O'REILLY
ONLamp.com
LAMP: THE OPEN SOURCE WEB PLATFORM

Published on **ONLamp.com** (http://www.onlamp.com/)
See this if you're having trouble printing code examples

Database Templates with MySQL

by Russell Dyer
03/18/2004

Once you've built several MySQL databases, you'll learn some shortcuts to database design.

Many databases are very similar. When creating new databases, developers often build the same basic tables with only slightly different names and some adjustments to columns. Rather than starting from scratch when putting together a new database, developers will sometimes copy tables from an existing database, give them new names, and then make modifications. This can be a big timesaver.

Why stop there? Take this trick a step further and put together a generic database with a set of empty, standard tables. With a well-designed MySQL template, you can quickly assemble the basics of any database as needed. A template also allows you to focus on the more interesting aspects of a database project.

In this article I will work through the process of creating a MySQL template and then apply the template to a scenario. Your clients' needs will dictate how a particular template might come together. However, in this article I will assume that we primarily develop databases for businesses that sell tangible products.

Core Tables

Databases for businesses that sell products typically consist of a core table with product descriptions and other product information. This is basically their catalog. For our template, we'll create a database called `product_template` with a core table named `products`. Let's create it with some basic, minimal columns:

```
CREATE DATABASE product_template;

USE product_template;

CREATE TABLE products (
        rec_id       INT             AUTO_INCREMENT PRIMARY KEY,
        rec_date     DATETIME,
        name         varchar(50),
        description  BLOB,
        category     VARCHAR(10),
        sub_category VARCHAR(10),
        status       ENUM('ac','in')
);

DESCRIBE products;
+--------------+-----------------+------+-----+---------+----------------+
| Field        | Type            | Null | Key | Default | Extra          |
+--------------+-----------------+------+-----+---------+----------------+
| rec_id       | int(11)         |      | PRI | NULL    | auto_increment |
| rec_date     | datetime        | YES  |     | NULL    |                |
| name         | varchar(50)     | YES  |     | NULL    |                |
| description  | blob            | YES  |     | NULL    |                |
| category     | varchar(10)     | YES  |     | NULL    |                |
| sub_category | varchar(10)     | YES  |     | NULL    |                |
```

```
| status       | enum('ac','in') | YES  |     | NULL    |                |
+--------------+-----------------+------+-----+---------+----------------+
```

Regardless of the kinds of products our clients stock, we can probably use all of these columns. Every product needs a unique number, so `rec_id` is the primary key. It's not always necessary to record the creation date, but it may prove useful in many cases so we've set up a column for it. Since each product needs a product name, we have a column called `name`. The columns `category` and `sub_category` can be handy in grouping and sub-grouping products. Their value, though, is debatable and limited.

I have found the `status` column to be particularly beneficial for all my tables. It allows me to remove a record temporarily from the user's view without having to delete it. Almost all of my `SELECT` statements contain `WHERE` clauses that filter out inactive records. Therefore, I just change the value of a record's status column to `in` (for inactive) to eliminate it from my result sets.

Of course, you may find it clearer to change the column to `active` with a Boolean type. This would let you select rows where the active status is true.

We could add several other basic columns to the `products` table. It's also missing columns that a particular client may need. However, for our purposes here and as a template these few columns are sufficient. Later in this article, after we've finished designing our template, we'll modify the table for a fictitious client to see how to apply a template.

Count Tables

Although the `products` table can contain a good deal of information on a client's merchandise, it should not contain a column for inventory quantities. We'll use another table called `inventory`. By having the inventory data in a separate table, we can have a separate row for the inventory of each item for each store location. The two tables would then be linked together by a key field.

```
CREATE TABLE inventory (
        rec_id      INT       PRIMARY KEY,
        rec_date    DATETIME,
        description BLOB,
        product_id  INT,
        quantity    INT,
        store_id    INT
);

DESCRIBE inventory;
+-------------+----------+------+-----+---------+----------------+
| Field       | Type     | Null | Key | Default | Extra          |
+-------------+----------+------+-----+---------+----------------+
| rec_id      | int(11)  |      | PRI | 0       | auto_increment |
| rec_date    | datetime | YES  |     | NULL    |                |
| description | blob     | YES  |     | NULL    |                |
| product_id  | int(11)  | YES  |     | NULL    |                |
| quantity    | int(11)  | YES  |     | NULL    |                |
| store_id    | int(11)  | YES  |     | NULL    |                |
+-------------+----------+------+-----+---------+----------------+
```

Several columns in this table have the same name as `products`. For some developers this can be confusing. For me, however, I like knowing the names of common columns without having to look again at a table's description. By reusing column names I can count on the key field of each of my tables as being `rec_id`, the record description column being `description`, and the status column being `status`.

Besides the standard columns, this *count* table includes the `product_id`, which links to `rec_id` in `products` in SQL queries. This is the first column to which we're giving a specific name. We use an identifying label here to reduce confusion, so that we know to which table the column will link. Of course we could use the same label for the key field in the main table in which it links (for example, `product_id` in `products` instead of `rec_id`), but that's not necessary. We know in the `products` table, for instance, that the primary key is the product's identification number. Giving the

key field a generic, reused name is partly a matter of style and partly a preference.

The next extra column above, `quantity`, stores quantity values. The final column is `store_id`. It links the data to yet another table, a reference table containing data on the client's store locations.

Reference Tables

To save time on typing and to speed up a database, many developers use reference tables for storing otherwise redundant data. Reference tables can hold data such as employee names, customer information, postal codes, or store locations. For our template we'll set up two minimal, generic reference tables. First we'll create one for basic data without address information:

```
CREATE TABLE reference (
        rec_id      INT       AUTO_INCREMENT PRIMARY KEY,
        rec_date    DATETIME,
        name        VARCHAR(25),
        description BLOB,
        status      ENUM('ac','in')
);

DESCRIBE reference;
+-------------+-----------------+------+-----+---------+-------+
| Field       | Type            | Null | Key | Default | Extra |
+-------------+-----------------+------+-----+---------+-------+
| rec_id      | int(11)         | YES  |     | NULL    |       |
| rec_date    | datetime        | YES  |     | NULL    |       |
| name        | varchar(25)     | YES  |     | NULL    |       |
| description | blob            | YES  |     | NULL    |       |
| status      | enum('ac','in') | YES  |     | NULL    |       |
+-------------+-----------------+------+-----+---------+-------+
```

By now the format of our generic tables probably seems a bit dreary and very familiar. However, it's this dullness that makes a template reusable and adaptable. It's the familiarity that allows a developer to create SQL statements intuitively and quickly.

For reference tables that will contain addresses, we need a table called `addresses` to contain many of the common columns:

```
CREATE TABLE addresses (
        rec_id       INT          AUTO_INCREMENT PRIMARY KEY,
        rec_date     DATETIME,
        name         VARCHAR(50),
        name2        VARCHAR(50),
        description  BLOB,
        category     VARCHAR(10),
        sub_category VARCHAR(10),
        address1     VARCHAR(50),
        address2     VARCHAR(50),
        city         VARCHAR(50),
        state        VARCHAR(8),
        postal_code  VARCHAR(10),
        country      VARCHAR(25),
        contact      VARCHAR(50),
        telephone    VARCHAR(15)
);

DESCRIBE addresses;
+--------------+-------------+------+-----+---------+----------------+
| Field        | Type        | Null | Key | Default | Extra          |
+--------------+-------------+------+-----+---------+----------------+
| rec_id       | int(11)     |      | PRI | NULL    | auto_increment |
| rec_date     | datetime    | YES  |     | NULL    |                |
| name         | varchar(50) | YES  |     | NULL    |                |
| name2        | varchar(50) | YES  |     | NULL    |                |
```

```
| description  | blob        | YES  |     | NULL    |                |
| category     | varchar(10) | YES  |     | NULL    |                |
| sub_category | varchar(10) | YES  |     | NULL    |                |
| address1     | varchar(50) | YES  |     | NULL    |                |
| address2     | varchar(50) | YES  |     | NULL    |                |
| city         | varchar(50) | YES  |     | NULL    |                |
| state        | varchar(8)  | YES  |     | NULL    |                |
| postal_code  | varchar(10) | YES  |     | NULL    |                |
| country      | varchar(25) | YES  |     | NULL    |                |
| contact      | varchar(50) | YES  |     | NULL    |                |
| telephone    | varchar(15) | YES  |     | NULL    |                |
+--------------+-------------+------+-----+---------+----------------+
```

This table has two name columns: one for the person's first name and another for the last name -- again, rename these columns to your preference. This address table is probably more suited for people rather than businesses, so you may want to create another table for businesses.

Depending on the client's needs, we will use the generic tables above to create several reference tables of different names and uses. When making copies, we could rename some of them, but I prefer to leave these columns unchanged. Invariably we will need to add more columns for specific reference tables. We'll see examples of this in the next section when we use our template.

Deploying a Template

Once we have constructed our products-database template with a basic set of generic tables, we can then deploy the template for developing databases for clients. For the examples in this section we will assume that a bookstore business with a few locations has contracted us to set up a database.

The first step is to copy the template to a new database. There are many methods and utilities we could use, but we'll create a new database `bookstore` and then copy the tables from `product_template` to the `products` table in the new database using a `CREATE TABLE...SELECT` statement.

```
CREATE DATABASE bookstore;

USE bookstore;

CREATE TABLE books
SELECT * FROM products_template.products;
```

Although this simple method of duplicating the needed generic table will work, it has two basic flaws: First, it copies the data along with the table layout. This shouldn't be a problem if we're copying a table that's part of our template: it won't contain any data. If it did, then a simple `DELETE` statement could clear out the data in the new table.

The second problem is with the primary key, `rec_id`. The method above doesn't duplicate the index, and the column does not have auto-increment enabled. The `rec_id` column is created, but it's not indexed and the `auto_increment` isn't there. We could just alter the new table:

```
ALTER TABLE books
CHANGE COLUMN rec_id
rec_id INT AUTO_INCREMENT PRIMARY KEY;
```

An alternative to `CREATE TABLE...SELECT` is to run a `SHOW CREATE TABLE` statement, modify the results for the new table name, and then run the modified results:

```
SHOW CREATE TABLE products_template.products;

CREATE TABLE `products` (
  `rec_id`       int(11)         NOT     NULL auto_increment,
  `rec_date`     datetime        default NULL,
  `name`         varchar(50)     default NULL,
  `description`  blob,
```

```
  `category`     varchar(10)     default NULL,
  `sub_category` varchar(10)     default NULL,
  `status`       enum('ac','in') default NULL,
  PRIMARY KEY  (`rec_id`)
) TYPE=MyISAM
```

The output shown here is without the table formatting lines; this is at the core of the results. We would copy this text (the `CREATE TABLE...` section) into a text editor and then change the table name from `products` to `books`. Next we would paste it back into the `mysql` client to create a new table with the proper settings and without the data. Of course, if we needed to do this more than once, it would be worth scripting.

Once we have the basics of `books` established, we can then change the table for the client's specific needs. The products table for a bookstore will require several additional columns. Also, `books` needs to link to `inventory` and a few reference tables. We'll have to alter the table to add the needed columns:

```
ALTER TABLE books
ADD COLUMN author_id INT,
ADD COLUMN genre INT,
ADD COLUMN publisher INT,
ADD COLUMN pub_year CHAR(4),
ADD COLUMN isbn VARCHAR(20);

DESCRIBE books;
+--------------+-----------------+------+-----+---------+----------------+
| Field        | Type            | Null | Key | Default | Extra          |
+--------------+-----------------+------+-----+---------+----------------+
| rec_id       | int(11)         |      | PRI | NULL    | auto_increment |
| rec_date     | datetime        | YES  |     | NULL    |                |
| name         | varchar(50)     | YES  |     | NULL    |                |
| description  | blob            | YES  |     | NULL    |                |
| category     | varchar(10)     | YES  |     | NULL    |                |
| sub_category | varchar(10)     | YES  |     | NULL    |                |
| status       | enum('ac','in') | YES  |     | NULL    |                |
| author_id    | int(11)         | YES  |     | NULL    |                |
| genre        | int(11)         | YES  |     | NULL    |                |
| publisher    | int(11)         | YES  |     | NULL    |                |
| pub_year     | varchar(4)      | YES  |     | NULL    |                |
| isbn         | varchar(20)     | YES  |     | NULL    |                |
+--------------+-----------------+------+-----+---------+----------------+
```

Based on the columns we've added to `products`, we will need to make copies of the reference tables: two copies of `reference` for authors and genres (plays and novels, for example) and a copy of `addresses` for publisher data. We can copy all three tables without adding very many columns to them.

The result of this exercise has been to show that in just a few minutes you can have a good starting point for a new client database. You have your products table, your inventory table, and various reference tables that all link to the main tables. All you need to do is to find out any additional columns that the client needs and add them to the appropriate tables. This can be quite impressive.

A Collection of Templates

The database examples given in this article were for clients that run a product sales business. Although it may easily be modifiable for various other types of clients, in time you may want to develop a set of templates for several different client types. It's not necessary to think ahead too much.

As you find yourself putting together a database for a new type of client, keep in mind which columns might be generic (and label them accordingly), and which columns are client specific. When you've finished the primary tables, make a copy of the database and strip out the extra client columns. Then add the new template to your toolbox for future use. In time you will be ready to set up any type of database and will be able to keep your workload manageable when taking on a new project.

Russell Dyer has worked full-time for several years as a free-lance writer of computer articles, primarily on MySQL.

Return to ONLamp.com.