

Chapter 1

Database Design with MySQL

IN THIS CHAPTER

- ◆ Identifying the problems that led to the creation of the relational database
- ◆ Learning the normalization process
- ◆ Examining advanced database concepts

THE BULK OF THIS CHAPTER is for those of you who have made it to the early twenty-first century without working with relational databases. If you're a seasoned database pro, having worked with Oracle, Sybase, or even something like Microsoft Access or Paradox, you may want to skip this little lesson on database theory. However, we do suggest that you look at the final section of this chapter, where we discuss some of MySQL's weirder points. MySQL's implementation of SQL is incomplete, so it might not support something you want to use.

Why Use a Relational Database?

If you're still here and are ready to read with rapt attention about database theory and the wonders of normalization, you probably don't know much about the history of the relational database. You may not even care. For that reason, I'll keep this very brief. Dr. E. F. Codd was a research scientist at IBM in the 1960s. A mathematician by training, he was unhappy with the available models of data storage, finding them all prone to error and redundancy. He worked on these problems and then, in 1970, published a paper with the rousing title "A Relational Model of Data for Large Shared Data Banks." In all honesty, nothing has been the same since.

A programmer named Larry Ellison read the paper and started work on software that could put Dr. Codd's theories into practice. If you've been a resident of this planet during the past 20 years, you may know that Ellison's product and company took the name Oracle and that he is now one of the richest individuals in the world. His earliest product was designed for huge mainframe systems. Responding to market demands over the years, Oracle, and many other companies that have sprung up since, have designed systems with a variety of features geared toward a variety of

4 Part I: Working with MySQL

operating systems. Now relational databases are so common that you can get one that runs on a Palm Pilot.

To understand why Dr. Codd's theories have revolutionized the data-storage world, it's best to have an idea of what the troubles are with other means of data storage. Take the example of a simple address book – nothing too complex, just something that stores names, addresses, phone numbers, emails, and the like. If you have no persistent, running program to put this information into, the file system of whatever OS you're running becomes the natural choice for storage.

For a simple address book, a delimited text file can be created to store the information. If the first row serves as a header and commas are used as delimiters, the text file might look something like this:

```
Name, Addr1, Addr2, City, State, Zip, Phone, Email
Jay Greenspan, 211 Some St, Apt 2, San Francisco, CA, 94107,
4155551212, jay@not.real
Brad Bulger, 411 Some St, Apt 6, San Francisco, CA, 94109,
4155552222, brad@not.real
John Doe, 444 Madison Ave. , New York, NY, 11234, 2125556666,
nobody@mysqlphpapps.com
```

This isn't much to look at, but it is at least machine-readable. Using whatever language you wish, you can write a script that opens this file and then parses the information. You will probably want it in some sort of two-dimensional or associative array so that you'll have some flexibility in addressing each portion of each line of the file. Any way you look at it, there's going to be a fair amount of code to write. If you want this information to be sortable and queryable by a variety of criteria, you're going to have to write scripts that will, for instance, sort the list alphabetically by name or find all people within a certain area code. What a pain.

You might face another major problem if your data needs to be used across a network by a variety of people. Presumably more than one person is going to need to write information to this file. What happens if two people try to make changes at once? For starters, it's quite possible that one person will overwrite another's changes. To prevent this from happening, the programmer has to specify file locking if the file is in use. While this might work, it's kind of a pain in the neck for the person who gets locked out. Obviously, the larger the system gets the more unmanageable this all becomes.

What you need is something more robust than the file system – a program or daemon that stays in memory seems to be a good choice. Furthermore, you'll need a data-storage system that reduces the amount of parsing and scripting that the programmer needs to be concerned with. No need for anything too arcane here. A plain, simple table like Table 1-1 should work just fine.

Now this is pretty convenient. It's easy to look at and if a running program accesses this table it should happen pretty quickly. What else might this program do? First, it should be able to address one row at a time without affecting the others. That way, if two or more people want to insert information into this table they

won't be tripping over each other. It would be even spiffier if the program provided a simple and elegant way to extract information from a table such as this. There should be a quick way to find all of the people from California that doesn't involve parsing and sorting the file. Furthermore, this wondrous program should be able to accept statements that describe what you want in a language very similar to English. That way you can just say: "Give me all rows where the contents of the state column equal CA."

Yes, this program is great, but it isn't enough. Major problems still need to be dealt with. These problems, which we'll discuss in the following pages, are the same ones that made Dr. Codd write his famous paper, and the same ones that made Larry Ellison a billionaire.

Blasted Anomalies

Dr. Codd's goal was to have a model of information that was dependable. All of the data-storage methods available to him had inherent problems. He referred to these problems as *anomalies*. There are three types of anomalies: update, delete, and insert.

The update anomaly

Now that you can assume that a table structure can quickly and easily handle multiple requests, you need to see what happens when the information gets more complex. Adding some more information to the previous table introduces some serious problems (Table 1-2).

Table 1-2 is meant to store information for an entire office, not just a single person. Since this company deals with other large companies, there will be times when more than one contact will be at a single office location. For example, in Table 1-2 two contacts are present at 1121 43rd St. At first this may appear to be okay; you can still get at all the information available relatively easily. The problem comes when the BigCo Company decides to up and move to another address. In that case, you'd have to update the address for BigCo in two different rows. This may not sound like such an onerous task, but consider the trouble if this table has 3,000 rows instead of 3 – or 300,000 for that matter. Someone, or some program, has to make sure the data are changed in every appropriate place.

Another concern is the potential for error. It's very possible that one of these rows could be altered while the other one remained the same. Or, if changes are keyed in one row at a time, it's likely that somebody will introduce a typo. Then you'd be left wondering if the correct address is 1121 or 1211.

The better way to handle this data is to take the company name and address and put that information in its own table. This process of separating a table out into multiple new tables is usually called *decomposition*. The two resulting tables will resemble Table 1-3 and Table 1-4.

Now the information pertinent to BigCo is in its own table, Companies. If you look at the next table (Table 1-4), Contacts, you'll see that we've inserted another

6 Part I: Working with MySQL

TABLE 1-1 SIMPLE TABLE FOR DATA STORAGE

name	addr1	addr2	city	state	zip	phone	email
Jay Greenspan	211 Some St.	Apt. 2	San Francisco	CA	94107	4155558888	jay@not.real
Brad Bulger	411 Some St.	Apt. 6	San Francisco	CA	94109	4155552222	brad@not.real
John Doe	444 Madison Ave.		New York	NY	11234	2125556666	nobody@mysql.phpapps.com

TABLE 1-2 PROBLEMATIC TABLE STORAGE

id	company_name	company_address	contact_name	contact_title	phone	email
1	BigCo Company	1121 43rd St.	Jay Greenspan	Vice President	4155551212	jay@not.real
2	BigCo Company	1121 43rd St.	Brad Bulger	President	4155552222	brad@not.real
3	LittleCo Company	4444 44th St.	John Doe	Lackey	2125556666	nobody@hotmail.com

TABLE 1-3 COMPANIES

company_id	company_name	company_address
1	BigCo Company	1121 43rd St.
2	LittleCo Company	4444 44th St.

TABLE 1-4 CONTACTS

contact_id	company_id	contact_name	contact_title	phone	email
1	1	Jay Greenspan	Vice President	415551212	jay@not.real
2	1	Brad Bulger	President	415552222	brad@not.real
3	2	John Doe	Lackey	2125556666	nobody@mysqlphpapps.com

8 Part I: Working with MySQL

column, `company_id`. This column references the `company_id` column of the Companies table. In Brad's row, you see that the `company_id` (the second column) equals 1. You can then go to the Companies table, look at the information for `company_id` 1, and see all the relevant address information. What's happened here is that you've created a relationship between these two tables—hence the name *relational database*.

You still have all the information you had in the previous setup, you've just segmented it. In this setup you can change the address for both Jay and Brad by altering only a single row. That's the kind of convenience you want to be after.

Perhaps this leaves you wondering how you get this information un-segmented. Relational databases give you the ability to merge, or join, tables. Consider the following statement, which is intended to give all the available information for Brad: "Give me all the columns from the contacts table where `contact_id` is equal to 1, and while you're at it throw in all the columns from the Companies table where the `company_id` field equals the value shown in Brad's `company_id` column."

In other words, in this statement, you are asking to join these two tables where the `company_id` fields are the same. The result of this request, or query, looks something like Table 1-5.

In the course of a couple of pages, you've learned how to solve a data-integrity problem by segmenting information and creating additional tables. But we have yet to give this problem a name.

When we learned the vocabulary associated with relational databases from a very thick and expensive book, this sort of problem was called an *update anomaly*. There may or may not be people using this term in the real world; if there are, we haven't met them (people in the real world call it "breach of contract" when addressing their consultants). However, we think this term is pretty apt. In Tables 1-1 and 1-2, if you were to update one row in the table, other rows containing the same information would not be affected.

The delete anomaly

Now take a look at Table 1-6, focusing on row 3.

Consider what happens if Mr. Doe is deleted from the database. This may seem like a simple change but suppose someone accessing the database wants a list of all the companies contacted over the previous year. In the current setup, when you remove row 3, you take out not only the information about John Doe, you remove information about the company as well. This problem is called a *delete anomaly*.

If the company information is moved to its own table, as you saw in the previous section, this delete anomaly won't be a problem. You can remove Mr. Doe and then decide independently if you want to remove the company he's associated with.

The insert anomaly

Our final area of concern is problems that will be introduced during an insert. Looking again at the Table 1-6, you can see that the purpose of this table is to store information on contacts, not companies. This becomes a drag if you want to add a

TABLE 1-5 QUERY RESULTS

company_id	company_name	company_address	contact_id	contact_name	contact_title	contact_phone	contact_email
1	BigCo Company	1121 43rd St.	2	Brad Bulger	President	4155552222	brad@not.real

TABLE 1-6 TABLE WITH DELETE ANOMALY

company_id	company_name	company_address	contact_name	contact_title	contact_phone	contact_email
1	BigCo Company	1121 43rd St	Jay Greenspan	Vice President	4155551212	jay@not.real
2	BigCo Company	1121 43rd St	Brad Bulger	President	4155552222	brad@not.real
3	LittleCo Company	4444 44th St	John Doe	Lackey	2125556666	nobody@mysqlphpapps.com

company but not an individual. For the most part, you'll have to wait to have a specific contact to add to the database before you can add company information. This is a ridiculous restriction. The solution is to store contact information in one table and company information in another. By storing company information in its own table, you can add a new company there even if you (as yet) have no contacts to go with it. Ditto for contacts with no matching companies.

Normalization

Now that we've shown you some of the problems you might encounter, you need to learn the ways to find and eliminate these anomalies. This process is known as *normalization*. Understanding normalization is vital to working with relational databases. But to anyone who has database experience normalization is not the be-all and end-all of data design. Experience and instinct also play a part in the creation of a good database. The examples in this book will usually be normalized. However, in some cases, a denormalized structure is preferable, for performance reasons, code simplification, or so on.

One other quick caveat. The normalization process consists of several *normal forms*. Normal forms are standards of database regulation that promote efficiency, predictability of results, and unambiguity.

In this chapter we cover first, second, and third normal forms. In addition to these, the normalization process can involve four other (progressively more rigorous) normal forms. (For the curious, these are called Boyce-Codd normal form, fourth normal form, fifth normal form, and Domain/Key normal form.) We know about these because we read about them in a book. In the real world, where real people actually develop database applications, these normal forms aren't discussed. If you get your data into third normal form that's about good enough – mainly because data in the third normal form meets the requirements of the first and second normal forms, by definition. Yes, a possibility exists that anomalies will exist in third normal form, but if you get this far you should be OK.

First normal form

Getting data into first normal form is fairly easy. Data need to be in a table structure and to meet the following criteria:

- ◆ Each column must have a unique name and define a single attribute of the table as a whole.
- ◆ Each row in the table must have a set of values that uniquely identifies the row (this is known as the *primary key* of the table).
- ◆ No two rows can be identical.

- ◆ Each cell must contain an *atomic* value, meaning that each cell contains only one value. No arrays or any other manner of representing more than one value can exist in any cell.
- ◆ No repeating groups of data are allowed.

The final item here is the only one that may require some explanation. Take a look at Table 1-7.

As you've already seen with these data, row 1 and row 2 contain two columns that contain identical information. This is a repeating group of data. Only when you remove these columns and place them in their own table will these data be in first normal form. The separation of tables that we did in Tables 1-3 and 1-4 will move this data into first normal form.

Before we move on to chat about second and third normal form, you're going to need a couple of quick definitions. The first is of the term *primary key*. The primary key is a column or set of columns by which each row can be uniquely identified.

Primary keys, while very important, are difficult to understand both in theory and in practice. The theory is straightforward: Each row in the column designated as the primary key must have a unique value. In practice, the easiest way to get a series of unique numbers is to use a series of sequential numbers, in which the value of the primary key column in each row increments the previous row's primary key value by one. Because this is such a popular solution to the primary key problem, all database servers of any consequence create the incremental values for you as records are created. MySQL has such a mechanism; you use it by designating your primary key column as type `auto_increment`.

Depending on your data, all kinds of values will work for a primary key. Social Security numbers work great, as do email addresses and URLs. The data just need to be unique. In some cases, two or more columns may comprise your primary key. For instance, to continue with the address-book example, if contact information needs to be stored for a company with many locations, it is probably best to store the switchboard number and mailing address information in a table that has the `company_id` and `company_location` as its primary key.

Next, we need to define the word *dependency*, which means pretty much what you think it means. A dependent column is one that is inexorably tied to the primary key. It can't exist in the table if the primary key is removed.

With that under your belt, you are ready to tackle second normal form.

Second normal form

This part of the process only comes into play when you end up with one of those multi-column primary keys that we just discussed. Assume that in the course of dividing up your address tables you end up with Table 1-8. Here, the `company_name` and `company_location` columns comprise the multi-column primary key.

12 Part I: Working with MySQL

TABLE 1-7 TABLE WITH REPEATING GROUPS OF DATA

company_id	company_name	company_address	contact_name	contact_title	phone	email
1	BigCo Company	1121 43rd St.	Jay Greenspan	Vice President	4155551212	jay@not.real
2	BigCo Company	1121 43rd St.	Brad Bulger	President	4155552222	brad@not.real
3	LittleCo Company	4444 44th St.	John Doe	Lackey	2125556666	nobody@hotmail.com

TABLE 1-8 TABLE NOT IN SECOND NORMAL FORM

company_name	company_location	company_ceo	company_address
BigCo Company	San Francisco	Bill Hurt	1121 43rd St.
LittleCo Company	Los Angeles	Bob Ouch	4444 44th St.

You should be able to see pretty quickly that an insertion anomaly would work its way in here if you were to add another location for BigCo Company. You'd have the CEO name, Bill Hurt, repeated in an additional row, and that's no good.

You can get this table into second normal form by removing rows that are only partially dependent on the primary key. Here, the CEO is dependent only on the `company_name` column. It is not dependent on the `company_location` column. To get into second normal form, you move rows that are only partially dependent on a multi-field primary key into their own table (see Tables 1-9 and 1-10). Second normal form does not apply to tables that have a single-column primary key.

TABLE 1-9 TABLE IN SECOND NORMAL FORM

company_id	company_name	company_ceo
1	BigCo Company	Bill Hurt
2	LittleCo Company	Bob Ouch

TABLE 1-10 TABLE IN SECOND NORMAL FORM

company_id	company_location	company_address
1	San Francisco	1121 43rd St.
2	Los Angeles	4444 44th St.

Third normal form

Finishing up the normalization process, third normal form is concerned with *transitive dependencies*. A transitive dependency describes a situation in which a column exists that is not directly reliant on the primary key. Instead, the field is reliant

14 Part I: Working with MySQL

on some other field, which in turn is dependent on the primary key. A quick way to get into third normal form is to look at all the fields in a table and ask if they all describe the primary key. If they don't, you're not there.

If your address book needs to store more information on your contacts, you might find yourself with a table like Table 1-11.

TABLE 1-11 TABLE NOT IN THIRD NORMAL FORM

contact_id	contact_name	contact_phone	assistant_name	assistant_phone
1	Bill Jones	4155555555	John Bills	2025554444
2	Carol Shaw	2015556666	Shawn Carlo	6505556666

You might think we're doing OK here. But look at the `assistant_phone` column and ask if that really describes the primary key (and the focus of this table), which is your contact. It's possible, even likely, that one assistant will serve many people, in which case it's possible that an assistant name and phone will end up listed in the table more than once. That would be a repeating group of data, which you already know you don't want. Tables 1-12 and 1-13 are in third normal form.

TABLE 1-12 TABLE IN THIRD NORMAL FORM

assistant_id	assistant_name	assistant_phone
1	John Bills	2025554444
2	Shawn Carlo	6505556666

TABLE 1-13 TABLE IN THIRD NORMAL FORM

contact_id	contact_name	contact_phone	assistant_id
1	Bill Jones	4155555555	1
2	Carol Shaw	2015556666	2

Types of Relationships

In the applications you'll see later in this book we create a bunch of tables that don't have anomalies. We include columns that maintain relationships among these tables. You'll encounter three specific types of relationships in database land.

The one-to-many relationship

This is by far the most common type of relationship that occurs between two tables. When one value in a column references multiple fields in another table, a one-to-many relationship is in effect (Figure 1-1).

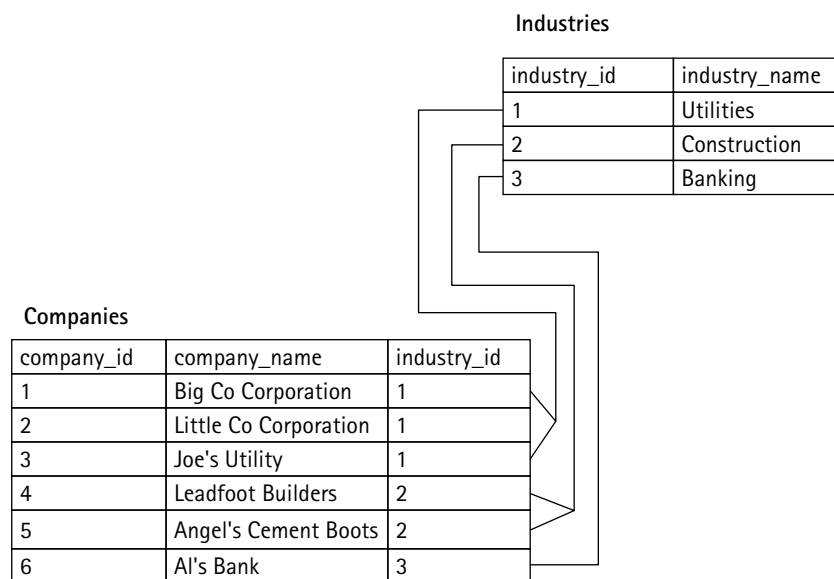


Figure 1-1: Tables with a one-to-many relationship

Figure 1-1 shows a classic one-to-many relationship. Here, each company is associated with a certain industry. As you can see, one industry listed in the industry table can be associated with one or more rows in the company table. This in no way restricts what you can do with the companies. You are absolutely free to use this table as the basis for other one-to-many relationships. Figure 1-2 shows that the Companies table can be on the “one” side of a one-to-many relationship with a table that lists city locations for all the different companies.

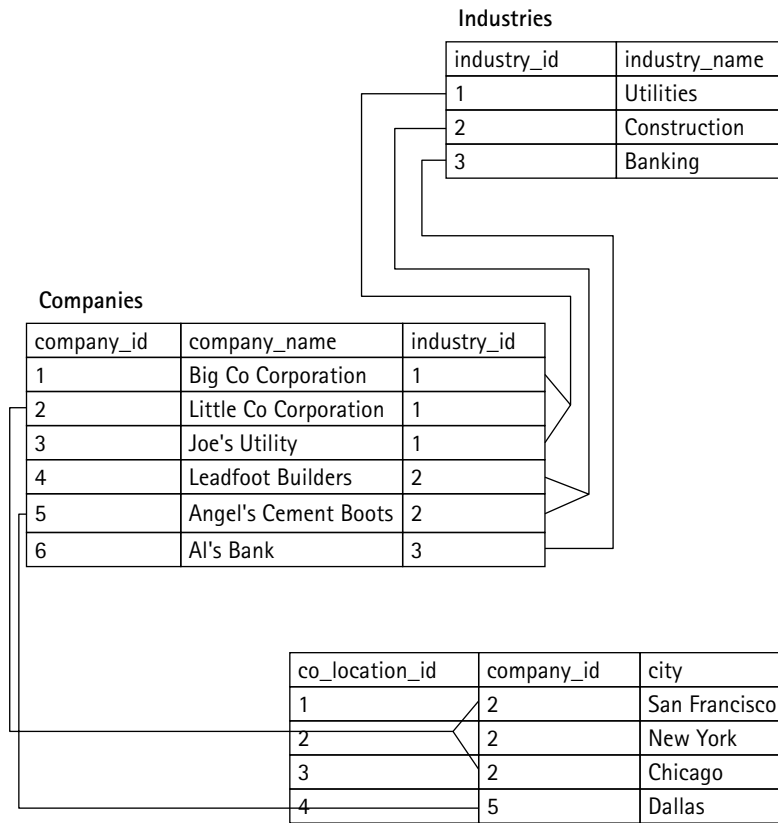


Figure 1-2: Tables with two one-to-many relationships

The one-to-one relationship

A one-to-one relationship is essentially a one-to-many relationship where only one row in a table is related to only one row in another table. During the normalization process, we mentioned a situation in which one table holds information about corporate executives and another holds information about their assistants. This could very well be a one-to-one relationship if each executive has one assistant and each assistant works for only one executive. Figure 1-3 gives a visual representation of this relationship.

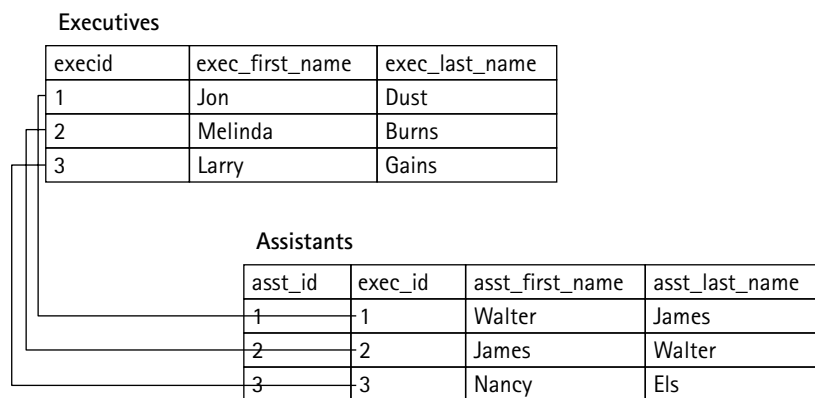


Figure 1-3: Tables with a one-to-one relationship

The many-to-many relationship

Many-to-many relationships work a bit differently from the other two kinds of relationships. For instance, suppose that the company keeping the data has a variety of newsletters that it sends to its contacts, and suppose that it needs to add this information to the database. There's a weekly, a monthly, a bi-monthly, and an annual newsletter, and to keep from annoying clients, the newsletters must only be sent to those who request them.

To start, you could add a table that stores the newsletter types (Table 1-14).

TABLE 1-14 NEWSLETTERS TABLE

newsletter_id	newsletter_name
1	Weekly
2	Monthly
3	Bi-monthly
4	Annual

Table 1-14 can't be directly related to another table that stores contact information. So it's not sufficient to define which clients have requested which types of newsletters. The only way to make that work is to add a column to the Contacts table that stores the newsletters that each contact receives. Right away, you should notice a problem with Table 1-15. In Table 1-15 the Newsletters column contains more

18 Part I: Working with MySQL

than one value. The value looks a lot like an array. As mentioned earlier, this should never occur within a database—you want only atomic values in each column.

TABLE 1-15 REVISED CONTACTS TABLE

contact_id	contact_first_name	contact_last_name	Newsletters
1	Jon	Doe	1,3,4
2	Al	Banks	2,3,4

In situations like this you'll need to create another table, of a type often known as a *mapping table* because it maps the relationship of one table to another. Figure 1-4 shows how the relationship between these values can be made to work.

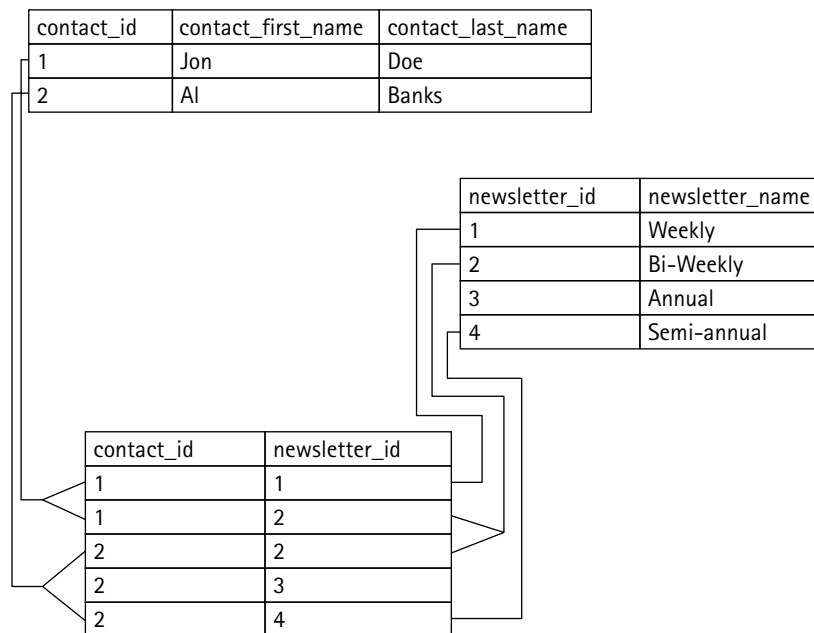


Figure 1-4: Tables with a many-to-many relationship

With this structure, any number of contacts can have any number of newsletters and any number of newsletters can be sent to any number of contacts.



Newcomers to databases often overlook many-to-many relationships and instead choose designs that require excessive columns within a table or arrays within a column. Make sure to consider a many-to-many relationship if your structure seems unmanageable.

Advanced Database Concepts

For a long time MySQL was a polarizing piece of software in the applications-development community. It had (and still has) aspects that many developers loved: it's free (at least, when used in applications that conform to the GNU Public License), it doesn't take up a whole lot of resources, it's very quick, and it's easy to learn compared to packages like Oracle and Sybase. However, it didn't originally offer features common in other databases, such as subselects or joins in updates, and these shortcomings kept many from adopting MySQL for their applications. But since the publication of the first edition of this book a lot of work has been done on MySQL, and it now offers at least partial support for the features discussed in the following sections.

Referential integrity

Every example used so far in this chapter has made use of *foreign keys*. A foreign key is a column that references the primary key of another table in order to maintain a relationship. In Table 1-4, the Contacts table contains a `company_id` column, which references the primary key of the Companies table (Table 1-3). This column is a foreign key to the Companies table.

In Chapter 2 we demonstrate how to create tables in MySQL. It's easy enough to create tables with all the columns necessary for primary keys and foreign keys. However, in MySQL foreign keys are not universally available.

In packages like Oracle, Sybase, or PostgreSQL, tables can be created that explicitly define foreign keys. For instance, with Oracle the database system could be made aware that the `company_id` column in the Contacts table has a relationship to the `company_id` column in the Companies table. This capability is potentially a very good thing and is known as a *foreign-key constraint*. If the database system is aware of a relationship, it can check to make sure the value being inserted into the foreign-key field exists in the referenced table. If it does not, the database system will reject the insert. The capability of the database server to reject records because they don't satisfy the requirements of linked tables is known as *referential integrity*.

With MySQL, at the time of this writing, foreign-key constraints are only available when you're using the InnoDB table type. You'll see how to work with foreign-key constraints in InnoDB in Chapter 2.

20 Part I: Working with MySQL

To demonstrate the importance of foreign-key constraints we'll show you how you'd achieve the same effect using MySQL table types other than InnoDB. Before inserting or updating records in your table, you have to take some extra steps.

To be ultra-safe, you would need to go through the following steps in order to insert a row in the Contacts table (Table 1-4), for example:

1. Get all the values for `company_id` in the Companies table.
2. Check to make sure the value for `company_id` to be inserted into the Contacts table exists in the data you retrieved in Step 1.
3. If it does, insert values.

The developers of MySQL had long argued that referential integrity was not necessary and that including it would slow down MySQL. Further, they argued that it is the responsibility of the application interacting with the database to ensure that the inserted data is correct. There is a logic to this way of thinking. In Parts III and IV of this book we present several applications that would work just fine without enforcing referential integrity or the method of checking shown above. In general, in these applications, all the possible values are pulled from a database anyway and there's very little opportunity for errors to creep into the system.

But there's no doubt that having the option of enforcing referential integrity is a good thing.

Transactions

In relational databases, things change in groups. As shown in a variety of applications in this book, many changes require that rows be updated in several tables concurrently. An e-commerce site may contain code that works in the following manner:

1. Insert a customer into the Customers table.
2. Check the inventory table to see that a sufficient quantity of the item exists to place the order.
3. Add invoice information into the Invoice table.
4. Reduce the quantity available for the item in the inventory table by the quantity ordered.

When you're working with a series of steps like this, serious problems can occur. If the operating system crashes or power goes out between steps three and four, the database will contain bad data. It's also important to remember that MySQL and other relational databases are multi-threaded, which means that they can process directives from multiple clients simultaneously. Imagine what would happen with the previous listing if two orders were placed almost simultaneously for an item that was nearly out of stock. Two threads (in the case of an e-commerce site, two

customers working through their browsers) could find themselves requesting the final item at the same time. If precautions are not taken, it's possible that one person might receive confirmation that the order is available when in fact it is not.

To prevent such occurrences, most sophisticated database systems make use of *transactions*. A transaction is a bundle of commands treated as an indivisible unit. If any one of these commands fails to go through, the whole group of commands fails, and the database returns to the state it was in before the first command was attempted. This is known as a commit/rollback approach. Either all the requests are committed to the database, or the database is rolled back to the state it was in prior to the transactions. This works both to prevent threads from stepping on each other and to protect data in the event of a crash.

With the example given above, if in Step 2 the application were to discover that no items are left, a `ROLLBACK` command will be given and no items will be removed from the inventory. In the case of a crash, the in-progress transactions will be automatically rolled back.

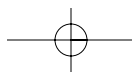
A transaction-capable database must support the four properties that go by the acronym ACID, which are defined as follows:

- ◆ **Atomicity** – The operations that make up each transaction are treated collectively as a single, or atomic, unit. Either all changes are committed or none are.
- ◆ **Consistency** – The available data will never be in an inconsistent state; either other threads will see the data in the state it was in prior to the transaction, or other threads will see the data in the state it winds up in after the transaction is completed.
- ◆ **Isolation** – Each transaction is isolated from all others. The effects of Transaction A are not visible to Transaction B until Transaction A is completed. If a transaction is in progress, the interim state of the data will not be visible to other transactions.
- ◆ **Durability** – When a transaction is complete, the changes are permanent. Even if a database crashes, the information from a committed transaction will be available and complete.

In older versions of MySQL transactions were not supported. This was a major problem for many developers, who could not fathom the idea of designing proper applications without this feature. Now MySQL features several table types (including InnoDB and BerkeleyDB) that support transactions. You read more about these tables in Chapter 2.

Stored procedures

The big fancy database systems allow for procedural code (real computer code, similar to PHP or Perl) to be placed within the database. Using stored procedures provides a couple of key advantages. First, it can reduce the amount of code needed in



22 Part I: Working with MySQL

middleware applications. If MySQL accepted stored procedures (which it unfortunately does not—yet), a single PHP command could be sent to the database to query data, do some string manipulation, and then return a value ready to be displayed in your page.

The other major advantage comes when you are working in an environment in which more than one front-end is accessing the same database. Consider a situation in which there happens to be one front-end written for the Web and another, accessible on Windows machines, written in Visual C++. It would be a pain to write all the queries and transactions in two different places. You'd be much better off writing stored procedures and accessing those from your various applications. Stored procedures are planned for MySQL version 5.0.

Summary

At this point you should have a pretty good idea of how relational databases work. The theory covered here is really important, as quality data design is one of the cornerstones of quality applications. If you fail in the normalization process, you could create difficulties that will haunt you for months or years.

In the applications in Parts III and IV of this book, you see how we approach and normalize several sets of data.

Now that you know how tables in a relational database work, move on to Chapter 2, where you see how to make these tables in MySQL.

