Fila sobre ella misma. Sin embargo, casi nunca tiene definido un Trigger de Sentencia (NO). Es decir, el Trigger de Sentencia se ejecuta antes o después de la sentencia DML, pero NO a la vez.

Las sentencias SQL escritas en el cuerpo del trigger tienen dos Restricciones:

- 1. NO PUEDEN Leer o Modificar ninguna Tabla Mutante de la sentencia que provoca el disparo. Esto incluye lógicamente la tabla de dicha sentencia y la tabla del trigger, que pueden ser distintas.



Ej.: Borramos una tupla de la tabla A que implica borrar tuplas de la tabla B (con una restricción **ON DELETE CASCADE** sobre A). Si este segundo borrado de B disparara algún trigger T, entonces, dicho trigger no podrá ni leer ni modificar las tablas A y B pues son mutantes. Si lo hace se producirá un error.

Sus tablas de restricción afectadas por la cláusula **ON DELETE CASCADE**, también son mutantes.

Ej.: Si borrar en A implica borrar en una tercera tabla C, el disparador sobre B no podrá tampoco acceder a C, si ese trigger se activa por borrar en A (aunque en C no se borre nada). Sí podría acceder a C si se activa por borrar directamente sobre B. También podría acceder a C si ésta no tuviera el **ON DELETE CASCADE**, pero hay que tener en cuenta que **NO** se podrán borrar valores en A, si están siendo referenciados en C (ORA-2292).

- 2. NO PUEDEN Modificar las columnas de PK, única o FK de una Tabla Padre a las que hace referencia la tabla del trigger, aunque Sí pueden modificarse las otras columnas.

Al escribir triggers hay que considerar qué Tablas son Mutantes y cuándo lo son.

Las Tablas Mutantes dependen del tipo de sentencia DML que se está ejecutando:

1. Todas las tablas afectadas por una operación **INSERT**, **DELETE** o **UPDATE** son mutantes
2. Si la tabla Hija (**Empleado**) tiene una FK (**#Dpto**) a otra tabla Padre (**Departamento**), ambas tablas son mutantes si:
 - Insertamos (**INSERT**) en la tabla Hija: Comprobar valores en la tabla padre.
 - Borramos (**DELETE**) de la tabla Padre: Impedir que tuplas hijas se queden sin padre.
 - Actualizamos (**UPDATE**) la tabla Padre o la tabla Hija: Las 2 operaciones anteriores.
3. Si existe la restricción **ON DELETE CASCADE**, ésta implica que si se borra de la tabla Padre, se borrarán las tuplas relacionadas en la tabla Hija y, a su vez, pueden borrarse tuplas de otras tablas hijas de esa tabla Hijo, y así sucesivamente (grafo de 'joins' o enlaces semánticos). En ese caso, todas esas tablas son mutantes. En triggers activados por **DELETE**, interesa saber si se pueden activar por un *borrado en cascada*, en cuyo caso no se podría acceder a todas esas tablas mutantes.

Las dos Restricciones ya dichas (transp. 25) en las sentencias SQL de un trigger se aplican a:

1. Todos los triggers a Nivel de Fila, salvo si la sentencia de disparo es un **INSERT** que afecta a una única fila, entonces esa tabla disparadora no se considera mutante en el trigger de Fila-Anterior.
 - ✓ Si se insertan varias filas con **INSERT INTO** Tabla **SELECT...** la tabla del trigger será mutante en ambos tipos de triggers de Fila.
 - ✓ Éste es el único caso en el que un trigger de Fila puede leer o modificar la tabla del trigger.
2. Los Triggers a Nivel de Sentencia cuando la sentencia de disparo se activa como resultado de una operación **ON DELETE CASCADE** (al borrar tuplas en la tabla Padre).

Los ERRORES por Tablas Mutantes se detectan y generan en Tº de Ejecución y no en la Compilación (ORA-4091).

Ej. T5: “Codificar un trigger que modifique el nº de empleados de un depto. (columna Dept.Num_Emp) cuando se ejecute **INSERT** o **DELETE** de algún empleado, y **UPDATE** en la columna #Dpto de la tabla Emp (Empleado). La tabla Dept es padre (PK) de la tabla Emp (FK), sin embargo el Trigger T5 es correcto, ya que modifica Num_Emp, que no es la PK. Este trigger no puede consultar la tabla Emp porque es mutante:

```
SELECT COUNT(*) FROM Empleado WHERE Dept = :new.Dept;
```

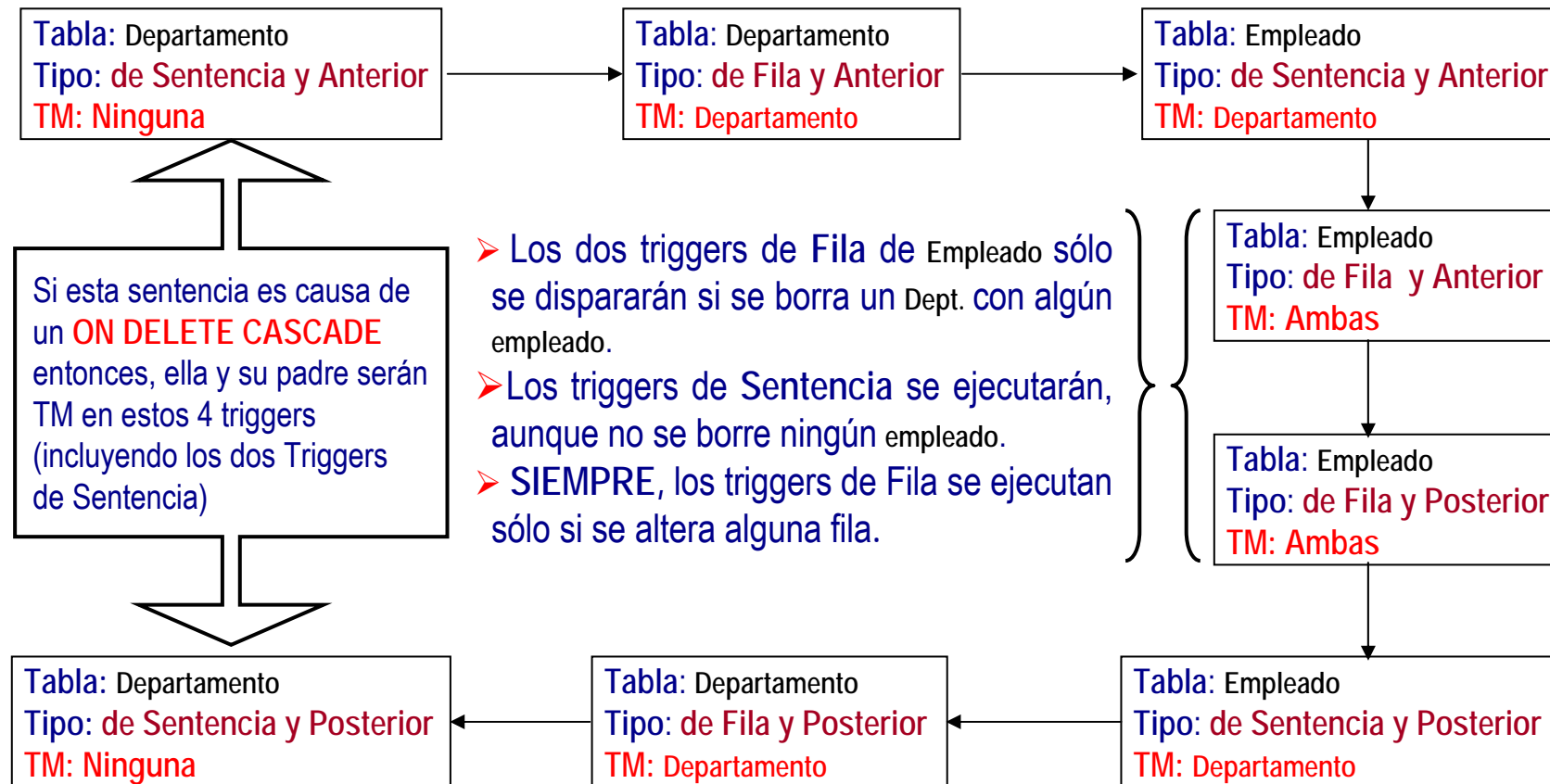
T5:

```
CREATE OR REPLACE TRIGGER Cuenta_Empleados
  BEFORE DELETE OR INSERT OR UPDATE OF #Dpto ON Empleado FOR EACH ROW
BEGIN
  IF INSERTING THEN
    UPDATE #Dpto SET Num_Emp = Num_Emp+1      WHERE NumDpto=:new.Dpto;
  ELSIF UPDATING THEN
    UPDATE #Dpto SET Num_Emp = Num_Emp+1      WHERE NumDpto=:new.Dpto;
    UPDATE #Dpto SET Num_Emp = Num_Emp-1      WHERE NumDpto=:old.Dpto;
  ELSE
    UPDATE #Dpto SET Num_Emp = Num_Emp-1      WHERE NumDpto=:old.Dpto;
  END IF;
END Cuenta_Empleados;
```

Si en la tabla Emp hubiera un **ON DELETE CASCADE** hacia Dept, entonces, al intentar borrar un dept habría que borrar sus empleados: Las dos tablas serían mutantes. Al borrar sus empleados se dispararía este *trigger* que intentaría modificar la tabla Dept que sería mutante y daría **ERROR** que resolveremos más adelante.

Sea la tabla Padre: Dept, y la Hija: Emp cuya FK es #Dpto con la restricción **ON DELETE CASCADE** que fuerza a borrar todos los empleados de un departamento si su departamento se borra. Supongamos que la sentencia **DELETE** tiene implementados los 4 tipos de triggers posibles (de fila y de sentencia, anterior y posterior) para las dos tablas.

Con **DELETE** en la tabla Dept se ejecutarán los siguientes triggers, con las TM indicadas, en este orden:



Sentencias sobre Departamento (Tabla Padre):

DELETE: Los 4 triggers de la tabla Hija Empleado sólo se dispararán si se borra un Departamento que tenga algún Empleado.

Si esta sentencia sobre el Padre es causa de un **ON DELETE CASCADE** sobre una tabla X (padre del padre) entonces, serán TM ella y su padre X, en sus 4 triggers. Ambas tablas también serán TM en los 4 triggers de la tabla Hija Empleado.

INSERT: Su tabla Hijo no se ve afectada: No se disparan sus triggers.

UPDATE: Su tabla Hijo no se afecta porque sólo se permite actualizar valores no referenciados en sus tablas Hijos (ORA-2292).

Sentencias sobre Empleado (Tabla Hijo): No se dispara ningún trigger del Padre.

DELETE: No afecta a la tabla Padre y sólo es TM la Hijo.

INSERT: ♦ Insertar una fila:

Las tabla Hijo no es mutante en el disparador de Fila-Anterior (a pesar de ser la tabla del disparador) y sí lo es en el de Fila-Posterior.

La tabla Padre es mutante sólo si se intenta modificar el atributo al que hace referencia la tabla Hijo y sólo en el trigger de Fila-Posterior, ya que durante la ejecución de triggers de Fila-Anterior aún no está mutando ninguna tabla.

♦ **Insertar varias filas:** Las tablas Padre e Hijo son TM en los dos triggers de Fila, pero la tabla Padre sólo si se modifica el atributo al que hace referencia la tabla Hijo.

UPDATE: Las tablas Padre e Hijo son mutantes en los dos triggers de Fila, pero la tabla Padre sólo si se modifica el atributo al que hace referencia la tabla Hijo.

El trigger `Cuenta_Empleados` (transpa 28) tiene dos inconvenientes:

- ✓ Modifica la columna `Num_Emp` aunque ésta no tenga el valor correcto.
- ✓ Si se modifica directamente esta columna, quedará incorrecta siempre.

Solución al Problema de la Tabla Mutante: Las TMs surgen básicamente en los triggers a nivel de Fila y, como no pueden acceder a las TMs, la Solución es Crear Dos Triggers:

1. A Nivel de Fila: En este trigger guardamos los valores importantes en la operación, pero no accedemos a TMs. Estos valores pueden guardarse en:
 - » Tablas de la BD especialmente creadas para esta operación.
 - » Variables o tablas PL/SQL de un paquete: Como cada sesión obtiene su propia instancia de estas variables, no tendremos que preocuparnos de si hay actualizaciones simultáneas en distintas sesiones.
2. A Nivel de Sentencia Posterior (AFTER): Utiliza los valores guardados en el disparador a Nivel de Fila para acceder a las tablas que ya no son mutantes.

Tablas Mutantes: Solución Si la columna `Emp.Dept` de la tabla hija `Empleado` tiene la restricción **ON DELETE CASCADE** hacia la tabla `Departamento`, y la sentencia disparadora es un **DELETE** sobre `Dept`, entonces la tabla `Dept` no puede modificarse porque también es mutante.

Trigger a Nivel de Fila:

```
CREATE OR REPLACE PACKAGE Empleado_Dept AS
  TYPE T_Dept IS TABLE OF Empleado.Dept%TYPE
    INDEX BY BINARY_INTEGER;
  Tabla_Dept T_Dept;
END Empleado_Dept;

CREATE OR REPLACE TRIGGER Fila_Cuenta_Empleados
  AFTER DELETE OR INSERT OR UPDATE OF Dept ON Empleado FOR EACH ROW
DECLARE Indice BINARY_INTEGER;
BEGIN Indice := Empleado_Dept.Tabla_Dept.COUNT + 1;
  IF INSERTING THEN
    Empleado_Dept.Tabla_Dept(Indice) := :new.Dept;
  ELSIF UPDATING THEN
    Empleado_Dept.Tabla_Dept(Indice) := :new.Dept;
    Empleado_Dept.Tabla_Dept(Indice+1) := :old.Dept;
  ELSE Empleado_Dept.Tabla_Dept(Indice) := :old.Dept;
  END IF;
END Fila_Cuenta_Empleados;
```

Trigger a Nivel de Sentencia:

```
CREATE OR REPLACE TRIGGER Sentencia_Cuenta_Empleados
  AFTER DELETE OR INSERT OR UPDATE OF Dept ON Empleado
DECLARE Indice BINARY_INTEGER;
         Total Departamento.Num_Emp%TYPE;
         Departamento Departamento.NumDept%TYPE;
BEGIN
  FOR Indice IN 1..Empleado_Dept.Tabla_Dept.COUNT LOOP
    Dept := Empleado_Dept.Tabla_Dept(Indice);
    SELECT COUNT(*) INTO Total FROM Empleado WHERE Dept = Dept;
    UPDATE Dept SET Num_Emp = Total WHERE NumDept = Dept;
  END LOOP;
  Empleado_Dept.Tabla_Dept.DELETE;
END Sentencia_Cuenta_Empleados;
```

- Variables Contenidas en un Paquete:

- ✓ Son Variables Globales de la Sesión y visibles a todos los programas de cada sesión.

- En el trigger con AFTER:

- ❖ Es necesario borrar la tabla PL/SQL, para que la siguiente sentencia empiece a introducir valores en esa tabla a partir de la posición 1.
 - » También se puede declarar otra variable en el paquete que mantenga el número de elementos de la tabla, poniéndola a cero al final, evitando usar los atributos de la tabla PL/SQL (**COUNT** y **DELETE**).
 - ❖ Hay que tener en cuenta que se ejecutará después de la sentencia, o sea, que los datos ya estarán actualizados.
 - ❖ Si este disparador produce un error la sentencia será deshecha (*rollback*).

Ese error puede ser:

- ✓ Producido por un error en el trigger o por una operación no válida del disparador.
 - ✓ Generado por el trigger de forma explícita porque se haya detectado que la sentencia disparadora no es admisible (por cualquier razón).
 - Esto se hace con el procedimiento **RAISE_APPLICATION_ERROR**.

Por supuesto, cualquier sentencia SQL de cualquier bloque PL/SQL siempre ha de Respetar todas las Reglas de Integridad.