tomers, you can just assign arbitrary keys, which the `AUTO_INCREMENT` feature that you saw in the last chapter makes easy.

In short, every table will be designed around some object that you're likely to search for a lot—an author, book, or customer, in this case—and that object will have a primary key. Don't choose a key that could possibly have the same value for different objects. The ISBN is a rare case for which an industry has provided a primary key that you can rely on to be unique for each product. Most of the time, you'll create an arbitrary key for this purpose, using `AUTO_INCREMENT`.

Normalization

The process of separating your data into tables and creating primary keys is called *normalization*. Its main goal is to make sure each piece of information appears in the database only once. Duplicating data is inefficient, because it makes databases larger than they need to be and therefore slows access. But, more important, the presence of duplicates creates a strong risk that you'll update only one row of duplicated data, creating inconsistencies in a database and potentially causing serious errors.

Thus, if you list the titles of books in the *Authors* table as well as the *Books* table, and you have to correct a typographic error in a title, you'll have to search through both tables and make sure you make the same change every place the title is listed. It's better to keep the title in one place and use the ISBN in other places.

But in the process of splitting a database into multiple tables, it's important not to go too far and create more tables than is necessary, which would also lead to inefficient design and slower access.

Luckily, E. F. Codd, the inventor of the relational model, analyzed the concept of normalization and split it into three separate schemas called *First*, *Second*, and *Third Normal Form*. If you modify a database to satisfy each of these forms in order, you will ensure that your database is optimally balanced for fast access, and minimum memory and disk space usage.

To see how the normalization process works, let's start with the rather monstrous database in Table 9-1, which shows a single table containing all of the author names, book titles, and (fictional) customer details. You could consider it a first attempt at a table intended to keep track of which customers have ordered books. Obviously, this is inefficient design, because data is duplicated all over the place (duplications are highlighted), but it represents a starting point.

Table 9-1. A highly inefficient design for a database table

Author 1	Author 2	Title	ISBN	Price $US	Customer Name	Customer Address	Purchase Date
David Sklar	*Adam Trachtenberg*	*PHP Cookbook*	*0596101015*	*44.99*	Emma Brown	1565 Rainbow Road, Los Angeles, CA 90014	Mar 03 2009
Danny Goodman		Dynamic HTML	0596527403	59.99	**Darren Ryder**	**4758 Emily Drive, Richmond, VA 23219**	**Dec 19 2008**
Hugh E Williams	David Lane	PHP And MySQL	0596005436	44.95	Earl B. Thurston	862 Gregory Lane, Frankfort, KY 40601	Jun 22 2009
David Sklar	*Adam Trachtenberg*	*PHP Cookbook*	*0596101015*	*44.99*	**Darren Ryder**	**4758 Emily Drive, Richmond, VA 23219**	**Dec 19 2008**
Rasmus Lerdorf	Kevin Tatroe & Peter MacIntyre	Programming PHP	0596006815	39.99	David Miller	3647 Cedar Lane, Waltham, MA 02154	Jan 16 2009

In the following three sections, we will examine this database design, and you'll see how we can improve it by removing the various duplicate entries and splitting the single table into multiple tables, each containing one type of data.

First Normal Form

For a database to satisfy the *First Normal Form*, it must fulfill three requirements:

- There should be no repeating columns containing the same kind of data.
- All columns should contain a single value.
- There should be a primary key to uniquely identify each row.

Looking at these requirements in order, you should notice straightaway that both the *Author 1* and *Author 2* columns constitute repeating data types. So we already have a target column for pulling into a separate table, as the repeated *Author* columns violate Rule 1.

Second, there are three authors listed for the final book, *Programming PHP*. I've handled that by making Kevin Tatroe and Peter MacIntyre share the *Author 2* column, which violates Rule 2—yet another reason to transfer the *Author* details to a separate table.

However, Rule 3 is satisfied, because the primary key of ISBN has already been created.

Table 9-2 shows the result of removing the *Authors* columns from Table 9-1. Already it looks a lot less cluttered, although there remain duplications that are highlighted.

Table 9-2. The result of stripping the Authors columns from Table 9-1

Title	ISBN	Price $US	Customer Name	Customer Address	Purchase Date
PHP Cookbook	*0596101015*	*44.99*	Emma Brown	1565 Rainbow Road, Los Angeles, CA 90014	Mar 03 2009
Dynamic HTML	0596527403	59.99	**Darren Ryder**	**4758 Emily Drive, Richmond, VA 23219**	**Dec 19 2008**
PHP and MySQL	0596005436	44.95	Earl B. Thurston	862 Gregory Lane, Frankfort, KY 40601	Jun 22 2009
PHP Cookbook	*0596101015*	*44.99*	**Darren Ryder**	**4758 Emily Drive, Richmond, VA 23219**	**Dec 19 2008**
Programming PHP	0596006815	39.99	David Miller	3647 Cedar Lane, Waltham, MA 02154	Jan 16 2009

The new *Authors* table shown in Table 9-3 is small and simple. It just lists the ISBN of a title along with an author. If a title has more than one author, additional authors get their own rows. At first, you may feel ill at ease with this table, because you can't tell which author wrote which book. But don't worry: MySQL can quickly tell you. All you have to do is tell it which book you want information for, and MySQL will use its ISBN to search the *Authors* table in a matter of milliseconds.

Table 9-3. The new Authors table

ISBN	Author
0596101015	David Sklar
0596101015	Adam Trachtenberg
0596527403	Danny Goodman
0596005436	Hugh E Williams
0596005436	David Lane
0596006815	Rasmus Lerdorf
0596006815	Kevin Tatroe
0596006815	Peter MacIntyre

As I mentioned earlier, the ISBN will be the primary key for the *Books* table, when we get around to creating that table. I mention that here in order to emphasize that the ISBN is not, however, the primary key for the *Authors* table. In the real world, the *Authors* table would deserve a primary key, too, so that each author would have a key to uniquely identify him or her.

So, in the *Authors* table, the ISBN is just a column for which—for the purposes of speeding up searches—we'll probably make a key, but not the primary key. In fact, it

cannot be the primary key in this table, because it's not unique: the same ISBN appears multiple times whenever two or more authors have collaborated on a book.

Because we'll use it to link authors to books in another table, this column is called a *foreign* key.

Keys (also called *indexes*) have several purposes in MySQL. The fundamental reason for defining a key is to make searches faster. You've seen examples in Chapter 8 in which keys are used in `WHERE` clauses for searching. But a key can also be useful to uniquely identify an item. Thus, a unique key is often used as a primary key in one table, and as a foreign key to link rows in that table to rows in another table.

Second Normal Form

The First Normal Form deals with duplicate data (or redundancy) across multiple columns. The *Second Normal Form* is all about redundancy across multiple rows. To achieve Second Normal Form, your tables must already be in First Normal Form. Once this has been done, we achieve Second Normal Form by identifying columns whose data repeats in different places and then removing them to their own tables.

So let's look again at Table 9-2. Notice how Darren Ryder bought two books and therefore his details are duplicated. This tells us that the *Customer* columns need to be pulled into their own tables. Table 9-4 shows the result of removing the *Customer* columns from Table 9-2.

Table 9-4. The new Titles table

ISBN	Title	Price
0596101015	PHP Cookbook	44.99
0596527403	Dynamic HTML	59.99
0596005436	PHP and MySQL	44.95
0596006815	Programming PHP	39.99

As you can see, all that's left in Table 9-4 are the *ISBN*, *Title*, and *Price* columns for four unique books, so this now constitutes an efficient and self-contained table that satisfies the requirements of both the First and Second Normal Forms. Along the way, we've managed to reduce the information to data closely related to book titles. This table could also include years of publication, page counts, numbers of reprints, and so on, as these details are also closely related. The only rule is that we can't put in any column that could have multiple values for a single book, because then we'd have to list the same book in multiple rows and would thus violate Second Normal Form. Restoring an *Author* column, for instance, would violate this normalization.

However, looking at the extracted *Customer* columns, now in Table 9-5, we can see that there's still more normalization work to do, because Darren Ryder's details are still duplicated. And it could also be argued that First Normal Form Rule 2 (all columns should contain a single value) has not been properly complied with, because the addresses really need to be broken into separate columns for *Address*, *City*, *State*, and *Zip code*.

Table 9-5. The Customer details from Table 9-2

ISBN	Customer Name	Customer Address	Purchase Date
0596101015	Emma Brown	1565 Rainbow Road, Los Angeles, CA 90014	Mar 03 2009
0596527403	Darren Ryder	4758 Emily Drive, Richmond, VA 23219	Dec 19 2008
0596005436	Earl B. Thurston	862 Gregory Lane, Frankfort, KY 40601	Jun 22 2009
0596101015	Darren Ryder	4758 Emily Drive, Richmond, VA 23219	Dec 19 2008
0596006815	David Miller	3647 Cedar Lane, Waltham, MA 02154	Jan 16 2009

What we have to do is split this table further to ensure that each customer's details are entered only once. Because the ISBN is not and cannot be used as a primary key to identify customers (or authors), a new key must be created.

Table 9-6 is the result of normalizing the *Customers* table into both First and Second Normal Forms. Each customer now has a unique customer number called *CustNo* that is the table's primary key, and that will most likely have been created via `AUTO_INCREMENT`. All the parts of customer addresses have also been separated into distinct columns to make them easily searchable and updateable.

Table 9-6. The new Customers table

CustNo	Name	Address	City	State	Zip
1	Emma Brown	1565 Rainbow Road	Los Angeles	CA	90014
2	Darren Ryder	4758 Emily Drive	Richmond	VA	23219
3	Earl B. Thurston	862 Gregory Lane	Frankfort	KY	40601
4	David Miller	3647 Cedar Lane	Waltham	MA	02154

At the same time, in order to normalize Table 9-6, we had to remove the information on customer purchases, because otherwise, there would be multiple instances of customer details for each book purchased. Instead, the purchase data is now placed in a new table called *Purchases* (see Table 9-7).

Table 9-7. The new Purchases table

CustNo	ISBN	Date
1	0596101015	Mar 03 2009
2	0596527403	Dec 19 2008
2	0596101015	Dec 19 2008
3	0596005436	Jun 22 2009
4	0596006815	Jan 16 2009

Here the *CustNo* column from Table 9-6 is reused as a key to tie both the *Customers* and the *Purchases* tables together. Because the ISBN column is also repeated here, this table can be linked with either of the *Authors* or the *Titles* tables, too.

The *CustNo* column can be a useful key in the *Purchases* table, but it's not a primary key. A single customer can buy multiple books (and even multiple copies of one book), so the *CustNo* column is not a primary key. In fact, the *Purchases* table has no primary key. That's all right, because we don't expect to need to keep track of unique purchases. If one customer buys two copies of the same book on the same day, we'll just allow two rows with the same information. For easy searching, we can define both *CustNo* and *ISBN* as keys—just not as primary keys.

There are now four tables, one more than the three we had initially assumed would be needed. We arrived at this decision through the normalization processes, by methodically following the First and Second Normal Form rules, which made it plain that a fourth table called *Purchases* would also be required.

The tables we now have are *Authors* (Table 9-3), *Titles* (Table 9-4), *Customers* (Table 9-6), and *Purchases* (Table 9-7), and we can link each table to any other using either the *CustNo* or the *ISBN* keys.

For example, to see which books Darren Ryder has purchased, you can look him up in Table 9-6, the *Customers* table, where you will see his *CustNo* is 2. Armed with this number, you can now go to Table 9-7, the *Purchases* table; looking at the ISBN column here, you will see that he purchased titles 0596527403 and 0596101015 on December 19, 2008. This looks like a lot of trouble for a human, but it's not so hard for MySQL.

To determine what these titles were, you can then refer to Table 9-4, the *Titles* table, and see that the books he bought were *Dynamic HTML* and *PHP Cookbook*. Should you wish to know the authors of these books, you could also use the ISBNs you just looked up on Table 9-3, the *Authors* table, and you would see that ISBN 0596527403,

Dynamic HTML, was written by Danny Goodman, and that ISBN 0596101015, *PHP Cookbook*, was written by David Sklar and Adam Trachtenberg.

Third Normal Form

Once you have a database that complies with both the First and Second Normal Forms, it is in pretty good shape and you might not have to modify it any further. However, if you wish to be very strict with your database, you can ensure that it adheres to the *Third Normal Form*, which requires that data that is *not* directly dependent on the primary key but *is* dependent on another value in the table should also be moved into separate tables, according to the dependence.

For example, in Table 9-6, the *Customers* table, it could be argued that the *State*, *City*, and *Zip code* keys are not directly related to each customer, because many other people will have the same details in their addresses, too. However, they are directly related to each other, in that the street *Address* relies on the *City*, and the *City* relies on the *State*.

Therefore, to satisfy Third Normal Form for Table 9-6, you would need to split it into Table 9-8 through Table 9-11.

Table 9-8. Third Normal Form Customers table

CustNo	Name	Address	Zip
1	Emma Brown	1565 Rainbow Road	90014
2	Darren Ryder	4758 Emily Drive	23219
3	Earl B. Thurston	862 Gregory Lane	40601
4	David Miller	3647 Cedar Lane	02154

Table 9-9. Third Normal Form Zip codes table

Zip	CityID
90014	1234
23219	5678
40601	4321
02154	8765

Table 9-10. Third Normal Form Cities table

CityID	Name	StateID
1234	Los Angeles	5
5678	Richmond	46
4321	Frankfort	17
8765	Waltham	21

Table 9-11. Third Normal Form States table

StateID	Name	Abbreviation
5	California	CA
46	Virginia	VA
17	Kentucky	KY
21	Massachusetts	MA

So, how would you use this set of four tables instead of the single Table 9-6? Well, you would look up the *Zip code* in Table 9-8, and then find the matching *CityID* in Table 9-9. Given this information, you could look up the city *Name* in Table 9-10 and then also find the *StateID*, which you could use in Table 9-11 to look up the State's *Name*.

Although using the Third Normal Form in this way may seem like overkill, it can have advantages. For example, take a look at Table 9-11, where it has been possible to include both a state's name and its two-letter abbreviation. It could also contain population details and other demographics, if you desired.

Table 9-10 could also contain even more localized demographics that could be useful to you and/or your customers. By splitting up these pieces of data, you can make it easier to maintain your database in the future, should it be necessary to add columns.

Deciding whether to use the Third Normal Form can be tricky. Your evaluation should rest on what data you may need to add at a later date. If you are absolutely certain that the name and address of a customer is all that you will ever require, you probably will want to leave out this final normalization stage.

On the other hand, suppose you are writing a database for a large organization such as the U.S. Postal Service. What would you do if a city were to be renamed? With a table such as Table 9-6, you would need to perform a global search-and-replace on every instance of that city. But if you have your database set up according to the Third Normal Form, you would have to change only a single entry in Table 9-10 for the change to be reflected throughout the entire database.

Therefore, I suggest that you ask yourself two questions to help you decide whether to perform a Third Normal Form normalization on any table:

- Is it likely that many new columns will need to be added to this table?
- Could any of this table's fields require a global update at any point?

If either of the answers is yes, you should probably consider performing this final stage of normalization.

When Not to Use Normalization

Now that you know all about normalization, I'm going to tell you why you should throw these rules out of the window on high-traffic sites. That's right—you should never fully normalize your tables on sites that will cause MySQL to thrash.

Normalization requires spreading data across multiple tables, and this means making multiple calls to MySQL for each query. On a very popular site, if you have normalized tables, your database access will slow down considerably once you get above a few dozen concurrent users, because they will be creating hundreds of database accesses between them. In fact, I would go so far as to say you should denormalize any commonly looked-up data as much as you can.

You see, if you have data duplicated across your tables, you can substantially reduce the number of additional requests that need to be made, because most of the data you want is available in each table. This means that you can simply add an extra column to a query and that field will be available for all matching results.

Of course, you have to deal with the downsides previously mentioned, such as using up large amounts of disk space, and ensuring that you update every single duplicate copy of data when one of them needs modifying.

Multiple updates can be computerized, though. MySQL provides a feature called *triggers* that make automatic changes to the database in response to changes you make. (Triggers are, however, beyond the scope of this book.) Another way to propagate redundant data is to set up a PHP program to run regularly and keep all copies in sync. The program reads changes from a "master" table and updates all the others. (You'll see how to access MySQL from PHP in the next chapter.)

However, until you are very experienced with MySQL, I recommend that you fully normalize all your tables (at least to First and Second Normal Form), as this will instill the habit and put you in good stead. Only when you actually start to see MySQL logjams should you consider looking at denormalization.

Relationships

MySQL is called a *relational* database management system because its tables store not only data but the *relationships* among the data. There are three categories of relationships.

One-to-One

A *one-to-one relationship* is like a (traditional) marriage: each item has a relationship to only one item of the other type. This is surprisingly rare. For instance, an author can write multiple books, a book can have multiple authors, and even an address can be associated with multiple customers. Perhaps the best example in this chapter so far

of a one-to-one relationship is the relationship between the name of a state and its two-character abbreviation.

However, for the sake of argument, let's assume that there can always be only one customer at any address. In such a case, the Customers–Addresses relationship in Figure 9-1 is a one-to-one relationship: only one customer lives at each address, and each address can have only one customer.

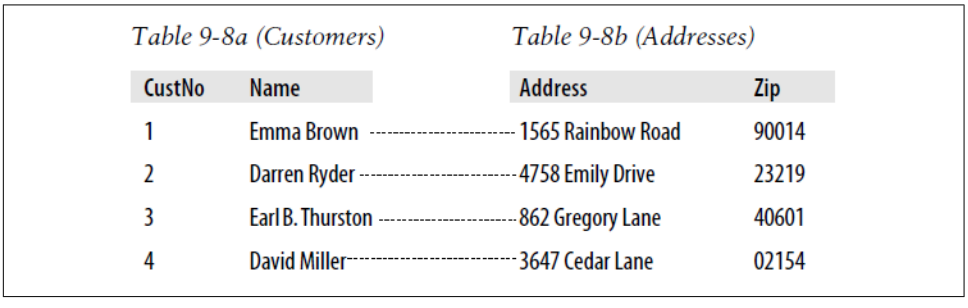

Figure 9-1. The Customers table, Table 9-8, split into two tables

Usually, when two items have a one-to-one relationship, you just include them as columns in the same table. There are two reasons for splitting them into separate tables:

- You want to be prepared in case the relationship changes later.
- The table has a lot of columns, and you think that performance or maintenance would be improved by splitting it.

Of course, when you build your own databases in the real world, you will have to create one-to-many Customer–Address relationships (*one* address, *many* customers).

One-to-Many

One-to-many (or many-to-one) relationships occur when one row in one table is linked to many rows in another table. You have already seen how Table 9-8 would take on a one-to-many relationship if multiple customers were allowed at the same address, which is why it would have to be split up if that were the case.

So, looking at Table 9-8a within Figure 9-1, you can see that it shares a one-to-many relationship with Table 9-7 because there is only one of each customer in Table 9-8a. However Table 9-7, the *Purchases* table, can (and does) contain more than one purchase from customers. Therefore, *one* customer has a relationship with *many* purchases.

You can see these two tables alongside each other in Figure 9-2, where the dashed lines joining rows in each table start from a single row in the lefthand table but can connect to more than one row on the righthand table. This one-to-many relationship

is also the preferred scheme to use when describing a many-to-one relationship, in which case you would normally swap the left and right tables to view them as a one-to-many relationship.

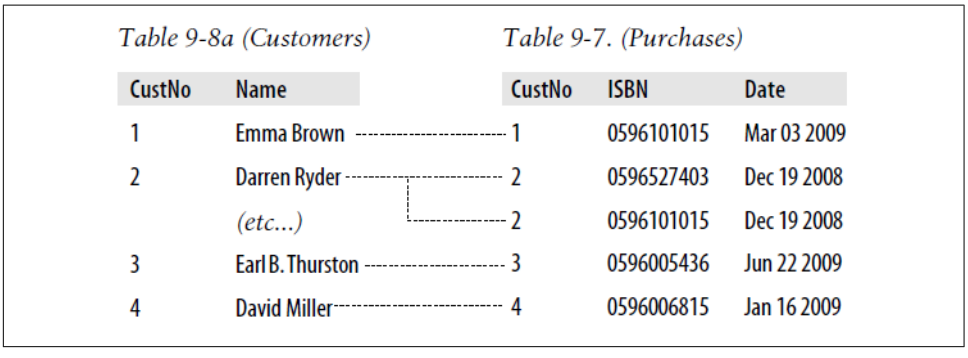

Figure 9-2. Illustrating the relationship between two tables

Many-to-Many

In a *many-to-many relationship*, many rows in one table are linked to many rows in another table. To create this relationship, add a third table containing the same key column from each of the other tables. This third table contains nothing else, as its sole purpose is to link up the other tables.

Table 9-12 is just such a table. It was extracted from Table 9-7, the *Purchases* table, but omits the purchase date information. It contains a copy of the ISBN of every title sold, along with the customer number of each purchaser.

Table 9-12. An intermediary table

Customer	ISBN
1	0596101015
2	0596527403
2	0596101015
3	0596005436
4	0596006815

With this intermediary table in place, you can traverse all the information in the database through a series of relations. You can take an address as a starting point and find out the authors of any books purchased by the customer living at that address.

For example, let's suppose that you want to find out about purchases in the 23219 zip code. Look that zip code up in Table 9-8b, and you'll find that customer number 2 has bought at least one item from the database. At this point, you can use Table 9-8a to

find out his or her name, or use the new intermediary Table 9-12 to see the book(s) purchased.

From here, you will find that two titles were purchased and can follow them back to Table 9-4 to find the titles and prices of these books, or to Table 9-3 to see who the authors were.

If it seems to you that this is really combining multiple one-to-many relationships, then you are absolutely correct. To illustrate, Figure 9-3 brings three tables together.

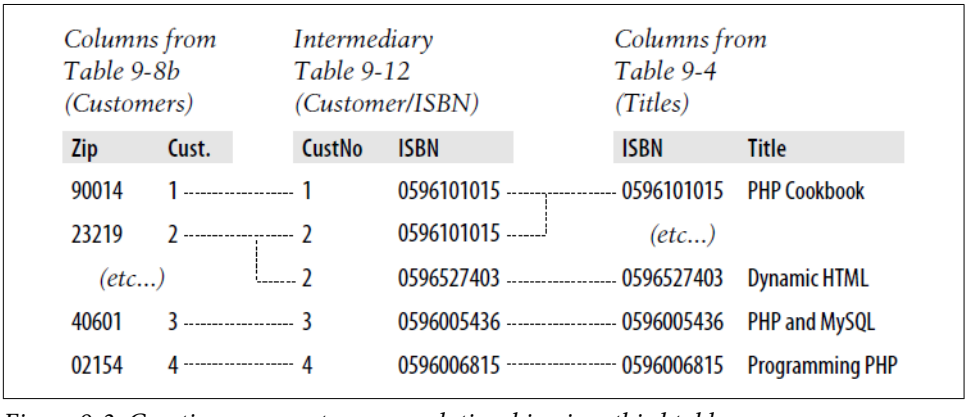

Figure 9-3. Creating a many-to-many relationship via a third table

Follow any zip code in the lefthand table to associated customer IDs. From there, you can link to the middle table, which joins the left and right tables by linking customer IDs and ISBNs. Now all you have to do is follow an ISBN over to the right-hand table to see which book it relates to.

You can also use the intermediary table to work your way backward from book titles to zip codes. The *Titles* table can tell you the ISBN, which you can use in the middle table to find ID numbers of customers who bought the books, and finally, you can use the *Customers* table to match the customer ID numbers to the customers' zip codes.

Databases and Anonymity

An interesting aspect of using relations is that you can accumulate a lot of information about some item—such as a customer—without actually knowing who that customer is. Note that in the previous example we went from customers' zip codes to customers' purchases, and back again, without finding out the name of a customer. Databases can be used to track people, but they can also be used to help preserve people's privacy while still finding useful information.