Normalization

Many people have trouble organizing data. One of the most challenging of these to maintain and rework suffer from what one author described as the “Spreadsheet Syndrome”: a tendency for the developer to lump every possible piece of information into as few table as possible, often into a single table. A schema that suffers from the “Spreadsheet Syndrome” is subject to data redundancies, data anomalies, and various inefficiencies. The cure for “Spreadsheet Syndrome” is database normalization.

Database normalization is a process by which an existing schema is modified to bring its component tables into compliance with a series of progressive normal forms. The concept of database normalization was first introduced by Edgar Frank Codd in his paper A Relational Model of Data for Large Shared Data Banks, section 4.

The goal of database normalization is to ensure that every non-key column in every table is directly dependent on the key, the whole key and nothing but the key and with this goal come benefits in the form of reduced redundancies, fewer anomalies, and improved efficiencies. While normalization is not the be-all and end-all of good design, a normalized schema provides a good starting point for further development.

There are seven recognized normal forms. We are going to focus on the first three.

Mike's Bookstore

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
- Publisher Address
- Description
- Review
- Reviewer Name

Let's start by adding the book that coined the term Spreadsheet Syndrome. Because this book has two authors, we are going to need to accommodate both in our table. Let's take a look at a typical approach

Table 1. Two Books
Beginning MySQL Database Design and Optimization

Chad Russell is a programmer and network administrator who owns his own Internet hosting company., Jon Stephens is a member of the MySQL AB documentation team.

MySQL, Database Design 1590593324

520 Apress

Lets take a look at some issues involved in this design:

First, this table is subject to several anomalies: we cannot list publishers or authors without having a book because the ISBN is a primary key which cannot be NULL (referred to as an insertion anomaly). Similarly, we cannot delete a book without losing information on the authors and publisher (a deletion anomaly). Finally, when updating information, such as an author's name, we must change the data in every row, potentially corrupting data (an update anomaly).

Note:

Normalization is a part of relational theory, which requires that each relation (AKA table) has a primary key. As a result, this article assumes that all tables have primary keys, without which a table cannot even be considered to be in first normal form.

Second, this table is not very efficient with storage. Lets imagine for a second that our publisher is extremely busy and managed to produce 5000 books for our database. Across 5000 rows we would need to store information such as a publisher name, address, phone number, URL, contact email, etc. All that information repeated over 5000 rows is a serious waste of storage resources.

Third, this design does not protect data consistency. Lets once again imagine that Jon Stephens has written 20 books. Someone has had to type his name into the database 20 times, and it is possible that his name will be misspelled at least once (i.e.. John Stevens instead of Jon Stephens). Our data is now in an inconsistent state, and anyone searching for a book by author name will find some of the results missing. This also contributes to the update anomalies mentioned earlier.

**First Normal Form**

The normalization process involves getting our data to conform to progressive normal forms, and a higher level of normalization cannot be achieved unless the previous levels have been satisfied (though many experienced designers can create normalized tables directly without iterating through the lower forms). The first normal form (or 1NF) requires that the values in each column of a table are atomic. By atomic we mean that there are no sets of values within a column.

In our example table, we have a set of values in our author and subject columns. With more than one value in a single column, it is difficult to search for all books on a given subject or by a specific author. In addition, the author names themselves are non-atomic: first name and last name are in fact different values. Without separating first and last names it becomes difficult to sort on last name.
One method for bringing a table into first normal form is to separate the entities contained in the table into separate tables. In our case this would result in Book, Author, Subject and Publisher tables.

Table 2. Book Table

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1590593324</td>
<td>Beginning MySQL Database Design and Optimization</td>
<td>520</td>
</tr>
</tbody>
</table>

Table 3. Author Table

<table>
<thead>
<tr>
<th>Author_ID</th>
<th>First_Name</th>
<th>Last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Chad</td>
<td>Russell</td>
</tr>
<tr>
<td>2</td>
<td>Jon</td>
<td>Stephens</td>
</tr>
<tr>
<td>3</td>
<td>Mike</td>
<td>Hillyer</td>
</tr>
</tbody>
</table>

Table 4. Subject Table

<table>
<thead>
<tr>
<th>Subject_ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>Database Design</td>
</tr>
</tbody>
</table>

Table 5. Publisher Table

<table>
<thead>
<tr>
<th>Publisher_ID</th>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Apress</td>
<td>2560 Ninth Street, Station 219</td>
<td>Berkeley</td>
<td>California</td>
<td>94710</td>
</tr>
</tbody>
</table>

Note

The Author, Subject, and Publisher tables use what is known as a surrogate primary key - an artificial primary key used when a natural primary key is either unavailable or impractical. In the case of author we cannot use the combination of first and last name as a primary key because there is no guarantee that each author's name will be unique, and we cannot assume to have the author's government ID number (such as SSN), so we use a surrogate key.

Some developers use surrogate primary keys as a rule, others use them only in the absence of a natural candidate for the primary key. From a performance point of view, an integer used as a surrogate primary key can often provide better performance in a join than a composite primary key across several columns. However, when using a surrogate primary key it is still important to create a UNIQUE key to ensure that duplicate records are not created inadvertently (but some would argue that if you need a UNIQUE key it would be better to stick to a composite primary key).

By separating the data into different tables according to the entities each piece of data represents, we can now overcome some of the anomalies mentioned earlier: we can add authors who have not yet written books, we can delete books without losing author or
publisher information, and information such as author names are only recoded once, preventing potential inconsistencies when updating.

Depending on your point of view, the Publisher table may or may not meet the 1NF requirements because of the Address column: on the one hand it represents a single address, on the other hand it is a concatenation of a building number, street number, and street name.

The decision on whether to further break down the address will depend on how you intend to use the data: if you need to query all publishers on a given street, you may want to have separate columns. If you only need the address for mailings, having a single address column should be acceptable (but keep potential future needs in mind).

**Defining Relationships**

As you can see, while our data is now split up, relationships between the tables 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. By naming the table appropriately, it becomes instantly clear which tables it connects in a many-to-many relationship (in the following example, between the Book and the Author table).

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Author_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1590593324</td>
<td>1</td>
</tr>
<tr>
<td>1590593324</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6. Book_Author Table

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:

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Subject_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1590593324</td>
<td>1</td>
</tr>
<tr>
<td>1590593324</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7. Book_Subject Table

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.
The case of a one-to-many relationship exists between the Book table and the Publisher table. 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 table representing the many, pointing to the primary key of the table representing the one. Here is the new Book table:

Table 8. Book Table

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Title</th>
<th>Pages</th>
<th>Publisher_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1590593324</td>
<td>Beginning MySQL Database Design and Optimization</td>
<td>520</td>
<td>1</td>
</tr>
</tbody>
</table>

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

In the tables above the values stored refer to primary key values from the Book, Author, Subject and Publisher 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 (DBMS) which support referential integrity constraints, such as the InnoDB storage engine for MySQL, defining a column as a foreign key will allow the DBMS 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 or if you're inserting NULL values. 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 atomicity of data, the Second Normal Form (or 2NF) deals with relationships between composite key columns and non-key columns. As stated earlier, the normal forms are progressive, so to achieve Second Normal Form, your tables must already be in First Normal Form.

The second normal form (or 2NF) any non-key columns must depend on the entire primary key. In the case of a composite primary key, this means that a non-key column cannot depend on only part of the composite key.

Let's introduce a Review table as an example.

Table 9. Review Table

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Author_ID</th>
<th>Summary</th>
<th>Author_URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1590593324</td>
<td>3</td>
<td>A great book!</td>
<td><a href="http://www.openwin.org">http://www.openwin.org</a></td>
</tr>
</tbody>
</table>
In this situation, the URL for the author of the review depends on the Author_ID, and not to the combination of Author_ID and ISBN, which form the composite primary key. To bring the Review table into compliance with 2NF, the Author_URL must be moved to the Author table.

**Third Normal Form**

Third Normal Form (3NF) requires that all columns depend directly on the primary key. Tables violate the Third Normal Form when one column depends on another column, which in turn depends on the primary key (a transitive dependency).

One way to identify transitive dependencies is to look at your table and see if any columns would require updating if another column in the table was updated. If such a column exists, it probably violates 3NF.

In the Publisher table the City and State fields are really dependent on the Zip column and not the Publisher_ID. To bring this table into compliance with Third Normal Form, we would need a table based on zip code:

Table 10. Zip Table

<table>
<thead>
<tr>
<th>Zip</th>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>94710</td>
<td>Berkeley</td>
<td>California</td>
</tr>
</tbody>
</table>

In addition, you may wish to instead have separate City and State tables, with the City_ID in the Zip table and the State_ID in the City table.

A complete normalization of tables is desirable, but you may find that in practice that full normalization can introduce complexity to your design and application. More tables often means more JOIN operations, and in most database management systems (DBMSs) such JOIN operations can be costly, leading to decreased performance. The key lies in finding a balance where the first three normal forms are generally met without creating an exceedingly complicated schema.

**Joining Tables**

With our tables now separated by entity, we join the tables together in our SELECT queries and other statements to retrieve and manipulate related data. When joining tables, there are a variety of JOIN syntaxes available, but typically developers use the INNER JOIN and OUTER JOIN syntaxes.

An INNER JOIN query returns one row for each pair or matching rows in the tables being joined. Take our Author and Book_Author tables as an example:

```
mysql> SELECT First_Name, Last_Name, ISBN
-> FROM Author INNER JOIN Book_Author ON Author.Author_ID = Book_Author.Author_ID;
+-----------------+---------+-------+
<table>
<thead>
<tr>
<th>First_Name</th>
<th>Last_Name</th>
<th>ISBN</th>
</tr>
</thead>
</table>
The third author in the Author table is missing because there are no corresponding rows in the Book_Author table. When we need at least one row in the result set for every row in a given table, regardless of matching rows, we use an OUTER JOIN query.

There are three variations of the OUTER JOIN syntax: LEFT OUTER JOIN, RIGHT OUTER JOIN and FULL OUTER JOIN. The syntax used determines which table will be fully represented. A LEFT OUTER JOIN returns one row for each row in the table specified on the left side of the LEFT OUTER JOIN clause. The opposite is true for the RIGHT OUTER JOIN clause. A FULL OUTER JOIN returns one row for each row in both tables.

In each case, a row of NULL values is substituted when a matching row is not present. The following is an example of a LEFT OUTER JOIN:

```sql
mysql> SELECT First_Name, Last_Name, ISBN
          -> FROM Author LEFT OUTER JOIN Book_Author ON Author.Author_ID = Book_Author.Author_ID;
```

```plaintext
+------------------------+--------+--------|
| First_Name | Last_Name | ISBN   |
|------------|-----------+--------|
| Chad       | Russell   | 1590593324 |
| Jon        | Stephens  | 1590593324 |
| Mike       | Hillyer   | NULL   |
+------------------------+--------+--------|
```

The third author is returned in this example, with a NULL value for the ISBN column, indicating that there are no matching rows in the Book_Author table.

**Conclusion**

Through the process of database normalization we bring our schema's tables into conformance with progressive normal forms. As a result our tables each represent a single entity (a book, an author, a subject, etc) and we benefit from decreased redundancy, fewer anomalies and improved efficiency.