Chapter 9. Using Stored Routines, Triggers, and Scheduled Events

Introduction

This chapter discusses stored database objects, which come in several varieties:

Stored functions and procedures
A stored function or procedure object encapsulates the code for performing an operation, enabling you to invoke the object easily by name rather than repeat all its code each time it's needed. A stored function performs a calculation and returns a value that can be used in expressions just like a built-in function such as `RAND()`, `NOW()`, or `LEFT()`. A stored procedure performs operations for which no return value is needed. Procedures are invoked with the `CALL` statement, not used in expressions. A procedure might update rows in a table or produce a result set that is sent to the client program.

Triggers
A trigger is an object that activates when a table is modified by an `INSERT`, `UPDATE`, or `DELETE` statement. For

example, you can check values before they are inserted into a table, or specify that any row deleted from a table should be logged to another table that serves as a journal of data changes. Triggers automate these actions so that you need not remember to do them yourself each time you modify a table.

Scheduled events

An event is an object that executes SQL statements at a scheduled time or times. Think of a scheduled event as something like a Unix*cron* job that runs within MySQL. For example, events can help you perform administrative tasks such as deleting old table rows periodically or creating nightly summaries.

NOTE

In this book, the term "stored routine" refers collectively to stored functions and procedures, and "stored program" refers collectively to stored routines, triggers, and events.

Stored programs are database objects that are user-defined but stored on the server side for later execution. This differs from sending an SQL statement from the client to the server for immediate execution. Each object also has the property that it is defined in terms of other SQL statements to be executed when the object is invoked. The object body is a single SQL statement, but that statement can use compound-statement syntax (a `BEGIN` … `END` block) that contains multiple statements. Thus, the body can range from very simple to

extremely complex. The following stored procedure is a trivial routine that does nothing but display the current MySQL version, using a body that consists of a single `SELECT` statement:

```
CREATE PROCEDURE show_version()
SELECT VERSION() AS 'MySQL Version';
```

More complex operations use a `BEGIN` … `END` compound statement:

```
CREATE PROCEDURE show_part_of_day()
BEGIN
  DECLARE cur_time, day_part TEXT;
  SET cur_time = CURTIME();
  IF cur_time < '12:00:00' THEN
    SET day_part = 'morning';
  ELSEIF cur_time = '12:00:00' THEN
    SET day_part = 'noon';
  ELSE
    SET day_part = 'afternoon or night';
  END IF;
  SELECT cur_time, day_part;
END;
```

Here, the `BEGIN` … `END` block contains multiple statements, but is itself considered to constitute a single statement. Compound statements enable you to declare local variables and to use conditional logic and looping constructs. These

capabilities provide considerably more flexibility for algorithmic expression than when you write inline expressions in noncompound statements such as `SELECT` or `UPDATE`.

Each statement within a compound statement must be terminated by a `;` character. That requirement causes a problem if you use the *mysql* client to define an object that uses compound statements because*mysql* itself interprets `;` to determine statement boundaries. The solution is to redefine *mysql*'s statement delimiter while you define a compound-statement object. Creating Compound-Statement Objects covers how to do this; be sure to read that recipe before proceeding to those that follow it.

This chapter illustrates stored routines, triggers, and events by example, but due to space limitations does not otherwise go into much detail about their extensive syntax. For complete syntax descriptions, see the *MySQL Reference Manual*.

Scripts for the examples shown in this chapter are located in the *routines*, *triggers*, and *events* directories of the `recipes` distribution. Scripts to create example tables are located in the *tables* directory.

In addition to the stored programs shown in this chapter, others can be found elsewhere in this book. See, for example, Converting the Lettercase of a String, Changing MySQL's Date Format, Using a Join to Fill or Identify Holes in a List, and Managing User Accounts.

Stored programs used here are created and invoked under the assumption that `cookbook` is the default database. To invoke a program from another database, qualify its name with the database name:

```
CALL cookbook.show_version();
```

Alternatively, create a database specifically for your stored programs, create them in that database, and always invoke them qualified with that name. Remember to grant users who will use them the `EXECUTE` privilege for that database.

PRIVILEGES FOR STORED PROGRAMS

When you create a stored routine (function or procedure), the following privilege requirements must be satisfied or you will have problems:

- To create or execute the routine, you must have the `CREATE ROUTINE` or `EXECUTE` privilege, respectively.
- If binary logging is enabled for your MySQL server, as is common practice, there are additional requirements for creating stored functions (but not stored procedures). These requirements are necessary to ensure that if you use the binary log for replication or for restoring backups, function invocations cause the same effect when re-executed as they do when originally executed:
  - You must have the `SUPER` privilege, and you must declare either that the function is deterministic or does not modify data by using one of the `DETERMINISTIC`, `NO SQL`, or `READS SQL DATA` characteristics. (It's possible to create functions that are not deterministic or that modify data, but they might not be safe for replication or for use in backups.)
  - Alternatively, if you enable the `log_bin_trust_function_creators` system variable, the server waives both of the preceding requirements. You can do this at server startup, or at runtime if you have the `SUPER` privilege.

To create a trigger, you must have the `TRIGGER` privilege for the table associated with the trigger.

To create a scheduled event, you must have the `EVENT` privilege for the database in which the event is created.

For information about granting privileges, see Managing User Accounts.

Creating Compound-Statement Objects

Problem

You want to define a stored program, but its body contains instances of the `;` statement terminator. The *mysql* client program uses the same terminator by default, so *mysql* misinterprets the definition and produces an error.

Solution

Redefine the *mysql* statement terminator with the `delimiter` command.

Discussion

Each stored program is an object with a body that must be a single SQL statement. However, these objects often perform complex operations that require several statements. To handle this, write the statements within a `BEGIN` … `END` block that forms a compound statement. That is, the block is itself a single statement but can contain multiple statements, each terminated by a `;` character. The `BEGIN` … `END` block can contain statements such as `SELECT` or `INSERT`, but compound statements also permit conditional statements such as `IF` or `CASE`, looping constructs such as `WHILE` or `REPEAT`, or other `BEGIN` … `END` blocks.

Compound-statement syntax provides flexibility, but if you define compound-statement objects within the *mysql* client, you

quickly encounter a problem: each statement within a compound statement must be terminated by a `;` character, but *mysql* itself interprets `;` to figure out where statements end so that it can send them one at a time to the server to be executed. Consequently, *mysql* stops reading the compound statement when it sees the first `;` character, which is too early. To handle this, tell *mysql*to recognize a different statement delimiter so that it ignores `;` characters within the object body. Terminate the object itself with the new delimiter, which *mysql* recognizes and then sends the entire object definition to the server. You can restore the *mysql* delimiter to its original value after defining the compound-statement object.

The following example uses a stored function to illustrate how to change the delimiter, but the principles apply to defining any type of stored program.

Suppose that you want to create a stored function that calculates and returns the average size in bytes of mail messages listed in the `mail` table. The function can be defined like this, where the body consists of a single SQL statement:

```
CREATE FUNCTION avg_mail_size()
RETURNS FLOAT READS SQL DATA
RETURN (SELECT AVG(size) FROM mail);
```

The `RETURNS FLOAT` clause indicates the type of the function's return value, and `READS SQL DATA` indicates that the function reads but does not modify data. The function body

follows those clauses: a single `RETURN` statement that executes a subquery and returns the resulting value to the caller. (Every stored function must have at least one `RETURN` statement.)

In *mysql*, you can enter that statement as shown and there is no problem. The definition requires just the single terminator at the end and none internally, so no ambiguity arises. But suppose instead that you want the function to take an argument naming a user that it interprets as follows:

- If the argument is `NULL`, the function returns the average size for all messages (as before).
- If the argument is non-`NULL`, the function returns the average size for messages sent by that user.

To accomplish this, the function has a more complex body that uses a `BEGIN` … `END` block:

```
CREATE FUNCTION avg_mail_size(user VARCHAR(8
RETURNS FLOAT READS SQL DATA
BEGIN
  DECLARE avg FLOAT;
  IF user IS NULL
  THEN # average message size over all users
    SET avg = (SELECT AVG(size) FROM mail);
  ELSE # average message size for given user
    SET avg = (SELECT AVG(size) FROM mail WHI
  END IF;
  RETURN avg;
END;
```

If you try to define the function within *mysql* by entering that definition as just shown, *mysql*improperly interprets the first semicolon in the function body as ending the definition. Instead, use the `delimiter` command to change the *mysql*delimiter, then restore the delimiter to its default value:

```
mysql> delimiter $$
mysql> CREATE FUNCTION avg_mail_size(user VAI
    -> RETURNS FLOAT READS SQL DATA
    -> BEGIN
    ->   DECLARE avg FLOAT;
    ->   IF user IS NULL
    ->   THEN # average message size over al
    ->     SET avg = (SELECT AVG(size) FROM r
    ->   ELSE # average message size for give
    ->     SET avg = (SELECT AVG(size) FROM r
    ->   END IF;
    ->   RETURN avg;
    -> END;
    -> $$
Query OK, 0 rows affected (0.02 sec)
mysql> delimiter ;
```

After defining the stored function, invoke it the same way as a built-in function:

```
mysql> SELECT avg_mail_size(NULL), avg_mail_s
+--------------------+--------------------+
| avg_mail_size(NULL) | avg_mail_size('barb'
```

```
+--------------------+--------------------
|        237386.5625 |                5223
+--------------------+--------------------
```

Using Stored Functions to Encapsulate Calculations

Problem

A particular calculation to produce a value must be performed frequently by different applications, but you don’t want to write the expression for it each time it’s needed. Or a calculation is difficult to perform inline within an expression because it requires conditional or looping logic.

Solution

Use a stored function to hide the ugly details and make the calculation easy to perform.

Discussion

Stored functions enable you to simplify your applications. Write the code that produces a calculation result once in a function definition, then simply invoke the function whenever you need to perform the calculation. Stored functions also enable you to use more complex algorithmic constructs than are available when you write a calculation inline within an expression. This section illustrates how stored functions can be useful in these

ways. (Granted, the example is not *that* complex, but the principles used here apply to writing much more elaborate functions.)

Different states in the US charge different rates for sales tax. If you sell goods to people from different states, you must charge tax using the rate appropriate for customer state of residence. To handle tax computations, use a table that lists the sales tax rate for each state, and a stored function that looks up the tax rate given a state.

To set up the `sales_tax_rate` table, use the *sales_tax_rate.sql* script in the *tables* directory of the `recipes` distribution. The table has two columns: `state` (a two-letter abbreviation), and `tax_rate` (a `DECIMAL` value rather than a `FLOAT`, to preserve accuracy).

Define the rate-lookup function, `sales_tax_rate()`, as follows:

```
CREATE FUNCTION sales_tax_rate(state_code CHA
RETURNS DECIMAL(3,2) READS SQL DATA
BEGIN
  DECLARE rate DECIMAL(3,2);
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET
  SELECT tax_rate INTO rate FROM sales_tax_ra
  RETURN rate;
END;
```

Suppose that the tax rates for Vermont and New York are 1 and 9 percent, respectively. Try the function to check whether the tax rate is returned correctly:

```
mysql> SELECT sales_tax_rate('VT'), sales_tax
+----------------------+--------------------
| sales_tax_rate('VT') | sales_tax_rate('NY'
+----------------------+--------------------
|                 0.01 |                0.09
+----------------------+--------------------
```

If you take sales from a location not listed in the table, the function cannot determine the rate for it. In this case, the function assumes a tax rate of 0 percent:

```
mysql> SELECT sales_tax_rate('ZZ');
+----------------------+
| sales_tax_rate('ZZ') |
+----------------------+
|                 0.00 |
+----------------------+
```

The function handles states not listed using a `CONTINUE` handler for `NOT FOUND`, which executes if a No Data condition occurs: if there is no row for the given `state_param` value, the `SELECT` statement fails to find a sales tax rate, the `CONTINUE` handler sets the rate to 0, and continues execution with the next statement after the `SELECT`. (This handler is an example of stored routine

logic not available in inline expressions. Handling Errors Within Stored Programsdiscusses handlers further.)

To compute sales tax for a purchase, multiply the purchase price by the tax rate. For example, for Vermont and New York, tax on a $150 purchase is:

```
mysql> SELECT 150*sales_tax_rate('VT'), 150*s
```

```
+--------------------------+----------------
| 150*sales_tax_rate('VT') | 150*sales_tax_ra
+--------------------------+----------------
|                     1.50 |
+--------------------------+----------------
```

Or write another function that computes the tax for you:

```
CREATE FUNCTION sales_tax(state_code CHAR(2)
RETURNS DECIMAL(10,2) READS SQL DATA
RETURN sales_amount * sales_tax_rate(state_c
```

And use it like this:

```
mysql> SELECT sales_tax('VT',150), sales_tax(
+--------------------+--------------------
| sales_tax('VT',150) | sales_tax('NY',150)
+--------------------+--------------------
```

```
|                 1.50 |                 13.50
+----------------------+----------------------
```

Using Stored Procedures to “Return”Multiple Values

Problem

An operation produces two or more values, but a stored function can return only a single value.

Solution

Use a stored procedure that has `OUT` or `INOUT` parameters, and pass user-defined variables for those parameters when you invoke the procedure. A procedure does not “return” a value the way a function does, but it can assign values to those parameters so that the user-defined variables have the desired values when the procedure returns.

Discussion

Unlike stored function parameters, which are input values only, a stored procedure parameter can be any of three types:

- An `IN` parameter is for input only. This is the default if you specify no type.
- An `INOUT` parameter is used to pass a value in, and can also pass a value out.
- An `OUT` parameter is used to pass a value out.

Thus, to produce multiple values from an operation, you can use `INOUT` or `OUT` parameters. The following example illustrates this, using an `IN` parameter for input, and passing back three values via `OUT` parameters.

Creating Compound-Statement Objects shows an `avg_mail_size()` function that returns the average mail message size for a given sender. The function returns a single value. To produce additional information, such as the number of messages and total message size, a function will not work. You could write three separate functions, but that is cumbersome. Instead, use a single procedure that retrieves multiple values about a given mail sender. The following procedure, `mail_sender_stats()`, runs a query on the `mail` table to retrieve mail-sending statistics about a given username, which is the input value. The procedure determines how many messages that user sent, and the total and average sizes of the messages in bytes, which it returns through three `OUT` parameters:

```
CREATE PROCEDURE mail_sender_stats(IN user VA
                                   OUT messag
                                   OUT total_
                                   OUT avg_s
BEGIN
  # Use IFNULL() to return 0 for SUM() and AV
  # no rows for the user (those functions ret
  SELECT COUNT(*), IFNULL(SUM(size),0), IFNUL
  INTO messages, total_size, avg_size
```

```
    FROM mail WHERE srcuser = user;
  END;
```

To use the procedure, pass a string containing the username, and three user-defined variables to receive the `OUT` values. After the procedure returns, access the variable values:

```
mysql> CALL mail_sender_stats('barb',@messag
mysql> SELECT @messages, @total_size, @avg_s
+-----------+-------------+-----------+
| @messages | @total_size | @avg_size |
+-----------+-------------+-----------+
|         3 |      156696 |     52232 |
+-----------+-------------+-----------+
```

This routine passes back calculation results. It's also common to use `OUT` parameters for diagnostic purposes such as status or error indicators.

If you call `mail_sender_stats()` from within a stored program, you can pass variables to it using routine parameters or program local variables, not just user-defined variables.

Using Triggers to Implement Dynamic Default Column Values

Problem

A table contains a column for which the initial value is not constant, but in most cases, MySQL permits only constant default values.

Solution

Use a `BEFORE INSERT` trigger. This enables you to initialize a column to the value of an arbitrary expression. In other words, the trigger performs dynamic column initialization by calculating the default value.

Discussion

Other than `TIMESTAMP` and `DATETIME` columns, which can be initialized to the current date and time (see Using TIMESTAMP or DATETIME to Track Row-Modification Times), default column values in MySQL must be constants. You cannot define a column with a `DEFAULT` clause that refers to a function call or other arbitrary expression, and you cannot define one column in terms of the value assigned to another column. That means each of these column definitions is illegal:

```
d        DATE DEFAULT NOW()
i        INT DEFAULT (... some subquery ...)
hash_val CHAR(32) DEFAULT MD5(blob_col)
```

You can work around this limitation by setting up a suitable trigger, which enables you to initialize a column however you

want. In effect, the trigger implements a dynamic (or calculated) default column value.

The appropriate type of trigger for this is `BEFORE INSERT`, which enables column values to be set before they are inserted into the table. (An `AFTER INSERT` trigger can examine column values for a new row, but by the time the trigger activates, it's too late to change the values.)

To see how this works, recall the scenario in Using Stored Functions to Encapsulate Calculations that created a `sales_tax_rate()` lookup function to return a rate from the `sales_tax_rate` table given a customer state of residence. Suppose that you anticipate a need to know at some later date the tax rate from the time of sale. It's not necessarily true that at that later date you could look up the value from the `sales_tax_rate` table; rates change and the rate in effect then might differ. To handle this, store the rate with the purchase invoice, initializing it automatically using a trigger.

A `cust_invoice` table for storing sales information might look like this:

```
CREATE TABLE cust_invoice
(
  id       INT NOT NULL AUTO_INCREMENT,
  state    CHAR(2),       # customer state o
  amount   DECIMAL(10,2), # sale amount
  tax_rate DECIMAL(3,2),  # sales tax rate a
  ... other columns ...
```

```
  PRIMARY KEY (id)
);
```

To initialize the sales tax column for inserts into the `cust_invoice` table, use a `BEFORE INSERT` trigger that looks up the rate and stores it in the table:

```
CREATE TRIGGER bi_cust_invoice BEFORE INSERT
FOR EACH ROW SET NEW.tax_rate = sales_tax_rat
```

Within the trigger, `NEW.` *`col_name`* refers to the new value to be inserted into the given column. By assigning a value to `NEW.` *`col_name`* within the trigger, you cause the column to have that value in the new row.

This trigger is simple and its body contains only a single SQL statement. For a trigger body that executes multiple statements, use `BEGIN` … `END` compound-statement syntax. In that case, if you use *mysql* to create the trigger, change the statement delimiter while you define the trigger, as discussed in Creating Compound-Statement Objects.

To test the implementation, insert a row and check whether the trigger correctly initializes the sales tax rate for the invoice:

```
mysql> INSERT INTO cust_invoice (state,amount
mysql> SELECT * FROM cust_invoice WHERE id =
+----+-------+--------+----------+
```

```
| id | state | amount | tax_rate |
+----+-------+--------+----------+
|  1 | NY    | 100.00 |     0.09 |
+----+-------+--------+----------+
```

The `SELECT` shows that the `tax_rate` column has the right value even though the `INSERT` provides no value for it.

Using Triggers to Simulate Function-Based Indexes

Problem

You need a function-based index, but MySQL doesn't support that capability.

Solution

Use a secondary column and triggers to simulate a function-based index.

Discussion

Some types of information are more easily analyzed using not the original values, but an expression computed from them. For example, if data values lie along an exponential curve, applying a logarithmic transform to them yields a more linear scale. Queries against a table that stores exponential values might therefore typically use expressions that refer to the log values:

```
SELECT * FROM expdata WHERE LOG10(value) < 2
```

A disadvantage of such expressions is that referring to the `value` column within a function call prevents the optimizer from using any index on it. MySQL must retrieve the values to apply the function to them, and the function values are not indexed. The result is diminished performance.

Some database systems permit an index to be defined on a function of a column, such that you can index `LOG10(value)`. MySQL does not support this capability, but there is a workaround:

1. Define a secondary column to store the function values and index that column.
2. Define triggers that keep the secondary column up to date when the original column is initialized or modified.
3. Refer directly to the secondary column in queries so that the optimizer can use the index on it for efficient lookups.

The following example illustrates this technique, using a table designed to store values that lie along an exponential curve:

```
CREATE TABLE expdata
(
  id          INT UNSIGNED NOT NULL AUTO_INCR
  value       FLOAT,  # original values
  log10_value FLOAT,  # LOG10() function of v
  INDEX (value),      # index on original val
```

```
  INDEX (log10_value) # index on function-bas
);
```

The table includes `value` and `log10_value` columns to store the original data values and those values transformed with `LOG10()`. The table also indexes both columns.

Create an `INSERT` trigger to initialize the `log10_value` value from `value` for new rows, and an `UPDATE` trigger to keep `log10_value` up to date when `value` changes:

```
CREATE TRIGGER bi_expdata BEFORE INSERT ON ex
FOR EACH ROW SET NEW.log10_value = LOG10(NEW

CREATE TRIGGER bu_expdata BEFORE UPDATE ON ex
FOR EACH ROW SET NEW.log10_value = LOG10(NEW
```

To test the implementation, insert and modify some data and check the result of each operation:

```
mysql> INSERT INTO expdata (value) VALUES (.(
mysql> SELECT * FROM expdata;
+----+-------+-------------+
| id | value | log10_value |
+----+-------+-------------+
|  1 |  0.01 |          -2 |
|  2 |   0.1 |          -1 |
|  3 |     1 |           0 |
```

```
|  4 |    10 |           1 |
|  5 |   100 |           2 |
+----+-------+-------------+
```

```
mysql> UPDATE expdata SET value = value * 10
mysql> SELECT * FROM expdata;
+----+-------+-------------+
| id | value | log10_value |
+----+-------+-------------+
|  1 |   0.1 |          -1 |
|  2 |     1 |           0 |
|  3 |    10 |           1 |
|  4 |   100 |           2 |
|  5 |  1000 |           3 |
+----+-------+-------------+
```

With that implementation, using a `log10_value` column that stores the `LOG10()` values of the `value` column, the `SELECT` query shown earlier can be rewritten:

```
SELECT * FROM expdata WHERE log10_value < 2;
```

The optimizer can use the index on `log10_value`, something not true of the original query that referred to `LOG10(value)`.

Using triggers this way to simulate a function-based index improves query performance for `SELECT` queries, but you should also consider the disadvantages of the technique:

- It requires extra storage for the secondary column.
- It requires more processing for statements that modify the original column (to activate the triggers that keep the secondary column and its index up to date).

The technique is therefore most useful if the workload for the table skews more toward retrievals than updates. It is less beneficial for a workload that is mostly updates.

The preceding example uses a `log10_value` column that is useful for several types of lookups, from single-row to range-based expressions. But functional indexes can be useful even for situations in which*most* queries select only a single row. Suppose that you want to store large data values such as PDF or XML documents in a table, but also want to look them up quickly later (for example, to access other values stored in the same row such as author or title). A `TEXT` or `BLOB` data type might be suitable for storing the values, but is not very suitable for finding them. (Comparisons in a lookup operation are slow for large values.) To work around this problem, use the following strategy:

1. Compute a hash value for each document and store it in the table along with the document. For example, use the `MD5()` function, which returns a 32-byte string of hexadecimal characters. That's still long for a comparison value, but much shorter than a full-column comparison based on contents of very long documents.

2. To look up the row containing a particular document, compute the document hash value and search the table for that value. For best performance, index the hash column. Because the hash value is a function of the document, the index on it is, in effect, a functional index.

The result is that lookups based on the hash-value column will perform much more efficiently than lookups based on the original document values.

Simulating TIMESTAMP Properties for Other Date and Time Types

Problem

The `TIMESTAMP` data type provides auto-initialization and auto-update properties. You would like to use these properties for other temporal data types that permit only constant values for initialization and don't auto-update.

Solution

Use an `INSERT` trigger to provide the appropriate current date or time value at row-creation time. Use an `UPDATE` trigger to update the column to the current date or time when the row is changed.

Discussion

Using TIMESTAMP or DATETIME to Track Row-Modification Times describes the special `TIMESTAMP` and `DATETIME` initialization and update properties enable you to record row-creation and row-modification times automatically. These properties are not available for other temporal types, although there are reasons you might like them to be. For example, if you use separate `DATE` and `TIME` columns to store row-modification times, you can index the `DATE` column to enable efficient date-based lookups.
(With `TIMESTAMP` or `DATETIME`, you cannot index just the date part of the column.)

One way to simulate `TIMESTAMP` properties for other temporal data types is to use the following strategy:

- When you create a row, initialize a `DATE` column to the current date and a `TIME` column to the current time.
- When you update a row, set the `DATE` and `TIME` columns to the new date and time.

However, this strategy requires all applications that use the table to implement the same strategy, and it fails if even one application neglects to do so. To place the burden of remembering to set the columns properly on the MySQL server and not on application writers, use triggers for the table. This is, in fact, a particular application of the general strategy discussed in Using Triggers to Implement Dynamic Default Column

Values that uses triggers to provide calculated values for initializing (or updating) row columns.

The following example shows how to use triggers to simulate `TIMESTAMP` properties for the `DATE` and `TIME` data types. (The same technique also serves to simulate `TIMESTAMP` properties for `DATETIME` for versions of MySQL older than 5.6.5, before `DATETIME` was given automatic properties.) Begin by creating the following table, which has a nontemporal column for storing data and columns for the `DATE` and `TIME` temporal types:

```
CREATE TABLE ts_emulate (data CHAR(10), d DA
```

The intent here is that when applications insert or update values in the `data` column, MySQL should set the temporal columns appropriately to reflect the time at which modifications occur. To accomplish this, set up triggers that use the current date and time to initialize the temporal columns for new rows, and to update them when existing rows are changed.
A `BEFORE INSERT` trigger handles row creation by invoking the `CURDATE()` and `CURTIME()` functions to get the current date and time and using those values to set the temporal columns:

```
CREATE TRIGGER bi_ts_emulate BEFORE INSERT ON
FOR EACH ROW SET NEW.d = CURDATE(), NEW.t = C
```

A `BEFORE UPDATE` trigger handles updates to the temporal columns when the `data` column changes value.
An `IF` statement is required here to emulate the `TIMESTAMP` property that an update occurs only if the `data` value in the row actually changes from its current value:

```
CREATE TRIGGER bu_ts_emulate BEFORE UPDATE ON
FOR EACH ROW # update temporal columns only i
IF NEW.data <> OLD.data THEN
  SET NEW.d = CURDATE(), NEW.t = CURTIME();
END IF;
```

To test the `INSERT` trigger, create a couple rows, but supply a value only for the `data` column. Then verify that MySQL provides the proper default values for the temporal columns:

```
mysql> INSERT INTO ts_emulate (data) VALUES(
mysql> INSERT INTO ts_emulate (data) VALUES(
mysql> SELECT * FROM ts_emulate;
+------+------------+----------+
| data | d          | t        |
+------+------------+----------+
| cat  | 2014-04-07 | 13:53:32 |
| dog  | 2014-04-07 | 13:53:37 |
+------+------------+----------+
```

Change the `data` value of one row to verify that the `BEFORE` `UPDATE` trigger updates the temporal columns of the changed row:

```
mysql> UPDATE ts_emulate SET data = 'axolotl
mysql> SELECT * FROM ts_emulate;
+---------+------------+----------+
| data    | d          | t        |
+---------+------------+----------+
| axolotl | 2014-04-07 | 13:53:49 |
| dog     | 2014-04-07 | 13:53:37 |
+---------+------------+----------+
```

Issue another `UPDATE`, but this time use one that does not change any `data` column values. In this case, the `BEFORE` `UPDATE` trigger should notice that no value change occurred and leave the temporal columns unchanged:

```
mysql> UPDATE ts_emulate SET data = data;
mysql> SELECT * FROM ts_emulate;
+---------+------------+----------+
| data    | d          | t        |
+---------+------------+----------+
| axolotl | 2014-04-07 | 13:53:49 |
| dog     | 2014-04-07 | 13:53:37 |
+---------+------------+----------+
```

The preceding example shows how to simulate the auto-initialization and auto-update properties offered

by `TIMESTAMP` columns. To implement only one of those properties and not the other, create only one trigger and omit the other.

Using Triggers to Log Changes to a Table

Problem

You have a table that maintains current values of items that you track (such as auctions being bid on), but you'd also like to maintain a journal (history) of changes to the table.

Solution

Use triggers to "catch" table changes and write them to a separate log table.

Discussion

Suppose that you conduct online auctions, and that you maintain information about each currently active auction in a table that looks like this:

```
CREATE TABLE auction
(
  id   INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ts   TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON
  item VARCHAR(30) NOT NULL,
  bid  DECIMAL(10,2) NOT NULL,
  PRIMARY KEY (id)
);
```

The `auction` table contains information about the currently active auctions (items being bid on and the current bid for each auction). When an auction begins, insert a row into the table. For each bid on an item, update its `bid` column so that as the auction proceeds, the `ts` column updates to reflect the most recent bid time. When the auction ends, the `bid` value is the final price and the row can be removed from the table.

To maintain a journal that shows all changes to auctions as they progress from creation to removal, set up another table that serves to record a history of changes to the auctions. This strategy can be implemented with triggers.

To maintain a history of how each auction progresses, use an `auction_log` table with the following columns:

```
CREATE TABLE auction_log
(
  action ENUM('create','update','delete'),
  id     INT UNSIGNED NOT NULL,
  ts     TIMESTAMP DEFAULT CURRENT_TIMESTAMP (
  item   VARCHAR(30) NOT NULL,
  bid    DECIMAL(10,2) NOT NULL,
  INDEX (id)
);
```

The `auction_log` table differs from the `auction` table in two ways:

- It contains an `action` column to indicate for each row what kind of change was made.
- The `id` column has a nonunique index (rather than a primary key, which requires unique values). This permits multiple rows per `id` value because a given auction can generate many rows in the log table.

To ensure that changes to the `auction` table are logged to the `auction_log` table, create a set of triggers. The triggers write information to the `auction_log` table as follows:

- For inserts, log a row-creation operation showing the values in the new row.
- For updates, log a row-update operation showing the new values in the updated row.
- For deletes, log a row-removal operation showing the values in the deleted row.

For this application, `AFTER` triggers are used because they activate only after successful changes to the `auction` table. (`BEFORE` triggers might activate even if the row-change operation fails for some reason.) The trigger definitions look like this:

```
CREATE TRIGGER ai_auction AFTER INSERT ON au
FOR EACH ROW
INSERT INTO auction_log (action,id,ts,item,b
VALUES('create',NEW.id,NOW(),NEW.item,NEW.bi

CREATE TRIGGER au_auction AFTER UPDATE ON au
```

```
FOR EACH ROW
INSERT INTO auction_log (action,id,ts,item,bi
VALUES('update',NEW.id,NOW(),NEW.item,NEW.bid

CREATE TRIGGER ad_auction AFTER DELETE ON auc
FOR EACH ROW
INSERT INTO auction_log (action,id,ts,item,bi
VALUES('delete',OLD.id,OLD.ts,OLD.item,OLD.bi
```

The `INSERT` and `UPDATE` triggers use `NEW.` *`col_name`* to access the new values being stored in rows. The `DELETE` trigger uses `OLD.` *`col_name`* to access the existing values from the deleted row.
The `INSERT` and `UPDATE` triggers use `NOW()` to get the row-modification times; the `ts` column is initialized automatically to the current date and time, but `NEW.ts` will not contain that value.

Suppose that an auction is created with an initial bid of five dollars:

```
mysql> INSERT INTO auction (item,bid) VALUES(
mysql> SELECT LAST_INSERT_ID();
+------------------+
| LAST_INSERT_ID() |
+------------------+
|              792 |
+------------------+
```

The `SELECT` statement fetches the auction ID value to use for subsequent actions on the auction. Then the item receives three more bids before the auction ends and is removed:

```
mysql> UPDATE auction SET bid = 7.50 WHERE ic
... time passes ...
mysql> UPDATE auction SET bid = 9.00 WHERE ic
... time passes ...
mysql> UPDATE auction SET bid = 10.00 WHERE :
... time passes ...
mysql> DELETE FROM auction WHERE id = 792;
```

At this point, no trace of the auction remains in the `auction` table, but the `auction_log` table contains a complete history of what occurred:

```
mysql> SELECT * FROM auction_log WHERE id = 7
+--------+-----+---------------------+------
| action | id  | ts                  | item
+--------+-----+---------------------+------
| create | 792 | 2014-01-09 14:57:41 | chint
| update | 792 | 2014-01-09 14:57:50 | chint
| update | 792 | 2014-01-09 14:57:57 | chint
| update | 792 | 2014-01-09 14:58:03 | chint
| delete | 792 | 2014-01-09 14:58:03 | chint
+--------+-----+---------------------+------
```

With the strategy just outlined, the `auction` table remains relatively small, and you can always find information about

auction histories as necessary bylooking in the `auction_log` table.

Using Events to Schedule Database Actions

Problem

You want to set up a database operation that runs periodically without user intervention.

Solution

Create an event that executes according to a schedule.

Discussion

MySQL provides an event scheduler that enables you to set up database operations that run at times that you define. This section describes what you must do to use events, beginning with a simple event that writes a row to a table at regular intervals. Why bother creating such an event? One reason is that the rows serve as a log of continuous server operation, similar to the `MARK` line that some Unix `syslogd` servers write to the system log periodically so that you know they're alive.

Begin with a table to hold the mark rows. It contains a `TIMESTAMP` column (which MySQL will initialize automatically) and a column to store a message:

```
CREATE TABLE mark_log
(
```

```
    ts      TIMESTAMP DEFAULT CURRENT_TIMESTAMI
    message VARCHAR(100)
  );
```

Our logging event will write a string to a new row. To set it up, use a `CREATE EVENT` statement:

```
  CREATE EVENT mark_insert
  ON SCHEDULE EVERY 5 MINUTE
  DO INSERT INTO mark_log (message) VALUES('--
```

The `mark_insert` event causes the message `'-- MARK --'` to be logged to the `mark_log` table every five minutes. Use a different interval for more or less frequent logging.

This event is simple and its body contains only a single SQL statement. For an event body that executes multiple statements, use `BEGIN` … `END` compound-statement syntax. In that case, if you use *mysql* to create the event, change the statement delimiter while you define the event, as discussed in Creating Compound-Statement Objects.

At this point, you should wait a few minutes and then select the contents of the `mark_log` table to verify that new rows are being written on schedule. However, if this is the first event that you've set up, you might find that the table remains empty no matter how long you wait:

```
mysql> SELECT * FROM mark_log;
Empty set (0.00 sec)
```

If that's the case, very likely the event scheduler isn't running (which is its default state until you enable it). Check the scheduler status by examining the value of the `event_scheduler` system variable:

```
mysql> SHOW VARIABLES LIKE 'event_scheduler';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| event_scheduler | OFF   |
+-----------------+-------+
```

To enable the scheduler interactively if it's not running, execute the following statement (which requires the `SUPER` privilege):

```
SET GLOBAL event_scheduler = 1;
```

That statement enables the scheduler, but only until the server shuts down. To start the scheduler each time the server starts, enable the system variable in your *my.cnf* option file:

```
[mysqld]
event_scheduler=1
```

When the event scheduler is enabled, the `mark_insert` event eventually creates many rows in the table. There are several ways that you can affect event execution to prevent the table from growing forever:

- Drop the event:

  ```
  DROP EVENT mark_insert;
  ```

  This is the simplest way to stop an event from occurring. But if you want it to resume later, you must re-create it.
- Disable event execution:

  ```
  ALTER EVENT mark_insert DISABLE;
  ```

  That leaves the event in place but causes it not to run until you reactivate it:

  ```
  ALTER EVENT mark_insert ENABLE;
  ```

- Let the event continue to run, but set up another event that "expires" old `mark_log` rows. This second event need not run so frequently (perhaps once a day). Its body should remove rows older than a given threshold. The following definition creates an event that deletes rows that are more than two days old:

```
CREATE EVENT mark_expire
ON SCHEDULE EVERY 1 DAY
DO DELETE FROM mark_log WHERE ts < NOW() -
```

If you adopt this strategy, you have cooperating events: one event that adds rows to the `mark_log` table, and another that removes them. They act together to maintain a log that contains recent rows but does not become too large.

Writing Helper Routines for Executing Dynamic SQL

Problem

Prepared SQL statements enable you to construct and execute SQL statements on the fly, but the supporting mechanism can be tedious to use.

Solution

Write a helper procedure that handles the drudgery.

Discussion

Using a prepared SQL statement involves three steps: preparation, execution, and deallocation. For example, if the `@tbl_name` and `@val` variables hold a table name and a value to insert into the table, you can create the table and insert the value like this:

```
SET @stmt = CONCAT('CREATE TABLE ',@tbl_name
PREPARE stmt FROM @stmt;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
SET @stmt = CONCAT('INSERT INTO ',@tbl_name,
PREPARE stmt FROM @stmt;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

To ease the burden of going through those steps for each dynamically created statement, use a helper routine that, given a statement string, prepares, executes, and deallocates it:

```
CREATE PROCEDURE exec_stmt(stmt_str TEXT)
BEGIN
  SET @_stmt_str = stmt_str;
  PREPARE stmt FROM @_stmt_str;
  EXECUTE stmt;
  DEALLOCATE PREPARE stmt;
END;
```

The `exec_stmt()` routine enables the same statements to be executed much more simply:

```
CALL exec_stmt(CONCAT('CREATE TABLE ',@tbl_na
CALL exec_stmt(CONCAT('INSERT INTO ',@tbl_nar
```

`exec_stmt()` uses an intermediary user-defined variable, `@_exec_stmt`, because `PREPARE` accepts a statement only when specified using either a literal string or a user-defined variable. A statement stored in a routine parameter does not work. (Avoid using `@_exec_stmt` for your own purposes, at least if you expect its value to persist across `exec_stmt()` invocations.)

Now, how about making it safer to construct statement strings that incorporate values that might come from external sources, such as web-form input or command-line arguments? Such information cannot be trusted and should be treated as a potential SQL injection attack vector:

- The `QUOTE()` function is available for quoting data values.
- There is no corresponding function for identifiers, but it's easy to write one that doubles internal backticks and adds a backtick at the beginning and end:

```
CREATE FUNCTION quote_identifier(id TEXT)
RETURNS TEXT DETERMINISTIC
RETURN CONCAT('`',REPLACE(id,'`','``'),'`'
```

Revising the preceding example to ensure the safety of data values and identifiers, we have:

```
SET @tbl_name = quote_identifier(@tbl_name);
SET @val = QUOTE(@val);
```

```
CALL exec_stmt(CONCAT('CREATE TABLE ',@tbl_na
CALL exec_stmt(CONCAT('INSERT INTO ',@tbl_nam
```

A constraint on use of `exec_stmt()` is that not all SQL statements are eligible for execution as prepared statements. See the *MySQL Reference Manual* for the limitations.

Handling Errors Within Stored Programs

Within stored programs, you can catch errors or exceptional conditions using condition handlers. A handler activates under specific circumstances, causing the code associated with it to execute. The code takes suitable action such as performing cleanup processing or setting a variable that can be tested elsewhere in the program to determine whether the condition occurred. A handler might even ignore an error if it occurs under certain permitted conditions and you want to catch it rather than have it terminate your program.

Stored programs can also produce their own errors or warnings to signal that something has gone wrong.

The following examples illustrate these techniques. For complete lists of available condition names, SQLSTATE values, and error codes, consult the *MySQL Reference Manual.*

Detecting End-of-Data Conditions

One common use of condition handlers is to detect “no more rows” conditions. To process a query result one row at a time, use a cursor-based fetch loop in conjunction with a condition handler that catches the end-of-data condition. The technique has these essential elements:

- A cursor associated with a `SELECT` statement that reads rows. Open the cursor to start reading, and close it to stop.
- A condition handler that activates when the cursor reaches the end of the result set and raises an end-of-data condition (`NOT FOUND`). We used a similar handler in Using Stored Functions to Encapsulate Calculations.
- A variable that indicates loop termination. Initialize the variable to `FALSE`, then set it to `TRUE` within the condition handler when the end-of-data condition occurs.
- A loop that uses the cursor to fetch each row and exits when the loop-termination variable becomes `TRUE`.

The following example implements a fetch loop that processes the `states` table row by row to calculate the total US population:

```
CREATE PROCEDURE us_population()
BEGIN
  DECLARE done BOOLEAN DEFAULT FALSE;
  DECLARE state_pop, total_pop BIGINT DEFAUL
  DECLARE cur CURSOR FOR SELECT pop FROM sta
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET
```

```
  OPEN cur;
  fetch_loop: LOOP
    FETCH cur INTO state_pop;
    IF done THEN
      LEAVE fetch_loop;
    END IF;
    SET total_pop = total_pop + state_pop;
  END LOOP;
  CLOSE cur;
  SELECT total_pop AS 'Total U.S. Population
END;
```

Clearly, that example is purely for illustration because in any real application you'd use an aggregate function to calculate the total. But that also gives us an independent check on whether the fetch loop calculates the correct value:

```
mysql> CALL us_population();
+-----------------------+
| Total U.S. Population |
+-----------------------+
|             308143815 |
+-----------------------+
mysql> SELECT SUM(pop) AS 'Total U.S. Populat
+-----------------------+
| Total U.S. Population |
+-----------------------+
|             308143815 |
+-----------------------+
```

`NOT FOUND` handlers are also useful for checking whether `SELECT ... INTO var_name` statements return any results. Using Stored Functions to Encapsulate Calculations shows an example.

Catching and Ignoring Errors

If you consider an error benign, you can use a handler to ignore it. For example, many `DROP` statements in MySQL have an `IF EXISTS` clause to suppress errors if objects to be dropped do not exist. But some `DROP` statements have no such clause and thus no way to suppress errors. `DROP USER` is one of these:

```
mysql> DROP USER 'bad-user'@'localhost';
ERROR 1396 (HY000): Operation DROP USER faile
```

To prevent errors from occurring for nonexistent users, invoke `DROP USER` within a stored procedure that catches code 1396 and ignores it:

```
CREATE PROCEDURE drop_user(user TEXT, host TE
BEGIN
  DECLARE account TEXT;
  DECLARE CONTINUE HANDLER FOR 1396
    SELECT CONCAT('Unknown user: ', account)
  SET account = CONCAT(QUOTE(user),'@',QUOTE(
  CALL exec_stmt(CONCAT('DROP USER ',account))
END;
```

If the user does not exist, `drop_user()` writes a message within the condition handler, but no error occurs:

```
mysql> CALL drop_user('bad-user','localhost');
+---------------------------------------+
| Message                               |
+---------------------------------------+
| Unknown user: 'bad-user'@'localhost' |
+---------------------------------------+
```

To ignore the error completely, write the handler using an empty `BEGIN` … `END` block:

```
DECLARE CONTINUE HANDLER FOR 1396 BEGIN END;
```

Another approach is to generate a warning, as demonstrated in the next section.

Raising Errors and Warnings

To produce your own errors within a stored program when you detect something awry, use the `SIGNAL` statement. This section shows some examples, and Using Triggers to Preprocess or Reject Datademonstrates use of `SIGNAL` within a trigger to reject bad data.

Suppose that an application performs a division operation for which you expect that the divisor will never be zero, and that you want to produce an error otherwise. You might expect that you could set the SQL mode properly to produce a divide-by-zero error (this requires `ERROR_FOR_DIVISION_BY_ZERO` plus strict mode, or just strict mode as of MySQL 5.7.4). But that works only within the context of data-modification operations such as `INSERT`. In other contexts, division by zero produces only a warning:

```
mysql> SET sql_mode = 'ERROR_FOR_DIVISION_BY_
mysql> SELECT 1/0;
+------+
| 1/0  |
+------+
| NULL |
+------+
1 row in set, 1 warning (0.00 sec)
mysql> SHOW WARNINGS;
+---------+------+---------------+
| Level   | Code | Message       |
+---------+------+---------------+
| Warning | 1365 | Division by 0 |
+---------+------+---------------+
```

To ensure a divide-by-zero error in any context, write a function that performs the division but checks the divisor first and

uses `SIGNAL` to raise an error if the"can't happen" condition occurs:

```
CREATE FUNCTION divide(numerator FLOAT, divi
RETURNS FLOAT DETERMINISTIC
BEGIN
  IF divisor = 0 THEN
    SIGNAL SQLSTATE '22012'
           SET MYSQL_ERRNO = 1365, MESSAGE_TI
  END IF;
  RETURN numerator / divisor;
END;
```

Test the function in a nonmodification context to verify that it produces an error:

```
mysql> SELECT divide(1,0);
ERROR 1365 (22012): unexpected 0 divisor
```

The `SIGNAL` statement specifies a SQLSTATE value plus an optional `SET` clause you can use to assign values to error attributes. `MYSQL_ERRNO` corresponds to the MySQL-specific error code, and `MESSAGE_TEXT` is a string of your choice.

`SIGNAL` can also raise warning conditions, not just errors. The following routine, `drop_user_warn()`, is similar to the `drop_user()` routine shown earlier, but instead of printing a message for nonexistent users, it generates a warning that can be displayed with `SHOW WARNINGS`. SQLSTATE

value `01000` and error 1642 indicate a user-defined unhandled exception, which the routine signals along with an appropriate message:

```
CREATE PROCEDURE drop_user_warn(user TEXT, ho
BEGIN
  DECLARE account TEXT;
  DECLARE CONTINUE HANDLER FOR 1396
  BEGIN
    DECLARE msg TEXT;
    SET msg = CONCAT('Unknown user: ', accou
    SIGNAL SQLSTATE '01000' SET MYSQL_ERRNO =
  END;
  SET account = CONCAT(QUOTE(user),'@',QUOTE(
  CALL exec_stmt(CONCAT('DROP USER ',account)
END;
```

Give it a test:

```
mysql> CALL drop_user_warn('bad-user','localh
Query OK, 0 rows affected, 1 warning (0.00 se
mysql> SHOW WARNINGS;
+---------+------+--------------------------
| Level   | Code | Message
+---------+------+--------------------------
| Warning | 1642 | Unknown user: 'bad-user'@
+---------+------+--------------------------
```

Using Triggers to Preprocess or Reject Data

Problem

There are conditions you want to check for data entered into a table, but you don't want to write the validation logic for every `INSERT`.

Solution

Centralize the input-testing logic into a `BEFORE INSERT` trigger.

Discussion

You can use triggers to perform several types of input checks:

- Reject bad data by raising a signal. This gives you access to stored program logic for more latitude in checking values than is possible with static constraints such as `NOT NULL`.
- Preprocess values and modify them, if you won't want to reject them outright. For example, map out-of-range values to be in range or sanitize values to put them in canonical form, if you permit entry of close variants.

Suppose that you have a table of contact information such as name, state of residence, email address, and website URL:

```
CREATE TABLE contact_info
(
  id      INT NOT NULL AUTO_INCREMENT,
  name    VARCHAR(30),    # name of person
  state   CHAR(2),        # state of residence
```

```
    email   VARCHAR(50),    # email address
    url     VARCHAR(255),   # web address
    PRIMARY KEY (id)
  );
```

For entry of new rows, you want to enforce constraints or perform preprocessing as follows:

- State of residence values are two-letter US state codes, valid only if present in the `states` table. (In this case, you could declare the column as an `ENUM` with 50 members, so it's more likely you'd use this lookup-table technique with columns for which the set of valid values is arbitrarily large or changes over time.)
- Email address values must contain an `@` character to be valid.
- For website URLs, strip any leading `http://` to save space.

To handle these requirements, create a `BEFORE INSERT` trigger:

```
CREATE TRIGGER bi_contact_info BEFORE INSERT
FOR EACH ROW
BEGIN
  IF (SELECT COUNT(*) FROM states WHERE abbr
    SIGNAL SQLSTATE 'HY000'
           SET MYSQL_ERRNO = 1525, MESSAGE_T
  END IF;
```

```
    IF INSTR(NEW.email,'@') = 0 THEN
      SIGNAL SQLSTATE 'HY000'
             SET MYSQL_ERRNO = 1525, MESSAGE_TI
    END IF;
    SET NEW.url = TRIM(LEADING 'http://' FROM N
  END;
```

To also handle updates, define a `BEFORE UPDATE` trigger with the same body as `bi_contact_info`.

Test the trigger by executing some `INSERT` statements to verify that it accepts valid values, rejects bad ones, and trims URLs:

```
mysql> INSERT INTO contact_info (name,state,e
    -> VALUES('Jen','NY','jen@example.com','h
mysql> INSERT INTO contact_info (name,state,e
    -> VALUES('Jen','XX','jen@example.com','h
ERROR 1525 (HY000): invalid state code
mysql> INSERT INTO contact_info (name,state,e
    -> VALUES('Jen','NY','jen','http://www.ex
ERROR 1525 (HY000): invalid email address
mysql> SELECT * FROM contact_info;
+----+------+-------+-----------------+-----
| id | name | state | email           | url
+----+------+-------+-----------------+-----
|  1 | Jen  | NY    | jen@example.com | www.e
+----+------+-------+-----------------+-----
```

Support Sign Out

/