Chapter 19. Triggers

Table of Contents

A trigger is a named database object that is associated with a table and that is activated when a particular event occurs for the table. For example, the following statements create a table and an `INSERT` trigger. The trigger sums the values inserted into one of the table's columns:

```
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
Query OK, 0 rows affected (0.03 sec)

mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
    -> FOR EACH ROW SET @sum = @sum + NEW.amount;
Query OK, 0 rows affected (0.06 sec)
```

This chapter describes the syntax for creating and dropping triggers, and shows some examples of how to use them. Discussion of restrictions on use of triggers is given in Section J.1, "Restrictions on Stored Routines and Triggers". Remarks regarding binary logging as it applies to triggers are given in Section 18.4, "Binary Logging of Stored Routines and Triggers".

For answers to some common questions about triggers in MySQL 5.1, see Section A.5, "MySQL 5.1 FAQ — Triggers".

19.1. `CREATE TRIGGER` Syntax

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    TRIGGER trigger_name trigger_time trigger_event
    ON tbl_name FOR EACH ROW trigger_stmt
```

This statement creates a new trigger. A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table. Currently, `CREATE TRIGGER` requires the `TRIGGER` privilege for the table associated with the trigger. (This statement requires the `SUPER` privilege prior to MySQL 5.1.6.)

The trigger becomes associated with the table named *tbl_name*, which must refer to a permanent table. You cannot associate a trigger with a **TEMPORARY** table or a view.

When the trigger is activated, the **DEFINER** clause determines the privileges that apply, as described later in this section.

trigger_time is the trigger action time. It can be **BEFORE** or **AFTER** to indicate that the trigger activates before or after the statement that activated it.

trigger_event indicates the kind of statement that activates the trigger. The *trigger_event* can be one of the following:

- **INSERT**: The trigger is activated whenever a new row is inserted into the table; for example, through **INSERT**, **LOAD DATA**, and **REPLACE** statements.
- **UPDATE**: The trigger is activated whenever a row is modified; for example, through **UPDATE** statements.
- **DELETE**: The trigger is activated whenever a row is deleted from the table; for example, through **DELETE** and **REPLACE** statements. However, **DROP TABLE** and **TRUNCATE** statements on the table do *not* activate this trigger, because they do not use **DELETE**. Dropping a partition does not activate **DELETE** triggers, either. See Section 13.2.9, "**TRUNCATE** Syntax".

It is important to understand that the *trigger_event* does not represent a literal type of SQL statement that activates the trigger so much as it represents a type of table operation. For example, an **INSERT** trigger is activated by not only **INSERT** statements but also **LOAD DATA** statements because both statements insert rows into a table.

A potentially confusing example of this is the **INSERT INTO ... ON DUPLICATE KEY UPDATE ...** syntax: a **BEFORE INSERT** trigger will activate for every row, followed by either an **AFTER INSERT** trigger or both the **BEFORE UPDATE** and **AFTER UPDATE** triggers, depending on whether there was a duplicate key for the row.

There cannot be two triggers for a given table that have the same trigger action time and event. For example, you cannot have two **BEFORE UPDATE** triggers for a table. But you can have a **BEFORE UPDATE** and a **BEFORE INSERT** trigger, or a **BEFORE UPDATE** and an **AFTER UPDATE** trigger.

trigger_stmt is the statement to execute when the trigger activates. If you want to execute multiple statements, use the **BEGIN ... END** compound statement construct. This also enables you to use the same statements that are allowable within stored routines. See Section 18.2.5, "**BEGIN ... END** Compound Statement Syntax". Some statements are not allowed in triggers; see Section J.1, "Restrictions on Stored Routines and Triggers".

MySQL stores the **sql_mode** system variable setting that is in effect at the time a trigger is

created, and always executes the trigger with this setting in force, *regardless of the current server SQL mode*.

Note: Currently, triggers are not activated by cascaded foreign key actions. This limitation will be lifted as soon as possible.

In MySQL 5.1, you can write triggers containing direct references to tables by name, such as the trigger named `testref` shown in this example:

```
CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE test4(
  a4 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  b4 INT DEFAULT 0
);

DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;
|

DELIMITER ;

INSERT INTO test3 (a3) VALUES
  (NULL), (NULL), (NULL), (NULL), (NULL),
  (NULL), (NULL), (NULL), (NULL), (NULL);

INSERT INTO test4 (a4) VALUES
  (0), (0), (0), (0), (0), (0), (0), (0), (0), (0);
```

Suppose that you insert the following values into table `test1` as shown here:

```
mysql> INSERT INTO test1 VALUES
    -> (1), (3), (1), (7), (1), (8), (4), (4);
Query OK, 8 rows affected (0.01 sec)
Records: 8  Duplicates: 0  Warnings: 0
```

As a result, the data in the four tables will be as follows:

```
mysql> SELECT * FROM test1;
+------+
| a1   |
+------+
|    1 |
|    3 |
|    1 |
|    7 |
|    1 |
|    8 |
```

```
|    4 |
|    4 |
+------+
8 rows in set (0.00 sec)

mysql> SELECT * FROM test2;
+------+
| a2   |
+------+
|    1 |
|    3 |
|    1 |
|    7 |
|    1 |
|    8 |
|    4 |
|    4 |
+------+
8 rows in set (0.00 sec)

mysql> SELECT * FROM test3;
+----+
| a3 |
+----+
|  2 |
|  5 |
|  6 |
|  9 |
| 10 |
+----+
5 rows in set (0.00 sec)

mysql> SELECT * FROM test4;
+----+------+
| a4 | b4   |
+----+------+
|  1 |    3 |
|  2 |    0 |
|  3 |    1 |
|  4 |    2 |
|  5 |    0 |
|  6 |    0 |
|  7 |    1 |
|  8 |    1 |
|  9 |    0 |
| 10 |    0 |
+----+------+
10 rows in set (0.00 sec)
```

You can refer to columns in the subject table (the table associated with the trigger) by using the aliases **OLD** and **NEW**. **OLD.***col_name* refers to a column of an existing row before it is updated or deleted. **NEW.***col_name* refers to the column of a new row to be inserted or an existing row after it is updated.

The **DEFINER** clause specifies the MySQL account to be used when checking access privileges at trigger activation time. If a *user* value is given, it should be a MySQL account in

'user_name'@'host_name' format (the same format used in the **GRANT** statement). The *user_name* and *host_name* values both are required. **CURRENT_USER** also can be given as **CURRENT_USER()**. The default **DEFINER** value is the user who executes the **CREATE TRIGGER** statement. (This is the same as **DEFINER = CURRENT_USER**.)

If you specify the **DEFINER** clause, you cannot set the value to any account but your own unless you have the **SUPER** privilege. These rules determine the legal **DEFINER** user values:

- If you do not have the **SUPER** privilege, the only legal *user* value is your own account, either specified literally or by using **CURRENT_USER**. You cannot set the definer to some other account.
- If you have the **SUPER** privilege, you can specify any syntactically legal account name. If the account does not actually exist, a warning is generated.

  Although it is possible to create triggers with a non-existent **DEFINER** value, it is not a good idea for such triggers to be activated until the definer actually does exist. Otherwise, the behavior with respect to privilege checking is undefined.

Note: Prior to MySQL 5.1.6, MySQL requires the **SUPER** privilege for the use of **CREATE TRIGGER**, so only the second of the preceding rules applies. As of 5.1.6, **CREATE TRIGGER** requires the **TRIGGER** privilege and **SUPER** is required only to be able to set **DEFINER** to a value other than your own account.

MySQL checks trigger privileges like this:

- At **CREATE TRIGGER** time, the user that issues the statement must have the **TRIGGER** privilege. (**SUPER** prior to MySQL 5.1.6.)
- At trigger activation time, privileges are checked against the **DEFINER** user. This user must have these privileges:
  - The **TRIGGER** privilege. (**SUPER** prior to MySQL 5.1.6.)
  - The **SELECT** privilege for the subject table if references to table columns occur via **OLD.***col_name* or **NEW.***col_name* in the trigger definition.
  - The **UPDATE** privilege for the subject table if table columns are targets of **SET NEW.***col_name* = *value* assignments in the trigger definition.
  - Whatever other privileges normally are required for the statements executed by the trigger.

19.2. DROP TRIGGER Syntax

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

This statement drops a trigger. The schema (database) name is optional. If the schema is omitted, the trigger is dropped from the default schema. **DROP TRIGGER** was added in MySQL 5.0.2. Its use requires the **TRIGGER** privilege for the table associated with the trigger. (This statement requires the **SUPER** privilege prior to MySQL 5.1.6.)

Use **IF EXISTS** to prevent an error from occurring for a trigger that does not exist. A **NOTE** is generated for a non-existent trigger when using **IF EXISTS**. See Section 13.5.4.31, "**SHOW WARNINGS** Syntax". The **IF EXISTS** clause was added in MySQL 5.1.14.

Note: When upgrading from a version of MySQL older than MySQL 5.0.10 to 5.0.10 or newer — including all MySQL 5.1 releases — you must drop all triggers *before upgrading* and re-create them afterward, or else **DROP TRIGGER** does not work after the upgrade. See Section 2.11.1, "Upgrading from MySQL 5.0 to 5.1", for a suggested upgrade procedure.

19.3. Using Triggers

This section discusses how to use triggers in MySQL 5.1 and some limitations regarding their use. Additional information about trigger limitations is given in Section J.1, "Restrictions on Stored Routines and Triggers".

A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table. Some uses for triggers are to perform checks of values to be inserted into a table or to perform calculations on values involved in an update.

A trigger is associated with a table and is defined to activate when an **INSERT**, **DELETE**, or **UPDATE** statement for the table executes. A trigger can be set to activate either before or after the triggering statement. For example, you can have a trigger activate before each row that is deleted from a table or after each row that is updated.

To create a trigger or drop a trigger, use the **CREATE TRIGGER** or **DROP TRIGGER** statement. The syntax for these statements is described in Section 19.1, "**CREATE TRIGGER** Syntax", and Section 19.2, "**DROP TRIGGER** Syntax".

Here is a simple example that associates a trigger with a table for **INSERT** statements. It acts as an accumulator to sum the values inserted into one of the columns of the table.

The following statements create a table and a trigger for it:

```
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
    -> FOR EACH ROW SET @sum = @sum + NEW.amount;
```

The **CREATE TRIGGER** statement creates a trigger named **ins_sum** that is associated with

the `account` table. It also includes clauses that specify the trigger activation time, the triggering event, and what to do with the trigger activates:

- The keyword `BEFORE` indicates the trigger action time. In this case, the trigger should activate before each row inserted into the table. The other allowable keyword here is `AFTER`.
- The keyword `INSERT` indicates the event that activates the trigger. In the example, `INSERT` statements cause trigger activation. You can also create triggers for `DELETE` and `UPDATE` statements.
- The statement following `FOR EACH ROW` defines the statement to execute each time the trigger activates, which occurs once for each row affected by the triggering statement In the example, the triggered statement is a simple `SET` that accumulates the values inserted into the `amount` column. The statement refers to the column as `NEW.amount` which means "the value of the `amount` column to be inserted into the new row."

To use the trigger, set the accumulator variable to zero, execute an `INSERT` statement, and then see what value the variable has afterward:

```
mysql> SET @sum = 0;
mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);
mysql> SELECT @sum AS 'Total amount inserted';
+-----------------------+
| Total amount inserted |
+-----------------------+
| 1852.48               |
+-----------------------+
```

In this case, the value of `@sum` after the `INSERT` statement has executed is `14.98 + 1937.50 - 100`, or `1852.48`.

To destroy the trigger, use a `DROP TRIGGER` statement. You must specify the schema name if the trigger is not in the default schema:

```
mysql> DROP TRIGGER test.ins_sum;
```

Trigger names exist in the schema namespace, meaning that all triggers must have unique names within a schema. Triggers in different schemas can have the same name.

In addition to the requirement that trigger names be unique for a schema, there are other limitations on the types of triggers you can create. In particular, you cannot have two triggers for a table that have the same activation time and activation event. For example, you cannot define two `BEFORE INSERT` triggers or two `AFTER UPDATE` triggers for a table. This should rarely be a significant limitation, because it is possible to define a trigger that executes multiple statements by using the `BEGIN ... END` compound statement construct after `FOR EACH ROW`. (An example appears later in this section.)

The **OLD** and **NEW** keywords enable you to access columns in the rows affected by a trigger. (**OLD** and **NEW** are not case sensitive.) In an **INSERT** trigger, only **NEW.***col_name* can be used; there is no old row. In a **DELETE** trigger, only **OLD.***col_name* can be used; there is no new row. In an **UPDATE** trigger, you can use **OLD.***col_name* to refer to the columns of a row before it is updated and **NEW.***col_name* to refer to the columns of the row after it is updated.

A column named with **OLD** is read-only. You can refer to it (if you have the **SELECT** privilege), but not modify it. A column named with **NEW** can be referred to if you have the **SELECT** privilege for it. In a **BEFORE** trigger, you can also change its value with **SET NEW.***col_name* = *value* if you have the **UPDATE** privilege for it. This means you can use a trigger to modify the values to be inserted into a new row or that are used to update a row.

In a **BEFORE** trigger, the **NEW** value for an **AUTO_INCREMENT** column is 0, not the automatically generated sequence number that will be generated when the new record actually is inserted.

OLD and **NEW** are MySQL extensions to triggers.

By using the **BEGIN ... END** construct, you can define a trigger that executes multiple statements. Within the **BEGIN** block, you also can use other syntax that is allowed within stored routines such as conditionals and loops. However, just as for stored routines, if you use the `mysql` program to define a trigger that executes multiple statements, it is necessary to redefine the `mysql` statement delimiter so that you can use the `;` statement delimiter within the trigger definition. The following example illustrates these points. It defines an **UPDATE** trigger that checks the new value to be used for updating each row, and modifies the value to be within the range from 0 to 100. This must be a **BEFORE** trigger because the value needs to be checked before it is used to update the row:

```
mysql> delimiter //
mysql> CREATE TRIGGER upd_check BEFORE UPDATE ON account
    -> FOR EACH ROW
    -> BEGIN
    ->     IF NEW.amount < 0 THEN
    ->         SET NEW.amount = 0;
    ->     ELSEIF NEW.amount > 100 THEN
    ->         SET NEW.amount = 100;
    ->     END IF;
    -> END;//
mysql> delimiter ;
```

It can be easier to define a stored procedure separately and then invoke it from the trigger using a simple **CALL** statement. This is also advantageous if you want to invoke the same routine from within several triggers.

There are some limitations on what can appear in statements that a trigger executes when activated:

- The trigger cannot use the **CALL** statement to invoke stored procedures that return data to the client or that use dynamic SQL. (Stored procedures are allowed to return data to

the trigger through **OUT** or **INOUT** parameters.)

- The trigger cannot use statements that explicitly or implicitly begin or end a transaction such as **START TRANSACTION**, **COMMIT**, or **ROLLBACK**.

MySQL handles errors during trigger execution as follows:

- If a **BEFORE** trigger fails, the operation on the corresponding row is not performed.
- A **BEFORE** trigger is activated by the *attempt* to insert or modify the row, regardless of whether the attempt subsequently succeeds.
- An **AFTER** trigger is executed only if the **BEFORE** trigger (if any) and the row operation both execute successfully.
- An error during either a **BEFORE** or **AFTER** trigger results in failure of the entire statement that caused trigger invocation.
- For transactional tables, failure of a statement should cause rollback of all changes performed by the statement. Failure of a trigger causes the statement to fail, so trigger failure also causes rollback. For non-transactional tables, such rollback cannot be done, so although the statement fails, any changes performed prior to the point of the error remain in effect.