Databases - Triggers and Views

Gordon Royle

School of Mathematics & Statistics
University of Western Australia

Triggers

Triggers are small procedural programs – stored routines – written in SQL, that are run automatically when specified *events* (such as rows being added, deleted or updated) occur on a particular table.

Triggers can be used:

- to perform stronger, or more complex and adaptive, integrity checks than are possible with the inbuilt mechanisms,
- to automatically perform routine operations, such as logging, tallying, or notification that should occur as a consequence of the original operation,
- to add additional functionality and/or overcome some of the limitations in any particular implementation of SQL.

There is a *wide variation* between the SQL standard and particular implementations, such as MySQL.

Basic syntax

In MySQL, the basic syntax is

```
CREATE TRIGGER name time event
ON table
FOR EACH ROW
BEGIN
...
END
```

Here, `name`, `time`, `event` and `table` are to be replaced with appropriate keywords or values.

What options?

Clearly `name` and `table` are replaced with an appropriate name (for the trigger) and the table to which the trigger is attached, while the other two values can take the following values:

- `time`
  Either `BEFORE` or `AFTER`
- `event`
  Either `INSERT`, `UPDATE` or `DELETE`

(MySQL does not implement `INSTEAD OF` which is in the SQL standard, but we can simulate this behaviour if necessary.)

Reading the docs

MySQL 5.7 Reference Manual :: 13 SQL Statement Syntax :: 13.1 Data Definition Statements :: 13.1.15 CREATE TRI

13.1.15 CREATE TRIGGER Syntax

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    TRIGGER 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.

What this means

SQL watches out for the specified event (`INSERT`, `UPDATE` or `DELETE`) on the specified table, and then runs the trigger either `BEFORE` or `AFTER` the action.

The trigger can do pretty much anything — prevent the event from happening, modify the event, record the event in another table, ensure that the action is legitimate — to ensure that everything in the database is in a consistent state.

Row-level only

The SQL standard specifies two "levels" of trigger

- Statement-level trigger
  Trigger runs once for each relevant type of *statement*, no matter how many rows it affects.
- Row-level trigger
  Trigger runs once for each affected *row*, maybe thousands of times if there are thousands of rows.

MySQL documentation shows `FOR EACH ROW` as compulsory, because MySQL only implements *row-level triggers* and not *statement-level triggers*.

Access to rows

In order to take effective action, the trigger needs to know

- The state of the row *before* the (proposed) event
- The state of the row *after* the (proposed) event,

These values are provided to the trigger as two tuples, called `NEW` and `OLD`.

NEW and OLD

- A tuple called NEW is available for any INSERT or UPDATE trigger
  This tuple contains the *new values* that the row will contain after the insertion or update.
- A tuple called OLD is available for any UPDATE or DELETE trigger
  This tuple contains the *old values* that the row did contain after the update or deletion.

For an UPDATE statement, OLD contains the row before the (proposed) update, and NEW contains the row after the (proposed) update.
For an INSERT statement, there is only an NEW row, and for a DELETE statement, there is only an OLD row.

Temporarily change delimiters

To define triggers, we need to do the same "delimiter gymnastics" as when defining all stored routines.

```
DELIMITER ++
CREATE TRIGGER name time event
ON table
FOR EACH ROW
BEGIN

A whole bunch of MySQL statements
each terminated with the usual semicolon

END++
DELIMITER ;
```

The first line temporarily changes the delimiter to ++, then the entire procedure is entered, and finally the delimiter is changed back again.

A log file

Suppose we have a table representing bank accounts

```
CREATE TABLE BankAccount(
  id INT PRIMARY KEY,
  name VARCHAR(64),
  balance REAL);
```

Now we'll make a table that will log banking transactions

```
CREATE TABLE BankAccountLog(
  id INT,
  time DATETIME,
  event ENUM('D', 'W'),
  preBalance REAL,
  postBalance REAL);
```

Adding triggers

```
DELIMITER ++

CREATE TRIGGER banklog
AFTER UPDATE
ON BankAccount
FOR EACH ROW
BEGIN
 INSERT INTO BankAccountLog
  VALUES(NEW.id, NOW(), 'D', OLD.balance, NEW.balance);
END++

DELIMITER ;
```

Notice the use of the built-in function `NOW()` to populate the attribute that has type `DATETIME`.

Adding logic

```
CREATE TRIGGER banklog
AFTER UPDATE
ON BankAccount
FOR EACH ROW
BEGIN
  IF NEW.balance > OLD.balance THEN
   INSERT INTO BankAccountLog
    VALUES(NEW.id, NOW(), 'D' ,OLD.balance,NEW.balance);
  ELSE
   INSERT INTO BankAccountLog
    VALUES(NEW.id, NOW(), 'W' ,OLD.balance,NEW.balance);
  END IF;
END
```

Adding variables

```
CREATE TRIGGER banklog
AFTER UPDATE
ON BankAccount
FOR EACH ROW
BEGIN
 DECLARE which CHAR;

 IF NEW.balance > OLD.balance THEN
  SET which = 'D';
 ELSE
  SET which = 'W';
 END IF;

 INSERT INTO BankAccountLog
      VALUES(NEW.id, NOW(),which,OLD.balance,NEW.balance);
END
```

Declare the variable

```
DECLARE which CHAR;
```

This statement *declares* a variable, whose name is `which` and that is of type `CHAR`.

The variable can be assigned a value, and then used whenever a `CHAR` is expected.

Assign a value to the variable

```
IF NEW.balance > OLD.balance THEN
 SET which = 'D';
ELSE
 SET which = 'W';
END IF;
```

This statement *compares* the new balance with the old balance, and assigns either 'D' or 'W' to the variable `which` depending on whether the change of balance constitutes a deposit or a withdrawal.

Use the variable

```
INSERT INTO BankAccountLog
     VALUES(NEW.id, NOW(),which,OLD.balance,NEW.balance);
```

Finally the variable is *used* in the `INSERT` statement into the log file.

Checking

The SQL standard specifies a type of data integrity constraint called a `CHECK` that is specified at the `CREATE TABLE` stage.

```
CREATE TABLE BankAccount (
 accountNumber INT,
 balance REAL CHECK (balance > -1000));
```

This will simply prevent any operation that would result in the balance dropping below -999 (perhaps this is the overdraft limit for this account).

However, this is *not implemented* in MySQL, which kindly says:

> The `CHECK` clause is parsed but ignored by all storage engines.

As a trigger

```
CREATE TRIGGER checkBalance
BEFORE UPDATE
ON BankAccount
FOR EACH ROW
BEGIN
IF NEW.balance <= -1000 THEN
  SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Balance too low';
END IF;
END
```

How it works

All of this trigger is familiar except the

```
SIGNAL SQLSTATE '45000'
  SET MESSAGE_TEXT = 'Balance too low';
```

The `SIGNAL` mechanism is used to deal with errors, halting execution and reporting a value and/or message back to the client responsible for the statement that generated the signal.

The value `45000` is a standard value that represents a "user-defined exception".

```
mysql> UPDATE BankAccount SET balance = -1500;
ERROR 1644 (45000): Balance too low
```

Question

Suppose that the `BankAccount` table has another column

```
CREATE TABLE BankAccount2(
  id INT PRIMARY KEY,
  name VARCHAR(64),
  balance REAL,
  minimumBalance REAL);
```

The new column should store the *overall minimum* that the balance has ever attained during a sequence of operations, where every operation has the following form:

```
UPDATE BankAccount2
SET balance = balance + ...
WHERE id = ...
```

Write a trigger to accomplish this.

Hints

- An update trigger cannot perform an `UPDATE` command on the same table
- The values of `OLD` are read-only, but the values of `NEW` can be both read *and written*, but only in a `BEFORE` trigger.

Why are these two restrictions necessary?

Views

A *view* is a named "virtual table" defined using a stored SQL `SELECT` query.

The *schema* of the view is determined by the types of the selected columns, and the *contents* of the view at any given moment are determined by running the SQL `SELECT` query.

For querying purposes, a view acts as though it is a regular table, although "under the hood", any query to the view is translated into an equivalent query on the underlying tables.

Jennifer on views

See

`https://www.youtube.com/watch?v=UcS2GSK3jaY`

`https://www.youtube.com/watch?v=jMgRah-2dr8`

for Jennifer Widom's recordings on the subject of *views*.

Why views?

The structure of a database is determined by a number of different factors:

- The logical design of the application
- Schema normalization and redundancy removal
- Security and access control
- Efficiency

This can lead to a structure that is not necessarily easy to *use*, or not necessarily easy for an "end-user" to use.

Why views 2

Views allow the DBA to define *customised tables* for each type of user — except that they are not really tables, but rather "virtual objects" that *behave* just like tables.

This additional layer of abstraction allows everyone to be happy - the user has easy-to-use tables that contain just the information they want (including redundant information), while the DBA has an optimised database that is easy to maintain.

As with all *abstraction layers*, an additional benefit is that the underlying *implementation* can be changed, and by simply changing the views, the end-users need not change any of their queries.

Example

Consider the following schema:

```
Student (id INT PRIMARY KEY, name VARCHAR(64))
Unit (code CHAR(8) PRIMARY KEY, uname VARCHAR(128))

CREATE TABLE Enrolled (
  sid INT,
  ucode VARCHAR(8),
  mark INT,
  FOREIGN KEY (sid)
    REFERENCES Student(id) ON UPDATE CASCADE,
  FOREIGN KEY (ucode)
    REFERENCES Unit(code) ON UPDATE CASCADE);
```

Create a view

```
CREATE VIEW AcademicRecord AS
SELECT id, name, COUNT(mark), AVG(mark) AS avg FROM
Student JOIN Enrolled ON Student.id = Enrolled.sid
GROUP BY id;
```

This creates a "summary line" for each student, just listing their name along with the number of units they have taken and the average mark they attained.

Using the view

```
mysql> select * from student;
+-----+-------+
| id  | name  |
+-----+-------+
| 123 | Amy   |
| 456 | Bill  |
| 567 | Chloe |
| 899 | Dan   |
+-----+-------+
mysql> select * from enrolled;
+------+----------+------+
| sid  | ucode    | mark |
+------+----------+------+
|  123 | CITS1402 |   80 |
|  456 | CITS1402 |   25 |
|  899 | CITS1402 |   65 |
|  123 | CITS2211 |   90 |
+------+----------+------+
```

Using the view

```
mysql> select * from AcademicRecord;
+-----+------+--------------+-----------+
| id  | name | COUNT(mark)  | avg |
+-----+------+--------------+-----------+
| 123 | Amy  |            2 |   85.0000 |
| 456 | Bill |            1 |   25.0000 |
| 899 | Dan  |            1 |   65.0000 |
+-----+------+--------------+-----------+
3 rows in set (0.00 sec)
```

Another example

From *Classic Models* the fact that an order comprises several rows from `orderdetails` makes it awkward to deal with orders and their prices. We can avoid this problem by defining a view

```
CREATE view pricedOrder AS
SELECT   ordernumber,
         SUM(priceeach * quantityordered) AS totalprice
FROM     orderdetails
GROUP BY ordernumber;
```

A new table

```
SELECT * from pricedOrder;
+-------------+------------+
| orderNumber | totalPrice |
+-------------+------------+
|       10100 |   10223.83 |
|       10101 |   10549.01 |
|       10102 |    5494.78
```

It looks and behaves just like a table with the same name — except that behind the scenes, MySQL is quickly rewriting every query into an equivalent query on the original tables.

A new table

A view can do pretty much anything a normal table can

```
SELECT orderDate,
       orderNumber,
       totalPrice
FROM   orders
       NATURAL JOIN pricedOrder;
+------------+-------------+------------+
| orderDate  | orderNumber | totalPrice |
+------------+-------------+------------+
| 2003-01-06 |       10100 |   10223.83 |
| 2003-01-09 |       10101 |   10549.01 |
| 2003-01-10 |       10102 |    5494.78 |
```

It looks and behaves just like a table with the same name — except that behind the scenes, MySQL is quickly rewriting every query into an equivalent query on the original tables.

More view terminology

There are two additional terms important in the discussion of views

- Updatable views
  A view is *updatable* if you can use it to alter the underlying tables
- Materialized views
  A view is *materialized* if the results of the query it represents have actually been *stored* in the database

Updatable views

A view is *updatable* if you can use `DELECT`, `INSERT` and `UPDATE` statements on the *view* and have the *underlying tables* altered in such a way that the view is changed in a manner consistent with the operation.

For a simple example, consider

```
CREATE VIEW marksOnly AS
SELECT sid, mark
FROM Enrolled;
```

This just pulls two fields out of the `Enrolled` table.

What should this statement do?

```
DELETE FROM marksOnly WHERE sid = 123;
```

What should this statement do?

```
DELETE FROM marksOnly WHERE sid = 123;
```

The only way to change the underlying tables so that the view no longer "sees" the student 123, is to delete those rows from `Enrolled` and so this is what happens.

Not updatable

On the other hand, the statement

```
UPDATE AcademicRecord
SET avg = 90
WHERE id = 123;
```

cannot be interpreted — there is no unique way to change the marks for Amy in order to get an average of 90.

Materialized views

Suppose a view represents a very complex query on some data that changes infrequently, but is queried often.

Then there may be advantages in actually storing the *results* of the query in the database, rather than re-running the very complex query every time a query is made.

This is called a *materialized view*, because the "virtual table" is now actually present - it has materialized out of thin air.

MySQL does not support materialized views, although they can be simulated in various ways — for example, having an *actual table* for the view, and using triggers to make sure it is always up-to-date.