Profesora: Olga Cuervo Miguélez

# TEMA 5

## PROGRAMACIÓN DE BASES DE DATOS. DISPARADORES O TRIGGERS

Módulo: **Bases de Datos**

Ciclo: **DAM**

# 9. Triggers o Disparadores

## 9.1 Creación de triggers. Diccionario de Datos

Los **triggers**, también llamados **desencadenadores** o **disparadores**, son programas almacenados que se ejecutan ("disparan") automáticamente en respuesta a algún suceso que ocurre en la base de datos.

En MySQL ese tipo de suceso se corresponde con alguna instrucción DML (INSERT, UPDATE, DELETE) sobre alguna tabla.

- ✓ Suponen un mecanismo para asegurar la integridad de los datos.
- ✓ Se emplean también como un método para realizar operaciones de auditoría sobre la base de datos.
- ✓ No hay que abusar de su utilización pues ello puede traer consigo una sobrecarga del sistema y por tanto un bajo rendimiento del mismo.
- ✓ Para poder crear triggers, en versiones anteriores a la 5.16 se necesita el privilegio SUPER. A partir de esta el privilegio es el de TRIGGER.
- ✓ No se pueden crear ni sobre una tabla temporal ni sobre una vista, solamente sobre tablas.

**Sintaxis:**

```
CREATE [DEFINER={cuenta_usuario | CURRENT_USER}] TRIGGER
nombre_trigger
    {BEFORE | AFTER}
    {UPDATE | INSERT | DELETE}
  ON tabla
  FOR EACH ROW
  cuerpo_del_trigger
```

**Aspectos a comentar de la sintaxis anterior:**

- ➢ **DEFINER ={cuenta_usuario | CURRENT_USER }:** Indica con qué privilegios se ejecutan las instrucciones del trigger. La opción por defecto es CURRENT_USER, que indica que las instrucciones se ejecutan con los privilegios del usuario que lanzó la instrucción de creación del trigger. La opción cuenta_usuario por otro lado hace que el trigger se ejecute con los privilegios de dicha cuenta.
- ➢ **Nombre del trigger:** Sigue las mismas normas que para nombrar cualquier objeto de la base de datos.
- ➢ **BEFORE | AFTER:** Indica cuando se ejecuta el trigger, antes (before) o después (after) de la instrucción DML que lo provocó.
- ➢ **UPDATE | INSERT | DELETE:** Define la operación DML asociada al trigger.
- ➢ **ON tabla**: Define la tabla base asociada al trigger.
- ➢ **FOR EACH ROW:** Indica que el trigger se ejecutará para cada fila de la tabla afectada por la operación DML. Esto es, si tenemos asociado un trigger a la operación de borrado de una tabla y se eliminan con una sola instrucción 6 filas de esta última, el trigger se ejecutará 6 veces, una por cada fila eliminada. Otros gestores de bases de datos (así como futuras implementaciones de MySQL) consideran también el otro estándar de ANSI, la cláusula FOR EACH

STATEMENT. Con esta segunda opción, el trigger se ejecutaría por cada operación DML realizada; en el ejemplo anterior, la instrucción de borrado daría lugar a que solo se ejecutara el trigger una sola vez en lugar de 6 (filas afectadas) con esta futura cláusula.

- **Cuerpo del trigger:** El conjunto de instrucciones que forman este programa almacenado. Si el conjunto de instrucciones es más de una, éstas irán agrupadas en un bloque BEGIN … END. NO pueden contener una instrucción CALL de llamada a un procedimiento almacenado.

### Referencias a las columnas afectadas:

Dentro del cuerpo del trigger se puede hacer referencia a los valores de las columnas que están siendo modificadas con una sentencia DML (la que provocó que se disparara el trigger), incluso pueden cambiarse si así se considerara. Para ello se utilizan los objetos **NEW** y **OLD**. De esta manera, en un trigger de tipo BEFORE UPDATE afectando a una columna *micolumna*, se utilizará la expresión *OLD.micolumna* para conocer el valor de la columna antes de ser modificada y *NEW.micolumna* será el nuevo valor de la columna después de la modificación.

Estos dos valores solo tienen sentido los dos juntos en una modificación (UPDATE), pues ante una inserción (INSERT) no existe valor antiguo (OLD) y ante un borrado (DELETE) no existe un valor nuevo (NEW) pues el que existe, se elimina.

**Dentro de un trigger de tipo BEFORE**, puede cambiarse el nuevo valor mediante una sentencia de asignación SET por lo que anularía por completo el efecto de la instrucción DML que provocó el trigger.

### Eventos de disparo:

Como se ha indicado, un trigger se ejecuta automáticamente (dispara) antes o después de una instrucción DML: INSERT, UPDATE o DELETE.

Además de la forma explícita anterior, pueden también ejecutarse las instrucciones del cuerpo del trigger si la modificación se produce de forma implícita, como sería el caso de una instrucción REPLACE pues en realidad equivale a una instrucción DELETE seguida de una INSERT (por lo tanto aquí se ejecutarían los triggers asociados a estas dos operaciones).

### ¿Triggers de tipo BEFORE o AFTER?

Prácticamente no hay diferencia. La ventaja que puede suponer utilizar los triggers de tipo BEFORE es que pueden cambiarse los valores modificados inicialmente por una instrucción UPDATE o INSERT mientras que con los triggers de tipo AFTER daría un error de ejecución.

Teniendo en cuenta los 2 tipos de triggers y las tres operaciones DML distintas, podríamos tener 6 triggers por tabla: BEFORE INSERT, AFTER INSERT, BEFORE UPDATE, AFTER UPDATE, BEFORE DELETE y AFTER DELETE.

## 9.2 Borrado de triggers

**Sintaxis:**

```
DROP TRIGGERS [IF EXISTS] [schema_name.]trigger_name
```

Se necesita el privilegio SUPER para borrar triggers.

## 9.3 Consulta de triggers

Podemos obtener información de los triggers creados con **SHOW TRIGGER**.

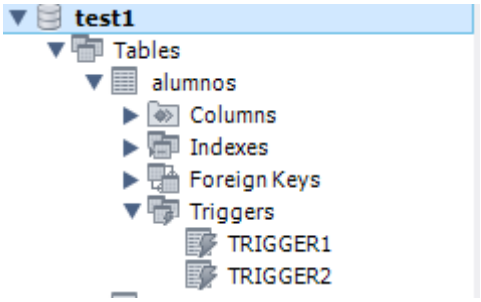
```
SHOW TRIGGERS [FROM  db_name] [LIKE  'patron' ]
```

***Nota***: Cuando se usa una cláusula `LIKE` con `SHOW TRIGGERS`, la expresión a cumplir (*`patron`*) se compara con el nombre de la tabla en que se declara el disparador, y no con el nombre del disparador:

Este comando nos permite mostrar triggers de una base de datos filtrándolo con un patrón.

Cuando creamos triggers se crea un nuevo registro en la tabla INFORMATION_SCHEMA llamada INFORMATION_SCHEMA.TRIGGERS que podemos visualizar con el siguiente comando:

**SELECT trigger_name, action statement FROM information_schema.triggers;**




## 9.4 Utilización de triggers

- **Copias o réplicas de datos:**

**Finalidad**: Mantener sincronizada una copia de seguridad de unos datos.

Para probar el ejemplo que viene a continuación lanzar antes la tabla:

```
CREATE TABLE alumnos_replica
```

```
(id      INT PRIMARY KEY,
 nombre VARCHAR(30));
```

y creamos el siguiente trigger:

```
-- COPIAS O REPLICAS
-- Ejemplo TRIGGER1
DROP TRIGGER IF EXISTS TRIGGER1;

DELIMITER $$
CREATE TRIGGER TRIGGER1
after INSERT ON ALUMNOS
FOR EACH ROW
BEGIN
    INSERT INTO ALUMNOS_REPLICA VALUES (NEW.ID, NEW.nombre);

END;$$

SHOW TRIGGERS FROM TEST;
```

Podemos probar con la siguiente instrucción:

```
INSERT INTO ALUMNOS VALUES (20,'Alberto Carrera');
```

Comprobamos que se ha insertado una nueva fila en la tabla '*ALUMNOS_REPLICA*'

```
SELECT * FROM ALUMNOS_REPLICA;
```

- **Auditoría:**

**Finalidad**: Auditar las operaciones que se realizan sobre una tabla.

Antes de lanzar el siguiente ejemplo, utilizaremos la tabla:

```
CREATE TABLE AUDITA (MENSAJE VARCHAR(200));
```

```
DROP TRIGGER IF EXISTS TRIGGER2;

DELIMITER $$
CREATE TRIGGER TRIGGER2
AFTER UPDATE ON ALUMNOS
FOR EACH ROW
BEGIN
    INSERT INTO AUDITA VALUES
     (CONCAT ('Modificacion realizada por: ' , USER(), ' el dia ', NOW(),
             ' Valores antiguos: ', OLD.ID, ' y ', OLD.nombre,
             ' Valores nuevos: ', NEW.ID, ' y ',NEW.nombre
            ));
END;$$

SHOW TRIGGERS FROM TEST;
```

Podemos probar con la instrucción

```
UPDATE ALUMNOS  SET
nombre = 'Mario Carrera'
WHERE ID = 20;
```

Comprobamos que se ha insertado una nueva fila en la tabla '*AUDITA*'

```
SELECT * FROM AUDITA;
```

| MENSAJE |
| --- |
| Modificacion realizada por: root@localhost el dia 2022-05-16 12:24:47 Valores antiguos: 20 y Alberto Carrera Valores nuevos: 20 y Mario Carrera |

# ÍNDICE