An Introduction to Database Normalization (part 2)

By W.J. Gilmore

All materials Copyright © 1997−2002 Developer Shed, Inc. except where otherwise noted.

Table of Contents

Introduction

As your Web application grows in size and complexity, the importance of employing sound design techniques begins to weigh particularly heavily into the project's success and longetivity. Since many of these large−scale productions incorporate database usage, taking time to adequately plan for the scaleability and manageability of the data to be stored within the database should be a high−priority item on your initial design checklist.

Last week's article, An Introduction to Database Normalization, introduced you to the basic vocabulary and underlying principles of creating efficient database table structures. This week, I'll build upon what you have learned thus far, creating a sample application using the popular MySQL database server. After finishing this tutorial, you'll be able to use what you have learned about MySQL syntax to begin building your own MySQL−powered applications, or use this syntax as the basis for learning how to create normalized databases on your database server of choice.

Before delving into this introduction of MySQL syntax as it applies to normalized databases, allow me to begin by introducing the sample application that will be created in this tutorial.

The Project

I've been messing around with XML and XSL transformations quite a bit lately, using my favorite scripting language, PHP, to create XML documents on−the−fly using data retrieved from a MySQL database. This is cool, because I can then use PHP 4.03's new Sablotron extension to transform these XML documents into a format viewable within a PC, PDA or cellular phone WAP browser. Although discussing the details of how this is accomplished is certainly out of the scope of this article, a database structure used within one of my new XML−oriented applications is perfectly suited for the purposes of this article.

The database that we'll create stores information used within an online news application. Using a Web interface, a select group of administrators add and submit newsworthy items that will then be inserted into the database. This information can fall under one of a group of categories, the name of each category being predetermined by the site administrator. To make things really interesting, the news service offers news items in different languages, and therefore each news item must contain some information specifying which language it is written in.

Note that in providing just this brief description, several questions already arise in terms of scaleability:

1. How many news items can be stored in the database?
2. How many administrators are able to add news items?
3. How many categories are there?
4. How many languages are there?

The answer to these questions are simple: We don't know. Therefore, it is in our best interests to build the most scaleable and manageable database table structure possible through proper database normalization.

To begin, take note that four tables are needed to properly store the news data: news, administrators, categories, and languages. I'll round out this project introduction with a synopsis of the columns used within each table found within the database.

Table: news − Contains the information relative to each news item.

news_id:	Unique identification number for the news item (uniquely identifies a given row).
category_id:	Specifies the identification number of the category under which this news item belongs.

author_id:	Specifies the identification number of the author who submitted this news item.
language_id:	Specifies the identification number of the language in which this news item is written.

title:	The title that is used as the header for the news item.
submit_time:	Specifies the date and time of the submission.
summary:	This is the actual information conveyed by the news item.

Table: administrators − Contains the information specific to each administrator.

admin_id:	Unique identification number for the author (uniquely identifies a given row).
name:	Specifies the author name.

email:	Specifies the author email address.
url:	Specifies the author URL, if available.

Table: categories − Contains the categories under which each news item can belong.

category_id:	Unique identification number for the category (uniquely identifies a given row).
name:	**Specifies the category name.**

Table: languages − Contains the languages under which each news item can be written in.

language_id:	Unique identification number for the language (uniquely identifies a given row).
name:	Specifies the language name.

Note that each table contains a unique identification number (primary key). This makes it possible to reference one specific row with a simple query. Furthermore, take note that the news table contains several foreign keys, which relate to another table in which that foreign key is the primary key of the respective table. Perhaps most importantly, however, is the fact that the grouping of data into well−defined tables allows for the subsequent expansion and editing of administrators, languages, categories without the need to perform massive updates to existing data.

In the next section, I'll show how these tables are created using MySQL syntax.

Creating the news tables in the MySQL database

Since its public release in October 1996, MySQL (http://www.mysql.com) has enjoyed a rapid gain in popularity. This is not surprising, considering the speed and power it offers to its users. Recently released under the GNU General Public License (GPL), MySQL is poised to become the 800−LB. gorilla in the realm of Open Source databasing technologies.

Let's begin by creating the three least complex tables: *languages*, *categories* and *administrators*:

```
mysql>create table languages ( ->language_id SMALLINT UNSIGNED
NOT NULL AUTO_INCREMENT, ->name CHAR(20) NOT NULL, ->PRIMARY
KEY (language_id) ); mysql>create table categories (
->category_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
->name CHAR(20) NOT NULL, ->PRIMARY KEY (category_id) );
mysql>create table administrators ( ->admin_id SMALLINT
UNSIGNED NOT NULL AUTO_INCREMENT, ->name CHAR(20) NOT NULL,
->email CHAR(40) NOT NULL, ->url CHAR(50) NULL, ->PRIMARY KEY
(admin_id) );
```

As you can see, several attributes accompany the column definitions. I'll introduce each here:

- SMALLINT: A SMALLINT is a relatively small integer having a range between −32768 and 32767.
- UNSIGNED: This attribute modifies the SMALLINT range to be between 0 and 65535. Therefore, 65535 languages will be allowed into the table, a number obviously larger than could ever be reached.
- NOT NULL: This ensures that the column cannot be left blank.
- NULL: This specifies that the column can be left blank. Alternatively, if neither NULL or NOT NULL are specified, NULL will become by default the attribute.
- AUTO_INCREMENT: An integer column attribute, AUTO_INCREMENT will result in the automatic insertion of value+1, where value is the largest column value currently found in the table.
  CHAR(N): A CHAR specifies a string of 1 or more characters. The maximum number of characters is specified by N.
- PRIMARY KEY: A PRIMARY KEY is essentially a unique key value for the table. There can only be one PRIMARY KEY in a given table. Any column defined to be the PRIMARY KEY must also be defined as NOT NULL.

Next I'll create the most complex table of the four, the news table. In addition to a primary key, this table contains several foreign keys:

```
mysql>create table news ( ->news_id SMALLINT UNSIGNED NOT NULL
AUTO_INCREMENT, ->category_id SMALLINT UNSIGNED NOT NULL
REFERENCES categories, ->admin_id SMALLINT UNSIGNED NOT NULL
REFERENCES administrators, ->language_id SMALLINT UNSIGNED NOT
```

```
NULL REFERENCES languages, ->title char(100) NOT NULL,
->submit_time TIMESTAMP(12) NOT NULL, ->summary TINYTEXT NOT
NULL, ->PRIMARY KEY (news_id) );
```

You've already been introduced to most of the column attributes used to create the news table. However, there are a few used within the *news* table creation worth noting:

- TIMESTAMP(N): The TIMESTAMP column contains the date and (optionally) time of the row insertion or update. In the case of the news table, I set N to be 12, which would result in the date and time being stored in the format YYMMDDHHMMSS. You can create TIMESTAMP values of different sizes, but 12 will suffice for our purposes in this case.
- TINYTEXT: This is essentially a string of characters limited to 255 characters or less.
- REFERENCES tablename: This specifies a Foreign Key constraint. The table reference refers to the PRIMARY Key specified within the table specified by *tablename.*

For those of you coming from other database disciplines, you may find it interesting that MySQL does not support Foreign Keys in the way that you would think. MySQL does not ensure that the value placed within the Foreign Key column is one that already exists within the Primary Key column of the table referenced within the Foreign Key column definition; It is up to you to ensure that this verification is a part of your application logic. The reason for the REFERENCES attribute is to facilitate the porting of code between other SQL implementations.

While you may find the MySQL developer's logic questionable in their choice to not implement Foreign Key support, their decision is not without reason; Continuous verification of value integrity between the Primary Key and Foreign Key values would result in a potentially substantial performance degradation.

Next section, I'll demonstrate how database normalization can vastly improve performance when querying tables.

Querying the MySQL database

As you likely already know, interaction with an SQL database takes place through queries. A query is nothing more than a request for information from a database, this request being a phrase constructed from various keywords and table / column references. In this section, I'll demonstrate using queries how database normalization makes an administrator's life considerably easier. To begin, assume that the news table has been filled with this information.

Using this information, assume I execute the following query:

```
mysql>SELECT n.title, a.name FROM news n, administrators a
WHERE –>n.admin_id = a.admin_id AND a.admin_id = 2;
```

Resulting in:

title	name
Nuevo Sitio: www.ziobudda.net	Michel

Knowing that the admin_id '2' maps to 'Michel', we can build our query without worrying that Michel will be misspelled. Furthermore, if the administrator later decides to changes his name to just the initials 'M.M.', the query will not need to be changed because the criteria is based upon an admin_id and not a name. Additionally, this name change will not require the consumption of potentially magnanimous amounts of resources, as would be the case if the name were included along with each row of the *news* table.

Considering another example, consider the query:

```
mysql>SELECT n.title FROM news n, categories c WHERE
–>n.category_id = c.category_id AND c.category_id = 2;
```

This yields:

title	name
Nuovo Sito: www.phpitalia.com	New Sites
Nuevo Sitio: www.ziobudda.net	New Sites

Now suppose that I want to update one of the category names found in the categories table, in particular I want to change the name 'New Sites' to 'Great Sites'. All that I need to do is update *one* row found in the *categories* table:

```
mysql>update categories set name = 'Great Sites' –>where
category_id = '2';
```

Again, executing the previous SELECT query would result in:

title	name
Nuovo Sito: www.phpitalia.com	Great Sites
Nuevo Sitio: www.ziobudda.net	Great Sites

Of course, your Web application is not likely to repeat the category name, but this will certainly make a difference when using dynamically−named table headers.

Conclusion

And thus concludes the two−part series on database normalization. Hopefully, I have succinctly explained the importance of taking time to plan for the efficient design of database tables, no matter how limited the initial intention of the Web application may be. For those of you interested in delving further into the depths of database normalization and advanced SQL queries, allow me to recommend the following online resources:

- "Introduction to Structured Query Language"
- "Database Normalization"
- "SQL by Design: Why You Need Database Normalization"