15.1.22 CREATE TRIGGER Statement

```
CREATE
    [DEFINER = user]
    TRIGGER [IF NOT EXISTS] trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    [trigger_order]
    trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

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. 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.

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.

`IF NOT EXISTS` prevents an error from occurring if a trigger having the same name, on the same table, exists in the same schema. This option is supported with `CREATE TRIGGER` beginning with MySQL 8.0.29.

This section describes `CREATE TRIGGER` syntax. For additional discussion, see Section 27.3.1, “Trigger Syntax and Examples”.

`CREATE TRIGGER` requires the `TRIGGER` privilege for the table associated with the trigger. If the `DEFINER` clause is present, the privileges required depend on the ***`user`*** value, as discussed in Section 27.6, “Stored Object Access Control”. If binary logging is enabled, `CREATE TRIGGER` might require the `SUPER` privilege, as discussed in Section 27.7, “Stored Program Binary Logging”.

The `DEFINER` clause determines the security context to be used when checking access privileges at trigger activation time, 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 each row to be modified.

Basic column value checks occur prior to trigger activation, so you cannot use `BEFORE` triggers to convert values inappropriate for the column type to valid values.

`trigger_event` indicates the kind of operation that activates the trigger. These ***`trigger_event`*** values are permitted:

- `INSERT`: The trigger activates whenever a new row is inserted into the table (for example, through `INSERT`, `LOAD DATA`, and `REPLACE` statements).
- `UPDATE`: The trigger activates whenever a row is modified (for example, through `UPDATE` statements).
- `DELETE`: The trigger activates whenever a row is deleted from the table (for example, through `DELETE` and `REPLACE` statements). `DROP TABLE` and `TRUNCATE TABLE` statements on the table do *not* activate this trigger, because they do not use `DELETE`. Dropping a partition does not activate `DELETE` triggers, either.

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 activates not only for `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 activates 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.

> **Note**
>
> Cascaded foreign key actions do not activate triggers.

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 ***`trigger_order`*** clause 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.

`trigger_body` is the statement to execute when the trigger activates. To execute multiple statements, use the `BEGIN ... END` compound statement construct. This also enables you to use the same statements that are permitted within stored routines. See Section 15.6.1, “BEGIN ... END Compound Statement”. Some statements are not permitted in triggers; see Section 27.8, “Restrictions on Stored Programs”.

Within the trigger body, 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.

Triggers cannot use `NEW.`***`col_name`*** or use `OLD.`***`col_name`*** to refer to generated columns. For information about generated columns, see Section 15.1.20.8, “CREATE TABLE and Generated Columns”.

MySQL stores the `sql_mode` system variable setting in effect when a trigger is created, and always executes the trigger body with this setting in force, *regardless of the current server SQL mode when the trigger begins executing*.

The `DEFINER` clause specifies the MySQL account to be used when checking access privileges at trigger activation time. If the `DEFINER` clause is present, the ***`user`*** value should be a MySQL account specified as `'`***`user_name`***`'@'`***`host_name`***`'`, `CURRENT_USER`, or `CURRENT_USER()`. The permitted ***`user`*** values depend on the privileges you hold, as discussed in Section 27.6, “Stored Object Access Control”. Also see that section for additional information about trigger security.

If the `DEFINER` clause is omitted, the default definer is the user who executes the `CREATE TRIGGER` statement. This is the same as specifying `DEFINER = CURRENT_USER` explicitly.

MySQL takes the `DEFINER` user into account when checking trigger privileges as follows:

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

Within a trigger body, the `CURRENT_USER` function returns the account used to check privileges at trigger activation time. This is the `DEFINER` user, not the user whose actions caused the trigger to be activated. For information about user auditing within triggers, see Section 8.2.23, “SQL-Based Account Activity Auditing”.

If you use `LOCK TABLES` to lock a table that has triggers, the tables used within the trigger are also locked, as described in LOCK TABLES and Triggers.

For additional discussion of trigger use, see Section 27.3.1, “Trigger Syntax and Examples”.