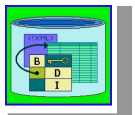


SQL 3 (Cuarta Parte)

Introducción

Poder expresar restricciones del lado del motor de base de datos resulta una estrategia fundamental, ya que minimiza las validaciones que hay que realizar en las aplicaciones cliente.

En este documento analizaremos qué restricciones pueden representarse en SQL2 y qué propuestas adicionales se incorporaron en SQL3.

3. Restricciones en el Modelo Relacional

El modelo relacional ofrece distintos tipos de restricciones:

- ***Restricciones de dominio*** (todo atributo debe ser atómico)
- ***Restricciones de clave*** (toda relación debe tener su clave)
- ***Restricciones de datos*** (ya sea de ***integridad de entidad***, que asegura que un atributo NULL no puede pertenecer a una clave, de ***integridad referencial***, que asegura que un conjunto de atributos en cierta relación puede referenciar la clave de otra relación en forma consistente, de ***dependencias funcionales*** y de ***dependencias multivaluadas)***.

Estas tres restricciones evitan que una base de datos se encuentre en *cierto estado* considerado inválido. Por ejemplo, no se permite insertar una tupla si viola la unicidad de la clave primaria. Precisamente previene que se realicen cambios no autorizados en una base de datos los cuales puedan resultar en una pérdida de consistencia de datos.

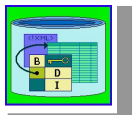
El lenguaje de consulta SQL tiene sentencias DDL que permiten representar estas y algunas otras restricciones.

SQL2 ofrece una variedad de técnicas para expresar restricciones dentro del esquema de base de datos, ya mencionadas en apuntes previos. Las mismas se especifican en el esquema de base de datos, con el objetivo de que el DBMS asegure su cumplimiento.

SQL3 también ofrece más formas de restricción, orientadas a disparar operaciones, más que a prevenir violación de estados: se especifican eventos y se indica qué acciones se realizarán en caso de que dichos eventos ocurran.

Aclaración

En lo que respecta a restricciones, los DBMS comerciales son más compatibles con SQL3 que con SQL2.



3.1 Restricciones en SQL 2

3.1.1 Restricción sobre el Dominio: TIPO, DEFAULT y CHECK

Recordemos la forma de la creación de los atributos en un sentencia CREATE TABLE, vista previamente:

Restricción para un atributo

```
CREATE TABLE nombre_tabla
(
     ...
     nombre_atributo_i   tipo_i   [DEFAULT default_i]   [restricción_i],
     ...
);
```

donde ***resticcion_i*** permite una expresión del tipo NOT NULL, PRIMARY KEY, UNIQUE o CHECK(***condición_i***) tal que dicha ***condición_i*** tiene la forma de cualquiera que se use en una cláusula WHERE

Esta restricción indica que todo atributo debe ser atómico en su dominio. Respecto del dominio de cierto atributo podemos decir que el mismo puede expresarse por medio de su ***tipo*** e indicando si puede o no valer **NULL**.

Ambas partes constituyen una importante restricción, sin embargo, puede ser importante expresar una restricción más fuerte que indique cuáles son los valores exactos admitidos. Esto típicamente ocurre cuando el rango de valores de su tipo es demasiado amplio para cierto atributo. En estos casos, se puede hacer uso de la cláusula **CHECK**. Dicha cláusula sólo se valida en inserciones o actualizaciones (INSERT o UPDATE), momento en el cual se controla si el valor a setear corresponde con los enumerados en la lista.

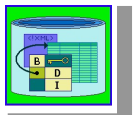
La cláusula CKECK sólo puede expresar una restricción sobre el *atributo* que está modificando (***constraint attributes***).

La restricción expresada en el CHECK sobre un atributo ***atrib*** será de la forma: ***atrib*** IS NOT NULL, ***atrib*** < 10, ***atrib*** BETWEEN 1 AND 5, ***atrib*** IN ('a', 'e', 'i', 'o', 'u'), o bien la validación contra una lista dinámica (SELECT)

Ejemplo 1

```
CREATE TABLE empleado
(
   nombre CHAR(100) NOT NULL,
   sueldo FLOAT CHECK( sueldo > 50),
   PRIMARY KEY (nombre)
);
```

Por medio de la cláusula CHECK, indicamos la lista de valores posibles que deberán proporcionarse en insert o update.



Ejemplo 2

Combinaremos ambas restricciones DEFAULT y CHECK. Si hay alguna incompatibilidad la creación de la tabla no falla, pero sí la inserción o actualización. Veamos un ejemplo:

```
CREATE TABLE empleado
(
   nombre CHAR(100) NOT NULL,
   sueldo FLOAT DEFAULT 100 CHECK( sueldo > 50),
   PRIMARY KEY (nombre)
);
```

Al intentar ejecutar ***INSERT INTO empleado VALUES ('Maria', 20)*** *el DBMS rechazaría la acción y enviaría un mensaje de error al estilo* ***"check constraint nombre_constraint violated"***

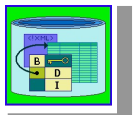
Ejemplo 3

La siguiente creación muestra una contradicción que **recién validaría el DBMS en inserción** con default.

```
CREATE TABLE empleado
(
   nombre CHAR(100) NOT NULL,
   sueldo FLOAT DEFAULT 100 CHECK( sueldo > 250),
   PRIMARY KEY (nombre)
);
```

Ejemplo 4

```
CREATE TABLE empleado
(
   nombre CHAR(100) NOT NULL,
   sueldo FLOAT DEFAULT 100 CHECK( sueldo > 50),
   cargo CHAR(20) CHECK( cargo IN ('Gerente', 'Secretaria', 'Vendedor')),
   PRIMARY KEY (nombre)
);
```



Ejemplo 5

En este caso se genera una lista dinámica.

```
CREATE TABLE cargo
(
   descripcion CHAR(20) NOT NULL,
   PRIMARY KEY(descripcion)
);


CREATE TABLE empleado
(
   nombre CHAR(100) NOT NULL,
   sueldo FLOAT DEFAULT 100 CHECK( sueldo > 50),
   cargo CHAR(20) CHECK( cargo IN (SELECT descripcion FROM cargo)),
   PRIMARY KEY (nombre)
);
```

Muy Importante

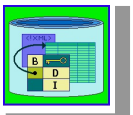
La validación que se obtiene con el CHECK que incluye un SELECT **NO ES IGUAL** al de una clave foránea.

¿Por qué? Porque el CHECK dinámico sólo se evalúa en el momento de la inserción o modificación, pero no se revé frente a cambios en las otras tablas.

En el ejemplo anterior, el DBMS chequea en cada inserción o modificación de una tupla de la **tabla *empleado*** si el valor proporcionado para el atributo *cargo* existe entre los valores del atributo *descripcion* correspondiente en la tabla *cargo*, y de no ser así rechaza la operación. Sin embargo, si se borrara más tarde de la **tabla *cargo*** una descripción, las tuplas en la tabla *empleado* que "referenciaban" a la tupla borrada no son consideradas como violadoras de la restricción, ya que la misma sólo se detecta en inserción o modificación. Por lo tanto, queda claro que **es superior** el uso de claves foráneas frente a este tipo de chequeos.

Aclaración

PostgreSQL no permiten el uso de subqueries (SELECT) en el CHECK, razón por la cual sólo se puede enumerar una lista fija. ¿Ya se imaginan por qué?



3.1.2 Restricción de Clave

Ya analizamos previamente la declaración de claves candidatas por medio de las cláusulas PRIMARY KEY y UNIQUE. Sin embargo, en SQL la declaración de claves es optativa.

Ejemplo 6

Supongamos que se quiere representar que sólo debe haber un empleado por cada puesto de trabajo y que los sueldos tampoco pueden repetirse. Lo podría expresar indistintamente de dos formas:

```
CREATE TABLE empleado
(
  nombre CHAR(100) NOT NULL,
  sueldo FLOAT DEFAULT 100 NOT NULL CHECK( sueldo > 50) UNIQUE,
  cargo CHAR(20) NOT NULL CHECK( cargo IN
                ('Gerente', 'Secretaria', 'Vendedor')) UNIQUE,
  PRIMARY KEY (nombre)
);
```

o bien

```
CREATE TABLE empleado
(
  nombre CHAR(100) NOT NULL,
  sueldo FLOAT DEFAULT 100 NOT NULL CHECK( sueldo > 50),
  cargo CHAR(20) NOT NULL CHECK( cargo IN
                          ('Gerente', 'Secretaria', 'Vendedor')),
  PRIMARY KEY (nombre),
  UNIQUE(sueldo),

  UNIQUE(cargo)
);
```

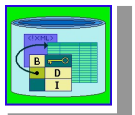
Nota

Esta es la forma de restringir el mapeo de una relación 1:1 sin participación total

Ejemplo 7

¿Expresa la siguiente tabla la misma restricción que el ejemplo anterior?

```
CREATE TABLE empleado
(
  nombre CHAR(100) NOT NULL,
  sueldo FLOAT DEFAULT 100 NOT NULL CHECK( sueldo > 50),
  cargo CHAR(20) NOT NULL CHECK( cargo IN
                          ('Gerente', 'Secretaria', 'Vendedor')),
  PRIMARY KEY (nombre),
  UNIQUE(sueldo ,cargo)
);
```



Obviamente NO!

La semántica del esquema permite insertar personas que cubran los mismos cargos con tal de que sus sueldos sean distintos.

En el esquema del ejemplo 6, si se tuvieran ya insertadas las siguientes dos tuplas

```
INSERT INTO empleado VALUES ('Ana', 3000, 'Gerente');
INSERT INTO empleado VALUES ('Juan', 1500, 'Vendedor');
```

y se quisiera insertar

```
INSERT INTO empleado VALUES ('Celia', 4000, 'Gerente');
```

se obtendría un error.

En cambio, en el esquema del ejemplo 7, estas 3 tuplas serían aceptadas y por lo tanto violarían el enunciado.

3.1.3 Restricciones de Datos

3.1.3.1 Restricción de Integridad de Entidad

En SQL2 se garantiza que todo atributo que es parte de una clave primaria no pueda ser NULL. Sin embargo, los atributos UNIQUE sí pueden valer NULL.

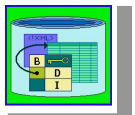
Aclaración

En PostgreSQL un atributo UNIQUE puede valer NULL. Pero recordar que considera dos valores NULL como diferentes (pueden haber tuplas repetidas en el valor NULL). O sea, UNIQUE y PRIMARY KEY no son realmente equivalentes, salvo que se defina EXPLICITAMENTE CON restricción NOT NULL => hacerlo explícito.

3.1.3.2 Restricción de Integridad Referencial

Ya analizamos, previamente en la materia, la declaración de claves foráneas por medio de la cláusula FOREIGN KEY la cual permite asegurar que los valores de atributos de una relación también aparecen en otra tabla.

Recordar que puede referenciarse a una clave PK o UNIQUE. En este último caso es obligatorio listar sus atributos, ya que puede haber varios UNIQUE definidos.



Ejemplo 8

```
CREATE TABLE cargo
(
   descripcion CHAR(20) NOT  NULL,
   PRIMARY KEY(descripcion)
);

CREATE TABLE empleado
(
   nombre CHAR(100) NOT NULL,
   sueldo FLOAT DEFAULT 100 CHECK( sueldo > 50),
   cargo CHAR(20),
   PRIMARY KEY (nombre),
   FOREIGN KEY (cargo)   REFERENCES cargo
);
```

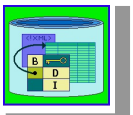
Si referencia una clave primaria sólo hace falta indicar el nombre de la tabla

Ejemplo 9

```
CREATE TABLE cargo
(
   codigo    INT NOT NULL,
   descripcion CHAR(20) NOT NULL,
   PRIMARY KEY(codigo),
   UNIQUE(descripcion)
);


CREATE TABLE empleado
(
   nombre CHAR(100) NOT NULL,
   sueldo FLOAT DEFAULT 100 CHECK( sueldo > 50),
   cargo CHAR(20),
   PRIMARY KEY (nombre),
   FOREIGN KEY (cargo)   REFERENCES   cargo(descripcion)
);
```

Definir todas las claves candidatas

Si referencia una clave candidata (UNIQUE) hace falta indicar el nombre de los atributos en la otra tabla



3.1.3.3 Restricción de Dependencias Funcionales y Multivaluadas

Este tipo de restricciones expresan restricciones sobre atributos de un esquema de relación completo y no una tabla en particular. Debería garantizarse que se siguen cumpliendo luego de la etapa de normalización (conservación de las dependencias).

Algunas se garantizan con PRIMARY KEY o UNIQUE y otras con programación (como veremos en las próximas secciones).

3.1.3.4 Otras Restricciones Avanzadas

Es muy común tener que representar restricciones complejas entre atributos, las cuales surgieron en la etapa de relevamiento y no se han podido representar en el modelo relacional.

Por ejemplo, que el sueldo de un empleado común jamás supere el sueldo de ningún gerente. Para ello se precisa mayor potencia expresiva. SQL ofrece dos formas para este tipo de restricciones: ***a nivel tabla*** (CHECK) y ***a nivel esquema*** (ASSERTION)

○ *Restricciones a Nivel Tabla*

Se utiliza la misma cláusula CHECK pero no modifica un atributo en particular. Permite expresar condiciones más avanzadas: comparar atributos de una misma tabla o de tablas distintas.

Sólo cuando se realiza una inserción o actualización de tuplas en dicha tabla, el DBMS verifica si se cumple o no la condición expresada y de no ser así rechaza la operación.

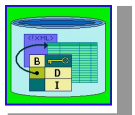
Restricción para un CHECK a nivel TABLA

```
CREATE TABLE nombre_tabla
(
     ...
     nombre_atributo_i    tipo_i   [DEFAULT default_i]   [restricción_i],
      ...
      CHECK (condicion_k),
      …
);
```

donde ***condición_k*** tiene la forma de cualquiera que se use en una cláusula WHERE

Muy Importante

El comportamiento del CHECK a nivel tabla es similar al CHECK para atributos individuales. Sólo se evalúa la condición en cada inserción o modificación, pero si una vez insertada una tupla, la referenciada por ella desaparece o cambia, ya no se considera que viola la restricción.



***Ejemplo 10**:*

Supongamos que queremos expresar la restricción de que un empleado común no supere un sueldo de 5000 pesos, salvo que sea gerente. Esta restricción **no puede expresarse** con dos check de atributos independientes (involucra más de un campo) pero sí con check a nivel tabla.

```
CREATE TABLE empleado
(
   nombre CHAR(100) NOT NULL,
   sueldo FLOAT DEFAULT 100,
   cargo CHAR(20),
   PRIMARY KEY (nombre),
   CHECK ( (sueldo BETWEEN 0 AND 5000) OR (cargo IN ('Gerente')))
);
```

Con esta restricción se podrían insertar las tuplas

```
INSERT INTO empleado VALUES ('Ana', 3000, 'Vendedor');
INSERT INTO empleado VALUES ('Juan', 7000, 'Gerente');
```

pero la siguiente inserción sería rechazada

```
INSERT INTO empleado VALUES ('Juan', 7000, 'Vendedor');
```

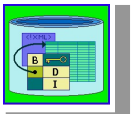
Ejemplo 11

Si se tuviera una tabla que guarda los cargos jerárquicos de la empresa y se quiere expresar la restricción de que un empleado no supere un sueldo de 5000 pesos salvo que sea jerárquico, se podría utilizar:

```
CREATE TABLE cargoJerarquico
(
   descripcion CHAR(20) NOT NULL,
   PRIMARY KEY(descripcion)
);

CREATE TABLE empleado
(
   nombre CHAR(100) NOT NULL,
   sueldo FLOAT DEFAULT 100,
   cargo CHAR(20),
   PRIMARY KEY (nombre),
   CHECK ( (sueldo BETWEEN 0 AND 5000)
                OR (cargo IN (SELECT descripcion FROM cargoJerarquico)))
);
```

Aclaración

PostgreSQL no permite el uso de subqueries (SELECT) en el CHECK de tuplas. Sólo se puede enumerar una lista fija.



○ *Restricciones a Nivel Esquema*

Para restricciones a nivel esquema se utiliza la cláusula ASSERTION que, a diferencia de los CHECKS, garantiza la consistencia en todo momento y no sólo en cada inserción o actualización.

Resultan ser más potentes que los FOREIGN KEY porque permiten expresar condiciones más complejas que una simple referencia a otro conjunto de atributos.

Estas cláusulas van fuera de la creación de las tablas, o sea a nivel esquema.

Sintaxis (fuera de las definiciones de tablas)

```
CREATE ASSERTION nombre_assertion CHECK (expresion);
```

donde ***expresion*** puede ser cualquiera que pueda usarse en el WHERE del SELECT

Muy Importante

Cualquiera sea la operación que se realice sobre una tabla de la base de datos, se chequean todas las aserciones declaradas que involucren a dicha tabla: si las condiciones son verificadas la operación se realiza, sino se rechaza.

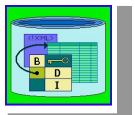
Ejemplo 12

Supongamos que queremos expresar la restricción de que el sueldo de cualquier empleado no supere un sueldo de gerente. Esta restricción es demasiado importante para expresarla con un CHECK porque debe garantizarse en todo momento (recordar el problema del antes citado).

```
CREATE TABLE empleado
(
  nombre CHAR(100) NOT NULL,
  sueldo FLOAT DEFAULT 100,
  cargo CHAR(20) CHECK( cargo IN ('Gerente', 'Secretaria', 'Vendedor')),
  PRIMARY KEY (nombre)
);
```

Debemos agregar la restricción:

```
CREATE ASSERTION SueldoTope
CHECK ( NOT EXISTS ( SELECT *
                     FROM empleado
                     WHERE cargo <> 'Gerente' AND
                            sueldo >ANY ( SELECT sueldo
                                          FROM empleado
                                          WHERE cargo = 'Gerente')
      ) );
```



Ejemplo 13

Supongamos que queremos expresar la restricción de que no haya puesto de trabajo con un promedio de sueldo para ese puesto, superior a los 5000. Si tenemos ya creada la tabla **empleado** del ejemplo anterior, agregaríamos la restricción:

```
CREATE ASSERTION SueldoPromedio
CHECK ( NOT EXISTS ( SELECT cargo
                     FROM empleado
                     GROUP BY cargo
                     HAVING AVG(sueldo) > 5000
       ) );
```

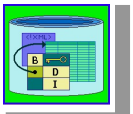
Ejemplo 14:

Supongamos que queremos expresar la restricción de que no haya más secretarias que vendedores. Si tenemos ya creada la tabla **empleado** del ejemplo anterior, agregaríamos la restricción:

```
CREATE ASSERTION NoDemasiadasSecretarias
CHECK (  (SELECT COUNT(*) FROM empleado WHERE cargo = 'Secretaria')
          <=
         (SELECT COUNT(*) FROM empleado WHERE cargo = 'Vendedor')
       );
```

Aclaración

PostgreSQL no tiene implementado ASSERTIONS ya que las reemplazaron por los llamados TRIGGERS



3.2 Restricciones en SQL3

SQL2 responde a una operación inválida rechazándola.

SQL3 introduce una novedad: permite llevar a la base de datos a un estado compatible con la restricción, para que la operación no falle. Esto implica permitir que el DBMS ejecute cierta "acción" para llevar a la base de datos a cierto estado, antes o luego de la operación solicitada.

De hecho, en SQL2 también existía este tipo de comportamiento, pero estaba restringido al FOREIGN KEY, o sea que no podía ser definido por los programadores de una aplicación. Tal es el caso de las opciones de SET NULL, CASCADE, RESTRICT o SET DEFAULT en dicha cláusula. Las mismas exigen realizar cierta "acción extra" sobre la base de datos para dejarla consistente.

El nuevo estándar introduce una posibilidad más general: permitir que los programadores puedan definir las acciones a seguir ante determinadas situaciones (no tienen por qué ser SET NULL, SET DEFAULT, RESTRICT o CASCADE). Agrega el concepto de *elemento activo*, que consiste en un código almacenado en la base de datos esperando para ser ejecutado.

Tiene diversas ***ventajas***: por ser código compilado (ya está en formato binario) no se pierde tiempo *parseando* la expresión en cada ejecución, por estar ya testeado disminuye entonces la probabilidad de error y por residir en forma centralizada reduce el tráfico entre las aplicaciones y el servidor.

3.2.1 Trigger (Disparador o Activador)

Es una idea más evolucionada que los *assertion.* Constituye la implementación de Sistemas de Bases de Datos Activos (Active Database Systems), o sea permiten crear y ejecutar “reglas” dentro de la base de datos que se activen frente a ciertos eventos. Dichas reglas (código) residirán en la base de datos junto con los datos propiamente dichos y serán ejecutadas por el DBMS automáticamente cuando los eventos declarados ocurran.

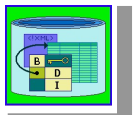
En ANSI se definió que los eventos deban producir modificación en los datos, o sea sólo pueden ser INSERT, DELETE o UPDATE. Para ello el programador debe:

- Especificar los eventos (**INSERT, DELETE o UPDATE**) que producirán el accionar del código (*trigger*).
- Especificar las condiciones bajo las cuales se ejecutará.
- Especificar las acciones a seguir en el caso de que el evento se produzca.

El mecanismo funciona de la siguiente manera: una vez detectada la solicitud del evento anunciado, se transfiere el control al código del *trigger*, el cual indica si las acciones del cuerpo del trigger se ejecutarán **ANTES** (**BEFORE**) o **DESPUES** (**AFTER**) del evento (INSERT, DELETE o UPDATE). En otras palabras, se "intercepta" el accionar lanzando un "*trigger*" que manipula el proceso de consistencia.

En forma opcional puede expresarse una condición (**WHEN**), que en el caso de evaluarse como verdadera producirá que se ejecuten las acciones programadas en el cuerpo del *trigger* y en caso contrario dichas acciones no se harán.

SQL3 también permite indicar que la acción se realice:



- **Una vez para cada tupla afectada** (***row-level-trigger***): si hay N tuplas involucradas, la regla se ejecuta N veces. Se indica con la cláusula FOR EACH ROW. Como en la codificación de la acción puede precisarse hacer referencia a la tupla previa o posterior a la modificación, existe la posibilidad de referir a la tupla vieja y a la tupla nueva.

- **Una sola vez para todas ellas (*statement-level-trigger***): Se especifica con la cláusula FOR EACH STATEMENT o sin especificar nada porque es la opción *default*.

Los *triggers* son más costosos en ejecución que las restricciones implementadas en SQL2 (claves primarias, claves foráneas, valores default o NOT NULL) por lo tanto **no definir *triggers* para ofrecer lo mismo que puede hacerse con SQL2.**

Típicamente se los utiliza para auditoría, calcular atributos derivados, implementar reglas complejas y permitir actualización de vistas complejas.

Sintaxis Genérica de Creación

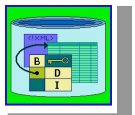
```
CREATE TRIGGER nombreTrigger
{ BEFORE | AFTER }
{ INSERT | DELETE | UPDATE [OF listaAtributos] } ON nombreTabla
[ REFERENCING OLD AS tuplasVieja NEW AS tuplaNueva
              OLD TABLE AS tablaVieja  NEW TABLE AS tablaNueva
]
[ FOR EACH  { ROW | STATEMENT }
  [ WHEN (condicion) ]
]
  unaAccion;
```

o bien

```
CREATE TRIGGER nombreTrigger
{ BEFORE | AFTER }
{ INSERT | DELETE | UPDATE [OF listaAtributos] } ON nombreTabla
[ REFERENCING OLD AS tuplasVieja NEW AS tuplaNueva
              OLD TABLE AS tablaVieja  NEW TABLE AS tablaNueva
]
[ FOR EACH  { ROW | STATEMENT } ]
  [ WHEN (condicion) ]
]
BEGIN ATOMIC
     variasAcciones;
END;
```

Sintaxis Genérica de Destrucción

```
DROP TRIGGER nombreTrigger;
```



Muy Importante

Las acciones o reglas programadas junto al evento **conforman una unidad que se ejecuta como un todo**. Por lo tanto, si alguna de ellas falla, se deshace TODA LA EJECUCION DEL *TRIGGER* (evento + acciones) asegurando que la base de datos queda en un estado consistente (igual al que se encontraba antes de lanzarse el *trigger*). Fallar no quiere decir que la condición WHEN se evaluó como FALSE, sino que se obtuvo una excepción (violación de PRIMARY KEY, etc)..

Para "forzar" a que falle un *trigger* se puede seguir alguna de las siguientes estrategias:

- Lanzar una excepción que no sea atrapada. De esta manera se deshace toda la ejecución del *trigger*, incluyendo el evento que tampoco se realizará.
- Invocar explícitamente un bloque PSM que lance una excepción que no sea atrapada. Si el bloque PSM falla, entonces falla todo el *TRIGGER,* con lo cual el evento tampoco se realizará.

Aclaración

Los DBMS comerciales como PostgreSQL, DB2, Oracle, extendieron el momento (TIMING) de realizar el evento respecto a las acciones asociadas, y además de BEFORE y AFTER ofrecen un **INSTEAD OF**.

Si se usa **INSTEAD OF** significa que cuando se produce el evento, el DBMS nunca lo va a ejecutar, porque lo reemplaza directamente por la acción definida, pero no se puede usar para especificar triggers sobre tablas (TABLES), sino sólo para triggers sobre vistas (VIEWS).

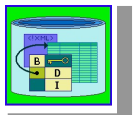
Los *triggers* proveen un esquema de autorización y seguridad avanzada: El creador (owner) del *trigger* debe tener privilegios sobre los objetos (tablas, etc) que refiere las acciones del *trigger*, pero los usuarios de dicho *trigger* (los que lo disparan) NO TIENEN por qué tener permisos sobre los objetos referidos en dichas acciones (sí sobre los eventos). Es decir, el DBMS valida los permisos sobre el creador del *trigger* y no sobre quien está ejecutando el *trigger*.

Por ejemplo: si Maria escribe un *trigger* tal que al modificar una tupla en la tabla AAA se encarga de insertar automáticamente una tupla en la tabla BBB, cuando el usuario Hernán quiera modificar la tabla AAA (tiene permisos sobre dicha tabla sobre la que se define el evento) automáticamente se insertará una tupla en BBB aunque él no tenga permisos sobre la tabla BBB. Así Hernán sólo precisa permisos de actualización sobre la tabla AAA, pero no precisa permisos de inserción sobre la tabla BBB. Obviamente Maria sí tendría permisos sobre las dos tablas.

Muy Importante

La idea del uso de *triggers* es que se automaticen las acciones sobre la base de datos, sin tener que dar permisos explícitos a otros usuarios: o sea Maria NO da permisos de inserción en la tabla BBB y así los usuarios sólo podrán insertar en BBB vía el *trigger* ideado para ello.

Para poder ejemplificar vamos a referirnos directamente a la sintaxis de los *triggers* en PostgreSQL, ya que tienen sus variantes.



3.2.1.1 Triggers en PostgreSQL

Sintaxis PostgreSQL Creación

```
CREATE TRIGGER nombreTrigger
{ BEFORE | AFTER | INSTEAD OF }
{ INSERT | DELETE | UPDATE [OF listaAtributos ] } ON nombreTabla
 [ FOR EACH  { ROW | STATEMENT }  -- for each statement es el default]
 [ WHEN (condicion) ]
 EXECUTE PROCEDURE fn(…);
```

Sintaxis PostgreSQL Destrucción

```
DROP TRIGGER nombreTrigger ON nombreTabla;
```

El trigger en PostgreSQL es muy distinto al ANSI (y a triggers de otros DBMS). Explicaremos cómo está definido:

Diferencias con el ANSI:

1. Las reglas no se colocan en el cuerpo del trigger. Lo único que puede hacerse es invocar un PSM "especial" que haga lo deseado (que contenga las reglas que se quieren implementar), al cual se lo llama "***trigger function***" y debe declarar un **RETURNS Trigger**. Esta "***trigger function***" es similar a una función PSM (vista en la clase 13), salvo que no pueden declarar parámetros formales y reciben **un entorno** desde el que lo invoca formado por variables que pueden ser usadas en su cuerpo:

   1.1 variable **old:** sólo puede ser usada en la combinación ***row-level-trigger*** y **UPDATE/DELETE.** Contiene el valor de la tupla en la BD en el momento en que se intercepta el evento. Tener en cuenta que la variable **old** no está inicializada (su intento de uso dará error en tiempo de ejecución) para los demás casos, es decir, ***row-level-trigger con INSERT*** o cualquier operación con ***statement-level-trigger***.

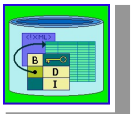
   1.2 variable **new:** sólo puede ser usada en la combinación ***row-level-trigger*** y **UPDATE/INSERT.** Contiene el valor de los valores de tupla proporcionados para persistir en la BD en el momento en que se intercepta el evento. Tener en cuenta que la variable **new** no está inicializada (su intento de uso dará error en tiempo de ejecución) para los demás casos, es decir, ***row-level-trigger con DELETE*** o cualquier operación con ***statement-level-trigger***.

   1.3 Variable **TG_NARGS y TG_ARGV[]** con la cantidad de parámetros actuales enviados y el arreglo con los mismos. Llegan en formato texto. Un intento fuera de los límites del arreglo devuelve null (Ej: TG_ARGS[-2] devuelve null).

La semántica de lo que debe devolver este "***trigger function***" debe corresponderse con lo siguiente:

1.4 Si es **row-level trigger**

1.4.1 con el timing BEFORE o INSTEAD OF, devolver alguno de estos tres valores según el efecto que se quiera producir:



a) **Terminación NORMAL y realización de todo: retornar algo diferente de NULL.** Es más, el valor retornado es el que usará para efectivizar el cambio en la BD si es un UPDATE o INSERT. Es la idea del ANSI. Si es un DELETE, el valor se ignora.

b) **Terminación ANORMAL y deshacer todo (ROLLBACK): lanzar excepción y no atraparla**. En este caso se deshace toda la transacción (las sentencias del ***trigger function*** invocado y la operación asociada al evento capturado). Es la idea del ANSI.

c) **Terminación NORMAL pero salteando las siguientes operaciones sobre la tupla (inclusive el evento en sí y siguientes triggers): retornar NULL**. Así, no sólo no se realizará la operación asociada al evento capturado (INSERT/UPDATE/DELETE) sobre la tupla sino que tampoco se van a seguir disparando otro triggers (si los hubiera). Como termina normalmente lo que se realizará será lo programado en el ***trigger function*** invocado. Este comportamiento está muy fuera del ANSI porque, claramente, hay cosas que se hacen y otras que no (no funciona como un todo). En el caso de INSTEAD OF sirve para informar que no produjo actualizaciones en tablas subyacentes, aunque realmente las haya hecho.

1.4.2 con timing AFTER, devolver alguno de estos dos valores según el efecto que se quiera producir:

a) Terminación NORMAL y realización de todo: retornar algún valor cualquiera, no importa cuál es (puede ser NULL o NOT NULL) porque el valor retornado será ignorado. El cambio en la BD será permanente. Es la idea del ANSI.

b) Terminación ANORMAL y deshacer todo (ROLLBACK): lanzar excepción y no atraparla. En este caso se deshace toda la transacción (las sentencias del ***trigger function*** invocado y la operación asociada al evento capturado). Es la idea del ANSI.

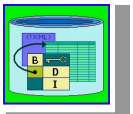
1.5 Si es **statement-level trigger (no puede usarse con INSTEAD OF)** no importa cuál sea el timing, devolver alguno de estos dos valores según el efecto que se quiera producir:

c) Terminación NORMAL y realización de todo: retornar algún valor cualquiera, no importa cuál es (puede ser NULL o NOT NULL) porque el valor retornado será ignorado. El cambio en la BD será permanente. Es la idea del ANSI.

d) Terminación ANORMAL y deshacer todo (ROLLBACK): lanzar excepción y no atraparla. En este caso se deshace toda la transacción (las sentencias del ***trigger function*** invocado y la operación asociada al evento capturado). Es la idea del ANSI.

Restricciones establecidas por PostgreSQL (para más información leer los manuales técnicos):

- **BEFORE** o **AFTER:** sólo puede ser usada con tablas (no con vistas).
- **INSTEAD OF:** sólo puede ser usada con vistas (no con tablas). No permite el uso de la cláusula WHEN. No puede usarse con trigger a nivel statement.
- Los **eventos** básicos son: INSERT, DELETE, UPDATE, pero agrega otros CREATE, ALTER, DROP, STARTUP, SHUTDOWN, LOGON y LOGOFF. Algunos de ellos precisan permisos



de administrador de la base de datos. Puede especificarse más de un INSERT o UPDATE en el mismo *trigger*, siempre que sea sobre la misma tabla o vista.

- En la cláusula **WHEN** no pueden colocarse subqueries ni se puede invocar un PSM.
- Un *trigger* no puede usar sentencias transaccionales, o sea no puede contener **COMMIT** ni **ROLLBACK** entre sus sentencias.
- No se pueden crear *triggers* recursivos (trigger cascading). Ejemplo: no se puede lanzar un UPDATE *tableA* dentro del cuerpo de un *trigger* cuyo evento sea UPDATE *tableA*.

Ejemplo 15

Sea las siguientes tablas, inicialmente vacías:

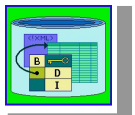
```
CREATE TABLE empleado
(
   legajo    INT NOT NULL PRIMARY KEY,
   nombre     VARCHAR(30),
   sueldo     DOUBLE PRECISION DEFAULT 14000 CHECK(sueldo >= 0)
);


CREATE TABLE auditoria
(
   fecha     TIMESTAMP NOT NULL,
   usuario   VARCHAR(30) NOT NULL
);
```

Caso A

Queremos interceptar el evento de "INSERT on EMPLEADO" y, sin que el usuario sepa, auditarlo. Lo hemos implementado así, haciendo uso de un **row-level trigger**.
(primero crear la función y luego el trigger):

```
CREATE TRIGGER BeforeInsertForEachRow
 BEFORE INSERT ON empleado
 FOR EACH ROW
 EXECUTE PROCEDURE SeeVbles();
```

```
CREATE OR REPLACE FUNCTION SeeVbles() RETURNS Trigger
AS $$
BEGIN

  INSERT INTO auditoria VALUES ( current_timestamp, user);

  raise notice 'new % % %', new.legajo, new.nombre, new.sueldo;
 --IMPOSIBLE IMPRIMIR OLD para INSERT:   raise notice 'old %', old;
  raise notice 'TG_NARGS %', TG_NARGS;
  raise notice 'TG_ARGV[0] %', TG_ARGV[0];

  RETURN new;  -- o sea,  hacer efectivo el evento
END;
$$ LANGUAGE plpgsql;
```



A.1) Explicar qué ocurriría si el usuario Salerno ejecutara:

INSERT INTO empleado (legajo) VALUES (10);

Rta:
La sentencia
INSERT INTO empleado (legajo) VALUES (10) está asociando una variable "new" que se corresponde con la tupla que se está proporcionando. En este caso, dicha tupla será **(10, null, 14000)** ya que no hay valor para el nombre a insertar y como no se proporcionó sueldo toma el valor indicado en la restricción default. O sea, new.legajo es 10, new.nombre es NULL y new.sueldo es 14000.
La variable old no está definida porque es un INSERT. Como no se están proporcionando parámetros actuales al invocar la función, TG_NARGS está en 0 y cualquier acceso al arreglo TG_ARG está en NULL.

El timing en el trigger es **BEFORE**, entonces primero se ejecuta el PSM y luego se realiza la operación asociada al evento (insert en tabla empleado).

El PSM, en su cuerpo, inserta una tupla en la tabla AUDITORIA. Luego imprime el valor de las variables explicadas previamente. Finalmente, y para que la operación se haga efectiva (tanto la inserción en la tabla AUDITORIA como en la tabla EMPLEADO) devuelve algo diferente de NULL que es lo que desea almacenar en la tabla EMPLEADO (1.4.1 A).

Lo que se obtiene es una **ejecución exitosa**, es decir las tablas quedan así:

EMPLEADO		
Legajo	Nombre	sueldo
10	NULL	14000

AUDITORIA	
Fecha	Usuario
2016-05-23 13:08:33	Salerno

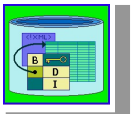
Y obtiene la siguiente información:

```
NOTICE:  new 10 <NULL>  14000
NOTICE:  TG_NARGS 0
NOTICE:  TG_ARGV[0] <NULL>
```

A.2) Explicar qué ocurriría si el usuario Salerno, con las tuplas previamente insertadas, ejecutara otra vez:

INSERT INTO empleado (legajo) VALUES (10);

Rta:
Nuevamente, primero se ejecuta el PSM con los valores explicados previamente. Es decir, se inserta en auditoría y se imprime lo mismo que antes. Sin embargo, aunque se realiza el return de la variable diferente de NULL para hacer efectiva las operaciones en la BD se obtiene una EXCEPCION por PK violation. Esta excepción no es manejada en el PSM y finaliza ANORMALMENTE la ejecución del PSM (1.4.1. B). Se hace ROLLBACK de toda la transacción.



Es decir, las tablas quedan exactamente iguales.
La ejecución ABORTA y devuelve una excepción SQLSTATE '23505' por llave duplicada.

Las tablas quedan intactas y se obtiene la siguiente información:

```
NOTICE:  new 10 <NULL>  14000
NOTICE:  TG_NARGS 0
NOTICE:  TG_ARGV[0] <NULL>
```

Notar que si queremos ABORTAR la ejecución del trigger (tanto la acción asociada al evento como lo del PSM), porque no valida alguna regla de negocios, justamente lo que debemos hacer es LANZAR una excepción de usuario como ser observa en el próximo caso.

CASO B)

Queremos interceptar el evento de "INSERT on EMPLEADO" y, sin que el usuario sepa, auditarlo. Pero, además, queremos implementar las siguientes 2 regla de negocios:

- si el usuario que está realizando la operación se está insertando "a él mismo" queremos rechazar la operación (el INSERT en EMPLEADO), salvo que se auto-agregue con sueldo 0.
- Se quiere guardar en mayúsculas el nombre en la tabla.

Lo hemos implementado haciendo uso de un **row-level trigger**. El PSM cambia así:

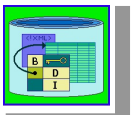
```
CREATE OR REPLACE FUNCTION SeeVbles() RETURNS Trigger
AS $$
BEGIN

  INSERT INTO auditoria VALUES ( current_timestamp, user);

  raise notice 'new % ', new;

  IF new.nombre = user AND new.sueldo > 0 THEN
      Raise exception 'NO PODES AUTO INSERTARTE' USING ERRCODE = 'PP111';
  ELSE
      new.nombre:= UPPER(new.nombre);  -- modificacion segun lo pedido
   END IF;

   raise notice 'new % ', new;
   RETURN new;  -- o sea,  hacer efectivo el evento

END;
$$ LANGUAGE plpgsql;
```

B.1) Explicar qué ocurriría si el usuario Salerno ejecutara:

INSERT INTO empleado (legajo, nombre, sueldo) VALUES (20, 'jorge', 10000);

Rta:

La ejecución es EXITOSA. Es decir, primero ejecuta el PSM, inserta en la tabla auditoría y como el nombre proporcionado (jorge) no coincide con Salerno, se hace efectiva toda la transacción, quedando:



EMPLEADO		
Legajo	Nombre	sueldo
10	NULL	14000
20	JORGE	10000

AUDITORIA	
Fecha	Usuario
2016-05-23 13:08:33	Salerno
2016-05-23 13:27:40	Salerno

Y obtiene la siguiente información:

```
NOTICE:  new (20, jorge, 10000)
NOTICE:  new (20, JORGE, 10000)
```

B.2) Explicar qué ocurriría si el usuario Salerno ejecutara:

INSERT INTO empleado (legajo, nombre) VALUES (30, 'Salerno');

Rta
Termina ANORMALMENTE la ejecución.
El PSM comienza insertando en la tabla AUDITORIA y se imprime los valores de las variables. Pero como new.nombre = user con sueldo 14000 (default) , entonces se lanza la excepción que finaliza anormalmente el PSM. Es decir, se hace rollback de toda la transacción. No hay nuevas tuplas en ninguna de las tablas.

La ejecución devuelve una excepción SQLSTATE 'PP111' No Podes auto insertarte.

Las tablas quedan intactas y se obtiene la siguiente información:

```
NOTICE:  new (30, Salerno ,14000)
```

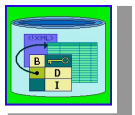
CASO C)

Ya discutimos qué ocurre si de devuelve un valor diferente de NULL y si se lanza excepción desde un trigger PSM.
Ahora analizaremos que pasa si se devuelve NULL:

Lo hemos implementado así:

```
CREATE TRIGGER BeforeInsertForEachRow
 BEFORE INSERT ON empleado
 FOR EACH ROW
 EXECUTE PROCEDURE SeeVbles();
```

```
CREATE OR REPLACE FUNCTION SeeVbles() RETURNS Trigger
AS $$
BEGIN

  INSERT INTO auditoria VALUES ( current_timestamp, user);

  raise notice 'new % ', new;
```



```
  IF new.nombre = user AND new.sueldo > 0 THEN
      Raise exception 'NO PODES AUTO INSERTARTE' USING ERRCODE = 'PP111';
  ELSE
      new.nombre:= UPPER(new.nombre);  -- modificacion segun lo pedido
   END IF;

   raise notice 'new % ', new;
   RETURN NULL;  -- o sea,  no hacer efectivo el evento

END;
$$ LANGUAGE plpgsql;
```

C.1)

Explicar qué ocurriría si el usuario Salerno ejecutara:

INSERT INTO empleado (legajo, nombre) VALUES (30, 'Salerno');

Rta:

Termina ANORMALMENTE la ejecución. Mismo efecto que el anterior porque abortó la operación.
El PSM comienza insertando en la tabla AUDITORIA y se imprime los valores de las variables. Pero como new.nombre = user con sueldo 14000 , entonces se lanza la excepción que finaliza anormalmente el PSM. Es decir, se hace rollback de toda la transacción. No hay nuevas tuplas en ninguna de las tablas.

La ejecución devuelve una excepción SQLSTATE 'PP111' No Podes auto insertarte.

Las tablas quedan intactas y se obtiene la siguiente información:

```
NOTICE:  new (30, Salerno ,14000)
```

C.2)

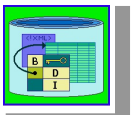
Explicar qué ocurriría si el usuario Salerno ejecutara:

INSERT INTO empleado (legajo, nombre, sueldo) VALUES (30, 'Salerno', 0);

La ejecución es exitosa. Es decir, primero ejecuta el PSM, inserta en la tabla auditoría y como el nombre proporcionado es Salerno pero el sueldo es 0, se hace efectiva lo implementado allí. Sin embargo, como se devuelve NULL, se "descarta" la operación que lanzó el trigger, es decir el INSERT en EMPLADO.

Las tablas quedan así:

EMPLEADO		
Legajo	Nombre	sueldo
10	NULL	14000
20	JORGE	10000



AUDITORIA	
Fecha	Usuario
2016-05-23 13:08:33	Salerno
2016-05-23 13:27:40	Salerno
2016-05-28 17:17:15	Salerno

Y obtiene la siguiente información:

```
NOTICE:  new (30,salerno,0)
NOTICE:  new (30,SALERNO,0)
```

CASO D)

¿En qué cambia la idea si se trata de un statement-level trigger?
Antes se seguir destruimos el otro trigger!

```
DROP TRIGGER BeforeInsertForEachRow ON EMPLEADO;
```

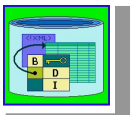
Al Nuevo, lo hemos implementado así:

```
CREATE TRIGGER BeforeInsertForEachStatement
 BEFORE INSERT ON empleado
 FOR EACH STATEMENT
 EXECUTE PROCEDURE SeeVbles();
```

```
CREATE OR REPLACE FUNCTION SeeVbles() RETURNS Trigger
AS $$
BEGIN

  INSERT INTO auditoria VALUES ( current_timestamp, user);

  raise notice 'new % ', new;

  IF new.nombre = user AND new.sueldo > 0 THEN
      Raise exception 'NO PODES AUTO INSERTARTE' USING ERRCODE = 'PP111';
  ELSE
      new.nombre:= UPPER(new.nombre);  -- modificacion segun lo pedido
   END IF;

   raise notice 'new % ', new;
   RETURN NULL;

END;
$$ LANGUAGE plpgsql;
```

Explicar qué ocurriría si el usuario Salerno ejecutara:

INSERT INTO empleado (legajo) VALUES (40);



Rta
Se obtiene Excepcion en tiempo de EJECUCION porque la variable "new.nombre" no tiene valor asignado.
El problema es que los triggers a **nivel statement NO PUEDEN referir a tuplas individuales** (sólo sirven para cosas globales).

CUIDADO CON ESO!!

CASO E)

Aquí mostramos la idea de un statement-level trigger. Usar información global.

Lo hemos implementado así:

```
CREATE TRIGGER BeforeInsertForEachStatement
 BEFORE INSERT ON empleado
 FOR EACH STATEMENT
 EXECUTE PROCEDURE SeeVbles();
```

```
create or replace FUNCTION SeeVbles() RETURNS Trigger
AS $$
BEGIN

  INSERT INTO auditoria VALUES ( current_timestamp, user);

  RETURN new;

END;
$$ LANGUAGE plpgsql;
```

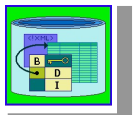
Explicar qué ocurriría si el usuario Salerno ejecutara:

```
INSERT INTO empleado (legajo)
    SELECT legajo + 100  FROM EMPLEADO;
```

Rta:
Se intercepta el evento asociado a esa única sentencia SQL, se ejecuta el PSM una sola vez (la inserción en auditoria una sola vez), pero las inserciones en la tabla empleado fueron múltiples.

Es decir, la operación resulta EXITOSA. Las tablas quedan así:

EMPLEADO		
Legajo	Nombre	sueldo
10	NULL	14000
20	JORGE	10000
110	NULL	14000
120	NULL	14000



AUDITORIA	
Fecha	Usuario
2016-05-23 13:08:33	Salerno
2016-05-23 13:27:40	Salerno
2016-05-28 17:17:15	Salerno
2016-05-28 17:36:09	**Salerno**

1 sola tupla en la tabla auditoría, aunque hubo inserción mútiple (2) en la tabla empleado

Notar que si esta inserción MASIVA se hubiera realizado con un trigger a nivel row, se hubieran insertado en la tabla AUDITORIA 2 tuplas en vez de 1, es decir una tupla por cada inserción en la tabla EMPLEADO realizada.

CASO F)
¿Si el statement-level trigger devuelve NULL qué ocurre?

Lo hemos implementado así:

```
CREATE TRIGGER BeforeInsertForEachRow
 BEFORE INSERT ON empleado
 FOR EACH STATEMENT
 EXECUTE PROCEDURE SeeVbles();
```

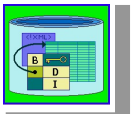
```
create or replace FUNCTION SeeVbles() RETURNS Trigger
AS $$
BEGIN

  INSERT INTO auditoria VALUES ( current_timestamp, user);

  RETURN null;

END;
$$ LANGUAGE plpgsql;
```

Explicar qué ocurriría si el usuario Salerno ejecutara:

```
INSERT INTO empleado (legajo)
    SELECT legajo + 500  FROM EMPLEADO;
```

Rta
En un statement-level trigger devolver NULL o diferente de NULL da igual porque se ignora el valor retornado.

Es decir, la operación resulta EXITOSA. Se producen inserciones en AMBAS tablas. Las tablas quedan así:



EMPLEADO		
Legajo	Nombre	sueldo
10	NULL	14000
20	JORGE	10000
110	NULL	14000
120	NULL	14000
510	**NULL**	**14000**
520	**NULL**	**14000**
610	**NULL**	**14000**
620	**NULL**	**14000**

AUDITORIA	
Fecha	Usuario
2016-05-23 13:08:33	Salerno
2016-05-23 13:27:40	Salerno
2016-05-28 17:17:15	Salerno
2016-05-28 17:36:09	Salerno
2016-05-28 17:39:01	**Salerno**

1 sola tupla en la tabla auditoría, aunque hubo inserción mútiple (4) en la tabla empleado

CASO G)

Si se produjera excepción (del sistema o de usuario) en el caso de statement-level trigger se hace ROLLBACK.

Ejemplo 16

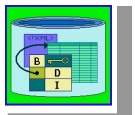
Seguimos con las tablas tal cual quedaron en el paso anterior.
Ahora vamos a analizar el caso de la combinación **UPDATE** con **BEFORE**.

CASO A)

Queremos interceptar el evento de "UPDATE on empleado" y sin que el usuario lo sepa auditarlo.
Lo hemos implementado así, haciendo uso de un **row-level trigger**.

```
CREATE TRIGGER BeforeUpdateForEachRow
 BEFORE UPDATE ON empleado
 FOR EACH ROW
 EXECUTE PROCEDURE SeeVbles();
```

```
create or replace FUNCTION SeeVbles() RETURNS Trigger
AS $$
BEGIN

  INSERT INTO auditoria VALUES ( current_timestamp, user);

  raise notice 'new %', new;
```



```
  raise notice 'old %', old;
  raise notice 'TG_NARGS %', TG_NARGS;
  raise notice 'TG_ARGV[0] %', TG_ARGV[0];

  RETURN new;  -- o sea,  hacer efectivo el evento
END;
$$ LANGUAGE plpgsql;
```

A.1) Explicar qué ocurriría si el usuario Salerno ejecutara:

UPDATE empleado SET nombre= 'Jorge' WHERE legajo = 10;

Rta:
La sentencia
UPDATE empleado SET nombre= 'Jorge ' WHERE legajo = 10 asocia una variable "old" y una "new" que se corresponde con la tupla previa y posterior que se está proporcionando. En este caso, dicha tupla será la old (10, null, 14000) y la new (10, 'Jorge', 14000).

Como no se están proporcionando parámetros actuales al invocar la función, TG_NARGS está en 0 y cualquier acceso al arreglo TG_ARG está en NULL.

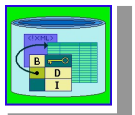
El timing en el trigger es **BEFORE**, entonces primero se ejecuta el PSM y luego se actualiza la operación indicada en el evento.

El PSM, en su cuerpo, inserta una tupla en la tabla AUDITORIA. Luego imprime el valor de las variables explicadas previamente. Finalmente, y para que la operación se haga efectiva (tanto la inserción en la tabla AUDITORIA como la actualización en la tabla EMPLEADO) devuelve algo diferente de NULL que es lo que desea almacenar en la tabla EMPLEADO. Si quisiera por ejemplo viendo que es NULL el valor del nombre cambiarlo por otra cosa, antes del return debería realizar new.nombre='xxx' para que quede dicho valor efectivo.

Lo que se obtiene es una ejecución EXITOSA, es decir las tablas quedan así:

EMPLEADO		
Legajo	Nombre	sueldo
10	**JORGE**	**14000**
20	JORGE	10000
110	NULL	14000
120	NULL	14000
510	NULL	14000
520	NULL	14000
610	NULL	14000
620	NULL	14000

AUDITORIA	
Fecha	Usuario
2016-05-23 13:08:33	Salerno
2016-05-23 13:27:40	Salerno
2016-05-28 17:17:15	Salerno
2016-05-28 17:36:09	Salerno
2016-05-28 17:39:01	Salerno
2016-05-28 17:44:51	**Salerno**



1 tuplas en la tabla auditoría, 1 tupla actualizada en la tabla empleado

Y obtiene la siguiente información:

```
NOTICE:  new  (10, Jorge, 14000)
NOTICE:  old  (10, <NULL>, 14000)
NOTICE:  TG_NARGS 0
NOTICE:  TG_ARGV[0] <NULL>
```

A.2) Explicar qué ocurriría si el usuario Salerno, con las tuplas previamente insertadas, ejecutara otra vez:

UPDATE empleado SET nombre= 'Jorge' WHERE legajo = -3;

Rta:
Ningún cambio en ninguna TABLA porque no llegó a interceptarse el evento. Postgres detecta que ese update es imposible y ni siquiera transfiere el control al PSM

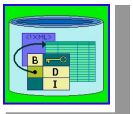
Ejemplo 17

Seguimos con las tablas tal cual quedaron en el paso anterior.
Ahora vamos a analizar el caso de la combinación **DELETE** con **BEFORE**.

CASO A)
Queremos interceptar el evento de "DELETE on empleado" y sin que el usuario lo sepa, auditarlo.
Lo hemos implementado así, haciendo uso de un **row-level trigger**.

```
CREATE TRIGGER BeforeDeleteForEachRow
 BEFORE DELETE ON empleado
 FOR EACH ROW
 EXECUTE PROCEDURE SeeVbles();
```

```
create or replace FUNCTION SeeVbles() RETURNS Trigger
AS $$
BEGIN

  INSERT INTO auditoria VALUES ( current_timestamp, user);

  raise notice 'old %', old;
  raise notice 'TG_NARGS %', TG_NARGS;
  raise notice 'TG_ARGV[0] %', TG_ARGV[0];

  RETURN new;  -- o sea,  hacer efectivo el evento
END;
$$ LANGUAGE plpgsql;
```



A.1) Explicar qué ocurriría si el usuario Salerno ejecutara:

DELETE FROM empleado WHERE legajo = 110;

Rta:
La sentencia DELETE empleado WHERE legajo = 110 asocia una variable "old" que se corresponde con la tupla previa. En este caso, dicha tupla será la old (110, null, 14000).
Como no se están proporcionando parámetros actuales al invocar la función, TG_NARGS está en 0 y cualquier acceso al arreglo TG_ARG está en NULL.

El timing en el trigger es **BEFORE**, entonces primero se ejecuta el PSM y luego se actualiza la operación indicada en el evento.
La operación resulta EXITOSA y las tablas quedan así.

EMPLEADO		
Legajo	Nombre	sueldo
10	JORGE	14000
20	JORGE	10000
110	NULL	14000
120	NULL	14000
510	NULL	14000
520	NULL	14000
610	NULL	14000
620	NULL	14000

AUDITORIA	
Fecha	Usuario
2016-05-23 13:08:33	Salerno
2016-05-23 13:27:40	Salerno
2016-05-28 17:17:15	Salerno
2016-05-28 17:36:09	Salerno
2016-05-28 17:39:01	Salerno
2016-05-28 17:44:51	Salerno
2016-05-28 17:51:44	Salerno

Es decir, se inserta una tabla en AUDITORIA pero NO SE BORRA la tupla en EMPLEADO. ¿Por qué? Porque se retornó NEW y ese valor estaba en null! Cuidado con eso. No trabajar con NEW si se está en un DELETE.

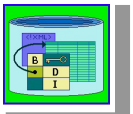
Para que se hiciera efectivo se debe retornar un valor diferente a NULL (no importa cual sea).

Ejemplo 18

Seguimos con las tablas tal cual quedaron en el paso anterior.
Borrar antes de seguir los trigger anteriores.

```
DROP TRIGGER beforedeleteforeachrow ON EMPLEADO;
DROP TRIGGER beforeinsertforeachstatement  ON EMPLEADO;
DROP TRIGGER beforeupdateforeachrow ON EMPLEADO;
```

Ahora vamos a analizar el caso de la combinación **AFTER** con **UPDATE**.



CASO A)
Queremos interceptar el evento de "UPDATE on empleado" y sin que el usuario lo sepa, auditarlo. Lo hemos implementado así, haciendo uso de un **row-level trigger**.

```
CREATE TRIGGER AfterUpdateForEachRow
 AFTER UPDATE ON empleado
 FOR EACH ROW
 EXECUTE PROCEDURE SeeVbles();
```

```
create or replace FUNCTION SeeVbles() RETURNS Trigger
AS $$
BEGIN

  INSERT INTO auditoria VALUES ( current_timestamp, user);

  raise notice 'old %', old;
  raise notice 'new %', new;
  raise notice 'TG_NARGS %', TG_NARGS;
  raise notice 'TG_ARGV[0] %', TG_ARGV[0];

  RETURN  NULL;  -- da lo mismo devolver NULL o NOT NULL porque es AFTER
END;
$$ LANGUAGE plpgsql;
```

A.1) Explicar qué ocurriría si el usuario Salerno ejecutara:

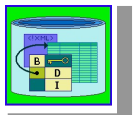
UPDATE empleado SET nombre = 'Maria' WHERE legajo = 110;

Rta:
La sentencia UPDATE empleado SET nombre = 'Maria' WHERE legajo = 110; asocia una variable "old" y "new" que se corresponde con la tupla previa y la nueva. En este caso, dicha tupla será la old (110, null, 14000) y (110, 'Maria', 14000).
Como no se están proporcionando parámetros actuales al invocar la función, TG_NARGS está en 0 y cualquier acceso al arreglo TG_ARG está en NULL.

El timing en el trigger es **AFTER**, entonces primero se actualiza la tupla EMPLEADO y luego se ejecuta el PSM. Por ese motivo, el valor retornado se ignora (la tupla ya se modificó). En este caso, devolver new o NULL era lo mismo, porque se ignora la devolución.
La único que se podría hacer es deshacer la transacción completa (hacer rollback) lanzando excepción.

Lo que se obtiene es una ejecución exitosa, es decir las tablas quedan así:

EMPLEADO		
Legajo	Nombre	sueldo
10	JORGE	14000
20	JORGE	10000
110	Maria	14000
120	NULL	14000
510	NULL	14000
520	NULL	14000
610	NULL	14000
620	NULL	14000



AUDITORIA	
Fecha	Usuario
2016-05-23 13:08:33	Salerno
2016-05-23 13:27:40	Salerno
2016-05-28 17:17:15	Salerno
2016-05-28 17:36:09	Salerno
2016-05-28 17:39:01	Salerno
2016-05-28 17:44:51	Salerno
2016-05-28 17:51:44	Salerno

Y obtiene la siguiente información:

```
NOTICE:  old (110, NULL,14000)
NOTICE:  new (110,Maria,14000)
NOTICE:  TG_NARGS 0
NOTICE:  TG_ARGV[0] <NULL>
```

A.2)
¿ Se podría hacer que Maria pase a mayúsculas en la inserción con esta forma de implementarlo?

Rta
NO, para eso está el BEFORE.

Ejemplo 19

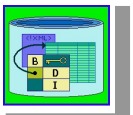
¿Qué sucede si interceptamos un evento, por ejemplo, de BORRADO y en el cuerpo del Trigger PSM hacemos una operación sobre la misma tabla?

Antes de esto, destruir todos los trigger previamente creados y comenzaremos con las tablas con 0 tuplas.

Supongamos que tenemos una regla diferente de negocios para la tabla EMPLEADO:
Queremos validar que si se inserta un empleado su sueldo no debe superar a los que existen hasta ahora (si los hubiera), caso contrario rechazarlo.

```
CREATE TRIGGER BeforeInsertForEachRow
 BEFORE INSERT ON empleado
 FOR EACH ROW
 EXECUTE PROCEDURE Checking();
```

```
create or replace FUNCTION Checking() RETURNS Trigger
AS $$
DECLARE
aSueldo FLOAT;
BEGIN

  SELECT MAX(sueldo) INTO aSueldo FROM empleado;
  IF (new.sueldo > aSueldo) THEN
      Raise exception 'SUELDO MUY ALTO' USING ERRCODE = 'PP111';
 END IF;

 Return NEW;
END;
$$ LANGUAGE plpgsql;
```



A) Qué pasa si se invoca con:

INSERT INTO empleado VALUES(10, 'Andrea', 10000);

Rta
La operación finaliza exitosamente. El sueldo 10000 es el primero.

EMPLEADO		
Legajo	Nombre	sueldo
10	Andrea	10000

B) Qué pasa si se invoca con:

INSERT INTO empleado VALUES(20, 'Lola', 2000);

Rta
La operación finaliza exitosamente. El sueldo 2000 no es mayor que el que existe

EMPLEADO		
Legajo	Nombre	sueldo
10	Andrea	10000
20	Lola	2000

C) Qué pasa si se invoca con:

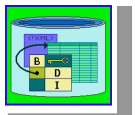
INSERT INTO empleado VALUES(30, 'Pedro', 15000);

Rta
La operación termina ANORMALMENTE porque 15000 es mayor que el máximo registrado y lanza la excepción de usuario. La tabla queda intacta.

Ejemplo 20

¿Qué pasa si un trigger lanza una operación con el mismo evento sobre una tabla que está siendo modificada?

```
create or replace FUNCTION Checking() RETURNS Trigger
AS $$
DECLARE
aSueldo FLOAT;
BEGIN
   insert into empleado values (new.legajo, new.nombre, new.sueldo);
  Return NEW;

END;
$$ LANGUAGE plpgsql;
```



Si se invoca con:

insert into empleado values (40, 'Sorpresa', 10)

el evento transfiere el control al Trigger PSM, pero este lanza un INSERT en empleado. O sea, un nuevo evento se produce que se captura, y así hasta que se genera un stack overflow.

Cuidado!!!
No colocar sentencias en el trigger PSM que generen eventos sobre los que se tengan triggers que puedan producir ciclos (conocido como Trigger en Cascada). Este tipo de recursión puede generarse por error inclusive indirectamente.

Ejemplo 21

Se tiene la base de datos con los estudiantes de la facultad.

Cuando los alumnos se inscriben, al comienzo de cuatrimestre, se los inserta en la tabla INSCRIPTO, registrando *legajo*, *fecha* (la de inscripción) y *código de la materia correspondiente*. Obviamente *nota* se encuentra en NULL.

Al final del cuatrimestre, se actualizan las tuplas con el acta de firma de TP proporcionada por el docente: *la fecha* que guardaba la inscripción se transforma en fecha de firma de TP, y la *nota* se transforma en nota de cursada. Obviamente este proceso se realiza sólo para aquellas tuplas cuya *nota* es NULL, porque el resto de las tuplas corresponden a cuatrimestres anteriores.

En ese momento, se desea que para aquellos alumnos que acaban de aprobar los TP se los inserte ***automáticamente*** en la tabla ***examen*** de donde se emitirá el listado de alumnos habilitados, el día del examen final.

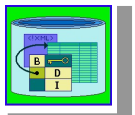
Supongamos que tenemos la siguiente tabla:

INSCRIPTO			
legajo	codMateria	Nota	Fecha
32198	1-A1	2	12/07/2008
32198	1-A1	NULL	07/07/2009
32198	23-B2	NULL	07/07/2009
28134	1-A1	NULL	07/07/2009
16345	23-B2	6	04/07/2006
16345	1-A1	NULL	07/07/2009
...	...	...	...

EXAMEN			
legajo	codMateria	Nota	fecha
16345	23-B2	2	23/11/2006
16345	23-B2	7	05/02/2007
...	...	...	...

La información en INSCRIPTO indica que el alumno con legajo 32198 se inscribió en la materia 1-A1 en 2008 y no aprobó los TP, se volvió a anotar en el 2009 y está actualmente cursando dicha materia (nota NULL). Además, está cursando la materia 23-B2 (nota NULL).

El alumno con legajo 28134 está cursando la materia 1-A1 (nota NULL).



El alumno con legajo 16345 cursó durante 2006 la materia con código 23-B2 y aprobó los TP con 6, y se encuentra actualmente inscripto en 1-A1 (nota NULL).

Lo que se nos pide realizar es un ***trigger*** que automáticamente maneje la anotación de alumnos en el acta de final cuando firmen los TP (nota >= 4). Para diseñar esto tenemos que pensar:

- Qué evento debería disparar las acciones: obviamente es la actualización de la tabla INSCRIPTO (**UPDATE**) sobre el campo NOTA.
- Qué acciones habría que seguir en caso de que llegue dicho UPDATE: primero verificar que la nota a modificar sea >=4 y sólo en ese caso habría que insertar una tupla en la tabla EXAMEN. Independiente de que la nota sea o no mayor o igual que 4, el evento de UPDATE se producirá.

Si el profesor de la materia **1-A1** presentara el siguiente listado de alumnos que cursaron su materia:

legajo	*nota*
32198	**3**
28134	**5**
16345	**9**

El operador de la base de datos realizaría la siguientes actualizaciones sobre la tabla INSCRIPTO:

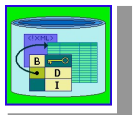
```
UPDATE INSCRIPTO
SET nota= 3, fecha=current_date
WHERE legajo= 32198 AND nota IS NULL AND codmateria= '1-A1';


UPDATE INSCRIPTO
SET nota= 5, fecha=current_date
WHERE legajo= 28134 AND nota IS NULL AND codmateria= '1-A1';

UPDATE INSCRIPTO
SET nota= 9, fecha=current_date
WHERE legajo= 16345 AND nota IS NULL AND codmateria= '1-A1';
```

Supongamos que **current_date** es **28/05/2016**

Obviamente desearíamos que se produjeran las tres actualizaciones y que automáticamente se insertaran las dos tuplas correspondientes a los alumnos que aprobaron los TP.

Muy Importante

Notar que, aunque en este caso, cada UPDATE sólo actualiza una sola tupla, pues se proporciona legajo+codMateria+NotaNull, como se desea referir en el trigger a las variables de correlación OLD/NEW, debe **necesariamente** usarse la cláusula FOR EACH ROW.



Vamos a discutir distintas implementaciones para analizar cuál es la diferencia entre el uso de OLD y NEW y BEFORE/AFTER

❖ **BEFORE / WHEN(new) / ACCION(new)**

Sin el FOR EACH ROW no puedo usar ***new/old*** y en este ejercicio lo necesito.

```
CREATE TRIGGER BeforeInsertForEachRow
 BEFORE UPDATE OF FECHA, NOTA ON INSCRIPTO
 FOR EACH ROW
WHEN ( new.nota >= 4)
 EXECUTE PROCEDURE ActaFinal();
```

```
CREATE OR REPLACE FUNCTION ActaFinal() RETURNS Trigger
AS $$
BEGIN

  INSERT INTO examen (legajo, codMateria, fecha) VALUES ( new.legajo,
new.codMateria, new.fecha);

   RETURN new;  -- hacer efectivo el evento

END;
$$ LANGUAGE plpgsql;
```

Cuando se ejecute un *UPDATE* el mismo es interceptado por el *trigger*, el cual antes de ejecutar el evento y para cada tupla actualizada (*FOR EACH ROW*) evalúa la condición del *WHEN* y si obtiene verdadero, ejecuta el BEGIN/END. Luego se procede a realizar el evento *UPDATE*, independientemente de que se haya verificado o no la condición *WHEN* como verdadera. Si en alguna de estas fases se produjera un error el DBMS deshace toda le ejecución de la sentencia (eventos+acción).

Para la primera actualización:

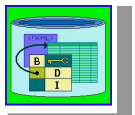
```
UPDATE INSCRIPTO
SET nota= 3, fecha=current_date
WHERE legajo= 32198 AND nota IS NULL AND codmateria= '1-A1';
```

se cumple que **new.nota < 4**, por lo tanto se pasa directamente al evento *UPDATE* y finaliza la ejecución de la sentencia.

Para la segunda actualización:

```
UPDATE INSCRIPTO
SET nota= 5, fecha=current_date
WHERE legajo= 28134 AND nota IS NULL AND codmateria= '1-A1';
```

se cumple que **new.nota >= 4**, por lo tanto primero se ejecuta el trigger PSM y como esa inserción no produce error, se pasa al evento *UPDATE* y finaliza la ejecución de la sentencia.



Para la tercera actualización:

```
UPDATE INSCRIPTO
SET nota= 9, fecha=current_date
WHERE legajo= 16345 AND nota IS NULL AND codmateria= '1-A1';
```

se cumple que **new.nota >= 4**, por lo tanto primero se ejecuta el trigger PSM y como esa inserción no produce error, se pasa al evento *UPDATE* y finaliza la ejecución de la sentencia.

Finalmente termina la transacción.

O sea que finalmente las tablas quedan con esta información:

INSCRIPTO			
legajo	codMateria	Nota	Fecha
32198	1-A1	2	12/07/2008
32198	1-A1	**3**	**28/05/2016**
32198	23-B2	NULL	07/07/2009
28134	1-A1	5	**28/05/2016**
16345	23-B2	6	04/07/2006
16345	1-A1	9	**28/05/2016**
...	...	...	...

EXAMEN			
legajo	codMateria	Nota	fecha
16345	23-B2	2	23/11/2006
16345	23-B2	7	05/02/2007
...	...	...	...
28134	1-A1	NULL	**28/05/2016**
16345	1-A1	NULL	**28/05/2016**

¿Es esta implementación del *trigger* correcta?
Sí, es lo solicitado por la lógica de negocios.

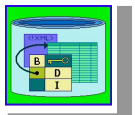
¿Qué hubiera pasado si en vez de usar *BEFORE* se colocara *AFTER* como *timing* del evento? Hubiera funcionado igual, porque primero se ejecutaría el evento *UPDATE* y luego se procedería a evaluar la cláusula *WHEN* por cada tupla y si resulta verdadera se ejecutaría el bloque BEGIN/END. Obviamente si en alguna de las etapas se produjera algún error, se desharía toda la sentencia (eventos+acción).

❖ BEFORE / WHEN(old) / ACCION(old)

¿Qué hubiera pasado si el *trigger* se implementa así?

```
CREATE TRIGGER BeforeInsertForEachRow
 BEFORE UPDATE OF FECHA, NOTA ON INSCRIPTO
 FOR EACH ROW
WHEN ( old.nota >= 4)
 EXECUTE PROCEDURE ActaFinal();
```

```
CREATE OR REPLACE FUNCTION ActaFinal() RETURNS Trigger
```



```
AS $$
BEGIN

  INSERT INTO examen (legajo, codMateria, fecha) VALUES ( old.legajo,
old.codMateria, old.fecha);

   RETURN new;  -- hacer efectivo el evento

END;
$$ LANGUAGE plpgsql;
```

Cuando se ejecute un *UPDATE* el mismo es interceptado por el *trigger*, el cual antes de ejecutar el evento y para cada tupla actualizada (*FOR EACH ROW*) evalúa la condición del *WHEN* y si obtiene verdadero, ejecuta el BEGIN/END. Luego se procede a realizar el evento *UPDATE*, independientemente de que se haya verificado o no la condición *WHEN* como verdadera. Si en alguna de estas fases se produjera un error el DBMS deshace toda le ejecución de la sentencia (eventos+acción).

El cliente ejecuta 3 *updates*, y para cada uno **old.nota es NULL,** por lo tanto al evaluar **NULL>= 4** se obtiene FALSE. La acción del *trigger* no se llega a realizar, pero se ejecuta directamente el evento *UPDATE*. Finalmente, las tablas quedan con esta información:

INSCRIPTO			
legajo	codMateria	Nota	Fecha
32198	1-A1	2	12/07/2008
32198	1-A1	**3**	**28/05/2016**
32198	23-B2	NULL	07/07/2009
28134	1-A1	5	**28/05/2016**
16345	23-B2	6	04/07/2006
16345	1-A1	9	**28/05/2016**
...	...	...	...

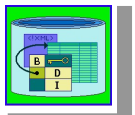
EXAMEN			
legajo	codMateria	Nota	fecha
16345	23-B2	2	23/11/2006
16345	23-B2	7	05/02/2007
...	...	...	...

¿Es esta implementación del *trigger* correcta? NO

Muy Importante

Obviamente esto no es lo que buscábamos, o sea este *trigger* fue **MAL PROGRAMADO!!!**

Aunque se usara AFTER se cometería el mismo error.



Ejemplo 22

En el ejemplo anterior, no hubo diferencia entre usar AFTER o BEFORE.

¿ Existen casos donde usar uno u otro no resulte indiferente?

Sí.

Supongamos que el Trigger PSM borra una tupla en la misma tabla donde el evento interceptado es UPDATE.

Tenemos las siguientes tuplas:

INSCRIPTO			
legajo	codMateria	Nota	Fecha
32198	1-A1	2	12/07/2008
32198	1-A1	**3**	**28/05/2016**
32198	23-B2	NULL	07/07/2009
28134	1-A1	5	**28/05/2016**
16345	23-B2	6	04/07/2006
16345	1-A1	9	**28/05/2016**

Si tuviéramos

```
CREATE OR REPLACE FUNCTION Sorpresa() RETURNS Trigger
AS $$
BEGIN

  DELETE FROM INSCRIPTO WHERE nota < 4;

   RETURN old;  -- hacer efectivo el evento algo que no sea null

END;
$$ LANGUAGE plpgsql;
```

Es lo mismo si el evento se lo declara BEFORE o AFTER?

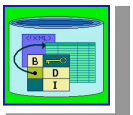
Rta
NOOOO

Si ejecutáramos UPDATE inscripto SET NOTA = 10

Con Before tendríamos: 4 tuplas en inscripto al terminar

Con After tendríamos: 6 tuplas en inscripto al terminar.

Por ese motivo PostgreSQL no permite el primer caso. Solo acepta AFTER.



Ejemplo 23

Supongamos que tenemos los siguientes esquemas de relación:

PROVEEDOR(<u>nombre</u>, direccion)
PROVEE(<u>nombre</u>, <u>codProd</u>, precio)

y se creó la siguiente vista:

```
CREATE VIEW maximal (nombre, maximoPrecio)
AS
SELECT proveedor.nombre, MAX( precio)
FROM provee, proveedor
WHERE provee.nombre = proveedor.nombre
GROUP BY proveedor.nombre;
```

Si quisiéramos borrar, a partir de la vista, los precios máximos de cada proveedor haríamos:

```
DELETE FROM maximal;
```

Pero, está operación no suele estar permitida en los DBMS ¿Por qué?

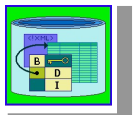
¿Cómo se podría hacer vía los *triggers*?

```
CREATE TRIGGER InsteadOfForEachRow
 INSTEAD OF DELETE ON MAXIMAL
 FOR EACH ROW
EXECUTE PROCEDURE BorraCaros();
```

```
CREATE OR REPLACE FUNCTION BorraCaros() RETURNS Trigger
AS $$
BEGIN

  DELETE FROM provee WHERE nombre = old.nombre and precio=
old.maximoPrecio;

   RETURN old;  -- hacer efectivo el evento algo que no sea null

END;
$$ LANGUAGE plpgsql;
```

Supongamos que la tabla prove y proveedor contienen:

PROVEEDOR	
<u>Nombre</u>	Dirección
A	Florida 100
B	**9 de Julio 200**
C	Lima 500



PROVEE		
Nombre	CodProD	Precio
A	Tornillo	1
A	**Tuerca**	**2**
A	Martillo	100
B	Resma Legal	500
B	Resma A4	500
C	Tonner	1200

Si se consulta la vista Maximal se obtiene
A 100
B 500
C 1200

Al ejecutar la primera vez DELETE FROM MAXIMAL se borran 4 tuplas de la tabla provee, gracias al trigger quedando

PROVEE		
Nombre	CodProD	Precio
A	Tornillo	1
A	**Tuerca**	**2**

Al volver a consultar maximal se obtiene
A 2

Faltaría ejecutar otras 2 veces DELETE FROM MAXIMAL para que la tabla provee quede sin tuplas.

Notar que la implementación de triggers con INSTEAD OF nos permite cambiar información de las tablas subyacentes a una vista compleja, eliminando las limitaciones que se tienen de actualización de las vistas.