



:: DEVELOPER ZONE

[Login](#) / [Register](#)

An Introduction to Database Normalization

By Mike Hillyer

When users ask for advice about their database applications, one of the first things I try to help them with is the normalization of their table structure. Normalization is the process of removing redundant data from your tables in order to improve storage efficiency, data integrity and scalability. This improvement is balanced against an increase in complexity and potential performance losses from the joining of the normalized tables at query-time.

Mike's Bookshop

Let's say you were looking to start an online bookstore. You would need to track certain information about the books available to your site viewers, such as:

- Title
- Author
- Author Biography
- ISBN
- Price
- Subject
- Number of Pages
- Publisher
- Description
- Review
- Reviewer Name

Let's start by adding a couple of books written by Luke Welling and Laura Thomson. Because this book has two authors, we are going to need to accommodate both in our table. Lets take a look at a typical approach I encounter:

Title	Author1	Author2	ISBN	Subject	Pages	Publisher
PHP and MySQL Web Development	Luke Welling	Laura Thomson	0672317842	PHP, MySQL	867	Sams
MySQL Tutorial	Luke Welling	Laura Thomson	0672325845	MySQL	300	Sams

Let's take a look at some issues involved in this table design:

First, this table is not very efficient with storage. Lets imagine for a second that Luke and Laura were extremely busy writers and managed to produce 500 books for our database. The combination of their two names is 25 characters long, and since we will repeat their two names in 500 rows we are wasting $25 \times 500 = 12,500$ bytes of storage space unnecessarily.

Second, this design does not protect data integrity. Lets once again imagine that Luke and Laura have written 500 books. Someone has had to type their names into the database 500 times, and it is very likely that one of their names will be misspelled at least once (i.e.. Thompson instead of Thomson). Our data is now corrupt, and anyone searching for book by author name will find some of the results missing. The same thing could happen with publisher name. Sams publishes hundreds of titles and if the publisher's name were misspelled even once the list of books by publisher would be missing titles.

Third, this table does not scale well. First of all, we have limited ourselves to only two authors, yet some books are written by over a dozen people. This kind of limitation is often exhibited in personal info tables where the typical design includes Phone1, Phone2, and Phone3 columns. In both cases we are limiting future growth of our data. Another limitation to scalability is that fact that with all our data in one table our one table file will grow faster and we will have more trouble with file size limitations of our underlying operating system.

First Normal Form

The normalization process involves getting our data to conform to three progressive normal forms, and a higher level of normalization cannot be achieved until the previous levels have been achieved (there are actually five normal forms, but the last two are mainly academic and will not be discussed). The First Normal Form (or 1NF) involves removal of redundant data from horizontal rows. We want to ensure that there is no duplication of data in a given row, and that every column stores the least amount of information possible (making the field atomic).

In our table above we have two violations of First Normal Form: First, we have more than one author field, and our subject field contains more than one piece of information. With more than one value in a single field, it would be very difficult to search for all books on a given subject. In addition, with two author fields we have two fields to search in order to look for a book by a specific author

We could get around these problems by modifying our table to have only one author field, with two entries for a book with two authors, as in the following example:

Title	Author	ISBN	Subject	Pages	Publisher
PHP and MySQL Web Development	Luke Welling	0672317842	MySQL	867	Sams
PHP and MySQL Web Development	Laura Thomson	0672317842	PHP	867	Sams
MySQL Tutorial	Laura Thomson	0672325845	MySQL	300	Sams
MySQL Tutorial	Luke Welling	0672325845	MySQL	300	Sams

While this approach has no redundant columns and the subject column has only one piece of information, we do have a problem that we now have two rows for a single book. Also, to ensure that we can do a search of author and subject (i.e. books on PHP by Luke Welling), we would need four rows to ensure that we had each combination of author and subject. Additionally, we would be violating the Second Normal Form, which will be described below.

A better solution to our problem would be to separate the redundant data into separate tables, with the tables related to each other. In this case we will create an Author table and a Subject table to store our information, removing that information from the Book table:

Author Table:

Author_ID	Last Name	First Name	Bio
1	Welling	Luke	Author of books on PHP and MySQL
2	Thomson	Laura	Another author of books on PHP and MySQL

Subject Table:

Subject_ID	Subject
1	PHP
2	MySQL

Book Table:

ISBN	Title	Pages	Publisher
0672317842	PHP and MySQL Web Development	867	Sams
0672325845	MySQL Tutorial	300	Sams

While creating the Author table, I also split author name down into separate first name and last name columns, in order to follow the requirement for storing as little information as possible in a given column. Each table has a primary key value which can be used for joining tables together when querying the data. A primary key value must be unique with in the table (no two books can have the same ISBN number), and

a primary key is also an INDEX, which speeds up data retrieval based on the primary key.

Defining Relationships

As you can see, while our data is now split up, relationships between the table have not been defined. There are various types of relationships that can exist between two tables:

- One to (Zero or) One
- One to (Zero or) Many
- Many to Many

The relationship between the Book table and the Author table is a many-to-many relationship: A book can have more than one author, and an author can write more than one book. To represent a many-to-many relationship in a relational database we need a third table to serve as a link between the two:

Book_Author Table:

ISBN	Author_ID
0672317842	1
0672317842	2
0672325845	1
0672325845	2

Similarly, the Subject table also has a many-to-many relationship with the Book table, as a book can cover multiple subjects, and a subject can be explained by multiple books:

Book_Subject Table:

ISBN	Subject_ID
0672317842	1
0672317842	2
0672325845	2

The one-to-many relationship will be covered in the next section. As you can see, we now have established the relationships between the Book, Author, and Subject tables. A book can have an unlimited number of authors, and can refer to an unlimited number of subjects. We can also easily search for books by a given author or referring to a given subject.

In the linking tables above the values stored refer to primary key values from the Book, Author, and Subject tables. Columns in a table that refer to primary keys from another table are known as foreign keys, and serve the purpose of defining data relationships. In database systems which support referential integrity, such as the InnoDB storage engine for MySQL, defining a column as a foreign key will also allow to database software to enforce the relationships you define. For example, with foreign keys defined, the InnoDB storage engine will not allow you to insert a row into the Book_Subject table unless the book and subject in question already exist in the book and subject tables. Such systems will also prevent the deletion of books from the book table that have 'child' entries in the book_subject or book_author tables.

Second Normal Form

Where the First Normal Form deals with redundancy of data across a horizontal row, Second Normal Form (or 2NF) deals with redundancy of data in vertical columns. As stated earlier, the normal forms are progressive, so to achieve Second Normal Form, your tables must already be in First Normal Form. Lets take another look at our new Book table:

Book Table:

ISBN	Title	Pages	Publisher
-------------	--------------	--------------	------------------

0672317842 PHP and MySQL Web Development	867	Sams
0672325845 MySQL Tutorial	300	Sams

This table is in First Normal Form; we do not repeat the same data in any two columns, and each column holds only the minimum amount of data. However, in the Publisher column we have the same publisher repeating in each row. This is a violation of Second Normal Form. As I stated above, this leaves the chance for spelling errors to occur. The user needs to type in the publisher name each time and such an approach is also inefficient in regards to storage usage, and the more data a table has the more IO requests are needed to scan the table, resulting in slower queries.

To normalize this table to Second Normal Form, we will once again break data off to a separate table. In this case we will move publisher information to a separate table as follows:

Publisher Table:

Publisher_ID	Publisher Name
1	Sams

In this case we have a one-to-many relationship between the book table and the publisher. A given book has only one publisher (for our purposes), and a publisher will publish many books. When we have a one-to-many relationship, we place a foreign key in the many, pointing to the primary key of the one. Here is the new Book table:

Book Table:

ISBN	Title	Pages	Publisher_ID
0672317842	PHP and MySQL Web Development	867	1
0672325845	MySQL Tutorial	300	1

Since the Book table is the 'many' portion of our one-to-many relationship, we have placed the primary key value of the 'one' portion in the Publisher_ID column as a foreign key.

The other requirement for Second Normal Form is that you cannot have any data in a table with a composite key that does not relate to all portions of the composite key. A composite key is a primary key that incorporates more than one column, as with our Book_Author table:

Let's say we wanted to add in tracking of reviews and started with the following table:

Review Table:

ISBN	Reviewer_ID	Review_Text	Reviewer_Name
0672325845	1	What a great book! I learned a lot!	Mike Hillyer

In this table the combination of Reviewer_ID and ISBN form the primary key (a composite key), ensuring that no reviewer writes more than one review for the same book. In this case the reviewer name is only related to the Reviewer_ID, and not the ISBN, and is therefore in violation of Second Normal Form.

In this case we solve the problem by having a separate table for the data that relates to only one part of the key. If the portion of the primary key that the field relates to is a foreign key to another table, move the data there (as in this situation where the Reviewer_ID will be a separate table):

Reviewer Table:

Reviewer_ID	Reviewer_Name	Reviewer_House	Reviewer_Street	Reviewer_City	Reviewer_Province	Reviewer_Pos
1	Mike Hillyer	123	Main Street	Calgary	Alberta	T1S-2N2

Review Table:

ISBN	Reviewer_ID	Review_Text
0672325845	1	What a great book! I learned a lot!

Third Normal Form

I have a confession to make; I do not often use Third Normal Form. In Third Normal Form we are looking for data in our tables that is not fully dependant on the primary key, but dependant on another value in the table. In the reviewer table above the Reviewer_Street and Reviewer_City fields are really dependant on the Reviewer_PostalCode and not the Reviewer_ID. To bring this table into compliance with Third Normal Form, we would need a table based on Postal Code:

PostalCode Table:

Postal_Code	Street	City
T1S-2N2	Main Street	Calgary

Now of course this new table violates Second Normal Form as the Street and City will be vertically redundant, and must be broken off to separate Street and City tables, and Province will need to be in it's own table which the City table will refer to as a foreign key.

The point of the last example is that Normalization is a tradeoff. In fact, there are two additional normal forms that are generally recognized, but are mainly academic. Additional normalization results in more complex JOIN queries, and this can cause a decrease in performance. In some cases performance can even be increased by de-normalizing table data, but de-normalization is beyond the scope of this article.

Conclusion

It is generally a good idea to make sure that your data at least conforms to Second Normal Form. Our goal is to avoid data redundancy to prevent corruption and make the best possible use of storage. Make sure that the same value is not stored in more than one place. With data in multiple locations we have to perform multiple updates when data needs to be changed, and corruption can begin to creep into the database.

Third Normal Form removed even more data redundancy, but at the cost of simplicity and performance. In our example above, do we really expect the City and Street names to change very regularly? In this situation Third Normal Form still prevents misspelling of City and Street names, but it is up to you to find a balance between Normalization and Speed/Simplicity. In an upcoming article we will look at how to form queries that join our normalized data together.

About the Author



Mike Hillyer is a Technical Writer for MySQL AB and lives in Alberta, Canada.

We are always looking for interesting articles about MySQL! Have you written something and would like to it published here? Please contact us via [feedback form](#) »

“We get much more performance out of MySQL Server than we did with Oracle. I am happy to be able to provide our customers with quick services over our web site and, to a large extent, this is thanks to MySQL.”—Kazushige Sato, IT manager in charge of online services for Aizawa Securities Co.

[Read more about how Aizawa Securities Co. uses MySQL software....](#)
