Capítulo 1

Actividad en la base de datos: Triggers

1.1. Introducción

Los *triggers* o *disparadores* son mecanismos que se utilizan para dotar a las bases de datos de cierta actividad, ejecutando un código cuando una determinada situación (inserción, borrado o modificación de tuplas) ocurre. Es decir, se consigue que la base de datos "reaccione" activamente ante los cambios.

La actividad en una base de datos tiene diversas utilidades:

- Evitar el almacenamiento de información errónea en la base de datos
- Aumentar la integridad referencial o conseguir una seguridad adicional
- Controlar restricciones de usuario: por ejemplo, una regla de negocio de una empresa puede especificar que un empleado no puede ganar más que su jefe. Mediante el uso de triggers, si se intenta incluir en la base de datos una información que incumpla esa regla, es la propia base de datos la que no lo permite.
- Un campo sólo admite ciertos valores, bien sea un rango, una colección de valores, o un formato determinado. Esta restricción es similar a las restricciones tipo `CHECK` de `SQL-2` (de hecho, en muchos casos se podría implementar con este tipo de restricciones). Así, un campo que especifica el sexo de una persona podría admitir solamente los valores `H` (hombre) y `M` (mujer). Un caso más complejo sería usar una regla que verifica que un NIF es correcto, comprobando si la letra corresponde al número.
- Mantenimiento de valores deducidos: Si tenemos una tabla con valores parciales de ventas y otra con totales, al insertar, borrar o modificar un valor parcial, automáticamente se actualizará el valor total.
- Mantener coherentes las tablas replicadas en una BD distribuida.

El uso de triggers nos proporciona ventajas, como son una mayor facilidad en la codificación de las aplicaciones que se ejecutan contra la base de datos, así como una mayor consistencia de los datos al no tener que realizar las mismas comprobaciones cada vez que se intentan modificar los datos. Así, en el ejemplo de la comprobación de la letra del NIF, si no usásemos un trigger

tendríamos que controlar en todas las partes del código de la aplicación (o aplicaciones) que el NIF introducido es correcto. Si en un solo lugar se omitiese la comprobación, esto daría lugar a errores en los datos. Sin embargo, utilizando un trigger el control se diseña una sola vez y cualquier forma de actualización de datos, desde cualquier aplicación, pasaría ese control. Lo mismo ocurriría en el caso de la actualización automática de los totales de ventas: sin triggers, si en algún lugar se omite la actualización de los totales, se produciría una inconsistencia en los datos.

Los triggers se denominan también reglas ECA (Evento-Condición-Acción), dado que estos son los tres elementos básicos de un trigger:

Evento: Es la modificación de datos que se controla con el trigger, y hará que este se *dispare*. Esta modificación puede ser una inserción (INSERT), un borrado (DELETE) o una actualización (UPDATE).

Condición: La condición que debe verificarse para que el trigger se dispare. Esta condición es opcional, ya que pueden darse casos en los que, al producirse un evento, automáticamente se dispare el trigger (podemos considerar una condición cierta en todos los casos).

Debemos tener en cuenta que la condición que hay que indicar en un trigger debe ser la condición *contraria* a la regla que queremos comprobar, para que el trigger se active si se da la condición que viola esta regla.

Acción: La acción (normalmente una modificación de datos o evitar que la operación en curso se realice) que se produce como reacción de la BD ante el cambio (el evento, cuando se da la condición).

Ejemplo 1.1.1 Consideremos el uso de un trigger para verificar que la letra del NIF de un cliente es correcta. La siguiente regla ECA realizaría ese control:

- Evento: Hay dos posibles eventos: la inserción de un cliente, o la actualización del NIF del mismo.
- Condición: El NIF es incorrecto, bien porque no se ha indicado la letra o porque se ha indicado una incorrecta. Como se ve, la condición que hace que se active el trigger es la contraria a la regla que queremos que se cumpla.
- Acción: No permitir la inserción del cliente, o la actualización de su NIF.

Como veremos, Oracle permite controlar más de un evento (o más de una condición) mediante un único trigger. Sin embargo, el estándar SQL:1999 no contempla ese caso, por lo que tendríamos que tener una regla ECA para cada uno de los eventos. □

Ejemplo 1.1.2 Consideremos una tabla de facturas y una tabla que contiene las líneas de factura. La tabla de facturas incluye un atributo `Total` que tendrá en todo momento el total actual de la factura (la suma de los totales de sus líneas).

Dado que para este caso habría los tres posibles eventos (la inserción, el borrado o la modificación de una línea de factura), consideremos la siguiente regla que contempla solamente el caso de la inserción de una nueva línea de factura.

- Evento: La inserción de una nueva línea de factura.

- Condición: Ninguna (no consideramos, por ejemplo, que el total de la línea debe ser un número mayor que cero, ya que este sería un control distinto).
- Acción: Sumar al total de la factura correspondiente el total de la línea insertada.

□

1.2. Triggers en SQL:1999

El último estándar de SQL, antiguamente denominado SQL-3 o SQL99, y actualmente denominado SQL:1999, incluye la definición de triggers para implementar la actividad en bases de datos. La Figura 1.1 muestra la sintaxis básica de la sentencia de creación de un trigger en SQL:1999. Sin embargo, antes de entrar en los detalles de esta sintaxis, debemos tener en cuenta dos consideraciones sobre los triggers en SQL:1999:

- El evento y la condición de un trigger está siempre asociado a una única tabla.
- Un trigger sólo puede controlar un evento (sobre la tabla asociada), ya sea de inserción, borrado o modificación.

Teniendo en cuenta estos aspectos, el Ejemplo 1.1.1 requeriría dos triggers, mientras que la regla ECA considerada en el Ejemplo 1.1.2 (que sólo tiene en cuenta la inserción de una línea de factura) podría hacerse con un solo trigger.

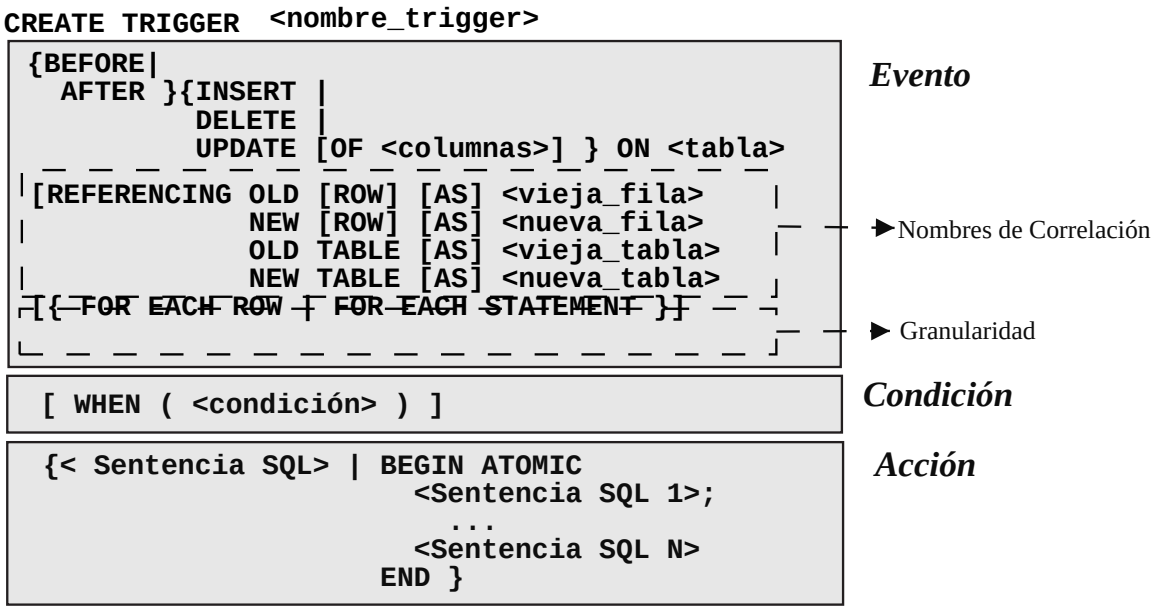

Figura 1.1: Sintaxis básica de creación

Para borrar un trigger, se usa la sentencia

```
DROP TRIGGER <nombre_trigger>;
```

Veamos ahora en detalle las tres partes básicas de la definición de cualquier trigger en SQL:1999.

1.2.1. El evento

Como ya se ha dicho, un trigger comprueba un único evento sobre una tabla. Este evento puede ser una inserción (`INSERT`), un borrado (`DELETE`) o una modificación (`UPDATE`). Dado que una inserción o borrado en una tabla siempre afecta a filas completas, no cabe hacer más específico el evento para estos casos. Sin embargo, una actualización puede afectar sólo a un subconjunto de los atributos de una tabla. Por ello, si queremos controlar específicamente la actualización de determinados atributos, utilizaremos la sintaxis `UPDATE OF columna1[, columna2, ...] ON tabla`. Si queremos controlar la actualización de cualquiera de los atributos, utilizaremos simplemente `UPDATE ON tabla`.

Además, dentro de esta parte de la definición debemos destacar lo siguiente:

Momento de activación: Cada trigger tiene momento de activación, y este puede ser antes (`BEFORE`) o después (`AFTER`) de que ocurra el evento que controla el trigger. En cualquier caso, los valores de la tabla antes y después de la modificación son conocidos en el trigger.

Es preciso indicar que sólo en los triggers de tipo `AFTER` se permite incluir, en la parte de acción del trigger, sentencias INSERT, DELETE o UPDATE.

Granularidad: La ejecución de una sentencia DML (una inserción, borrado o modificación) puede afectar a una o más tuplas. La granularidad de un trigger especifica si el trigger se activa una vez por cada fila afectada (`FOR EACH ROW`) o una sola vez por cada sentencia DML (`FOR EACH STATEMENT`). Si no se indica nada, la granularidad por defecto es `FOR EACH STATEMENT`.

Nombres de correlación: Como ya se ha dicho, tanto en los trigger tipo `BEFORE` como en los `AFTER` se conocen los valores de las filas antes y después de la ejecución de la sentencia que activa el trigger. Los nombres de correlación nos sirven para dar nombres más significativos a estos valores.

Si la granularidad del trigger es a nivel de tupla, el valor de la tupla que está siendo modificada se conoce como "OLD ROW", y el valor después de la modificación como "NEW ROW". Estos valores no se conocerán si la granularidad es a nivel de sentencia, ya que se considera sólo una ejecución del trigger por sentencia y esto puede afectar a una o más filas.

Cuando la granularidad del trigger es a nivel de sentencia, sin embargo, sí se conoce el estado de la tabla antes de la modificación y después de esta. Estos valores se suelen denotar como "OLD TABLE" y "NEW TABLE", respectivamente. Sin embargo, para este caso no se conoce el contenido específico de una fila en concreto.

Si utilizamos los nombres de correlación, como por ejemplo

```
REFERENCING OLD ROW AS tupla_vieja
            NEW ROW AS tupla_nueva
```

en un trigger con granularidad de fila, podremos referenciar los valores viejo y nuevo de las filas con los nombres indicados. Para usar nombres de correlación para las filas, la palabra `ROW` es opcional. Para las tablas, la palabra `TABLE` es obligatoria.

Se debe notar que si queremos acceder a los valores viejos o nuevos es obligatorio usar los nombres de correlación. Finalmente, debemos indicar ciertas salvedades en los valores conocidos durante la ejecución del trigger:

- El valor `NEW ROW` no existe en un trigger de borrado, ya que la tupla deja de existir.
- El valor `OLD ROW` no existe en un trigger de inserción, puesto que antes de la ejecución no existía la tupla.
- Los valores `OLD ROW` y `NEW ROW` sólo son aplicables en triggers con granularidad de tupla, ya que si fuese de sentencia el trigger se ejecutaría una sola vez por sentencia, y no se sabe la(s) tupla(s) afectada(s) por la misma. Si la granularidad es de sentencia sólo son visibles `OLD TABLE` y `NEW TABLE`.

  Esto quiere decir que si deseamos verificar alguna condición sobre los valores de las filas que se insertan, modifican o borran, la granularidad debe ser `FOR EACH ROW`.

Ejemplo 1.2.1 Consideremos la tabla `EMPL` de empleados, y la sentencia SQL de la Tabla 1.1.

Cuadro 1.1: Valores viejos y nuevos en una actualización

Tabla vieja				*Tabla nueva*		
Código	*Jefe*	*Salario*	`UPDATE EMPL`	*Código*	*Jefe*	*Salario*
0001	1000	1500	`SET SALARIO=3100`	0001	1000	1500
0002	**1000**	**1600**	`WHERE SALARIO>1540`	**0002**	**1000**	**3100**
0010	**1300**	**1550**		**0010**	**1300**	**3100**
1000	**1100**	**3000**	$\Longrightarrow$	**1000**	**1100**	**3100**

La Tabla 1.1 muestra los valores de la tabla vieja (antes de la actualización) y nueva (después de la actualización). Si se hubiese definido un trigger que controlase la actualización, ambos valores serían visibles, ya que la sentencia es un `UPDATE`. Esta sentencia de actualización afecta sólo a las tres filas **resaltadas**, por lo que si la granularidad fuese a nivel de fila (`FOR EACH ROW`), en cada una de las ejecuciones se podrían ver los valores viejos y nuevos para cada fila. En cambio, si la granularidad fuese de sentencia (`FOR EACH STATEMENT`), sólo se podría acceder a las versiones vieja y nueva de la tabla en conjunto. □

1.2.2. La condición

La condición que debe verificarse para que el trigger se active se especifica, después del evento, mediante la cláusula

```
WHEN (<condición>)
```

Esta cláusula admite cualquier expresión condicional válida de SQL, es decir, el mismo tipo de condiciones que se podrían incluir en la cláusula `WHERE` de una sentencia `SELECT`. Además, en los triggers con granularidad `FOR EACH ROW` se puede hacer referencia a los valores nuevos y/o viejos de los atributos de la fila en consideración.

1.2.3. La acción

La acción especificada en la definición del trigger, y que se ejecutará cuando se produzca el evento y se verifique la condición, puede ser una única sentencia (normalmente una sentencia SQL) o un bloque delimitado por

```
BEGIN ATOMIC
  <sentencias terminadas en ;>
END ATOMIC
```

Las sentencias más comúnmente usadas son de DML (`INSERT`, `DELETE` o `UPDATE`), que sólo pueden ser utilizadas en triggers de tipo `AFTER`, o la sentencia `SET` que se usa normalmente en triggers de tipo `BEFORE` para asignar valores a atributos, por ejemplo

```
SET tupla_nueva.Atributo1 = tupla_vieja.Atributo1
```

Hay algunas consideraciones importantes con respecto a los triggers:

- La activación del trigger *no evita que el evento disparador se ejecute*. Así, si tenemos un trigger de tipo `BEFORE UPDATE`, se activa el trigger y luego se ejecuta la sentencia `UPDATE`. En un trigger de tipo `AFTER`, en primer lugar se ejecuta la sentencia y luego el trigger.

- La sentencia de creación de un trigger crea el trigger, pero en ningún caso lo ejecuta. Además, los triggers no pueden ejecutarse directamente, sino que se activan siempre cuando se da el evento y se verifica la condición.

1.2.4. Ejemplos

Considerando la tabla creada con la siguiente sentencia SQL:

```
CREATE TABLE EMPL(
   Codigo NUMBER(4) NOT NULL,
   Jefe NUMBER(4),
   Salario NUMBER(7,2),
   Edad NUMBER(3),
   Primary key (Codigo),
   Foreign key (Jefe) references EMPL(Codigo) );
```

Considera las siguientes restricciones:

Ejercicio 1.1 *La edad es un número positivo menor de 150.*

Solución:

Aún cuando esta regla podría controlarse mediante una cláusula check de la forma

```
ALTER TABLE EMPL
    ADD CONSTRAINT edad_valida CHECK ((Edad>0) and (Edad<150))
```

veamos cómo se podrían implementar triggers que realizasen la misma función.

La verificación de la edad debemos hacerla tanto al añadir un empleado como al cambiar la edad de uno ya existente. Dado que un trigger en SQL:1999 sólo puede controlar un evento, necesitaremos dos triggers para forzar la verificación de esta regla. La creación de ambos triggers se indica a continuación: `edad_valida_ins`, que será de tipo `AFTER` y controlará la regla en las inserciones, y `edad_valida_upd`, de tipo `BEFORE` y controlará la regla en las actualizaciones.

Control de inserciones

```
CREATE TRIGGER edad_valida_ins
    AFTER INSERT ON EMPL
    REFERENCING NEW ROW AS tupla_nueva
    FOR EACH ROW
    WHEN ((tupla_nueva.edad<=0) OR (tupla_nueva.edad>150))
    DELETE FROM EMPL WHERE CODIGO=tupla_nueva.Codigo;
```

Veamos algunos aspectos importantes sobre esta definición de trigger:

- Controla sólo un evento, la inserción, y es de tipo `AFTER` ya que en la acción borraremos la tupla recién insertada, y las sentencias `DELETE` sólo son admisibles en este tipo de trigger.
- La granularidad es `FOR EACH ROW`, para poder acceder a los valores de la nueva tupla. Además, es obligatorio usar el nombre de correlación para acceder a estos valores.
- La condición establecida en la cláusula `WHEN` es la condición inválida (edad menor o igual que cero, o mayor o igual a 150), ya que el trigger se debe activar cuando la edad es inválida. Como se ve, en esta cláusula se usa el nombre de correlación `tupla_nueva` para acceder a los valores que se están insertando.
- La acción, una única sentencia SQL, consiste en borrar la tupla recién insertada. Para ello también usamos el nombre de correlación, borrando la tupla que corresponde al nuevo código (que es la clave de la tabla)

Control de actualizaciones

```
CREATE TRIGGER edad_valida_upd
    BEFORE UPDATE OF EDAD ON EMPL
    REFERENCING NEW AS tupla_nueva
                OLD AS tupla_vieja
    FOR EACH ROW
    WHEN ((tupla_nueva.edad <= 0) OR (tupla_nueva.edad > 150))
    SET tupla_nueva.Edad = tupla_vieja.Edad;
```

De este trigger podemos destacar lo siguiente:

- De nuevo controla sólo un evento, en este caso la actualización. Dado que sólo controlamos la actualización de la edad, lo indicamos explícitamente indicando `UPDATE OF EDAD ON EMPL`. Además, el trigger es de tipo `BEFORE` porque en la acción vamos a modificar el valor que se va a usar para modificar la fila (recuérdese que la ejecución del trigger no evita la ejecución del evento disparador).
- La granularidad es de nuevo `FOR EACH ROW`, para poder acceder a los valores, en este caso viejos y nuevos, de los atributos de la tupla que está siendo actualizada. Se usan también los nombres de correlación, y en este caso se ha decidido omitir la palabra clave opcional `ROW` en la declaración de estos nombres.
- La condición de la cláusula `WHEN` es de nuevo la condición inválida, que hace que el trigger se active cuando la edad es inválida, y se usan también los nombres de correlación.

- En la acción se asigna el valor antiguo (que será válido) al atributo edad. Así, al actualizar la tabla la sentencia `UPDATE`, usará el valor antiguo por lo que realmente no se actualiza la tabla y se conserva un valor válido para la edad.

Ejercicio 1.2 *Un empleado no puede ganar más que su jefe*

Solución:

Como en el caso anterior, tenemos que crear dos triggers, uno para controlar la inserción de un nuevo empleado, y otro para controlar las posibles actualizaciones.

Control de inserciones: En este caso sólo tenemos que controlar la inserción de un subordinado, verificando que no cobre más que el jefe que le asignemos (si le asignamos alguno). No hay que controlar la inserción de un nuevo jefe, ya que (debido a la integridad referencial) no se dará el caso de insertar un empleado que ya sea jefe de otros. La definición del trigger es la siguiente:

```
CREATE TRIGGER  emp_valido_ins
   AFTER INSERT ON EMPL
   REFERENCING NEW ROW AS tupla_nueva
   FOR EACH ROW
   WHEN ( EXISTS (SELECT Codigo FROM EMPL WHERE Codigo = tupla_nueva.Jefe)
          AND
      (tupla_nueva.Salario > (SELECT Salario FROM EMPL
                                    WHERE Codigo = tupla_nueva.Jefe)) )
   DELETE FROM EMPL WHERE CODIGO=tupla_nueva.Codigo;
```

En la parte `WHEN` se comprueba que el empleado que estamos insertando tiene jefe, y que este jefe cobra menos que el nuevo empleado. Si es así, el empleado se borrará. Nótese que se confía en que la cláusula `WHEN` se evalúe con cortocircuito (si el empleado no tiene jefe, la condición es falsa y no verifica el salario). Si esto no está asegurado, la condición no sería válida, ya que la sentencia `SELECT` que obtiene el salario del jefe debe obtener *exactamente* un salario para ser válida.

Una condición alternativa que sería siempre válida por no depender del modo de evaluación sería

```
... WHEN (1 = (SELECT COUNT(*) FROM EMPL
                 WHERE Codigo=tupla_nueva.Jefe
                   AND Salario < tupla_nueva.Salario))
```

Es decir, se verifica que el nuevo empleado tiene exactamente un jefe (no puede tener más) que cobra menos que él.

Control de actualizaciones: El caso de las actualizaciones es más complejo, ya que se pueden dar más situaciones en donde se infrinja la regla especificada:

- Si modificamos un empleado aumentando su salario, no debe cobrar más que su jefe. Además, si el empleado cambia de jefe debemos verificar la misma condición.
- Si modificamos un jefe (decrementando su salario), debemos verificar que sus empleados no cobren más que él. No es necesario comprobar el cambio del código de empleado de un jefe, ya que eso se controla mediante la integridad referencial.

```
CREATE TRIGGER  emp_valido_upd
   BEFORE UPDATE OF Salar, Jefe ON EMPL
   REFERENCING NEW ROW AS tupla_nueva
               OLD ROW AS tupla_vieja
   FOR EACH ROW
   WHEN (  ( EXISTS (SELECT Codigo FROM EMPL       -- Comprueba si hay subordinados
                WHERE Jefe=tupla_nueva.Codigo      -- que cobran más que el jefe
                  AND Salario>tupla_nueva.Salario)
           )
         OR
         (1 = (SELECT COUNT(*) FROM EMPL           -- Comprueba si el jefe cobra
                   WHERE Codigo=tupla_nueva.Jefe   -- menos que el subordinado
                     AND Salario < tupla_nueva.Salario
          )

BEGIN ATOMIC
   SET  tupla_nueva.Salario=tupla_vieja.Salario;
   SET  tupla_nueva.Jefe=tupla_vieja.Jefe;
END;
```

Como se ve, el trigger se activa cuando cualquiera de las dos situaciones inválidas indicadas se verifica. En la acción se utilizan dos sentencias `SET` para volver a la situación anterior, evitando el cambio tanto del jefe como del salario del empleado.

1.3. Triggers en Oracle

Oracle, como la mayoría de los gestores de bases de datos relacionales actuales, soporta la definición de triggers para dotar de actividad a sus bases de datos. Sin embargo, también como la mayoría de los gestores, la sintaxis y potencia de los triggers definidos en Oracle diferirán de la definida en SQL:1999. Esto es debido a que todos estos gestores empezaron a incluir triggers antes de la definición del estándar. Por ejemplo, la acción de los triggers en Oracle será un bloque PL/SQL.

Además, como cualquier elemento de una base de datos de Oracle, la definición de los triggers se almacena en el catálogo de la base de datos. En concreto, todos los triggers del sistema pueden verse en la tabla `ALL_TRIGGERS`, y los definidos por el usuario actual en la tabla `USER_TRIGGERS`, cuya definición se muestra a continuación.

```
SQL> describe ALL_TRIGGERS

Name                           Null?    Type
------------------------------ -------- ----
OWNER                          NOT NULL VARCHAR2(30)
TRIGGER_NAME                   NOT NULL VARCHAR2(30)
TRIGGER_TYPE                            VARCHAR2(16)
TRIGGERING_EVENT                        VARCHAR2(26)
TABLE_OWNER                    NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
```

REFERENCING_NAMES	VARCHAR2(87)
WHEN_CLAUSE	VARCHAR2(2000)
STATUS	VARCHAR2(8)
DESCRIPTION	VARCHAR2(2000)
TRIGGER_BODY	LONG

A continuación veremos la sintaxis básica de la definición de los triggers en Oracle, e implementaremos los triggers de los ejemplos usados en SQL:1999. Además, veremos también algunas de las ventajas y limitaciones que Oracle presenta con respecto a la definición de triggers en SQL:1999.

1.3.1. Sintaxis básica

La sintaxis básica de creación de un trigger en Oracle es la siguiente:

```
CREATE [OR REPLACE] TRIGGER <nombre_trigger>
   {BEFORE|AFTER} <Evento> [OR <Evento>...] ON <Tabla>
   [REFERENCING [NEW AS tupla_nueva]
                [OLD AS tupla]_vieja]
   ]
   [FOR EACH ROW]
   [WHEN <Condicion_SQL_simple>]
<Bloque PL/SQL>
```

Si el trigger se crea correctamente, así nos lo indicará Oracle. En caso contrario da el aviso "Aviso: Disparador creado con errores de compilación".

En cuanto a la sintaxis de Oracle, encontramos una serie de diferencias con respecto al estándar SQL:1999:

- Podemos crear triggers nuevos, usando `CREATE TRIGGER ...`, y en el caso de que ya exista un trigger con ese nombre recibiremos un error. Si usamos `CREATE OR REPLACE TRIGGER ...` se crea el trigger en caso de que no exista, y si existe lo sobrescribe con la nueva definición.

- Se puede controlar más de un evento en un sólo trigger, uniendo todos los eventos con `OR`. Así, un trigger de tipo `BEFORE INSERT OR UPDATE ON ...` controlaría los eventos de inserción y actualización en una tabla.

  Para saber qué evento ha disparado un trigger que controla más de uno, existen los predicados `INSERTING`, `UPDATING`, y `DELETING`, que devuelven cierto o falso si el evento es, respectivamente, una inserción, una modificación o un borrado.

- Los momentos de activación son `BEFORE` y `AFTER`, con el mismo significado que los triggers de SQL:1999.

  Además, Oracle incorpora el momento de activación `INSTEAD OF`, que hace que se ejecute la acción del trigger *en vez de* ejecutar la sentencia DML que lo activa. Este tipo de triggers no se puede utilizar con tablas, sólo con vistas, con lo que se ofrece un mecanismo manual para actualizar vistas que de otro modo no serían actualizables.

- La cláusula `REFERENCING` permite referenciar los valores nuevos y viejos de tuplas, pero no de tablas. Esta cláusula es opcional en Oracle, y si no se incluye, los nombres de correlación por defecto son `OLD` para la tupla vieja y `NEW` para la tupla nueva (sin usar la palabra `ROW`). Cuando se usen los nombres de correlación en la parte de la acción, deberán ir precedidos de dos puntos (`:`), de la forma `:NEW.NombreAtributo`. En la parte de la condición (`WHEN`) no llevan los dos puntos.
- La condición establecida en la cláusula `WHEN` es una condición SQL simple, en la que *no se admiten subconsultas*. En caso de que necesitásemos hacer una comprobación que requiriese una consulta compleja, esta se trasladará al bloque PL/SQL en el que se establece la acción.
- La acción es un bloque PL/SQL:

```
[DECLARE
  -- Declaración de variables ]
BEGIN
  -- Código PL/SQL
END;
```

Como veremos, en los triggers `FOR EACH ROW` no será posible acceder (ni para consultar ni para actualizar) a la tabla que esté "mutando" (*mutating table*), que es la tabla sobre la que se define el trigger, ni a todas aquellas que están siendo consultadas o modificadas por ser referenciadas por esta tabla mediante claves foráneas y con la opción `ON DELETE CASCADE` (se denominan *constraining tables*). Sólo es posible, en algunos casos, acceder a estas tablas en los triggers de tipo `BEFORE INSERT`. Esta es una importantísima limitación para la definición de triggers en Oracle.

Por ejemplo, acciones tales como (en el control de inserciones del Ejercicio 1.1) borrar la tabla que se acaba de insertar, no es posible definirlas en Oracle. Alternativamente podemos generar una excepción (mediante `RAISE_APPLICATION_ERROR`) para deshacer la acción. Sin embargo, debemos tener en cuenta que esto desharía todas las sentencias desde el inicio de la transacción. Teóricamente, podríamos hacer también un rollback (usando la sentencia `ROLLBACK`), pero no nos está permitido dentro de un trigger. Este es un caso en el que podemos superar artificialmente esa deficiencia, pero veremos algún ejemplo en el que, a no ser que apliquemos un método realmente retorcido que implica el uso de tablas temporales y/o triggers con granularidad de sentencia, no se podrán resolver.

1.3.2. Ejemplos

Veamos cómo se pueden implementar en Oracle los ejercicios ya resueltos con triggers en SQL:1999.

Ejercicio 1.3 *La edad de un empleado debe ser un número positivo menor de 150.*

Solución:

Podemos implementar un solo trigger para controlar esta regla tanto en las inserciones como los borrados.

```
CREATE OR REPLACE TRIGGER ora_edad_valida
    AFTER INSERT OR UPDATE OF EDAD ON EMPL
```

```
    FOR EACH ROW
    WHEN ((NEW.edad <= 0) OR (NEW.edad > 150))
BEGIN
    RAISE_APPLICATION_ERROR(-20001, 'La edad introducida no es válida');
END;
```

Veamos algunos aspectos importantes sobre este trigger:

- Como se ve, se indican los dos eventos en el mismo trigger.
- No se usa la cláusula `REFERENCING` para indicar los nombres de correlación para los valores nuevo y viejo de la tupla que se está insertando o modificando, usando el valor `NEW` para la nueva tupla (no necesitamos acceder al valor viejo en este caso).
- Dado que la condición es simple y no necesita subconsultas, podemos usar la cláusula `WHEN`.
- Elevamos una excepción en la parte de la acción, con lo que se hace un rollback y además el usuario obtiene un mensaje de error. Si usásemos simplemente `ROLLBACK` se desharían los cambios silenciosamente.

  Como ejercicio adicional, se puede comprobar qué pasa si se intenta borrar la tupla que se acaba de insertar.

Esto podría resolver ambos eventos en un solo trigger. Sin embargo, para el caso de la actualización, la transacción puede continuar normalmente si asignamos el valor anterior de la edad (que sería válido) en un trigger de tipo `BEFORE`. Por ello podríamos eliminar el evento de actualización del trigger anterior y añadir el siguiente:

```
CREATE OR REPLACE TRIGGER ora_edad_valida_upd
    BEFORE UPDATE OF EDAD ON EMPL
    REFERENCING NEW AS tupla_nueva
                OLD AS tupla_vieja
    FOR EACH ROW
    WHEN ((tupla_nueva.Edad <= 0) OR (tupla_nueva.Edad > 150))
BEGIN
    :tupla_nueva.Edad := :tupla_vieja.Edad;
END;
```

Como se ve, en este caso usamos los nombres de correlación, que en la parte de la acción (es decir, del bloque PL/SQL) van precedidos de dos puntos. Dado que la sentencia de asignación es válida, la actualización continúa, pero con el valor viejo que le hemos asignado, por lo que realmente no se actualiza la edad del empleado.

Una alternativa a usar estos dos triggers es la utilización, en un único trigger, de los predicados `INSERTING` y `UPDATING`, de la siguiente forma:

```
CREATE OR REPLACE TRIGGER ora_edad_valida_un_solo_trigger
    AFTER INSERT OR UPDATE OF EDAD ON EMPL
    FOR EACH ROW
    WHEN ((NEW.edad <=0 ) OR (NEW.edad > 150))
BEGIN
```

```
  IF UPDATING THEN
     :NEW.Edad := :OLD.Edad; ELSIF
  INSERTING THEN
    RAISE_APPLICATION_ERROR(-20001, 'La edad introducida no es válida');
  END IF;
END;
```

Ejercicio 1.4 *Un empleado no puede ganar más que su jefe.*

Solución: Aunque también podríamos implementar esta regla con un único trigger para ambos eventos, usaremos de nuevo dos triggers, con lo que la transacción que incluya la actualización podrá continuar, como en el ejercicio anterior.

Control de inserciones: Una posible definición del trigger que controla la inserción de empleados con salario inválido es la siguiente:

```
CREATE OR REPLACE TRIGGER ora_emp_valido_ins
   BEFORE INSERT ON EMPL
   REFERENCING NEW AS tupla_nueva
   FOR EACH ROW
DECLARE
   NumJefesCobranMenos   INT;
BEGIN
   SELECT COUNT(*) INTO NumJefesCobranMenos
       FROM EMPL
       WHERE Codigo=:tupla_nueva.Jefe
         AND Salario < :tupla_nueva.Salario;
   IF NumJefesCobranMenos=1
      THEN
          RAISE_APPLICATION_ERROR(-20002, 'El salario del empleado no es válido');
   END IF;
END;
```

Como se ve, este trigger es bastante diferente de los anteriores. No existe la cláusula `WHEN`, debido a que la condición requiere una subconsulta, y esto no es válido para Oracle. Por lo tanto, la comprobación se hace mediante un `IF` en el bloque PL/SQL de la acción. Además, este bloque ahora empieza con la cláusula `DECLARE` ya que necesitamos una variable local (`NumJefesCobranMenos`) que usamos para almacenar el número de jefes que cobran menos que el empleado que estamos insertando. Si este valor es 1, se hace un rollback para evitar la inserción.

En el siguiente apartado veremos los errores que se producen (no al crear el trigger sino al activarse) si declaramos el trigger de tipo `AFTER INSERT` o si especificamos una sentencia `ROLLBACK` en la acción del trigger.

Control de actualizaciones: Para controlar la modificación tendríamos de nuevo una condición compleja, por lo que no usamos la cláusula `WHEN` y trasladamos la comprobación de la condición al bloque de la acción.

```
CREATE TRIGGER ora_emp_valido_upd  -- No funcionará!!!
    BEFORE UPDATE ON EMPL
```

```
    REFERENCING NEW AS tupla_nueva
                OLD AS tupla_vieja
    FOR EACH ROW
--Declaramos variables locales
DECLARE jefe_emp        NUMBER(4);
        sal_jefe        NUMBER(7,2);
        num_subords     INTEGER;
        max_sal_subords NUMBER(7,2);
BEGIN
  -- Condición 1: Salario nuevo subordinado > salario jefe
  SELECT Jefe INTO jefe_emp
     FROM EMPL
     WHERE Codigo=:tupla_nueva.jefe;
  IF jefe_emp IS NOT NULL THEN                        -- El empleado tiene jefe
     SELECT Salario INTO sal_jefe
         FROM EMPL
         WHERE Codigo=jefe_emp;
     IF :tupla_nueva.Salario>sal_jefe THEN            -- Si cobra más que el jefe...
         :tupla_nueva.Salario := :tupla_vieja.salario;
     END IF;
  END IF;

  -- Condición 2: Salario nuevo (del jefe) < Salario de algún subordinado
  SELECT count(*) INTO num_subords
     FROM EMPL
     WHERE Jefe=:tupla_nueva.Codigo;
  IF num_subords > 0 THEN                          -- El jefe tiene subordinados
     SELECT max(salario) INTO max_sal_subords
        FROM EMPL
        WHERE Jefe=:tupla_nueva.codigo;
     IF :tupla_nueva.Salario<max_sal_subords THEN  -- El subordinado cobra más
        :tupla_nueva.Salario := :tupla_vieja.Salario;
     END IF;
  END IF;
END;
```

Aunque el trigger se crea sin errores, al intentar ejecutarlo nos dará un error al intentar seleccionar datos de la tabla (que está "mutando") sobre la que se define el trigger, como veremos a continuación.

1.3.3. Limitaciones de Oracle y triggers mal definidos: errores de ejecución

Como se había indicado, una de las mayores limitaciones de los triggers en Oracle es que no se pueden seleccionar ni actualizar datos en las tablas que están mutando. Estas tablas son la que está asociado el trigger y aquellas otras tablas que esta referencia con claves foráneas y restricción `ON DELETE CASCADE`. El trigger se crea correctamente, pero no funciona como es de esperar al insertar datos en la tabla.

Así, vemos que si creamos en el Ejercicio 1.1 el trigger para controlar las inserciones de empleados que cobran más que su jefe de tipo `AFTER INSERT`, este se crea correctamente.

```
CREATE OR REPLACE TRIGGER ora_emp_valido_ins
   AFTER INSERT ON EMPL
   REFERENCING NEW AS tupla_nueva
   FOR EACH ROW
DECLARE
   NumJefesCobranMenos   INT;
BEGIN
   SELECT COUNT(*) INTO NumJefesCobranMenos
       FROM EMPL
       WHERE Codigo=:tupla_nueva.Jefe
         AND Salario < :tupla_nueva.Salario;
   IF NumJefesCobranMenos=1
      THEN
          ROLLBACK;
   END IF;
END;
Disparador creado.
```

Sin embargo, como se muestra a continuación, si tratamos de insertar una fila válida en la tabla `EMPL` (que actualmente está vacía), se produce el siguiente error.

```
INSERT INTO EMPL (codigo,salario) VALUES(1212,1000);
INSERT INTO EMPL (codigo,salario) VALUES(1212,1000)
            *
ERROR en línea 1:
ORA-04091: la tabla SYSTEM.EMPL está mutando,
puede que el disparador/la función no puedan verla
ORA-06512: en "SYSTEM.ORA_EMP_VALIDO_INS", línea 4
ORA-04088: error durante la ejecución del disparador
'SYSTEM.ORA_EMP_VALIDO_INS'
```

Dado que Oracle no tiene limitadas las acciones que se pueden indicar en la acción para los triggers tipo `BEFORE` o `AFTER`, podemos crear el mismo trigger perfectamente de tipo `BEFORE`, en el que la tabla aún no está mutando:

```
CREATE OR REPLACE TRIGGER ora_emp_valido_ins
   BEFORE INSERT ON EMPL
   REFERENCING NEW AS tupla_nueva
   FOR EACH ROW
DECLARE
   NumJefesCobranMenos   INT;
BEGIN
   SELECT COUNT(*) INTO NumJefesCobranMenos
       FROM EMPL
       WHERE Codigo=:tupla_nueva.Jefe
         AND Salario < :tupla_nueva.Salario;
```

```
   IF NumJefesCobranMenos=1
      THEN
          ROLLBACK;
   END IF;
END;
Disparador creado.
```

Si intentamos de nuevo insertar la tupla válida, esta se inserta sin problemas:

```
INSERT INTO EMPL (codigo,salario) VALUES(1212,1000);
1 fila creada.
```

Sin embargo, tal como está el trigger, sigue sin ser correcto. Funciona para las tuplas que se insertan correctamente, pero si tratamos de insertar un empleado con un salario no válido, no obtenemos el resultado esperado, ya que no nos permite hacer un rollback dentro de un trigger:

```
INSERT INTO EMPL (codigo,jefe,salario) VALUES(1111,1212,1100);
INSERT INTO EMPL (codigo,jefe,salario) VALUES(1111,1212,1100)
            *
ERROR en línea 1:
ORA-04092: no se ha podido ROLLBACK en un disparador
ORA-06512: en "SYSTEM.ORA_EMP_VALIDO_INS", línea 10
ORA-04088: error durante la ejecución del disparador
'SYSTEM.ORA_EMP_VALIDO_INS'
```

Si en vez de la sentencia `ROLLBACK` generásemos una excepción, utilizando `RAISE_APPLICATION_ERROR`, sí obtendríamos el resultado esperado (tal como se hizo en el apartado anterior).

En cuanto al trigger de actualización, parece que está definido de forma correcta, pero tampoco funcionará, por acceder (para consulta) a la tabla que está mutando.

```
UPDATE EMPL SET SALARIO=80 WHERE CODIGO=20
       *
ERROR en línea 1:
ORA-04091: la tabla SYSTEM.EMPL está mutando,
puede que el disparador/la función no puedan verla
ORA-06512: en "SYSTEM.ORA_EMP_VALIDO_UPD", línea 7
ORA-04088: error durante la ejecución del disparador 'SYSTEM.ORA_EMP_VALIDO_UPD'
```

Para este caso, no hay una forma directa de solucionar este problema, y no consideramos aquí alternativas que incluyan el uso de arrays o tablas temporales ni sucesiones de triggers sobre ellas, por lo que este trigger no se definirá. En consecuencia, o bien tratamos las actualizaciones de forma separada, o el uso de triggers no sirve para hacer cumplir la regla "un empleado no puede ganar más que su jefe".

Como vemos, el hecho de que no podamos seleccionar datos de la tabla sobre la que se define un trigger hace que la teórica potencia de los triggers decrezca enormemente en Oracle.

1.4. Ejercicios

1.4.1. Ejercicios resueltos

Ejercicio 1.5 *Consideremos una base de datos para la gestión de un almacén. Para esta base de datos consideraremos el siguiente esquema:*

- FACTURA(COD_FAC, FECHA)
- LINEA(COD_FAC,COD_PROD, PRECIO_UNITARIO, CANTIDAD)
- TOTALES(COD_PROD, TOTAL_VENDIDO)

Las tablas de facturas y líneas de factura son las convencionales, con la salvedad de que no admitimos dos líneas con el mismo producto en una factura (se sumarían las cantidades y formaríamos una sola línea), por lo que usaremos el código del producto en combinación con el código de factura como clave primaria de la clave de líneas. Asimismo, omitimos en este modelo la tabla de productos, en donde estaría la descripción del mismo.

La tabla de totales nos indica para cada producto el importe total vendido hasta el momento. Si no hay ninguna venta de un producto de código X, no habrá una línea en la tabla de totales, mientras que si sí hay ventas, la tupla correspondiente en totales contendrá la suma del importe vendido para ese código.

Se pide para esta base de datos la implementación en Oracle de lo siguiente:

1. *Las sentencias SQL para crear las tablas.*
2. *Las sentencias de creación de los triggers correspondientes para que la tabla de totales se actualice automáticamente al insertar, borrar o modificar líneas de factura.*

Solución:

Creación de las tablas: La creación correcta de las tablas, incluyendo las claves primarias y foráneas, es la siguiente. El atributo COD_PROD en la tabla TOTALES referencia a un producto en la línea de factura, pero dado que no es clave primaria ni tiene restricción de unicidad, no se puede establecer una clave foránea en este caso.

```
CREATE TABLE FACTURA(
   COD_FAC NUMBER(4),
   FECHA DATE,
   CONSTRAINT pk_factura PRIMARY KEY(COD_FAC)
   );

CREATE TABLE LINEA(
   COD_FAC NUMBER(4),
   COD_PROD NUMBER(4),
   PRECIO_UNITARIO NUMBER(7,2),
   CANTIDAD NUMBER(3),
   CONSTRAINT pk_linea PRIMARY KEY(COD_FAC, COD_PROD),
   CONSTRAINT fk_factura FOREIGN KEY(COD_FAC)
       REFERENCES FACTURA(COD_FAC)
```

```
   );

CREATE TABLE TOTALES(
   COD_PROD NUMBER(4),
   TOTAL NUMBER(9,2),
   CONSTRAINT pk_totales PRIMARY KEY(COD_PROD);
   );
```

Creación de los triggers: En este caso tendremos tres triggers, ya que tenemos que considerar la inserción, el borrado, y la modificación de una tupla de ventas. Se necesitan los tres triggers, uno para cada evento, porque el tratamiento (la acción a realizar) es distinto en cada uno de los eventos. En todos los casos crearemos los triggers de tipo `AFTER` y con granularidad `FOR EACH ROW`.

Para controlar la inserción, se intenta actualizar la fila correspondiente al mismo producto en la tabla `TOTALES`. Si no está (`SQL%NOTFOUND` será cierto), se inserta la fila.

```
CREATE OR REPLACE TRIGGER insertar_venta
   AFTER INSERT ON LINEA
   REFERENCING NEW AS tupla_nueva
   FOR EACH ROW
BEGIN
   UPDATE TOTALES
      SET TOTAL = TOTAL + (:tupla_nueva.precio_unitario * :tupla_nueva.cantidad)
      WHERE COD_PROD = :tupla_nueva.cod_prod;
   IF SQL%NOTFOUND THEN
      INSERT INTO TOTALES (COD_PROD, TOTAL)
         VALUES(:tupla_nueva.cod_prod,
     :tupla_nueva.precio_unitario * :tupla_nueva.cantidad);
   END IF;
END;
```

En el caso de la eliminación de una venta, decrementamos el total en la tabla `TOTALES`, que debe existir si no se han borrado filas manualmente. En el caso de que el total quede a cero, se debe borrar la fila.

```
CREATE OR REPLACE TRIGGER borrar_venta
   AFTER DELETE ON LINEA
   REFERENCING OLD AS tupla_vieja
   FOR EACH ROW
DECLARE
   TotalDespues TOTALES.TOTAL%TYPE;
BEGIN
   UPDATE TOTALES
      SET TOTAL = TOTAL - (:tupla_vieja.precio_unitario * :tupla_vieja.cantidad)
      WHERE COD_PROD = :tupla_vieja.cod_prod;
   SELECT TOTAL INTO TotalDespues
      FROM TOTALES
      WHERE COD_PROD = :tupla_vieja.cod_prod;
```

```
   IF TotalDespues = 0 THEN
      DELETE FROM TOTALES
         WHERE COD_PROD = :tupla_vieja.cod_prod;
   END IF;
END;
```

La actualización de una fila de ventas puede ser simple o complicada, dependiendo de lo que intentemos controlar en la actualización. Lo más sencillo, además de ser lo más común, es controlar sólo la modificación de la cantidad y el precio unitario, y no considerar Si se quisiese considerar la modificación del código de la factura o del producto, algo realmente inusual, habría que añadir esas comprobaciones en el trigger. En vez de controlarlo en el mismo trigger, aquí usaremos otro (`no_mod_codigos`, en este caso de tipo `BEFORE`) para no permitir las modificaciones de esos códigos.

```
CREATE OR REPLACE TRIGGER modificar_venta
   AFTER UPDATE OF PRECIO_UNITARIO, CANTIDAD ON LINEA
   REFERENCING OLD AS tupla_vieja
               NEW AS tupla_nueva
   FOR EACH ROW
BEGIN
      UPDATE TOTALES
      SET TOTAL = TOTAL + (:tupla_nueva.precio_unitario * :tupla_nueva.cantidad)
                        - (:tupla_vieja.precio_unitario * :tupla_vieja.cantidad)
      WHERE COD_PROD = :tupla_vieja.cod_prod;
END;

CREATE OR REPLACE TRIGGER no_mod_codigos
   BEFORE UPDATE OF COD_PROD, COD_FAC ON LINEA
   REFERENCING OLD AS tupla_vieja
               NEW AS tupla_nueva
   FOR EACH ROW
   WHEN ((tupla_nueva.cod_prod <> tupla_vieja.cod_prod)
         OR (tupla_nueva.cod_fac <> tupla_vieja.cod_fac))
BEGIN
   RAISE_APPLICATION_ERROR ( -20005,
      'No se puede modificar el código del producto o de la factura de una línea');
END;
```

Verificación de los triggers Una vez creados los triggers, comprobemos su funcionamiento. Para ello, comenzaremos con las tres tablas vacías e iremos insertando, borrando y modificando ventas.

```
-- Tablas inicialmente vacías
SELECT * FROM FACTURA;
ninguna fila seleccionada

SELECT * FROM LINEA;
ninguna fila seleccionada
```

```
SELECT * FROM TOTALES;
ninguna fila seleccionada

-- Insertamos una factura y una línea
INSERT INTO FACTURA (COD_FAC, FECHA) VALUES(1, SYSDATE);
1 fila creada.

INSERT INTO LINEA (COD_FAC, COD_PROD,PRECIO_UNITARIO,CANTIDAD)
     VALUES (1, 1000, 23.5, 7);
1 fila creada.

-- Vemos cómo se actualiza el total
SELECT * FROM TOTALES;
  COD_PROD      TOTAL
---------- ----------
      1000      164,5

-- Insertamos otra factura con otra línea, del mismo producto
INSERT INTO FACTURA (COD_FAC, FECHA) VALUES(2, SYSDATE);
1 fila creada.

INSERT INTO LINEA (COD_FAC, COD_PROD,PRECIO_UNITARIO,CANTIDAD)
     VALUES (2, 1000, 23.5, 3);
1 fila creada.

--Vemos cómo se actualizan los valores
SELECT * FROM TOTALES;
  COD_PROD      TOTAL
---------- ----------
      1000        235

-- Intentamos modificar el código de factura de una línea
-- y no es posible
UPDATE LINEA SET COD_FAC=1
   WHERE COD_FAC=2 AND COD_PROD=1000;
UPDATE LINEA SET COD_FAC=1
*
ERROR en línea 1:
ORA-20005: No se puede modificar el código del producto o de la factura de una
línea
ORA-06512: en "SYSTEM.NO_MOD_CODIGOS", línea 2
ORA-04088: error durante la ejecución del disparador 'SYSTEM.NO_MOD_CODIGOS'

-- Modificamos el precio y cantidad de una línea
-- y vemos cómo los cambios se ven reflejados
UPDATE LINEA SET PRECIO_UNITARIO=20
```

```
   WHERE COD_FAC=2 AND COD_PROD=1000;
1 fila actualizada.

UPDATE LINEA SET CANTIDAD=10
   WHERE COD_FAC=2 AND COD_PROD=1000;
1 fila actualizada.

SELECT * FROM TOTALES;

  COD_PROD      TOTAL
---------- ----------
      1000      364,5

-- Borramos una linea
DELETE FROM LINEA
   WHERE COD_FAC=2 AND COD_PROD=1000;
1 fila borrada.

SELECT * FROM TOTALES;
  COD_PROD      TOTAL
---------- ----------
      1000      164,5

-- Borramos la última línea
DELETE FROM LINEA;
1 fila borrada.

SELECT * FROM TOTALES;
ninguna fila seleccionada
```

Ejercicio 1.6 *En nuestra empresa vendemos una serie de productos, que tienen un precio de venta al público. La estructura de la tabla que los almacena es la siguiente:*

- PRODUCTO(CODIGO, DESCRIPCION, CATEGORIA, PRECIO)

Queremos tener en una tabla almacenados los productos más baratos de cada categoría, por lo que tendremos la siguiente tabla

- OFERTAS(CODIGO, CATEGORIA, PRECIO)

Como vemos, el esquema es similar, pero esta tabla almacena un único producto por categoría (por ello se ha elegido la categoría como clave primaria), que corresponde al más barato de la misma.

Se pide la implementación de los triggers necesarios para la alimentación automática de esa tabla.

Solución:

En primer lugar, creemos las tablas:

```
CREATE TABLE PRODUCTO(
   CODIGO NUMBER(4),
   DESCRIPCION VARCHAR2(40),
   CATEGORIA VARCHAR2(20),
   PRECIO NUMBER(7,2),
   CONSTRAINT pk_producto PRIMARY KEY (CODIGO)
   );

CREATE TABLE OFERTAS(
   CODIGO NUMBER(4),
   CATEGORIA VARCHAR2(20),
   PRECIO NUMBER(7,2),
   CONSTRAINT pk_producto2 PRIMARY KEY (CATEGORIA),
   CONSTRAINT fk_producto FOREIGN KEY (CODIGO)
        REFERENCES PRODUCTO(CODIGO)
   );
```

Como en los casos anteriores, usaremos un trigger para cada uno de los tres posibles eventos, ya que las acciones son distintas. Veamos la creación de los tres triggers, y comprobaremos que el de actualización y borrado no funcionan, por intentar acceder a tablas que están mutando.

Sin embargo, veamos de todas formas cómo se haría *teóricamente*.

Control de inserciones Para gestionar las inserciones, comprobamos si hay la fila correspondiente en la tabla de ofertas (si no hay filas, el mínimo devolverá nulo). Si no existe, se inserta. En otro caso, si el valor del mínimo precio obtenido es mayor que el precio del producto que se está insertando, se actualiza ese valor. Este trigger sí funciona de forma correcta.

```
CREATE OR REPLACE TRIGGER insertar_producto
   AFTER INSERT ON PRODUCTO
   FOR EACH ROW
DECLARE
  MinPrecio PRODUCTO.PRECIO%TYPE;
BEGIN
  SELECT MIN(PRECIO) INTO MinPrecio
     FROM OFERTAS
     WHERE CATEGORIA = :new.categoria;

  IF MinPrecio IS NULL THEN --No había la fila
    DBMS_OUTPUT.PUT_LINE('No había. Insertamos');
    INSERT INTO OFERTAS (CODIGO, CATEGORIA, PRECIO)
      VALUES (:new.codigo, :new.categoria, :new.precio);
  ELSIF MinPrecio > :new.precio THEN
    DBMS_OUTPUT.PUT_LINE('Sí había. Actualizamos');
     UPDATE OFERTAS
     SET PRECIO = :new.precio
       WHERE CATEGORIA = :new.categoria;
  END IF;
END;
```

Control de borrados Cuando se borra un producto, debemos comprobar (después de borrarlo) cuál es el mínimo precio de la categoría del producto borrado. La solución, teóricamente, sería la siguiente (nota: no funciona).

```
CREATE OR REPLACE TRIGGER borrar_producto
   AFTER DELETE ON PRODUCTO
   FOR EACH ROW
DECLARE
   MinPrecio PRODUCTO.PRECIO%TYPE;
BEGIN
  SELECT MIN(PRECIO) INTO MinPrecio
     FROM PRODUCTO                              --oh, oh ...
     WHERE CATEGORIA =:old.categoria;
  IF MinPrecio IS NULL THEN
     DELETE FROM PRODUCTO                       --oh, oh ...
         WHERE CATEGORIA = :old.categoria;
  ELSE
      UPDATE OFERTAS
         SET PRECIO = MinPrecio
         WHERE CODIGO= :old.codigo
           AND CATEGORIA = :old.categoria;
  END IF;
END;
```

Control de actualizaciones Controlaremos únicamente la actualización del precio de un producto. En esta actualización debemos considerar dos posibilidades: que un producto pase a ser el que menos cuesta de su categoría cuando antes no lo era (al decrementar su precio), o que un producto que era el que menos costaba, ahora no lo sea (al incrementar su producto). Sin embargo, la acción a tomar en ambos casos puede considerarse la misma: actualizar la fila de la tabla de ofertas con el mínimo de los precios de la categoría del producto que estamos modificando. Por ello, la solución, en teoría, sería la siguiente.

```
CREATE OR REPLACE TRIGGER modificar_producto
   AFTER UPDATE OF PRECIO ON PRODUCTO
   FOR EACH ROW
BEGIN
      UPDATE OFERTAS
         SET PRECIO = (SELECT MIN(PRECIO)
                         FROM PRODUCTO
                         WHERE CATEGORIA =:new.categoria)
         WHERE CODIGO= :new.codigo
           AND CATEGORIA = :new.categoria;
END;
```

Verificación de los triggers Vamos a comprobar el funcionamiento de los triggers. Como se ve en el siguiente script, el trigger de inserción funciona, pero el de borrado y el de actualización no, porque intentan seleccionar datos de la tabla `PRODUCTO`, que durante el proceso está mutando.

```
-- Activamos SERVEROUTPUT para ver los mensajes
SET SERVEROUTPUT ON;
-- Insertamos la primera fila y como no la hay en la
-- tabla de ofertas, se añade
INSERT INTO PRODUCTO(CODIGO,DESCRIPCION,CATEGORIA,PRECIO)
    VALUES (1001,'Libreta roja A4 XGR','LIBRETA A4', 1.5);
No había. Insertamos
1 fila creada.
SELECT * FROM OFERTAS;
    CODIGO CATEGORIA                PRECIO
---------- -------------------- ----------
      1001 LIBRETA A4                   1,5

-- Insertamos un producto más barato y se actualiza la oferta
INSERT INTO PRODUCTO(CODIGO,DESCRIPCION,CATEGORIA,PRECIO)
    VALUES (1002,'Libreta roja A4 Barata','LIBRETA A4', 0.5);
Sí había. Actualizamos
1 fila creada.
SELECT * FROM OFERTAS;
    CODIGO CATEGORIA                PRECIO
---------- -------------------- ----------
      1001 LIBRETA A4                   ,5

-- Los triggers de actualización y borrado no funcionan
-- al intentar seleccionar de la tabla PRODUCTO
UPDATE PRODUCTO
   SET PRECIO=0.4 WHERE CODIGO=1001;
UPDATE PRODUCTO SET PRECIO=0.4 WHERE CODIGO=1001
       *
ERROR en línea 1: ORA-04091: la tabla SYSTEM.PRODUCTO está
mutando, puede que el disparador/la función no puedan verla
ORA-06512: en "SYSTEM.MODIFICAR_PRODUCTO", línea 2
ORA-04088:  error durante la ejecución del disparador 'SYSTEM.MODIFICAR_PRODUCTO'

DELETE FROM PRODUCTO  WHERE CODIGO=1001;
DELETE FROM PRODUCTO
   WHERE CODIGO=1001
           *
ERROR en línea 1:
ORA-04091: la tabla SYSTEM.PRODUCTO está mutando,
puede que el disparador/la función no puedan verla
ORA-06512: en "SYSTEM.BORRAR_PRODUCTO", línea 4
ORA-04088: error durante la ejecución del disparador 'SYSTEM.BORRAR_PRODUCTO'
```

1.4.2. Ejercicios sin resolver

Ejercicio 1.7 *Supóngase una tabla de socios de un videoclub, con el esquema siguiente:*

- SOCIO(CODIGO, NOMBRE, FECHA_ULT_MOD)

donde FECHA_ULT_MOD*, de tipo fecha, debe contener la fecha de última modificación de la fila del socio (o la de inserción, si no se ha modificado).*

Impleméntense los triggers necesarios para gestionar esta fecha automáticamente.

Ejercicio 1.8 *Consideremos una biblioteca que presta libros, gestionado mediante la siguiente base de datos (se muestran sólo las tablas necesarias)*

- SOCIO(DNI, NOMBRE)
- EJEMPLAR(ISBN, NUMERO_EJEMPLAR, LOCALIZACION)
- PRESTAMO(DNI, ISBN, NUMERO_EJEMPLAR, FECHA_PRESTAMO, FECHA_DEVOLUCION)

La tabla préstamos almacena el socio (DNI) y el ejemplar (ISBN y Número de Ejemplar) del libro prestado. Cuando se realiza un préstamo se indica en su fecha la fecha actual, y un nulo en la fecha de devolución, guardándose el histórico de todos los préstamos. Cuando se devuelve el libro, se indica la fecha de devolución, que pasa a tener la fecha actual en vez de un nulo.

Impleméntese un trigger para evitar que se realice un préstamo a un socio si éste tiene más de cinco libros sin devolver, creando además una entrada en una tabla LOG_INTENTOS(DNI, ISBN, FECHA_INTENTO) *que guarde los intentos no válidos de retirar un libro.*