# How To Automate Your Database Using Triggers and Events

Published July 10, 2011

Databases are incredibly powerful. Unfortunately, they have a reputation for being complex and mysterious beasts ruled by dark overlords who speak in strange tongues. I’m guilty of developing web applications which shunned great database techniques I didn’t understand. I don’t think I’m alone — many developers prefer to re-invent wheels in PHP rather than implement database functionality which could save time and effort while increasing performance.

Today, we’re going to look at **triggers** and **events** — features supported by most popular commercial and open source database systems…

## Triggers

A trigger is code run just before or just after an INSERT, UPDATE or DELETE SQL event occurs on a particular database table. That code can verify or modify the incoming data, perform calculations, run further SQL commands, etc.

Triggers are supported in MySQL, PostgreSQL, SQLite, Firebird, DB2, Microsoft SQL Server and Oracle. Implementation varies so you should check the appropriate documentation before coding.

## Events (or Temporal Triggers)

Events are often referred to as “temporal triggers” because they’re scheduled by time rather than a table update. Events can scheduled to run once or more during specific periods. Effectively, they’re database-only cron jobs and could be used to archive data, clean logs, or calculate information for complex reports.

Fewer databases support events but most provide a similar solution. Not all will be easy to implement within a typical web application so please check your documentation.

## When Should You Use Triggers?

Triggers can be as simple or complex as you like. That said, there are a few situations where they should be avoided:

1. Triggers should not be used in place of [foreign key constraints](http://www.sitepoint.com/mysql-foreign-keys-quicker-database-development/). Foreign keys enforce referential integrity so it becomes impossible to add, remove or edit data which would leave orphan records. For example, if you delete a user from your social networking system, you don’t necessarily want messages from that person to be retained.
2. Triggers are not a substitute for [transactions](http://www.sitepoint.com/mysql-transactions-php-emulation/). Trigger code can fail so you should still wrap related updates in a single transaction.
3. Be wary about splitting your business logic between the database and your back-end systems (written in PHP, C#, Java etc). Maintenance will become more difficult if your code is separated.
4. Avoid duplication of effort. Your back-end code should be sanitizing user input so it should not be necessary to repeat those checks within the database.
5. Triggers incur a performance overhead. They will execute quicker than a series of SQL commands passed by back-end code, but you should refrain from using complex triggers on regularly modified tables.

Ideally, triggers should be considered when they automate changes which are specific to the database or management of its data. An audit log is a good example. Consider a WordPress-like CMS system with a ‘blog’ table containing titles and body content. An ‘audit’ table could record the date and time an article is added, edited or removed. Your web system may never present that information or even know it’s recorded so a trigger would be ideal.

Our ‘blog’ table would have a one-to-many relationship with the ‘audit’ table; in other words, one or more audit records would point at a single post. Assume our system wants to retrieve the title, body content and last update date, i.e. the post itself and its last recorded entry in the ‘audit’ table. We can obtain that information but the SELECT command is complex and requires sub-selects to ensure we only retrieve the last audit record.

A trigger could help us reduce the complexity and increase performance. For example, we could add a ‘last\_audit\_id’ field to our ‘blog’ table. Whenever a post is updated, a new entry would be added to the ‘audit’ table and its ID would be recorded in blog.last\_audit\_id.

## When Should You Use Events?

Scheduled events are ideal for batch processing large volumes of data during off-peak periods.

Assume we want to delete a post from our ‘blog’ table. Rather than executing a SQL DELETE, we would set a boolean ‘deleted’ field to true so that post can be undeleted. After a while, our table will become larger, slower and perhaps contain a large proportion of deleted articles. To solve this problem, a scheduled event could be run once a week to move older deleted posts and audit records into archive tables. It could also physically delete a post once it reaches a specific age.

These are only simple suggestions and your application will have different requirements. Fortunately, triggers and events are supported in the web’s most-used relational database system: MySQL 5.x. In my following articles, we’ll discuss how to create a similar blog database to that described above.

See also:

* [Top 10 MySQL Mistakes Made By PHP Developers](http://www.sitepoint.com/mysql-mistakes-php-developers/)
* [How to Use MySQL Foreign Keys for Quicker Database Development](http://www.sitepoint.com/mysql-foreign-keys-quicker-database-development/)
* [MySQL Transactions & Why They Can’t Be Emulated in PHP](http://www.sitepoint.com/mysql-transactions-php-emulation/)

# How to Create Triggers in MySQL

Published July 12, 2011

This article was written in 2011 and remains one of our most popular posts. If you’re keen to learn more about MySQL, you may find this[*recent article on administering MySQL*](http://www.sitepoint.com/dbninja-mysql-client/)of great interest.

*This is the second article in a series about*[*database automation with triggers and events*](http://www.sitepoint.com/database-triggers-events/)*. A trigger is SQL code which is run just before or just after an INSERT, UPDATE or DELETE event occurs on a particular database table. Triggers have been supported in MySQL since version 5.0.2.*

## Our Database Plan

*We’ll create a small example database for a blogging application. Two tables are required:*

* *`blog`: stores a unique post ID, the title, content, and a deleted flag.*
* *`audit`: stores a basic set of historical changes with a record ID, the blog post ID, the change type (NEW, EDIT or DELETE) and the date/time of that change.*

*The following SQL creates the `blog` and indexes the deleted column:*

*CREATE TABLE `blog` (*

*`id` mediumint(8) unsigned NOT NULL AUTO\_INCREMENT,*

*`title` text,*

*`content` text,*

*`deleted` tinyint(1) unsigned NOT NULL DEFAULT '0',*

*PRIMARY KEY (`id`),*

*KEY `ix\_deleted` (`deleted`)*

*) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='Blog posts';*

*The following SQL creates the `audit` table. All columns are indexed and a foreign key is defined for audit.blog\_id which references blog.id. Therefore, when we physically DELETE a blog entry, it’s full audit history is also removed.*

*CREATE TABLE `audit` (*

*`id` mediumint(8) unsigned NOT NULL AUTO\_INCREMENT,*

*`blog\_id` mediumint(8) unsigned NOT NULL,*

*`changetype` enum('NEW','EDIT','DELETE') NOT NULL,*

*`changetime` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,*

*PRIMARY KEY (`id`),*

*KEY `ix\_blog\_id` (`blog\_id`),*

*KEY `ix\_changetype` (`changetype`),*

*KEY `ix\_changetime` (`changetime`),*

*CONSTRAINT `FK\_audit\_blog\_id` FOREIGN KEY (`blog\_id`) REFERENCES `blog` (`id`) ON DELETE CASCADE ON UPDATE CASCADE*

*) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;*

## Creating a Trigger

*We now require two triggers:*

* *When a record is INSERTed into the blog table, we want to add a new entry into the audit table containing the blog ID and a type of ‘NEW’ (or ‘DELETE’ if it was deleted immediately).*
* *When a record is UPDATEd in the blog table, we want to add a new entry into the audit table containing the blog ID and a type of ‘EDIT’ or ‘DELETE’ if the deleted flag is set.*

*Note that the changetime field will automatically be set to the current time.*

*Each trigger requires:*

1. *A****unique name****. I prefer to use a name which describes the table and action, e.g. blog\_before\_insert or blog\_after\_update.*
2. *The****table****which triggers the event. A single trigger can only monitor a single table.*
3. ***When the trigger occurs****. This can either be BEFORE or AFTER an INSERT, UPDATE or DELETE. A BEFORE trigger must be used if you need to modify incoming data. An AFTER trigger must be used if you want to reference the new/changed record as a foreign key for a record in another table.*
4. *The****trigger body****; a set of SQL commands to run. Note that you can refer to columns in the subject table using OLD.col\_name (the previous value) or NEW.col\_name (the new value). The value for NEW.col\_name can be changed in BEFORE INSERT and UPDATE triggers.*

*The basic trigger syntax is:*

*CREATE*

*TRIGGER `event\_name` BEFORE/AFTER INSERT/UPDATE/DELETE*

*ON `database`.`table`*

*FOR EACH ROW BEGIN*

*-- trigger body*

*-- this code is applied to every*

*-- inserted/updated/deleted row*

*END;*

*We require two triggers — AFTER INSERT and AFTER UPDATE on the blog table. It’s not necessary to define a DELETE trigger since a post is marked as deleted by setting it’s deleted field to true.*

*The first MySQL command we’ll issue is a little unusual:*

*DELIMITER $$*

*Our trigger body requires a number of SQL commands separated by a semi-colon (;). To create the full trigger code we must change delimiter to something else — such as $$.*

*Our AFTER INSERT trigger can now be defined. It determines whether the deleted flag is set, sets the @changetype variable accordingly, and inserts a new record into the audit table:*

*CREATE*

*TRIGGER `blog\_after\_insert` AFTER INSERT*

*ON `blog`*

*FOR EACH ROW BEGIN*

*IF NEW.deleted THEN*

*SET @changetype = 'DELETE';*

*ELSE*

*SET @changetype = 'NEW';*

*END IF;*

*INSERT INTO audit (blog\_id, changetype) VALUES (NEW.id, @changetype);*

*END$$*

*Finally, we set the delimiter back to a semi-colon:*

*DELIMITER ;*

*The AFTER UPDATE trigger is almost identical:*

*DELIMITER $$*

*CREATE*

*TRIGGER `blog\_after\_update` AFTER UPDATE*

*ON `blog`*

*FOR EACH ROW BEGIN*

*IF NEW.deleted THEN*

*SET @changetype = 'DELETE';*

*ELSE*

*SET @changetype = 'EDIT';*

*END IF;*

*INSERT INTO audit (blog\_id, changetype) VALUES (NEW.id, @changetype);*

*END$$*

*DELIMITER ;*

*It’s beyond the scope of this article, but you could consider calling a single stored procedure which handles both triggers.*

## Trigger Happy?

*Let’s see what happens when we insert a new post into our blog table:*

*INSERT INTO blog (title, content) VALUES ('Article One', 'Initial text.');*

*A new entry appears in the `blog` table as you’d expect:*

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **title** | **content** | **deleted** |
| 1 | Article One | Initial text | 0 |

*In addition, a new entry appears in our `audit` table:*

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **blog\_id** | **changetype** | **changetime** |
| 1 | 1 | NEW | 2011-05-20 09:00:00 |

*Let’s update our blog text:*

*UPDATE blog SET content = 'Edited text' WHERE id = 1;*

*As well as changing the post, a new entry appears in the `audit` table:*

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **blog\_id** | **changetype** | **changetime** |
| 1 | 1 | NEW | 2011-05-20 09:00:00 |
| 2 | 1 | EDIT | 2011-05-20 09:01:00 |

*Finally, let’s mark the post as deleted:*

*UPDATE blog SET deleted = 1 WHERE id = 1;*

*The `audit` table is updated accordingly and we have a record of when changes occurred:*

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **blog\_id** | **changetype** | **changetime** |
| 1 | 1 | NEW | 2011-05-20 09:00:00 |
| 2 | 1 | EDIT | 2011-05-20 09:01:00 |
| 3 | 1 | DELETE | 2011-05-20 09:03:00 |

*This is a simple example but I hope it’s provided some insight into the power of MySQL triggers. In my next post we’ll implement a scheduled event to archive deleted posts.*

If you enjoyed reading this post, you’ll love[*Learnable*](https://learnable.com/); the place to learn fresh skills and techniques from the masters. Members get instant access to all of SitePoint’s ebooks and interactive online courses, like[*PHP & MySQL Web Development for Beginners*](https://learnable.com/courses/php-mysql-web-development-for-beginners-13).

**How to Create Scheduled Events in MySQL**

Published July 20, 2011

This is the third and last article in a series about [database automation with triggers and events](http://www.sitepoint.com/database-triggers-events/). If you’ve not done so already, please read [How to Create Triggers in MySQL](http://www.sitepoint.com/how-to-create-mysql-triggers/) which introduces many of the concepts discussed here.

An event is similar to a trigger. However, rather than running in response to a data change, events can be scheduled to run any number of times during a specific period. In effect, it’s a database-only cron job.

Events have been supported in MySQL since version 5.1. They are ideal for maintenance tasks such as data archiving or report generation which can be scheduled during off-peak times.

## Our Database Plan

Our blog database has a problem. Old posts are marked as deleted rather than being removed from the `blog` table. Our table will grow indefinitely and become slower over time. We could purge the old posts but that would remove them forever. Therefore, we’ll move posts and their associated audit records to archive tables. The archive tables can grow without affecting the speed of the main web application and we can undelete old posts if necessary.

Two archive tables are required:

* `blog\_archive`: identical to the `blog` table except it does not require a deleted flag or an auto-incrementing ID.
* `audit\_archive`: identical to the `audit` table except the timestamp is not automatically generated and it does not require an auto-incrementing ID.

The following SQL creates both tables:

CREATE TABLE `blog\_archive` (

`id` mediumint(8) unsigned NOT NULL,

`title` text,

`content` text,

PRIMARY KEY (`id`),

) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='Blog posts archive';

--

CREATE TABLE `audit\_archive` (

`id` mediumint(8) unsigned NOT NULL,

`blog\_id` mediumint(8) unsigned NOT NULL,

`changetype` enum('NEW','EDIT','DELETE') NOT NULL,

`changetime` timestamp NOT NULL,

PRIMARY KEY (`id`),

KEY `ix\_blog\_id` (`blog\_id`),

KEY `ix\_changetype` (`changetype`),

KEY `ix\_changetime` (`changetime`),

CONSTRAINT `FK\_audit\_blog\_archive\_id` FOREIGN KEY (`blog\_id`) REFERENCES `blog\_archive` (`id`) ON DELETE CASCADE ON UPDATE CASCADE

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## Starting MySQL’s Event Scheduler

MySQL events are executed by a special event scheduler thread. It’s disabled by default so use the following MySQL command can determine whether it’s running:

***SHOW PROCESSLIST;***

If the scheduler is running, at least two rows will be shown and one will have its user field set to “event\_scheduler”. If only one row is returned, the scheduler is disabled and events will not run.

You can ensure the scheduler starts when MySQL is launched with the command-line option **--event-scheduler=ON** or setting **event\_scheduler=ON** in your MySQL configuration file (my.cnf or my.ini on Windows).

Alternatively, you can start the scheduler from the MySQL command line:

***SET GLOBAL event\_scheduler = ON;***

## Creating an Event

We require a scheduled event which:

* Copies posts from `blog` to `blog\_archive` when the deleted flag is set to 1.
* Copies the associated audit entries for those posts from `audit` to `audit\_archive`.
* Physically deletes archived posts from the `blog` table. Referential integrity has been defined with a foreign key so all associated audit entries for those posts will also be removed.

Assuming you have MySQL rights to create events, the basic syntax is:

CREATE EVENT `event\_name`

ON SCHEDULE schedule

[ON COMPLETION [NOT] PRESERVE]

[ENABLE | DISABLE | DISABLE ON SLAVE]

DO BEGIN

-- event body

END;

The schedule can be assigned various settings, e.g.

* Run once on a specific date/time:  
  AT ‘YYYY-MM-DD HH:MM.SS’  
  e.g. AT ’2011-06-01 02:00.00′
* Run once after a specific period has elapsed:  
  AT CURRENT\_TIMESTAMP + INTERVAL n [HOUR|MONTH|WEEK|DAY|MINUTE]  
  e.g. AT CURRENT\_TIMESTAMP + INTERVAL 1 DAY
* Run at specific intervals forever:  
  EVERY n [HOUR|MONTH|WEEK|DAY|MINUTE]  
  e.g. EVERY 1 DAY
* Run at specific intervals during a specific period:  
  EVERY n [HOUR|MONTH|WEEK|DAY|MINUTE] STARTS date ENDS date  
  e.g. EVERY 1 DAY STARTS CURRENT\_TIMESTAMP + INTERVAL 1 WEEK ENDS ’2012-01-01 00:00.00′

An event is normally dropped once its schedule has expired (ON COMPLETION NOT PRESERVE). Set ON COMPLETION PRESERVE to prevent that behavior. The [MySQL CREATE EVENT Syntax documentation](http://dev.mysql.com/doc/refman/5.5/en/create-event.html) provides further details.

We can now define our event (remembering to set the DELIMITER first). We’ll set it to run every week starting on a Sunday morning:

DELIMITER $$

CREATE

EVENT `archive\_blogs`

ON SCHEDULE EVERY 1 WEEK STARTS '2011-07-24 03:00:00'

DO BEGIN

-- copy deleted posts

INSERT INTO blog\_archive (id, title, content)

SELECT id, title, content

FROM blog

WHERE deleted = 1;

-- copy associated audit records

INSERT INTO audit\_archive (id, blog\_id, changetype, changetime)

SELECT audit.id, audit.blog\_id, audit.changetype, audit.changetime

FROM audit

JOIN blog ON audit.blog\_id = blog.id

WHERE blog.deleted = 1;

-- remove deleted blogs and audit entries

DELETE FROM blog WHERE deleted = 1;

END \*/$$

DELIMITER ;

This is a simple example but you could add more functionality, e.g. only move posts which were deleted at least 1 month ago and purge all archieved posts over 1 year old. I hope you’ve enjoyed this series and are considering database triggers and events in your next project.