MySQL Tutorial 3: Triggers and Automation

Getting to know the MySQL database engine

— P. Lutus — Message Page —

(double-click any word to see its definition)

— Navigate this multi-page article with the arrows and drop-down lists at the top and bottom of each page —

Note: In this article, footnotes are marked with a light bulb over which one hovers .

Introduction

MySQL has the ability to automatically process records that are inserted into, or updated within, a table. An example might be a table of commercial transactions, where a record might be entered describing a purchase: a quantity of widgets at a particular price. At that point, a MySQL trigger would begin an automatic process in which the associated code computes the remainder of the record's fields: a quantity discount if applicable, sales tax, and so forth.

Another example might be the automatic generation of a table with numeric values automatically inserted, from an original table without such numbers. MySQL is not supposed to be able to do this, but if certain procedures are followed, it can.

Both the above examples are shown in this article, and example code is provided. Because of the complexity of the examples, downloadable MySQL code files are provided to set up the demonstrations.

Automatic Invoice Computation

In this demonstration, a table is created that represents an invoice, and the table includes a trigger and associated code to automatically update the table on each entered purchase. Each record in the table contains:

- Provided by the user:
  - The item name or description
  - The quantity purchased
  - The purchase price
- Automatically computed by the trigger code:
  - A quantity discount field that uses the entered quantity to select a discount
  - A pre-tax subtotal for those who need before-tax amounts to be tallied separately
  - A sales tax amount that applies local sales tax rates to the purchase amount
  - A subtotal for the present record
  - A running total for the invoice as a whole (all records in the table)

Again, because of this example's complexity and unlike the earlier examples, we won't be typing in the code this time. Instead we will use a downloadable MySQL code file to set things up, and the detailed description includes user interaction with the table and its trigger code. Let's get started.

First steps:

- This demonstration relies on this file: invoice_trigger_demonstration.sql
- By clicking the link above, you can view the MySQL code that creates the table and provides the trigger actions
- By right-clicking the link and choosing "Save As ..." or "Save Link As ...", depending on your browser, you can download the MySQL code file.
- After downloading the file, put a copy in a directory convenient to the MySQL command-line interpreter.

- Now let MySQL read the file:

```
mysql> source (path)/invoice_trigger_demonstration.sql;
```

- Let's examine the "purchases" table structure:

```
mysql> describe purchases;

+----------+---------------+------+-----+---------+----------------+
| Field    | Type          | Null | Key | Default | Extra          |
+----------+---------------+------+-----+---------+----------------+
| Item     | text          | NO   |     | NULL    |                |
| Quan     | int(11)       | NO   |     | NULL    |                |
| Price    | decimal(10,2) | NO   |     | NULL    |                |
| Disc %   | decimal(10,2) | YES  |     | NULL    |                |
| PreTax   | decimal(10,2) | YES  |     | NULL    |                |
| Tax      | decimal(10,2) | YES  |     | NULL    |                |
| Subtotal | decimal(10,2) | YES  |     | NULL    |                |
| Total    | decimal(10,2) | YES  |     | NULL    |                |
| pk       | int(11)       | NO   | PRI | NULL    | auto_increment |
+----------+---------------+------+-----+---------+----------------+
```

- Now let's examine the table itself (I've included some sample records in the invoice_trigger_demonstration.sql source file):

```
mysql> select * from purchases;

+------------+------+-------+--------+--------+-------+----------+--------+----+
| Item       | Quan | Price | Disc % | PreTax | Tax   | Subtotal | Total  | pk |
+------------+------+-------+--------+--------+-------+----------+--------+----+
| Gadgets    |   10 |  5.98 |  10.00 |  59.70 |  5.07 |    64.77 |  64.77 |  1 |
| Widgets    |    8 |  4.33 |   0.00 |  34.64 |  2.94 |    37.58 | 102.35 |  2 |
| Tools      |   16 | 32.88 |  10.00 | 525.98 | 44.71 |   570.69 | 673.04 |  3 |
| Hardware   |   34 |  2.45 |  10.00 |  83.20 |  7.07 |    90.27 | 763.31 |  4 |
| Mousetraps |   22 |  5.93 |  10.00 | 130.36 | 11.08 |   141.44 | 904.75 |  5 |
+------------+------+-------+--------+--------+-------+----------+--------+----+
```

Now let's add a new record and see what happens:

```
mysql> insert into purchases (Item,Quan,Price) values('Cat Food',20,2.55);
mysql> select * from purchases;

+------------+------+-------+--------+--------+-------+----------+--------+----+
| Item       | Quan | Price | Disc % | PreTax | Tax   | Subtotal | Total  | pk |
+------------+------+-------+--------+--------+-------+----------+--------+----+
| Gadgets    |   10 |  5.98 |  10.00 |  59.70 |  5.07 |    64.77 |  64.77 |  1 |
| Widgets    |    8 |  4.33 |   0.00 |  34.64 |  2.94 |    37.58 | 102.35 |  2 |
| Tools      |   16 | 32.88 |  10.00 | 525.98 | 44.71 |   570.69 | 673.04 |  3 |
| Hardware   |   34 |  2.45 |  10.00 |  83.20 |  7.07 |    90.27 | 763.31 |  4 |
| Mousetraps |   22 |  5.93 |  10.00 | 130.36 | 11.08 |   141.44 | 904.75 |  5 |
| Cat Food   |   20 |  2.55 |  10.00 |  50.90 |  4.33 |    55.23 | 959.98 |  6 |
+------------+------+-------+--------+--------+-------+----------+--------+----+
```

The advantages to this kind of automation should be obvious:

- The record-completing computations are conceptually located very close to the source of the data — the data table itself.
- The computations are carried out on every insertion or update, without any explicit action required by the table's user.
- All queries on the table (and all views) include the computed values along with the user-supplied values for each record.

But there's one drawback to this specific trigger — if a record is deleted, the rightmost "Total" column becomes wrong. Here's an example:

```
mysql> delete from purchases where pk = 3;
mysql> select * from purchases;

+------------+------+-------+--------+--------+-------+----------+--------+----+
| Item       | Quan | Price | Disc % | PreTax | Tax   | Subtotal | Total  | pk |
+------------+------+-------+--------+--------+-------+----------+--------+----+
| Gadgets    |   10 |  5.98 |  10.00 |  59.70 |  5.07 |    64.77 |  64.77 |  1 |
| Widgets    |    8 |  4.33 |   0.00 |  34.64 |  2.94 |    37.58 | 102.35 |  2 |
| Hardware    |   34 |  2.45 |  10.00 |  83.20 |  7.07 |    90.27 | 763.31 |  4 |
| Mousetraps  |   22 |  5.93 |  10.00 | 130.36 | 11.08 |   141.44 | 904.75 |  5 |
| Cat Food    |   20 |  2.55 |  10.00 |  50.90 |  4.33 |    55.23 | 959.98 |  6 |
+------------+------+-------+--------+--------+-------+----------+--------+----+
```

The totals from row 4 onward are now in error. The reason should be obvious — the automated subtotals are limited to a single row, as insertions and updates trigger the bookkeeping code. But if a row is deleted, the subtotals aren't updated to reflect the deletion because the deletion doesn't produce a table-wide recalculation of the totals.

This is an important issue with this kind of automation — it's a problem because MySQL triggers can only change values in the row that produced the trigger. The reason for that is a bit subtle — if trigger code could change the content of rows other than the originating row, those changes would set off more triggers, and the system would go into an infinite recursion of triggers and lock up.

But there are solutions for this kind of problem:

- Drop the a total column from this table — add a total column later, and perform the total before displaying and using the completed invoice.
- Instead of relying on a row trigger to create an accurate total, issue a periodic or event-driven update to the invoice while it's being edited, an update like this:

```
mysql> update purchases set Total = 0;
mysql> select * from purchases;

+------------+------+-------+--------+--------+-------+----------+--------+----+
| Item       | Quan | Price | Disc % | PreTax | Tax   | Subtotal | Total  | pk |
+------------+------+-------+--------+--------+-------+----------+--------+----+
| Gadgets    |   10 |  5.98 |  10.00 |  59.70 |  5.07 |    64.77 |  64.77 |  1 |
| Widgets    |    8 |  4.33 |   0.00 |  34.64 |  2.94 |    37.58 | 102.35 |  2 |
| Hardware   |   34 |  2.45 |  10.00 |  83.20 |  7.07 |    90.27 | 192.62 |  4 |
| Mousetraps |   22 |  5.93 |  10.00 | 130.36 | 11.08 |   141.44 | 334.06 |  5 |
| Cat Food   |   20 |  2.55 |  10.00 |  50.90 |  4.33 |    55.23 | 389.29 |  6 |
+------------+------+-------+--------+--------+-------+----------+--------+----+
```

The reason the above works is because the chosen "update" command updates all the rows, in order, from bottom to top. But this specific update command cannot be made part of our row trigger, for reasons given above — it has to originate elsewhere.

Conclusion? This is a nice kind of automation, it takes care of some important bookkeeping on the fly, but one can try to do too much in code that can affect only one row of a table.

Session Variables

Views are a nice feature, but there are some things they cannot do. One limitation is shown in this example:

```
mysql> create or replace view `MyView` as select @index as index, Name, Address from mytable;
ERROR 1351 (HY000): View's SELECT contains a variable or parameter
```

It seems my having included the session variable "@index" invalidates the view definition. I don't actually know the reason for this prohibition — I think it may have to do with the fact that there's no assurance that a given numerical value can be reliably associated with a record, if there are differences in how a query is designed and executed. But that's just a guess — and in any case, if you think of a way to include them, session variables works just fine in views.

Before we move on, a brief digression into how session variables work:

```
mysql> set @x = 2;
mysql> select @x;

+------+
| @x   |
+------+
|    2 |
+------+
mysql> select sqrt(@x);

+--------------------+
| sqrt(@x)           |
+--------------------+
| 1.4142135623730951 |
+--------------------+
mysql> select pow(@x ,32);

+-------------+
| pow(@x ,32) |
+-------------+
|  4294967296 |
+-------------+
```

A session variable (a variable whose name begins with "@") remains in existence as long as the session (from user logon to logoff) does, so it can be used for procedures that require some degree of persistence. And there's one more thing to understand about session variables:

```
mysql> select @y;

+------+
| @y   |
+------+
| NULL |
+------+
```

The meaning of this example is that all session variables exist implicitly, they just have a default value of NULL if they haven't been assigned another value. This is important to understand when a procedure that uses session variables isn't behaving as it should — the first thing to check is whether a session variable is being used before being assigned a value.

Including Numbers in Views

To organize my fairly large DVD collection, I have a two-tiered system. The key to the system is a master table containing all the DVD names and associated information. The physical DVDs are stored in book-like organizers, each page of which has four DVD pockets. Parenthetically, *I don't recommend this storage method to anyone* — occasionally I ruin a DVD by storing it this way (they scratch too easily), and as to Blu-Ray disks, this storage method is an unmitigated disaster (Blu-Ray disks are much more sensitive to scratching). But this is just an example of a database problem, not at all a recommendation for storing media.

My goal is to have a system in which the organizer data tables are automatically updated whenever the master DVD list is updated. The organizer tables contain titles and comments for the DVDs assigned to that organizer, plus a page and pocket number for each DVD (again, each page has pockets for four DVDs). Ideally, when a new record is entered into the master list, the subsidiary tables/views would be automatically updated. As it turns out, this is easier said than done.

For each DVD, the master list contains a title, a comment, an organizer assignment code, and a few other things. A master list record looks like this:

```
mysql> select * from master_dvd_list where Title='XXX';

+-------+------------+--------+------+---------------------+-----+
| Title | Comment    | BluRay | Org  | LastModified        | pk  |
+-------+------------+--------+------+---------------------+-----+
| XXX   | Vin Diesel | n      | 4    | 2012-09-17 01:15:50 | 748 |
+-------+------------+--------+------+---------------------+-----+
```

Pretty simple, nothing fancy. Now the interesting part — the second tier. In the second tier are tables for each DVD organizer, with much the same information as the master DVD list, but with "Page" and "Pocket" fields to locate a particular DVD. Like this:

```
mysql> select * from dvd4_list where Title='XXX';

+-------+------------+--------+------+--------+---------------------+
| Title | Comment    | BluRay | Page | Pocket | LastModified        |
+-------+------------+--------+------+--------+---------------------+
| XXX   | Vin Diesel | n      |   13 |      4 | 2012-09-17 01:15:50 |
+-------+------------+--------+------+--------+---------------------+
```

In the original system, I would enter a new DVD into the master list, then run a Python program that would laboriously replace all the subsidiary tables with content generated by an algorithm to create the page and pocket numbers, and insert completed records into an empty table. Crude and cumbersome.

As time passed and I learned more about MySQL, it occurred to me that I could create views, one per organizer, that when invoked would read the master table and create a result for a specific organizer. The views would automatically update if the master table did, so there would be no need for a housekeeping program to create the organizer tables. This was all easily accomplished except for the page and pocket fields — they have numeric values, and as the above example shows, MySQL prevents use of session variables in views.

So after some thought and experimentation, I have a practical solution — I have a way to automatically generate the individual organizer tables on demand, without a need to perform any kind of global update after adding records to the master list. And the subsidiary tables have page and pocket numbers generated along with the view's query result. Here's how I did it:

- First, in a MySQL script (click here to see the script) I write a view definition that calls a function for certain column contents, and calls another function for the desired organizer identifier:

```
create or replace algorithm = temptable view dvd_generic_view as
  select Title,Comment,BluRay, row_index(0) as Page,row_index(1) as Pocket,
  LastModified from master_dvd_list where Org = get_org() order by Title;
```

- The functions `row_index()` and `get_org()` are part of the script. `row_index()` provides two different forms of the row number, depending on the argument provided to the function:

```
-- return a row index for page and pocket calculations

 CREATE function row_index(mode int) returns int
 BEGIN
   if(mode = 0)
   then -- need increment and page value
     set @rowindex = @rowindex + 1;
     return floor(@rowindex/4) + 1;
   else -- need pocket value
     return mod(@rowindex,4) + 1;
   end if;
 END|

-- return the current org definition

 CREATE function get_org() returns text
 BEGIN
   return @org;
 END|
```

  There is one more function that sets initial values — for the desired organizer, and the row counter:

```
CREATE function setup(org text) returns text
 BEGIN
   set @org = org;
   set @rowindex = -1;
   return org;
 END|
```

- Now for the hackiest part of this scheme — actually invoking the view:

```
mysql> select y.* from (select setup(4)) x join dvd_generic_view y where Title = 'XXX';

+-------+------------+--------+------+--------+---------------------+
| Title | Comment    | BluRay | Page | Pocket | LastModified        |
+-------+------------+--------+------+--------+---------------------+
| XXX   | Vin Diesel | n      |   13 |      4 | 2012-09-17 01:15:50 |
+-------+------------+--------+------+--------+---------------------+
```

- If I want an additional field that identifies the organizer, I can instead say:

```
mysql> select * from (select setup(4) as Org) x join dvd_generic_view y where Title = 'XXX';

+------+-------+------------+--------+------+--------+---------------------+
| Org  | Title | Comment    | BluRay | Page | Pocket | LastModified        |
+------+-------+------------+--------+------+--------+---------------------+
| 4    | XXX   | Vin Diesel | n      |   13 |      4 | 2012-09-17 01:15:50 |
+------+-------+------------+--------+------+--------+---------------------+
```

- A few words of explanation:
  - The above is a very hacky solution to get around a limitation on views that, based on how well this method works, probably doesn't need to exist.
  - The invocation line —

    ```
    select * from (select setup(4) as Org) x join dvd_generic_view y where Title = 'XXX';
    ```

    — works by creating two temporary tables (x and y), "join"s them as a single result in such a way that the `setup()` function is only called once, and allows the user to choose whether to display the organizer field.
  - The `setup()` function takes an argument that identifies the desired organizer content. This is used by the view query to select only the desired records from the master list.
  - The view code uses a special argument "algorithm = temptable" to make sure the view creates and returns a complete table. If this isn't done, and if a WHERE clause is then used to select specific records as in my example, the row numbering won't come out right.

This is a rather hacky method that (a) works just fine for small tables, but (b) would probably be a disaster with a large database, because of all the inefficient things it does to achieve its result. But it does show a way to use numeric values in a view, and it is completely reliable. This makes me wonder whether the prohibition against using numeric values in views really needs to exist.

Topical Links

- MySQL: Using Triggers
- MySQL: Restrictions on Views

— Navigate this multi-page article with the arrows and drop-down lists at the top and bottom of each page —

Home | MySQL | MySQL Tutorial 3: Triggers and Automation Share This Page