

## 27.3.1 Trigger Syntax and Examples

To create a trigger or drop a trigger, use the `CREATE TRIGGER` or `DROP TRIGGER` statement, described in Section 15.1.22, “CREATE TRIGGER Statement”, and Section 15.1.34, “DROP TRIGGER Statement”.

Here is a simple example that associates a trigger with a table, to activate for `INSERT` operations. The trigger acts as an accumulator, summing the values inserted into one of the columns of the table.

```
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 (0.01 sec)
```

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 action time, the triggering event, and what to do when the trigger activates:

- The keyword `BEFORE` indicates the trigger action time. In this case, the trigger activates before each row inserted into the table. The other permitted keyword here is `AFTER`.
- The keyword `INSERT` indicates the trigger event; that is, the type of operation that activates the trigger. In the example, `INSERT` operations cause trigger activation. You can also create triggers for `DELETE` and `UPDATE` operations.
- The statement following `FOR EACH ROW` defines the trigger body; that is, the statement to execute each time the trigger activates, which occurs once for each row affected by the triggering event. In the example, the trigger body is a simple `SET` that accumulates into a user variable 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;
```

If you drop a table, any triggers for the table are also dropped.

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.

It is possible to define multiple triggers for a given table that have the same trigger event and action time. For example, you can have two `BEFORE UPDATE` triggers for a table. By default, triggers that have the same trigger event and action time activate in the order they were created. To affect trigger order, specify a clause after `FOR EACH ROW` that indicates `FOLLOWS` or `PRECEDES` and the name of an existing trigger that also has the same trigger event and action time. With `FOLLOWS`, the new trigger activates after the existing trigger. With `PRECEDES`, the new trigger activates before the existing trigger.

For example, the following trigger definition defines another `BEFORE INSERT` trigger for the `account` table:

```
mysql> CREATE TRIGGER ins_transaction BEFORE INSERT ON account
      FOR EACH ROW PRECEDES ins_sum
      SET
        @deposits = @deposits + IF(NEW.amount>0,NEW.amount,0),
        @withdrawals = @withdrawals + IF(NEW.amount<0,-NEW.amount,0);
Query OK, 0 rows affected (0.01 sec)
```

This trigger, `ins_transaction`, is similar to `ins_sum` but accumulates deposits and withdrawals separately. It has a `PRECEDES` clause that causes it to activate before `ins_sum`; without that clause, it would activate after `ins_sum` because it is created after `ins_sum`.

Within the trigger body, the `OLD` and `NEW` keywords enable you to access columns in the rows affected by a trigger. `OLD` and `NEW` are MySQL extensions to triggers; they 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. You can refer to a column named with `NEW` 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 used to update a row. (Such a `SET` statement has no effect in an `AFTER` trigger because the row change has already occurred.)

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

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 permitted 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 must 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 execute the same code from within several triggers.

There are 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 permitted 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`. (`ROLLBACK TO SAVEPOINT` is permitted because it does not end a transaction.).

See also Section 27.8, “Restrictions on Stored Programs”.

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 any `BEFORE` triggers and the row operation 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 nontransactional tables, such rollback cannot be done, so although the statement fails, any changes performed prior to the point of the error remain in effect.

Triggers can contain 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 four tables contain the following data:

```
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)
```