Eventos en MySQL. Disparadores múltiples en el mismo evento.

Crear disparadores múltiples

Resumen: en este documento, aprenderás a crear múltiples disparadores que se ativan con el mismo evento y tiempo de acción, para una tabla dada.

> Este tutorial es relevante para la versión 5.7.2, o superiores, de MySQL. Si tiene una versión anterior de MySQL, las declaraciones en el tutorial no funcionarán.

Antes de la versión 5.7.2, sólo se puede crear un disparador para un evento en una tabla, por ejemplo, solo puede crear un trigger para el evento BEFORE UPDATE o AFTER UPDATE. MySQL 5.7.2+ eliminó esta limitación y permite crear múltiples disparadores para una tabla determinada que tienen el mismo evento y tiempo de acción. Estos disparadores se activarán secuencialmente cuando ocurra un evento.
Aquí está la sintaxis para definir un disparador que se activará antes o después de un disparador existente en respuesta al mismo evento y tiempo de acción:

```
DELIMITER $$

CREATE TRIGGER trigger_name
{BEFORE|AFTER}{INSERT|UPDATE|DELETE}
ON table_name FOR EACH ROW
{FOLLOWS|PRECEDES} existing_trigger_name
BEGIN
    -- statements
END$$

DELIMITER ;
```

En esta sintaxis, FOLLOWS o PRECEDES especifica si se debe invocar el nuevo disparador antes o después de un disparador existente.

- FOLLOWS permite que el nuevo disparador se active después de un disparador existente.
- PRECEDES permite que el nuevo disparador se active antes que un disparador existente.

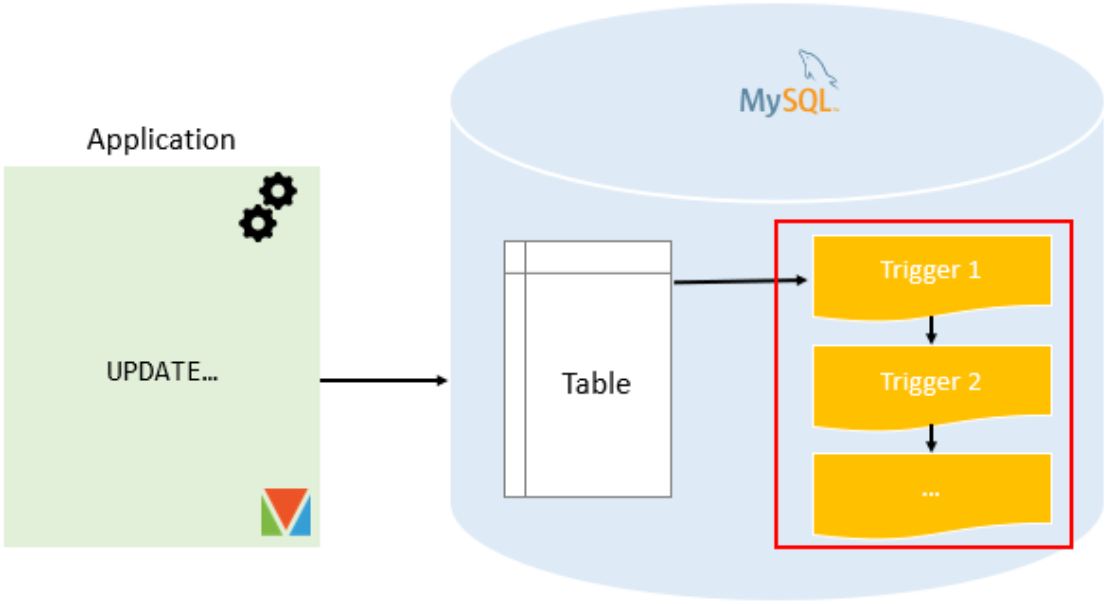

Figura 1: MySQL Multiple Trigger

Ejemplo de múltiples disparadores MySQL

Utilizaremos la tabla de productos en la base de datos de muestra para la demostración.
Suponga que desea cambiar el precio de un producto (columna MSRP) y registrar el precio anterior en una tabla separada llamada PriceLogs.
Primero, cree una nueva tabla price_logs utilizando la siguiente instrucción CREATE TABLE:

```
CREATE TABLE PriceLogs (
    id INT AUTO_INCREMENT,
    productCode VARCHAR(15) NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    updated_at TIMESTAMP NOT NULL
                        DEFAULT CURRENT_TIMESTAMP
            ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (id),
    FOREIGN KEY (productCode)
        REFERENCES products (productCode)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

En segundo lugar, cree un nuevo disparador que se active cuando ocurra el evento BEFORE UPDATE de la tabla de productos:

```
DELIMITER $$

CREATE TRIGGER before_products_update
   BEFORE UPDATE ON products
   FOR EACH ROW
BEGIN
```

```
     IF OLD.msrp <> NEW.msrp THEN
         INSERT INTO PriceLOgs(product_code,price)
         VALUES(old.productCode,old.msrp);
     END IF;
END$$

DELIMITER ;
```

En tercer lugar, verifique el precio del producto S12_1099:

```
SELECT
    productCode,
    msrp
FROM
    products
WHERE
    productCode = 'S12_1099';
```

En cuarto lugar, cambie el precio de un producto utilizando la siguiente declaración de ACTUALIZACIÓN:

```
UPDATE products
SET msrp = 200
WHERE productCode = 'S12_1099';
```

Quinto, consultar datos de la tabla PriceLogs:

```
SELECT * FROM PriceLogs
```

Funciona como se esperaba.
Suponga que desea registrar al usuario que cambió el precio. Para lograr esto, puede agregar una columna adicional a la tabla PriceLogs.
Sin embargo, con el propósito de la demostración de disparadores múltiples, crearemos una nueva tabla separada para almacenar los datos de los usuarios que hicieron los cambios.
Quinto, cree la tabla UserChangeLogs:

```
CREATE TABLE UserChangeLogs (
    id INT AUTO_INCREMENT,
    productCode VARCHAR(15) DEFAULT NULL,
    updatedAt TIMESTAMP NOT NULL
        DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP,
    updatedBy VARCHAR(30) NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (productCode)
        REFERENCES products (productCode)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

Sexto, cree un desencadenador BEFORE UPDATE para la tabla de productos. Este disparador se activa después del disparador before_products_update.

```
DELIMITER $$

CREATE TRIGGER before_products_update_log_user
   BEFORE UPDATE ON products
   FOR EACH ROW
   FOLLOWS before_products_update
BEGIN
    IF OLD.msrp <> NEW.msrp THEN
        INSERT INTO
            UserChangeLogs(productCode,updatedBy)
        VALUES
            (OLD.productCode,USER());
    END IF;
END$$

DELIMITER ;
```

Hagamos una prueba rápida.
Séptimo, actualice el precio de un producto utilizando la siguiente instrucción UPDATE:

```
UPDATE
    products
SET
    msrp = 220
WHERE
    productCode = 'S12_1099';
```

Octavo, consultar datos de las tablas PriceLogs y UserChangeLogs:

```
SELECT * FROM PriceLogs;


SELECT * FROM UserChangeLogs;
```

Como puede ver, ambos disparadores se activaron en la secuencia como se esperaba.

Información sobre el orden de activación de los triggers

Si usa la instrucción SHOW TRIGGERS para mostrar los disparadores, no verá el orden en que los disparadores se activan para el mismo evento y tiempo de acción.

```
SHOW TRIGGERS
FROM classicmodels
WHERE `table` = 'products';
```

Para encontrar esta información, debe consultar la columna action_order en la tabla de desencadenadores de la base de datos information_schema de la siguiente manera:

```
SELECT
    trigger_name,
    action_order
FROM
    information_schema.triggers
```

```
WHERE
    trigger_schema = 'classicmodels'
ORDER BY
    event_object_table ,
    action_timing ,
    event_manipulation;
```