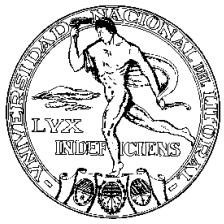

Universidad Nacional del Litoral

Facultad de Ingeniería y Ciencias Hídricas

Departamento de Informática

Bases de Datos

SQL: Guía de Trabajo Nro. 7

Triggers

Parte 1

Msc. Lic. Hugo Minni

2020

Triggers

Los triggers son diferentes de otros tipos de constraints o procedimientos que hemos visto:

1. Se activan cuando ocurren ciertos **eventos** (llamados **triggering events**), especificados por el desarrollador. Los tipos de eventos permitidos son normalmente insert, update o delete sobre una determinada tabla.

2. En algunos motores de bases de datos, una vez activado por su triggering event, el trigger evalúa una **Condition**. Si la condición no aplica, el trigger finaliza sin ejecutar nada.

3. Se ejecuta la **Action** asociada al trigger.
En el caso de que el motor de bases de datos permita Condition, esta Action se ejecuta solamente si la Condition evalúa a true.

Entonces tenemos las siguientes definiciones:

Triggering event

Eventos
Triggering event

Activan el trigger.
Los tipos de eventos son normalmente `INSERT`, `UPDATE` o `DELETE` sobre una determinada tabla.

En el caso particular de los eventos `UPDATE`, podemos especificar que los mismos deben estar limitados a un atributo (columna) o conjunto de atributos en particular.

Condición

Condition

Es una condición que se debe cumplir a fin de que el trigger lleve a cabo alguna acción. Si no se cumple, no sucede nada.
Esta condition es opcional.
Si no está presente, la **Action** es ejecutada cada vez que el trigger es activado. Si está presente, la action es ejecutada **sólo si la Condition es verdadera**.
Su función es básicamente evitar la ejecución innecesaria de un bloque de código.

Acción

Action

Es un bloque de código que puede modificar en alguna manera los efectos del evento. Inclusive puede abortar la transacción de la cual el evento forma parte.

No es necesario que este código esté vinculado necesariamente al evento que activó el trigger. El código puede llevara a cabo **cualquier secuencia de operaciones** de base de datos.

Recordemos que el evento que está por suceder es un INSERT, UPDATE o DELETE.

La **acción** se va a interponer y puede modificar estas situaciones, o aún evitar por completo que se lleven a cabo.

Ejemplo 1

Queremos crear un trigger que cree una copia de la tabla `Titles` por cada modificación que hagamos sobre sus tuplas en el transcurso del año 2015.

PL/pgSQL

```
A  CREATE TRIGGER trTitles
       {BEFORE | AFTER}  C
       UPDATE ON Titles  B
   WHEN (date_part('year', now()) = '2015')  D
         EXECUTE PROCEDURE test200();  E
                G          F
```

A. La sentencia `CREATE TRIGGER` crea el trigger `trTitles`.

B. La cláusula que indica el **Triggering event:** En este caso, una sentencia `UPDATE` sobre la tabla `Titles`.

Por ahora ignoraremos las cláusulas `BEFORE` y `AFTER` (**C**).

PL/pgSQL admite **Condition.** La **Condition** es (**D**). Se expresa con la keyword `WHEN` y una expresión booleana. En este caso estamos diciendo que sólo ejecutaremos la **Action** cuando el año actual sea `2015`.

Habíamos dicho que la Action era un bloque de código que podía modificar en alguna manera los efectos del evento. En el caso de PL/pgSQL la **Action** es la invocación a una función especial creada por nosotros (**E**) llamada **trigger function**.

En el ejemplo la trigger function es *test200()* (**F**).
La trigger function es precedida por la cláusula `EXECUTE PROCEDURE` (**G**)

Action - La trigger function

Es una función PostgreSQL -como las que vimos en la Guía de Trabajo Nro. 4, pero con las siguientes particularidades:

- Se debe declarar **sin parámetros**.
- Su cláusula `RETURNS` debe especificar `trigger`

Un ejemplo de triggger function:

```
CREATE OR REPLACE FUNCTION test7000()
   RETURNS trigger
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
   BEGIN
      IF ((SELECT SUM(price*qty)
            FROM titles20 t INNER JOIN sales s
                                ON t.title_id = s.title_id
            WHERE pub_id = NEW.pub_id) <= 1500) THEN
         ...
      ELSE
         ...;
      END IF;

      ...
   END
   $$;
```

Más de un triggering event

Si tenemos más de un triggering event, los mismos se separan con `OR`:

```
CREATE TRIGGER trTitles
   [BEFORE | AFTER] UPDATE OR INSERT ON Titles
    ...
```

T-SQL

```
CREATE TRIGGER trTitles
   ON Titles   (A)
   FOR UPDATE  (B)
(C) AS
      IF (YEAR(CURRENT_TIMESTAMP) = '2015')        (D)
         SELECT * INTO auditoria FROM titles
      RETURN
        (E)
```

T-SQL es bastante diferente a PL/pgSQL en la definición del trigger.

(**A**) y (**B**) son las cláusulas que indican el **Triggering event**:

Primero se especifica sobre que tabla se evalúa el evento (**A**) y luego de qué evento de trata (en este caso una sentencia `UPDATE`) (**B**).

En T-SQL tenemos que escribir la keyword `FOR` antes del nombre del o los triggering events. En las últimas versiones de SQL Server, `FOR` se puede reemplazar por `AFTER`

T-SQL no permite especificar **Condition**.

Luego se especifica la keyword `AS` (**C**).

En T-SQL la **action** es un bloque de código de cualquier tipo que se escribe directamente luego de la keyword `AS` (**D**).

Si tenemos más de un triggering event, los mismos se separan con coma. Por ejemplo:
`FOR UPDATE, INSERT, DELETE`

La sentencia `RETURN` (**E**) no puede especificar valor de retorno.

El trigger y el estado de la base de datos

Estado de la base de datos

Recordemos que llamábamos **estado de la base de datos** al conjunto de las **instancias actuales** de **todas las tablas** de la base de datos.

Hay algo crucial respecto a los triggers y es que la evaluación de la **Condition** y la ejecución de la **Action** del trigger **puede** tener efecto sobre el estado de la base de datos **anterior** a la ocurrencia del **triggering event** (`BEFORE`) o sobre el estado de la base de datos **posterior** a la ocurrencia del **triggering event** (`AFTER`).

BEFORE trigger
AFTER trigger

Se denomina **BEFORE TRIGGER** a un trigger que evalúa su **Condition** y ejecuta su **Action** sobre el estado de la base de datos previo a la ejecución del Triggering event.

Se denomina **AFTER TRIGGER** a un trigger que evalúa su **Condition** y ejecuta su **Action** sobre el estado de la base de datos posterior a la ejecución del Triggering event.

Siguiendo con nuestro ejemplo en PL/pgSQL:

```
CREATE TRIGGER trTitles
   AFTER (A)
   UPDATE ON Titles
WHEN (date_part('year', now()) = '2015')
      EXECUTE PROCEDURE test200();
```

...aquí estamos indicando que el trigger hará una evaluación del estado de la base de datos **posterior** al **triggering event**.

Triggers y el estado de la base de datos

En T-SQL **todos los triggers son triggers AFTER**. **T-SQL no soporta triggers BEFORE**. La keyword `FOR` ha sido sustituida por `AFTER` en las nuevas versiones de SQL Server a fines de que quede más claro que se trata de un trigger AFTER:

```
CREATE TRIGGER trTitles
   ON Titles
   AFTER UPDATE
   AS
      IF (YEAR(CURRENT_TIMESTAMP) = '2015')
         SELECT * INTO auditoria FROM titles
```

Como T-SQL no soporta Condition, en este ejemplo la reemplazamos por un condicional IF

Granularidad del trigger

Sabemos que una sentencia SQL puede afectar una o varias tuplas.

Por ejemplo:

```
UPDATE titles2
   SET price = 50
   WHERE title_id = 'BU1032'
```

...aquí estamos realizando un `UPDATE` filtrando por Primary Key, lo cual nos da la garantía de que la sentencia afectará una **única tupla** de la tabla `titles2`

Pero podríamos tener una sentencia `UPDATE` como la siguiente:

```
UPDATE titles2
   SET price = price * 1.50
   WHERE type = 'business'
```

Hay cuatro publicaciones de tipo `'business'`. Por lo tanto, cuatro tuplas se verían afectadas por esta sentencia.

En SQL el programador tiene la opción de especificar que el trigger se ejecuta:

a) Una vez por cada tupla modificada

b) Una vez para todas las tuplas que han sido modificadas en la sentencia SQL. (en el caso de que la sentencia de modificación SQL afecte muchas tuplas).

Entonces tenemos las siguientes definiciones:

row-level trigger

Es un trigger que se ejecuta una (1) vez por cada tupla modificada.

statement-level trigger

Es un trigger que se ejecuta una (1) vez no importa cuantas hayan sido las tuplas modificadas.

En el ejemplo recién planteado:

```
UPDATE titles2
   SET price = price * 1.50
   WHERE type = 'business'
```

Si el trigger fuese definido como statement-level, se ejecutaría una única vez aún cuando fuesen cuatro las tuplas afectadas por la sentencia.

En cambio, si el trigger es definido como row-level, el trigger se ejecutaría cuatro veces, una por cada tupla afectada.

En nuestro ejemplo PL/pgSQL:

```
CREATE TRIGGER trTitles
   AFTER
   UPDATE ON Titles
A  FOR EACH ROW
WHEN (date_part('year', now()) = '2015')
      EXECUTE PROCEDURE test200();
```

A. La cláusula indica que el trigger se ejecuta una (1) vez por cada tupla modificada.

En T-SQL todos los triggers son statement-level triggers. **T-SQL no soporta row-level triggers**.

Los triggers PL/pgSQL son por omisión **statement-level triggers**.

Triggering events de tipo UPDATE

En PL/pgSQL, cuando el Triggering event es de tipo `UPDATE`, podemos especificar una cláusula adicional (opcional) para acotar el o los atributos (columnas) que deben ser modificados para que el trigger sea activado.

Supongamos que queremos impedir que se disminuyan los precios de las publicaciones. Podemos escribir la segunda línea como:

```
CREATE TRIGGER trTitles2
   AFTER UPDATE OF price ON Titles
   FOR EACH ROW
   WHEN (Condition)
      Action
```

Estamos especificando que el triggering event debe ser un update sólo de las columnas listadas luego de la keyword `OF`.

Si se debe especificar más de una columna, éstas se separan con coma.

La cláusula `OF` obviamente no es válida para eventos `INSERT` o `DELETE`, ya que éstos eventos tienen sentido solo para tuplas completas.

T-SQL no soporta la cláusula `OF` para especificar el update de sólo una serie de atributos listados.

Vinculado a esta característica no proporcionada, en cambio, T-SQL permite que dentro del cuerpo de la **Action** utilicemos la función `UPDATE()`.`UPDATE()` permite determinar si una columna dada ha sido afectada por una sentencia `INSERT`, `UPDATE` o `DELETE` que es triggering event de un trigger. Esta función es accesible únicamente dentro de la **Action** de un trigger:

```
UPDATE (nombre-columna)
```

`nombre-columna` es el nombre de la columna para la cual se desea testear la existencia de modificaciones. La función devuelve un valor verdadero si la columna especificada ha sido afectada por una sentencia `INSERT` o `UPDATE`.

`UPDATE()` considera que la columna ha sido afectada por un `UPDATE` cuando la cláusula `SET` afecta directamente la columna en cuestión.

`UPDATE()` considera que la columna ha sido afectada por un `INSERT` en los siguientes casos:

Valores de tuplas anteriores y posteriores

Los triggers permiten que tanto la **Condition** como la **Action** puedan hacer referencia a antiguos valores de tuplas y/o nuevos valores de tuplas en dependencia de las alteraciones provocadas por el **Triggering event**.

Valores de tuplas anteriores y posteriores

En PostgreSQL, para acceder a los valores de las tuplas anteriores y posteriores, la trigger function dispone de dos variables de tipo `RECORD`:

`NEW`
Es una variable de tipo `RECORD` con exactamente la misma estructura de una tupla de la tabla a la que se asocia el trigger, y que contiene la nueva tupla para triggering events `INSERT` o `UPDATE` en row-level triggers.
Esta variable es `NULL` para operaciones `DELETE`.

`OLD`
Es una variable de tipo `RECORD` con exactamente la misma estructura de una tupla de la tabla la que se asocia el trigger, y que contiene la tupla antigua para triggering events `UPDATE` o `DELETE` en row-level triggers.
Esta variable es `NULL` para Triggering events de tipo `INSERT`.

Estas variables **sólo están disponibles en row-level triggers**.

En la **Condition**, PL/pgSQL también permite, en row-level triggers, que hagamos referencia a las columnas antiguas y nuevas con la sintaxis `OLD.nombre-columna` y `NEW.nombre-columna` respectivamente. Por supuesto, los triggers con triggering event `INSERT` no pueden referenciar a `OLD` y los triggers con triggering event `DELETE` no pueden referenciar a `NEW`.

Ejemplo 1

```
CREATE FUNCTION test()
   RETURNS trigger
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
   BEGIN
      UPDATE Titles
         SET price = OLD.price
         WHERE title_id = NEW.title_id;
      RETURN NULL;
   END
   $$;


CREATE TRIGGER trTitles3
   AFTER UPDATE OF price ON Titles
   FOR EACH ROW
   WHEN (OLD.price > NEW.price)
      EXECUTE PROCEDURE test();
```

Cuando el triggering event de un row-level trigger es un `UPDATE`, tendremos una tupla antigua y una nueva (las tuplas previas y posteriores al `UPDATE`, respectivamente).

Valores de tuplas anteriores y posteriores

T-SQL no permite el acceso a la tupla antigua y la nueva. Pero tampoco tendría sentido, ya que no soporta row-level triggers.

Triggers BEFORE row-level

Como vimos, en los triggers BEFORE, la **Condition** es evaluada sobre el estado de la base de datos previo a la ejecución del **Triggering event**.
Si la **Condition** es verdadera, la **Action** del trigger es ejecutada ***sobre ese estado***.

Uso de los triggers BEFORE row-level

Podemos decir que el espíritu detrás de la idea de los triggers BEFORE es dar al programador la posibilidad de actuar en última instancia.
Podemos distinguir dos maneras diferentes de actuar:

Validación de datos entrantes

En el caso de los triggers FOR UPDATE o FOR INSERT, los triggers BEFORE permiten al desarrollador realizar algún tipo de validación sobre los valores entrantes.

En otras palabras, el desarrollador puede corregir o retocar algún aspecto de las tuplas que forman parte de una sentencia `INSERT` o `UPDATE` **antes** de que sean realmente plasmadas en la base de datos.

Cancelación lisa y llana de la transacción

En los triggers FOR INSERT, FOR UPDATE o FOR DELETE, los triggers BEFORE permiten al desarrollador realizar algún control o evaluación de alguna regla de negocios lo suficientemente compleja como para que no haya podido ser resuelta con ninguna constraint.

En este último caso, el trigger puede darse la potestad de **cancelar la transacción**.
Es decir, impedir que el evento que disparó el trigger se plasme en la base de datos.

Triggers BEFORE row-level en PL/pgSQL

Analizaremos estos aspectos sobre PL/pgSQL, ya que como vimos SQL Server no admite triggers BEFORE.

En todos los casos de triggers *BEFORE row-level*, PL/pgSQL define el curso de acción o comportamiento a través del *valor de retorno* de la trigger function
Lo revisaremos para cada caso.

A. Aprobar el Triggering event sin intervenir

El caso más sencillo es aquel en el que el trigger **no interviene**, sino que simplemente deja que el triggering event `INSERT` o `UPDATE` que disparó el trigger simplemente se lleve a cabo para la tupla actual.
En este caso, el valor de retorno de la trigger function debe ser la table row **`NEW`**.

Si se trata de un triggering event `DELETE` y efectivamente deseamos que el mismo efectivamente ocurra para la tupla actual, el valor de retorno de la trigger function **no posee relevancia**, pero debe ser diferente de `NULL`.
Normalmente se estila retornar la table row **`OLD`**.

B. Aprobar el Triggering event interviniendo

Acá ya entraríamos en la categoría de **Validación de datos entrantes** recién planteada**.**

Supongamos que se trata de un triggering event INSERT o UPDATE y `UPDATE` y deseamos que el mismo efectivamente ocurra para la tupla actual.

Pero, sin embargo, queremos corregir o retocar algún aspecto de las tuplas que forman parte de una sentencia `INSERT` o `UPDATE` **antes** de que sean realmente plasmadas en la base de datos.

En este caso tenemos la posibilidad de intervenir modificando o ajustando algún aspecto de la tupla que está por ser insertada o modificada.

Una vez realizada esta intervención, el valor de retorno de la trigger function puede ser la tupla `NEW` "adulterada" u otra table row completamente nueva con la misma estructura.

La tupla retornada se convierte en la tupla que será insertada o que reemplazará a la tupla que está siendo insertada/actualizada.

Ejemplo

Supongamos que queremos insertar tuplas de publicaciones, pero existen algunas de las que desconocemos el precio inicial.

Lo que podemos hacer es implementar un trigger que verifique que el atributo `price`, y, si el mismo es nulo, lo reemplace por un valor adecuado (tal vez uno que calculamos de alguna manera compleja).

En el siguiente ejemplo queremos reemplazar el precio por el valor `15` (algo que podríamos haber implementado a través de una constraint `DEFAULT`, pero sirve como ejemplo):

```
CREATE FUNCTION test()
   RETURNS trigger
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
   BEGIN
      NEW.price:=15;
      RETURN NEW;
   END
   $$;

CREATE TRIGGER trTitles4
   BEFORE INSERT ON Titles
   FOR EACH ROW
   EXECUTE PROCEDURE test();
```

Ejemplo

Supongamos que la tabla `Publishers` posee dos columnas adicionales: `FechaHoraAlta` está destinada a guardar la fecha y hora en que se da de alta una editorial. `UsuarioAlta` se utilizará para registrar el usuario que realizó la operación de inserción:

```
ALTER TABLE publishers
   ADD COLUMN FechaHoraAlta DATE NULL;

ALTER TABLE publishers
   ADD COLUMN UsuarioAlta VARCHAR(255) NULL;
```

Nos solicitan que, ante la inserción de una nueva editorial, se registre la fecha y hora de la operación (función `CURRENT_TIMESTAMP`) y el usuario que llevó a cabo la operación (función `SESSION_USER`);

Podemos escribir un trigger como el siguiente:

```
CREATE TRIGGER tr_Editoriales
   BEFORE
   INSERT
   ON publishers20
   FOR EACH ROW
      EXECUTE PROCEDURE test604();


CREATE OR REPLACE FUNCTION test604()
   RETURNS trigger
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
      --recTitle RECORD;
   BEGIN
       NEW.FechaHoraAlta := (SELECT CURRENT_TIMESTAMP);
       NEW.UsuarioAlta  := SESSION_USER;
       RETURN NEW;
   END
   $$;
```

C. Cancelar un Triggering Event

Acá ya entraríamos en la categoría de **Cancelación lisa y llana de la transacción** recién planteada**.**

Si se trata de un Triggering event `INSERT`, `UPDATE` o `DELETE`, y deseamos cancelar el mismo para la tupla actual, el valor de retorno de la trigger function debe ser `NULL`.
Esto equivale a deshacer la transacción.

Ejemplo

Supongamos que deseamos impedir que se den de alta publicaciones de editoriales que no hayan vendido por más de `$2500` (tabla `Sales`).

Por ejemplo, La editorial `'1389'` posee un monto de ventas que debería permitir la inserción de sus publicaciones. En cambio, para la editorial `'0736'` seguramente se debería impedir la inserción de publicaciones.

La siguiente sentencia debería permittirse:

```
INSERT INTO titles
   SELECT 'PC4545', 'Prueba 1', 'trad_cook', '1389',
          14.99, 8000.00, 10, 4095, 'Prueba 1', '06/12/91'
```

...pero la siguiente no:

```
INSERT INTO titles
   SELECT 'PC4646', 'Prueba 2', 'trad_cook', '0736',
          14.99, 8000.00, 10, 4095, 'Prueba 1', '06/12/91'
```

```
CREATE OR REPLACE FUNCTION test7000()
   RETURNS trigger
   LANGUAGE plpgsql
   AS
   $$
   DECLARE
   BEGIN
      IF ((SELECT SUM(price*qty)
            FROM titles20 t INNER JOIN sales s
                              ON t.title_id = s.title_id
            WHERE pub_id = NEW.pub_id) <= 2500) THEN
         RETURN NULL;
      ELSE
         RETURN NEW;
      END IF;
   END
   $$;



CREATE TRIGGER tr_ejercicio5
   BEFORE
   INSERT
   ON Titles20
   FOR EACH ROW
      EXECUTE PROCEDURE test7000();
```

Como se mencionó antes, T-SQL no soporta BEFORE triggers.
Sin embargo, proporciona la posibilidad de cancelar lisa y llanamente la transacción. Lo veremos más adelante.

Triggers AFTER

Uso de los triggers AFTER

A diferencia de los triggers BEFORE, los triggers AFTER no están pensados para actuar ANTES de que los datos sean plasmados en la base de datos, ni validando ni cancelando eventualmente la transacción.

La idea detrás de los triggers AFTER es que el desarrollador pueda ejecutar alguna actividad asociada al triggering event (INSERT, UPDATE o DELETE) que se está llevando a cabo.
Esta actividad puede ser el registro de la operación, por ejemplo.

Supongamos que se está llevando a cabo una operación bancaria delicada, se maneja dinero, y se desea llevar el registro de qué usuario está llevando a cabo la operación, en que fecha y hora y qué columnas está modificando.
Inclusive pude se necesario guardar los datos tal como estaban antes de la operación.
Este es un uso clásico de un trigger AFTER

En SQL Server todos los triggers son AFTER, así que es natural el rol de los mismos como herramienta de registro o log.

Ejemplo

Supongamos que tenemos una tabla de registro (log) con la siguiente estructura:

```
CREATE TABLE Registro
   (
    fecha DATE NULL,
    tabla varchar(100) NULL,
    operacion varchar(30) NULL,
    CantFilasAfectadas Integer NULL
   )
```

Nos solicitan registrar en ella una entrada por cada sentencia `DELETE` que afectan más de una tupla.
El siguiente trigger lleva a cabo esta tarea:

```
CREATE TRIGGER tr_ejercicio9
   ON Employee
   AFTER
   DELETE
   AS
   DECLARE
      @CantFilas INTEGER;
   BEGIN
      SET @CantFilas = (SELECT COUNT(*) FROM deleted);
      IF (@CantFilas > 1)
         INSERT INTO Registro
            SELECT CURRENT_TIMESTAMP, 'Employee', 'DELETE', @CantFilas;
      --END IF;
      RETURN
   END
```

Triggers AFTER row-level

Valor de retorno de la trigger function

El valor de retorno de una trigger function en un AFTER trigger row-level es ignorado. Puede ser `NULL`.

Statement-level triggers

Creamos un statement-level trigger especificando la cláusula `FOR EACH STATEMENT`.

Acceso a los valores de las tuplas anteriores y posteriores

En PL/pgSQL es imposible acceder a las tuplas anteriores o posteriores desde un statement-level trigger.

Como dijimos antes, todos los triggers T-SQL son statement-level AFTER triggers.
T-SQL sí permite acceder al conjunto de tuplas "anteriores" a la aplicación del triggering event a través de una "tabla virtual" denominada **deleted**, y al conjunto de tuplas "posteriores" a la aplicación del triggering event a través de una "tabla virtual" denominada **inserted**.

Estas tablas virtuales pueden ser manipuladas como cualquier tabla, con la salvedad de que son de solo lectura.

inserted posee entonces la estructura de la tabla a la que se asocia el trigger, y contiene las nuevas tuplas para triggering events `INSERT` o `UPDATE`.
inserted no está definida para triggering events `DELETE`.

deleted posee la misma estructura de la tabla a la que se asocia el trigger, y que contiene las antiguas tuplas para triggering events UPDATE o DELETE.
deleted no está definida para triggering events de tipo `INSERT`.

Valor de retorno en un statement-level trigger

Valor de retorno de la trigger function

El valor de retorno de una trigger function asociada a un statement-level trigger (BEFORE o AFTER) es ignorado. Puede ser `NULL`.

Validación de datos entrantes en T-SQL

Ya sabemos que SQL Server no posee triggers BEFORE. Ahora analizaremos que punto de vista adopta respecto a la validación de datos entrantes y eventual cancelación de la transacción.

A. Aprobar el Triggering event sin intervenir

Ya sea que se trate de un triggering event INSERT, UPDATE o DELETE, en SQL Server el no intervenir equivale a no hacer nada.
El trigger simplemente deja que la transacción ocurra.

B. Aprobar el Triggering event interviniendo

Esta era la categoría de **Validación de datos entrantes** antes planteada**.**

Supongamos que se trata de un triggering event INSERT o UPDATE y `UPDATE` y deseamos que el mismo efectivamente ocurra para la tupla actual.

Sin embargo, queremos corregir o retocar algún aspecto de las tuplas que forman parte de una sentencia `INSERT` o `UPDATE` **antes** de que sean realmente plasmadas en la base de datos.

Como en SQL Server los triggers son siempre statement-level, en el mejor de los casos, si solo una tupla ha sido afectada, tendremos una "tabla virtual" INSERTED o DELETED con una única tupla.

Sin embargo, SQL Server no permite modificar estas tuplas. Son solo lectura.

En definitiva, los triggers T-SQL **no permiten intervenir modificando o ajustando algún aspecto de la tupla que está por ser insertada o modificada.**

Tampoco permiten generar una tupla desde cero.

C. Cancelar un Triggering Event

Esta era la categoría de **Cancelación lisa y llana de la transacción** recién planteada**.**

Si se trata de un Triggering event `INSERT`, `UPDATE` o `DELETE`, y deseamos cancelar el mismo para la tupla –o tuplas- actuales, SQL Server permite cancelar la transacción a través de la sentencia `ROLLBACK TRANSACTION`.

Ejemplo

El promedio actual de precios de publicaciones es $15. Supongamos que queremos impedir que este promedio aumente. En T-SQL podemos escribir un trigger como el siguiente:

```
CREATE TRIGGER AveragePriceTrigger
   ON Titles
   AFTER UPDATE
   AS
      IF(15 < (SELECT AVG(price) FROM Titles))   (A)
         ROLLBACK TRANSACTION
```

Estamos evaluando (**A**) si el promedio de precios de publicaciones quedó por encima de los $15 **después de la actualización**.

Ejemplo

Supongamos que deseamos impedir que se den de alta publicaciones de editoriales que no hayan vendido por más de `$2500` (tabla `Sales`).

Por ejemplo, La editorial `'1389'` posee un monto de ventas que debería permitir la inserción de sus publicaciones. En cambio, para la editorial `'0736'` seguramente se debería impedir la inserción de publicaciones.

La siguiente sentencia debería permitirse:

```
INSERT INTO titles
   SELECT 'PC4545', 'Prueba 1', 'trad_cook', '1389',
          14.99, 8000.00, 10, 4095, 'Prueba 1', '06/12/91'
```

...pero la siguiente no:

```
INSERT INTO titles
   SELECT 'PC4646', 'Prueba 2', 'trad_cook', '0736',
          14.99, 8000.00, 10, 4095, 'Prueba 1', '06/12/91'
```

```
CREATE TRIGGER tr_Publicaciones
   ON titles2
   FOR INSERT
   AS
      DECLARE @pub_id char(4),
              @monto float
      SELECT @pub_id = pub_id
         FROM inserted

      SELECT @monto = SUM(price*qty)
         FROM titles t INNER JOIN sales s
                       ON t.title_id = s.title_id
         WHERE pub_id = @pub_id

      PRINT @pub_id + ' vendio ' + CONVERT(varchar(10), @monto)

      IF @monto <= 2500
         ROLLBACK TRANSACTION
      -- END IF
      RETURN
```

Eliminar triggers

```
DROP TRIGGER <Nombre-trigger>
```

En PostgreSQL la sintaxis especifica la tabla sobre la que se aplica el trigger a eliminar:

```
DROP TRIGGER <Nombre-trigger>
   ON <Tabla-asociada>
```

Información disponible en triggers PL/pgSQL

En una trigger function PL/PgSQL disponemos de una variable de tipo `text` llamada `TG_OP`. Esta variable posee un valor string `"INSERT"`, `"UPDATE"` o `"DELETE"` indicando cuál fue el triggering event que desencadenó el trigger.

Otras variables que pueden resultar útiles son las siguientes:
`TG_NAME` proporciona el nombre del trigger en ejecución.
`TG_WHEN` proporciona el tipo de trigger: `BEFORE`, `AFTER` o `INSTEAD OF`.
`TG_LEVEL` proporciona la granularidad del trigger: `ROW` o `STATEMENT`.
`TG_TABLENAME` proporciona el nombre de la tabla a la cual está asociada el trigger.

Síntesis

	PL/pgSQL		T-SQL	
	INSERT/UPDATE	**DELETE**	**INSERT/UPDATE**	**DELETE**
Aprobar triggering event sin alterarlo	- BEFORE - ROW-LEVEL - RETURN **NEW**	- BEFORE - ROW-LEVEL - RETURN **OLD**	Sin particularidades	
Aprobar triggering event alterándolo	- BEFORE - ROW-LEVEL **- Adulterar NEW o crearlo desde cero**. - RETURN **NEW**	No tiene sentido	No se puede	No tiene sentido
Cancelar triggering event	BEFORE ROW-LEVEL RETURN **NULL**		**ROLLBACK TRANSACTION**	
Logging	AFTER (podría ser BEFORE) ROW-LEVEL o STATEMENT-LEVEL. RETURN NULL (se ignora)		AFTER (sin opción)	