

# **Beginning Databases with PostgreSQL**

From Novice to Professional, Second Edition

NEIL MATTHEW AND RICHARD STONES

Apress®

## **Beginning Databases with PostgreSQL: From Novice to Professional, Second Edition**

**Copyright © 2005 by Neil Matthew and Richard Stones**

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-478-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Jason Gilmore

Contributing Author: Jon Parise

Technical Reviewer: Robert Treat

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Assistant Publisher: Grace Wong

Project Manager: Sofia Marchant

Copy Manager: Nicole LeClerc

Copy Editor: Marilyn Smith

Production Manager: Kari Brooks-Copony

Production Editor: Katie Stence

Compositor: Susan Glinert

Proofreader: Elizabeth Berry

Indexer: John Collin

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail [orders@springer-ny.com](mailto:orders@springer-ny.com), or visit <http://www.springer-ny.com>. Outside the United States: fax +49 6221 345229, e-mail [orders@springer.de](mailto:orders@springer.de), or visit <http://www.springer.de>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail [info@apress.com](mailto:info@apress.com), or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section.



# Relational Database Principles

In this chapter, we will examine what makes a database system, particularly a relational one like PostgreSQL, so useful for real-world data. We will start by looking at spreadsheets, which have much in common with relational databases but also have significant limitations. We will learn how a relational database, such as PostgreSQL, has many advantages over spreadsheets. Along the way, we will continue our rather informal look at SQL.

In particular, this chapter will cover the following topics:

- Spreadsheets: their problems and limitations
- How databases store data
- How to access data in a database
- Basic database design, with multiple tables
- Relationships between tables
- Some basic data types
- The NULL token, used to indicate an unknown value

## Limitations of Spreadsheets

Spreadsheet applications, such as Microsoft Excel, are widely used as a way of storing and inspecting data. It's easy to sort the data in different ways, and see the features and patterns in the data just by looking at it.

Unfortunately, people often mistake a tool that is good for inspecting and manipulating data for a tool suitable for storing and sharing complex and perhaps business-critical data. The two needs are often very different.

Most people will be familiar with one or more spreadsheets and quite at home with data being arranged in a set of rows and columns. Figure 2-1 shows a typical example—an OpenOffice (<http://www.openoffice.org/>) spreadsheet holding data about customers.

customers.csv - OpenOffice.org 1.1.1

File Edit View Insert Format Tools Data Window Help

ch02/customers.csv

Arial 10 B I U

D4 The Barn

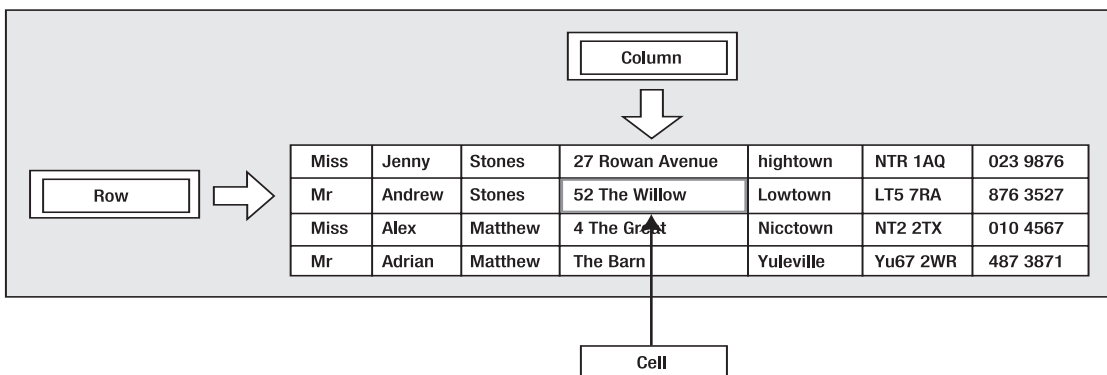
	A	B	C	D	E	F	G	H
1	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876	
2	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527	
3	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567	
4	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871	
5	Mr	Simon	Cozens	7 Shady Lane	Oakenham	OA3 6QW	514 5926	
6	Mr	Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232	
7	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982	
8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982	
9	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432	
10	Mr	Mike	Howard	86 Dysart Street	Tibbsville	TB3 7FG	505 5482	
11	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264	
12	Mr	Richard	Neill	42 Thatched Way	Winnersby	WB3 6GQ	505 6482	
13	Mrs	Laura	Hardy	73 Margarita Way	Oxbridge	OX2 3HX	821 2335	
14	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234	
15	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526	
16								
17								
18								
19								

Sheet1

Sheet 1 / 1 Default 100% STD Sum=0

**Figure 2-1.** A simple spreadsheet

Certainly, such information is easy to see and modify. Each customer has a separate *row*, and each piece of information about the customer is held in a separate *column*, as labeled in Figure 2-2. The intersection of a column and a row is a *cell*.



**Figure 2-2.** Some spreadsheet terminology

This simple spreadsheet incorporates several features that will be handy to remember when we start designing databases. For example, the first and last names are held in separate columns, which makes it easy to sort the data by last name if required.

So what is wrong with storing customer information in a spreadsheet? Spreadsheets are fine, as long as you:

- Don't have too many customers
- Don't have many complex details for each customer
- Don't need to store any other repeating information, such as the various orders each customer has placed
- Don't want several people to be able to update the information simultaneously
- Do ensure the spreadsheet gets backed up regularly if it holds important data

Spreadsheets are a fantastic idea, and they are great tools for many types of problems. However, just as you wouldn't (or at least shouldn't) try to hammer in a nail with a screwdriver, sometimes spreadsheets are not the right tool for the job.

Just imagine what it would be like if a large company, with tens of thousands of customers, kept the master copy of its customer list in a simple spreadsheet. In a big company, it's likely that several people would need to update the list. Although file locking can ensure that only one person updates the list at any one time, as the number of people trying to update the list grows, they will spend longer and longer waiting for their turn to edit the list. What we would like is to allow many people to simultaneously read, update, add, and delete rows, and let the computer ensure there are no conflicts. Clearly, simple file locking will not be adequate to efficiently handle this problem.

Another problem with spreadsheets is their strict two dimensions. Suppose we also wanted to store details of each order a customer placed. We could start putting order information next to each customer, but as the number of orders per customer grew, the spreadsheet would get more and more complex. Consider the outcome when we start trying to add some basic order information for each customer, as shown in Figure 2-3.

Unfortunately, it's not looking quite so elegant anymore. We now have rows of arbitrary length, which does not give us an easy way to calculate how much each customer has spent with us. Eventually, we will exceed the number of columns allowed in each row. It's the repeating groups problem we saw in the previous chapter. Multiple sheets inside a spreadsheet can help, but they are not an ideal solution to the problem.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876		22 Jun 2004	\$15.30	25 Jul 2004	\$27.89	4 Oct 2
2	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527						
3	Miss	Alex	Matthew	4 The Street	Nicotown	NT2 2TX	010 4567		2 Jun 2004	\$32.67	11 Jul 2004	\$23.65	18 Nov 2
4	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871		18 Jun 2004	\$56.32	4 Aug 2004	\$73.11	
5	Mr	Simon	Cozens	7 Shady Lane	Oakenham	OA3 6QW	514 5926						
6	Mr	Neil	Matthew	5 Pasture Lane	Nicotown	NT3 7RT	267 1232						
7	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982		27 Jun 2004	\$32.34			
8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982						
9	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432		12 Jun 2004	\$17.43	18 Jul 2004	\$32.54	
10	Mr	Mike	Howard	86 Dysart Street	Tibbsville	TB3 7FG	505 5482		12 Sep 2004	\$76.23			
11	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264						
12	Mr	Richard	Neill	42 Thatched Way	Winnersby	WB3 6GQ	505 6482						
13	Mrs	Laura	Hardy	73 Margarita Way	Oxbridge	OX2 3HX	821 2335						
14	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234						
15	Mr	David	Hudson	4 The Square	Miltown	MT2 6RT	961 4526		4 Nov 2004	\$12.45			
16													
17													
18													

Figure 2-3. Spreadsheet with repeating order information

### A SPREADSHEET CHALLENGE

Here is an example of how easily you can exceed the capabilities of a spreadsheet. An acquaintance was trying to set up a spreadsheet as a favor for friends who run a small business. This small business makes leather items, and the price of the item depended not only on the time and effort required to make the item, but also on the unit cost of the leather used in the manufacture. The owners would buy leather in batches of different types, each of which would have a unit price that varied significantly depending on both the grade and the timing of the purchase. Then they would use their stock on a first in, first used basis as they made items for sale, normally many per batch of leather purchased. The challenge was to create a spreadsheet to do the following:

- Track the overall current stock value.
- Track how many batches of leather are in stock of each grade.
- Track how much had been paid for the batch and grade currently being used on a particular item being made.

After days of effort, they discovered that this apparently straightforward stockkeeping requirement is a surprisingly difficult problem to transfer to a spreadsheet. The variable nature of the number of stock records does not fit well with the spreadsheet philosophy.

The point we are making here is that spreadsheets are great in their place, but there are limits to their usefulness.

## Storing Data in a Database

When you look at it superficially, a relational database, such as PostgreSQL, has many similarities to a spreadsheet. However, when you know about a database's underlying structure, you can see that it is much more flexible, principally because of its ability to relate tables together in complex ways. It can efficiently store much more complex data than a spreadsheet, and it also has many other features that make it a better choice as a data store. For example, a database can manage multiple simultaneous users.

Let's first look at storing our simple, single-sheet customer list in a database, to see what benefits this might have. Later in the chapter, we will extend this and see how PostgreSQL can help us solve our customer orders problem.

As we saw in the previous chapter, databases are made up of *tables*, or in more formal terminology, *relations*. We will stick to using the term *tables* in this book. A table contains *rows* of data (more formally called *tuples*), and each data row consists of a number of *columns*, or *attributes*.

First, we need to design a table to hold our customer information. The good news is that a spreadsheet of data is often an almost ready-made solution, since it holds the data in a number of rows and columns. To get started with a basic database table, we need to decide on three things:

- How many columns do we need to store the attributes associated with each item?
- What type of data goes in each attribute (column)?
- How can we distinguish different rows containing different items?

Note that the order of rows doesn't matter in a database table. In a spreadsheet, the order of the rows is normally very important, but in a database table, there is no order. That's because when you ask to look at the data in a database table, the database is free to give you the rows of data in any order it chooses, unless you specifically ask for it ordered in a particular way. If you need to see the data in a particular order, you achieve this by the way it is *retrieved* from the database, rather than how it is stored. We will see how to retrieve ordered data in Chapter 4, when we look at the `ORDER BY` clause of the `SELECT` statement.

### Choosing Columns

If you look back at our original spreadsheet for our customer information in Figure 2-1, you can see that we have already decided on what seems a sensible set of columns for each customer: first name, last name, ZIP code, and so on. So, we've already answered the question of how many columns we should have.

An important difference between spreadsheet rows and database rows is that the number of columns in a database table must be the same for all the rows. That's not a problem in our original version of the spreadsheet.

### Choosing a Data Type for Each Column

The second criterion is to determine what type of data goes in each column. While spreadsheets allow each cell to have a different type, in a database table, each column must have the same

type. Just like most programming languages, databases use *types* to classify different data values. Most of the time, the basic types are all you need to know. The main choices are integer numbers, floating-point numbers, fixed-length text, variable-length text, and dates. Often, the easiest way to decide the appropriate type is simply to look at some sample data.

In our customer data, it might be appropriate to use a text type for all the columns, even though the phone numbers are numbers. Storing the phone number as a simple number often presents some problems: it could easily result in the loss of leading zeros, prevent us from storing international dial codes (+), disallow using brackets around area codes, and so on. Obviously, a phone number can be much more than a simple string of numerals. Then again, using a character string to store the phone number might not be the best decision, since we could also accidentally store all sorts of strange characters, but it seems a better starting point than a number type. The initial design can always be refined later.

We can see that the length of the title (Mr, Mrs, Dr) is always very short—probably never longer than four characters. Similarly, ZIP codes also have a fixed maximum length. Therefore, we will make both of these columns fixed-length fields, but leave all the other columns as variable length, since there is no easy way of knowing how long a person’s last name might be, for example.

We will come back to PostgreSQL data types in the “Basic Data Types” section later in this chapter and also in Chapter 8.

## Identifying Rows Uniquely

Our last problem in transforming our spreadsheet into a database table is a little more subtle, as it comes from the way databases manage relations between tables. We need to decide what makes each row of customer data different from any other customer row in the database. In other words, how do we tell our customers apart? In a spreadsheet, we tend not to worry about the exact details of what distinguishes customers. However, in a database design, this is a key question, since relational database rules require each row to be unique in some way.

The obvious solution to distinguishing customers might seem to be by name, but unfortunately, that’s often not good enough. It is quite possible that two customers will have the same name. Another item you might choose is the phone number, but that fails when two customers live at the same address. At this point, you might suggest using a combination of name and phone number.

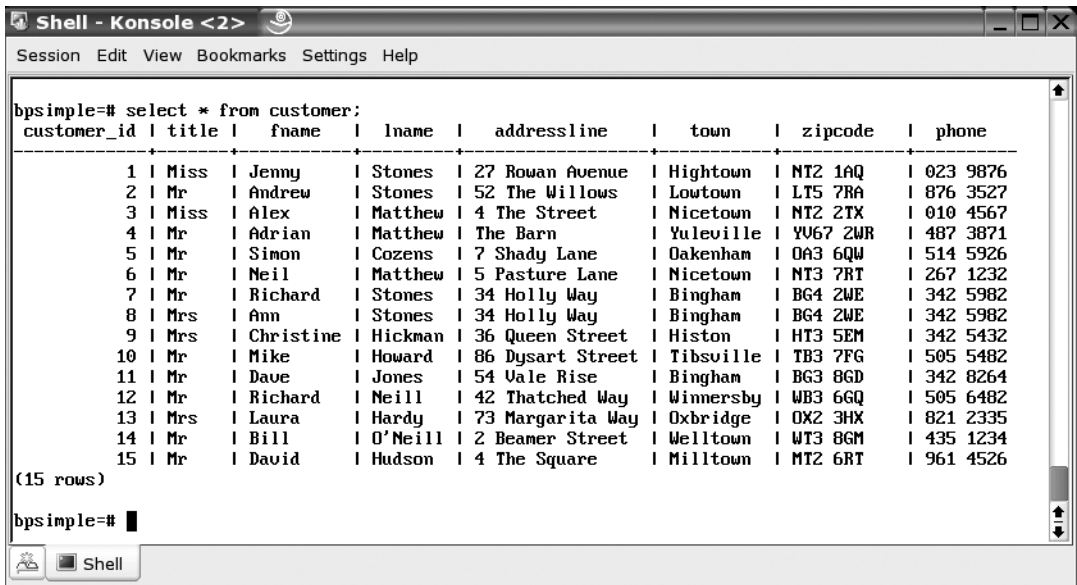
Certainly, it’s unlikely that two customers will have both the same name and the same phone number, but quite apart from being inelegant, another problem is lurking. What happens if a customer changes to a new phone provider and subsequently the phone number changes? By our definition, a unique customer must then be a new customer, because it is different from the customer we had before. Of course, we know that it is the same customer, with a new phone number. In a database, it’s generally bad practice to pick a unique identifying feature for a customer that might subsequently change, as it’s hard to manage changes to unique identifiers.

This sort of problem, identifying uniqueness, turns up frequently in database design. What we have been doing is looking for a *primary key*—an easy way to distinguish one row of customer data from all the other rows. Unfortunately, we have not yet succeeded, but all is not lost, since the standard solution is to assign a unique number to each customer.



We simply give each customer a unique number, and bingo, we have a distinct way to tell customers apart, regardless of whether they change their phone number, move to a new residence, or even change their name. This type of addition to a row to provide a unique key when no good choice exists in the actual data is called adding a *surrogate key*. This is such a common occurrence in real-world data that there is even a special data type in most databases, the serial data type, to help solve the problem. We will discuss this type later in the chapter, in the “Basic Data Types” section.

Now that we have decided on a database design for our initial table, it’s time to store our data in a database. Figure 2-4 shows our data in a PostgreSQL database being viewed using a simple command-line tool, `psql`, in a terminal window on a Linux machine.



```

Shell - Konsole <2>
Session Edit View Bookmarks Settings Help

bpsimple=# select * from customer;
 customer_id | title | fname | lname | addressline | town | zipcode | phone
-----
1 | Miss | Jenny | Stones | 27 Rowan Avenue | Hightown | NT2 1AQ | 023 9876
2 | Mr | Andrew | Stones | 52 The Willows | Lowtown | LT5 7RA | 876 3527
3 | Miss | Alex | Matthew | 4 The Street | Nicetown | NT2 2TX | 010 4567
4 | Mr | Adrian | Matthew | The Barn | Yuleville | YU67 ZWR | 487 3871
5 | Mr | Simon | Cozens | 7 Shady Lane | Oakenham | OA3 6QW | 514 5926
6 | Mr | Neil | Matthew | 5 Pasture Lane | Nicetown | NT3 7RT | 267 1232
7 | Mr | Richard | Stones | 34 Holly Way | Bingham | BG4 2WE | 342 5982
8 | Mrs | Ann | Stones | 34 Holly Way | Bingham | BG4 2WE | 342 5982
9 | Mrs | Christine | Hickman | 36 Queen Street | Histon | HT3 5EM | 342 5432
10 | Mr | Mike | Howard | 86 Dysart Street | Tibsville | TB3 7FG | 505 5482
11 | Mr | Dave | Jones | 54 Vale Rise | Bingham | BG3 8GD | 342 8264
12 | Mr | Richard | Neill | 42 Thatched Way | Wimmersby | WB3 6GQ | 505 6482
13 | Mrs | Laura | Hardy | 73 Margarita Way | Oxbridge | OX2 3HX | 821 2335
14 | Mr | Bill | O'Neill | 2 Beamer Street | Welltown | WT3 8GM | 435 1234
15 | Mr | David | Hudson | 4 The Square | Milltown | MT2 6RT | 961 4526

(15 rows)

bpsimple=#

```

Figure 2-4. Command-line viewing of customer data from a database

Notice that we have added an extra column, `customer_id`, as our unique way of referencing a customer. It is our primary key for the table. As you can see, the data looks much as it did in a spreadsheet, laid out in rows and columns. In later chapters, we will explain the actual mechanics of defining a database table, storing, and accessing the data, but rest assured, it’s not difficult.

## Accessing Data in a Database

You can easily view your PostgreSQL data using the `psql` tool from the command line, as you saw in Figure 2-4. However, PostgreSQL is not restricted to command-line use. Figure 2-5 shows the more user-friendly graphic approach of pgAdmin III, a free tool available from <http://www.pgadmin.org/>, and also bundled with the Windows distributions of PostgreSQL from version 8. We will see more about graphical interfaces in Chapter 5.

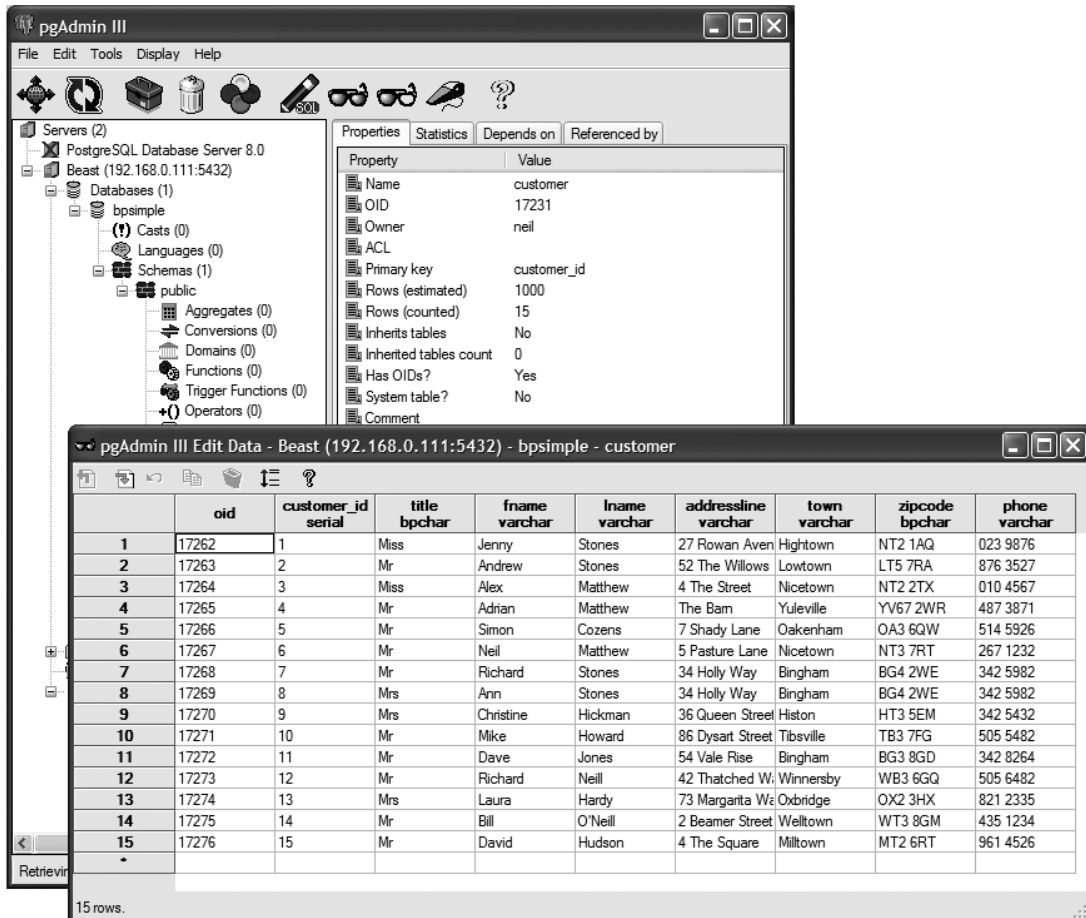


Figure 2-5. Viewing customer data from a database with pgAdmin III

## Accessing Data Across a Network

Of course, if we could only access our data on the machine on which it was physically stored, the situation really wouldn't have improved much over the single spreadsheet file being shared among different users.

PostgreSQL is a server-based database, and as described in the previous chapter, once configured, will accept requests from clients across a network. Although the client can be on the same machine as the database server, for multiuser access, this won't normally be the case. For Microsoft Windows users, an ODBC driver is available, so we can arrange to connect any Windows desktop application that supports ODBC across a network to a server holding our data. Figure 2-6 shows Microsoft Access on a Windows PC accessing a PostgreSQL database running on a Linux machine. This is done using linked external tables via an ODBC connection across the network.

Microsoft Access

db1 : Database (Access 2000 file format)

public\_customer : Table

	customer_id	title	fname	lname	addressline	town	zipcode	phone
	1	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876
	2	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527
	3	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567
	4	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871
	5	Mr	Simon	Cozens	7 Shady Lane	Oakenham	OA3 6QW	514 5926
	6		Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232
	7	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
	8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
	9	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432
	10	Mr	Mike	Howard	86 Dysart Street	Tibsville	TB3 7FG	505 5482
	11	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264
	12	Mr	Richard	Neill	42 Thatched Way	Winnersby	WB3 6GQ	505 6482
	13	Mrs	Laura	Hardy	73 Margarita Way	Oxbridge	OX2 3HX	821 2335
	14	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234
	15	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526

Record: 15 of 15

Datasheet View

**Figure 2-6.** Accessing the same data from Microsoft Access

Now we can access the same data from many machines across the network at the same time. We have one copy of the data, securely held on a central server, accessible to multiple desktops running different operating systems, across a network.

We will see the technical details of configuring an ODBC connection in Chapter 5.

## Handling Multiuser Access

PostgreSQL, like all relational databases, can automatically ensure that conflicting updates to the database can never occur. It looks to the users as though they all have unrestricted access to all the information at the same time, but behind the scenes, PostgreSQL is monitoring changes and preventing conflicting updates.

This ability to allow many people to apparently have simultaneous read and write access to the same data, but ensure that it remains consistent, is a very important feature of databases. When a user changes a column, you either see it before it changes or after it changes; you never see partial updates.

A classic example is a bank database transferring money between two accounts. If, while the money was being transferred, someone were to run a report on the amount of money in all the accounts, it's very important that the total be correct. It may not matter in the report which account the money was in at the instant the report was run, but it is important that the report doesn't see the in-between point, where one account has been debited but the other not credited.

Relational databases like PostgreSQL hide any intermediate states, so they cannot be seen by other users. This is termed *isolation*. The report operation is isolated from the money-transfer operation, so it appears to happen either before or after, but never at exactly the same instant. We will come back to this concept of isolation in Chapter 9 when we look at Transactions.

## Slicing and Dicing Data

Now that we have seen how easy it is to access the data once it is in a database table, let's have a first look at how we might actually process that data. We frequently need to perform two very basic operations on big sets of data: selecting rows that match a particular set of values and selecting a subset of the columns of the data. In database terminology, these are called *selection* and *projection* respectively. That may sound somewhat complex, but accomplishing selection and projection is actually quite simple.

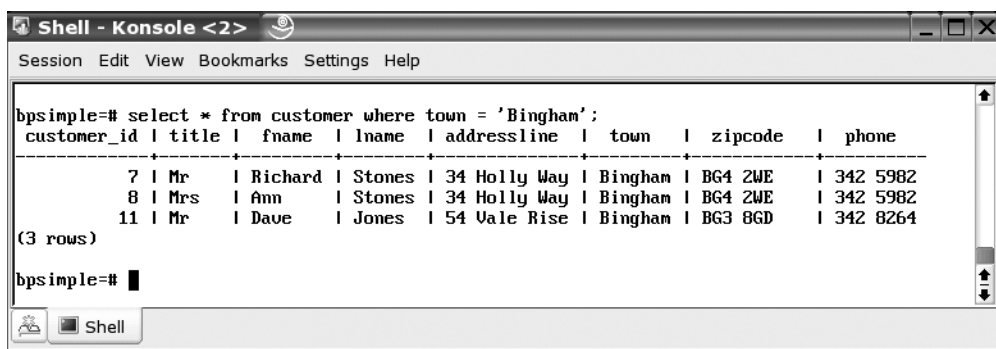
### Selection

Let's start by looking at selection, where we are selecting a subset of the rows. Suppose we want to see all our customers who live in the town Bingham. Let's return to PostgreSQL's standard command-line tool, `psql`, to see how we can use the SQL language to ask PostgreSQL to get the data we want. The SQL command we need is very simple:

```
SELECT * FROM customer WHERE town = 'Bingham'
```

If you are typing in your SQL statements (using a command-line tool like `psql` or a graphical tool such as pgAdmin III), you also need to add a semicolon at the end. The semicolon tells `psql` that this is the end of a command, because longer commands might extend over more than one line. Generally, in this book, we will show the semicolon.

PostgreSQL responds by returning all the rows in the `customer` table, where the `town` column contains Bingham, as shown in Figure 2-7.



```
Shell - Konsole <2>
Session Edit View Bookmarks Settings Help

bpsimple=# select * from customer where town = 'Bingham';
customer_id | title | fname | lname | addressline | town | zipcode | phone
-----
7 | Mr | Richard | Stones | 34 Holly Way | Bingham | BG4 2WE | 342 5982
8 | Mrs | Ann | Stones | 34 Holly Way | Bingham | BG4 2WE | 342 5982
11 | Mr | Dave | Jones | 54 Vale Rise | Bingham | BG3 8GD | 342 8264
(3 rows)

bpsimple=#
```

Figure 2-7. Selecting a subset of the data rows

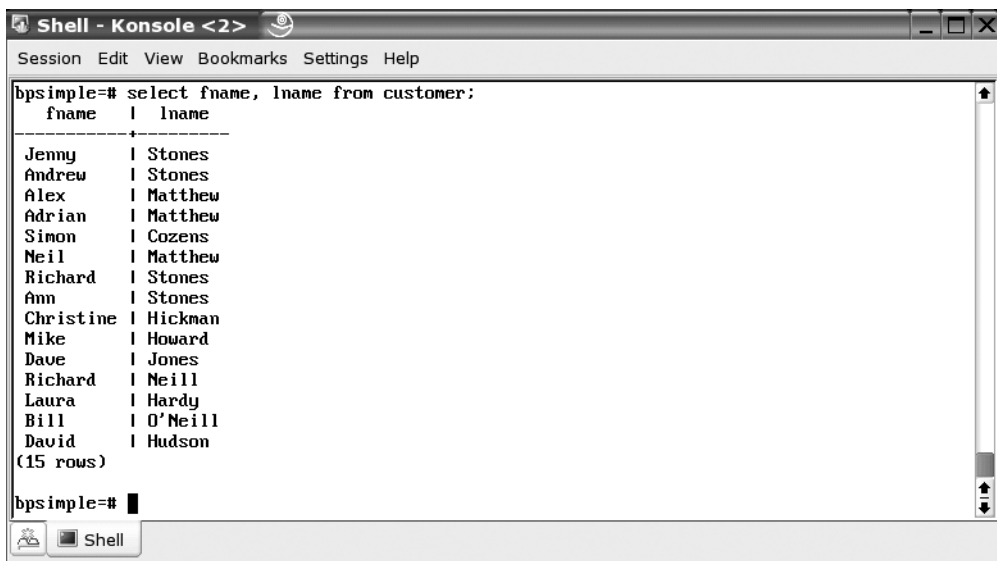
So that was selection, where we choose particular rows from a table. As you can see, that was pretty easy. Don't worry about the details of the SQL statement yet. We will come back to that more formally in Chapter 5.

## Projection

Now let's look at projection, where we are selecting particular columns from a table. Suppose we wanted to select just the first name and last names from our customer table. You will remember that we called those columns `fname` and `lname`. The command to retrieve the names is also quite simple:

```
SELECT fname, lname FROM customer;
```

PostgreSQL responds by returning the appropriate columns, as shown in Figure 2-8.

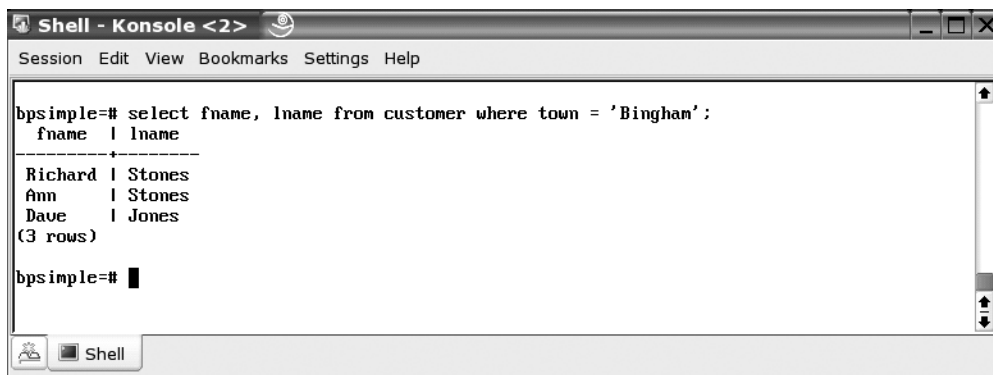


**Figure 2-8.** *Selecting a subset of the data columns*

You might reasonably suppose that sometimes we want to do both operations on the data at the same time; that is, select particular column values but only from particular rows. That's pretty easy in SQL as well. For example, suppose we wanted to know only the first names and last names of all our customers who live in Bingham. We can simply combine our two SQL statements into a single command:

```
SELECT fname, lname FROM customer WHERE town = 'Bingham';
```

PostgreSQL responds with our requested data, as shown in Figure 2-9.



```
Shell - Konsole <2>
Session Edit View Bookmarks Settings Help

bpsimple=# select fname, lname from customer where town = 'Bingham';
  fname | lname
-----+-----
 Richard | Stones
    Ann  | Stones
    Dave  | Jones
(3 rows)

bpsimple=#
```

**Figure 2-9.** *Selecting a subset of both columns and rows*

There is one very important thing to notice here. In many traditional programming languages, such as C or Java, when searching for data in a file, we would have written some code to scan through all the lines in the file, printing out names each time we came across one with the town we were searching for. Although it might be possible to squeeze that much logic onto a physical single line of code, it would be a very long and complex line, unlike the succinct line of SQL shown here. This is because C, Java, and similar languages are essentially procedural languages. You specify in the language how the computer should behave. In SQL, which is termed a *declarative language*, you tell the computer what you are trying to achieve, and PostgreSQL works some internal magic to handle this task for you.

This might seem a little strange if you have never used a declarative language before, but once you get used to the idea, it seems obvious that it's a much better idea to tell the computer what you want, rather than how to do it. You will wonder how you have managed without such languages till now.

## Adding Information

So far, all we have looked at is our database emulating a single worksheet in a spreadsheet, and we've just touched the surface of SQL's features. As we will see in this book, however, relational databases such as PostgreSQL are very rich in useful features, which take them well beyond the realms of spreadsheet capabilities. One of the most important capabilities of databases is their ability to link data together across tables, and that is what we will look at now.

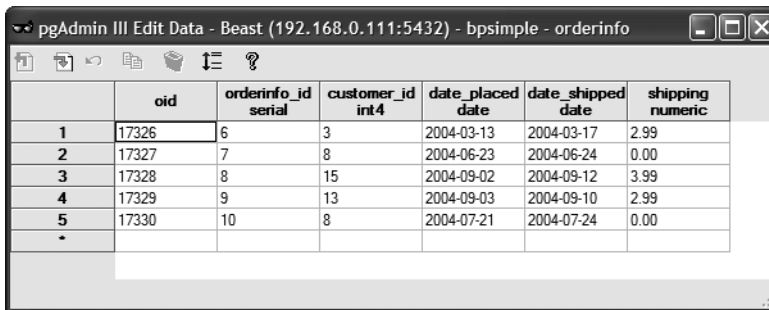
## Using Multiple Tables

Recall our customer order problem, where our simple customer spreadsheet suddenly became very untidy once additional order information was stored for each customer. How do we store information about orders from customers when we don't know in advance how many orders a customer might make? As you can probably guess from the title of this section, the way to solve this problem with a relational database is to add another table to store this information.

Just as we designed our customer table, we start by deciding what information we want to store about each order. For now, let's assume that we want to store the name of the customer who placed the order, the date the order was placed, the date it was shipped, and how much we charged for delivery. As in our customer table, we will also add a unique reference number for each order, rather than try to make any assumptions about what might be unique. There is obviously no need to store all the customer details again. We already know that given a `customer_id`, we can find all the details of that customer in the customer table.

You might be wondering why we've omitted the details of what was ordered. Certainly, that is an important aspect of orders to most customers—they like to get what they ordered. If you're thinking that it's a similar problem to not knowing in advance how many orders a customer will place, you're quite right. We have no idea how many items will be on each order. The repeating groups problem is never far away. We will leave this aside for now and deal with it in the "Creating a Simple Database Design" section later in this chapter.

Figure 2-10 shows our order information table with some sample data, again shown in the graphical tool, pgAdmin III.



The screenshot shows the pgAdmin III interface with the 'orderinfo' table selected. The table has the following columns: `oid`, `orderinfo_id serial`, `customer_id int4`, `date_placed date`, `date_shipped date`, and `shipping numeric`. The data rows are as follows:

	oid	orderinfo_id serial	customer_id int4	date_placed date	date_shipped date	shipping numeric
1	17326	6	3	2004-03-13	2004-03-17	2.99
2	17327	7	8	2004-06-23	2004-06-24	0.00
3	17328	8	15	2004-09-02	2004-09-12	3.99
4	17329	9	13	2004-09-03	2004-09-10	2.99
5	17330	10	8	2004-07-21	2004-07-24	0.00
*						

**Figure 2-10.** Some order information viewed in pgAdmin III

We haven't put too much data in the table, as it is easier to experiment on smaller amounts of data. You will notice an extra column, `oid`, which isn't part of our user data. This is a special column used internally by PostgreSQL. The current version of PostgreSQL defaults to creating this column on all tables, but hides it from the `SELECT *` command. We will discuss this column in Chapter 8.

## Relating a Table with a Join Operation

Now we have details of our customers, and at least summary details of their orders, stored in our database. In many ways, this is no different from using a pair of spreadsheets: one for our customer details and one for their order details. It's time to look at what we can do using these tables in combination, and start to see the power of databases. We do this by selecting data from both tables at the same time. This is called a *join*, which, after selection and projection from a single table, is the third most common SQL data-retrieval operation.

Suppose we want to list all the orders and the customers who placed them. In a procedural language, such as C, we would need to write code to scan one of the tables, perhaps starting with the customer table, then for each customer, we look for and print out any orders they have placed. That's not difficult, but it's certainly a bit time-consuming and tedious to code. I'm sure you will be pleased to know we can find the answer much more easily with SQL, using a join operation. All we need to do is tell SQL three things:

- The columns we want
- The tables we want the data retrieved from
- How the two tables relate to each other

The command we need is the example presented in the previous chapter:

```
SELECT * FROM customer, orderinfo
WHERE customer.customer_id = orderinfo.customer_id;
```

As you can probably guess, this asks for all columns from our two tables, and tells SQL that the column `customer_id` in the table `customer` holds the same information as the `customer_id` column in the `orderinfo` table. Note the convenient *table.column* notation, which enables us to specify both a table name and a column within that table. The `*` in our command means all columns. We could instead use named columns to select only specified columns, if we just wanted names and amounts, for example.

Now that we have a database with some tables and data, we can see how PostgreSQL responds in Figure 2-11.

```
bpsimple=# select * from customer, orderinfo where customer.customer_id = orderinfo.customer_id;
```

customer_id	title	fname	lname	addressline	town	zipcode	phone	orderinfo_id	shipping
3	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567	6	
3									2.99
8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982	7	
8									0.00
8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982	10	
8									0.00
13	Mrs	Laura	Hardy	73 Margarita Way	Oxbridge	OX2 3HX	821 2335	9	
13									2.99
15	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526	8	
15									3.99

```
(5 rows)
bpsimple=#
```

**Figure 2-11.** Selecting data from two tables in one operation

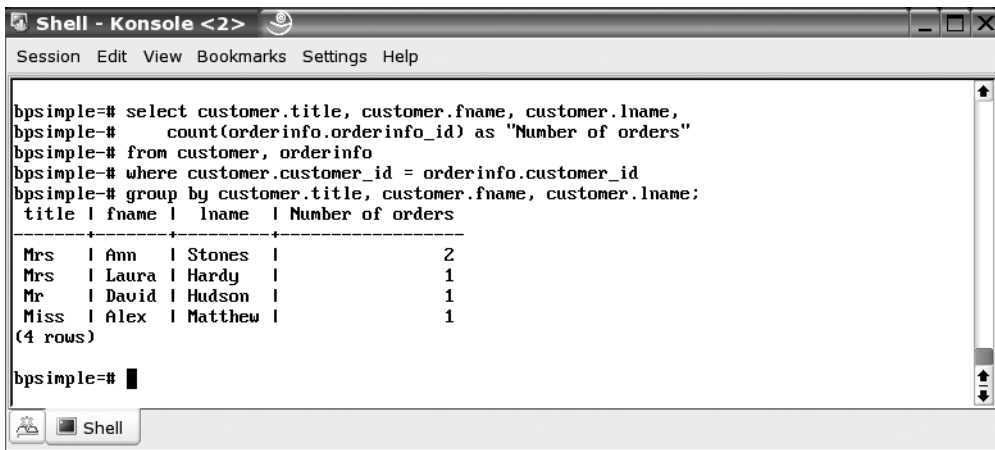


This is a bit untidy, since the rows wrap to fit in the window, but you can see how PostgreSQL has answered our query, without us needing to specify exactly how to solve the problem.

Let's leap ahead briefly, and see a much more complex query we could perform using SQL on these two tables. Suppose we wanted to see how frequently different customers had placed orders with us. This requires a significantly more advanced bit of SQL:

```
SELECT customer.title, customer.fname, customer.lname,
       count(orderinfo.orderinfo_id) AS "Number of orders"
FROM customer, orderinfo
WHERE customer.customer_id = orderinfo.customer_id
GROUP BY customer.title, customer.fname, customer.lname;
```

That's a complex bit of SQL, but without going into the details, you can see that we still have not told SQL how to answer the question; we've just specified the question in a very precise way using SQL. We also managed it all in a single statement. For the record, Figure 2-12 shows how PostgreSQL responds.



```
Shell - Konsole <2>
Session Edit View Bookmarks Settings Help

bpsimple=# select customer.title, customer.fname, customer.lname,
bpsimple=#         count(orderinfo.orderinfo_id) as "Number of orders"
bpsimple=# from customer, orderinfo
bpsimple=# where customer.customer_id = orderinfo.customer_id
bpsimple=# group by customer.title, customer.fname, customer.lname;
 title | fname | lname | Number of orders
-----+-----+-----+-----
 Mrs   | Ann   | Stones |                2
 Mrs   | Laura | Hardy  |                1
 Mr    | David | Hudson |                1
 Miss  | Alex  | Matthew |               1
(4 rows)

bpsimple=#
```

**Figure 2-12.** Retrieving order frequency

Some database experts may like typing SQL directly into a window using a command-line tool, and it certainly is useful sometimes, but it's not everyone's preference. If you prefer to build your queries graphically, that's not a problem. As noted earlier in this chapter, you can simply access the database via an ODBC driver and use a Windows graphical user interface (GUI), for example. Figure 2-13 shows the same query being designed and executed in Access on a Windows machine, using the PostgreSQL ODBC driver and linked external tables. We will see some other GUI tools, such as Rekall running on a Linux desktop, in Chapter 5.

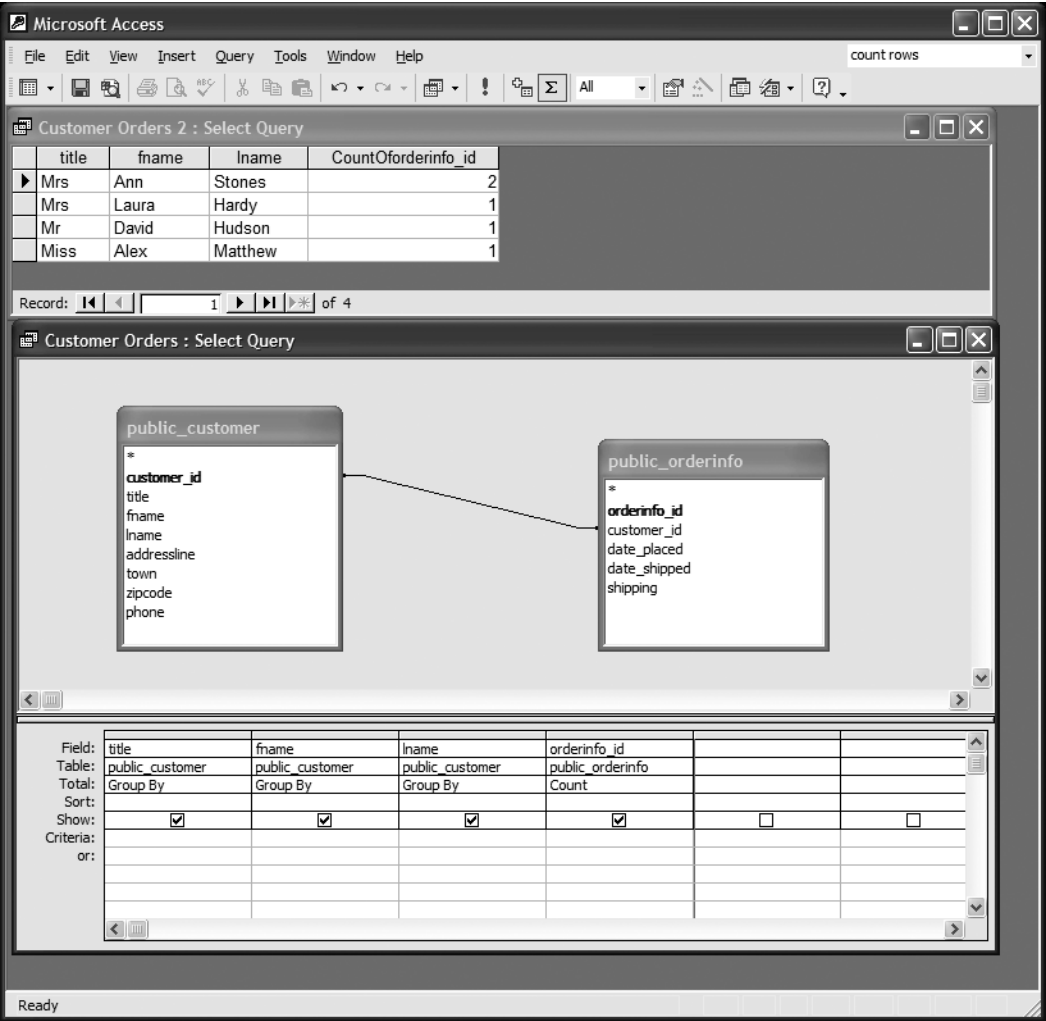


Figure 2-13. Building a query graphically

In our particular environment, the data is still stored on a Linux machine, but the user hardly needs to be aware of the technical details. Generally, in this book, we will use the command line for teaching SQL, because that way you will learn the basics before moving on to more complex SQL commands. Of course, you are welcome to use a GUI rather than a command-line tool to construct your SQL commands; it's your choice.

## Designing Tables

So far, we have only two tables in our database, and we have not really talked about how we decide what goes in each table, except in the very informal way of doing what looked reasonable. This design, which includes tables, columns, and relationships, is more correctly called a *schema*.

Designing a database schema with more than a couple of dozen tables can be quite challenging if the data is complex. Database designers earn their money by being good at this difficult task. Fortunately, for relatively simple databases, with up to perhaps ten tables, it's possible to come up with a fairly good design just by applying some basic rules of thumb, rather than needing to apply rules in a more formal way.

In this section, we are going to look at the simple sample database we are starting to build, and figure out a way to decide what tables we need.

## Understanding Some Basic Rules of Thumb

When a database is designed, it is often *normalized*; that is, a set of rules is applied to ensure that data is broken down in an appropriate fashion. In Chapter 12, we will look at database design in a formal way. To get started, all we require are some simple ground rules. These rules are just to help you understand the initial database, named *bpsimple*, we will be using to explore SQL and PostgreSQL in this and the following chapters. We strongly suggest that you don't just read these rules, and then dash off to design a database with 20 tables. Work your way through the book—at least until Chapter 12.

---

**Tip** If you're interested in learning more about normal forms, we suggest Joe Celko's *SQL for Smarties* (ISBN 1-55860-576-2). It has some excellent definitions of the various rules of normalization, as well as other rules Dr. E. F. Codd defined for the relational model and many advanced examples of SQL usage.

---

### Rule One: Break Down the Data into Columns

The first rule is to put only one piece of information, or data *attribute*, in each column. This comes naturally to most people, provided they consciously think about it. In our original spreadsheet, we have already quite naturally broken down the information for each customer into different columns, so the name was separate from the ZIP code, for example.

In a spreadsheet, this rule just makes it simpler to work on the data; for example, to sort by the ZIP code. In a database, however, it is essential that the data is correctly broken down into attributes.

Why is this so important in databases? From a practical point of view, it is difficult to specify that you want the data between the twenty-ninth and thirty-fifth characters from an address column, because that happens to be where the ZIP code lives. There is bound to be some place where the rule does not hold, and you get the wrong piece of data. Another reason for the data to be correctly broken down is that all columns in a database must have the same type, unlike a spreadsheet, which is quite forgiving about the types of data in a column.

### Rule Two: Have a Unique Way of Identifying Each Row

You will remember that when we tried to decide how to identify each row in the spreadsheet example at the beginning of this chapter, we had a problem of not being sure what would be unique. As was mentioned, this was because there was no primary key. In general, it doesn't need to be a single column that is unique; it could be a pair of columns taken together,

or occasionally even the combination of three columns that uniquely identifies a row. It is rare, and probably a mistake, if you find yourself requiring more than three columns to uniquely identify a row.

In any case, there must be a way of saying, with absolute certainty, if I look at the contents of a particular column, or group of columns in this row, I know it will have a value different from all other rows in this table. If you cannot find a column, or at most a combination of three columns, that uniquely identifies each row, it's time to add an extra column to fulfill that purpose. In our customer table, we added an extra column, `customer_id`, to identify each row.

### Rule Three: Remove Repeating Information

Recall that when we tried to store order information in the customer table, it looked rather untidy because of the repeating groups. For each customer, we repeated order information as many times as was required. This meant that we could never know how many columns were needed for orders. In a database, the number of columns in a table is effectively fixed by the design. So we must decide in advance how many columns we need, what type they are, and name each column before we can store any data. Never try to store repeating groups of data in a single row.

The way around this restriction is to do exactly what we did with our orders and customers data: split the data into separate tables. Then you can join the tables together when you need data from both tables. In our example, we used the column `customer_id` to join the two tables.

More formally, what we had was a *many-to-one relationship*; that is, there could be many orders received from a single customer.

### Rule Four: Get the Naming Right

This is occasionally the hardest rule to implement well. What do we call a table or column? Tables and columns should have short, meaningful names. If you cannot decide what to call something, it's often a clue that all is not well in your table and column design.

In addition to coming up with appropriate names, most database designers have their own personal rules of thumb, or *naming conventions*, that they use to ensure the naming of tables and columns in a database is consistent. Don't have some table names singular and some plural. For example, rather than naming one table `office` and the other `departments`, use `office` and `department`. If you decide on a naming rule for an `id` column—perhaps the table name with an appended `_id`—stick to that rule. If you use abbreviations, always use them consistently. If a column in one table is a key to another table (a *foreign key*, as explained in Chapter 12), try to give them the same base name. In a complex database, it can get very annoying when names are not quite consistent, such as `customer_id`, `customer_ident`, `cust_id`, and `cust_no`.

Achieving this apparently simple goal of getting the names right is often surprisingly challenging, but the rewards in simplified maintenance are considerable.

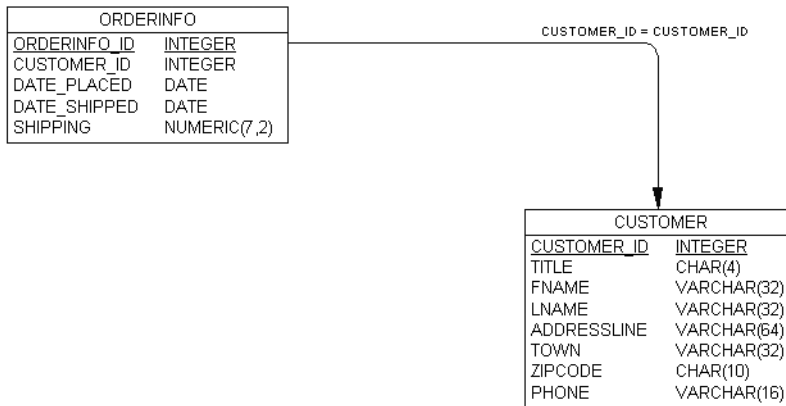
## Creating a Simple Database Design

We can draw our database design, or schema, using an entity relationship diagram. For our two-table database, such a diagram might look like Figure 2-14.

---

**Note** An *entity relationship diagram* is a graphical way of representing the logical structure of our data. It helps us visualize how the different entities in our data relate to each other.

---



**Figure 2-14.** A simple entity relationship diagram

This diagram shows our two tables, the column, the data types, and the sizes in each column, and also tells us that `customer_id` is the column that joins the two tables together. Notice that the arrow goes from the `orderinfo` table to the `customer` table. This is a hint that for each `orderinfo` entry, there is at most a single entry in the `customer` table, but that for each customer there may be many orders. Also notice that some columns are underlined, which indicates that the column is guaranteed to be unique. These columns form the primary key for the tables.

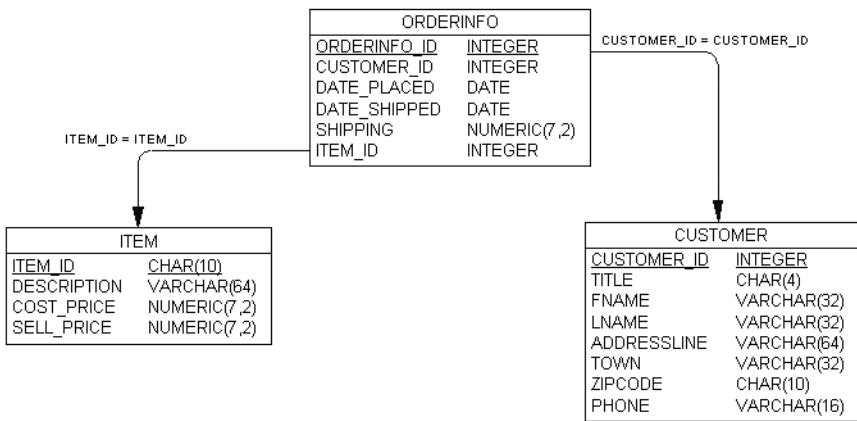
It's important that you remember which way a one-to-many relationship goes; getting it confused can cause a lot of problems. You should also notice that we have been very careful to name the column we want to use to join the two tables the same in each table: `customer_id`. This is not essential. We could have called the two columns `foo` and `bar` if we had wanted to, but, as noted in the previous section, consistent naming is a great help in the long run.

The next stage is to extend our very simple two-table design into something slightly more realistic. We will design it as a simple order-management database, called `bpsimple`.

## Extending Beyond Two Tables

Clearly, the information we have so far is lacking, in that we don't know what items were in each order. You may remember that we deliberately omitted the actual items from each order, promising to come back to that problem. It's now time to sort out the actual items in each order.

The problem we have is that we don't know in advance how many items there will be in each order. It's almost the same as not knowing in advance how many orders a customer might place. Each order might have one, two, three, or a hundred items in it. We must separate the information that a customer placed an order from the details of what was in that order. Basically, what we might try is something like what is shown in Figure 2-15.



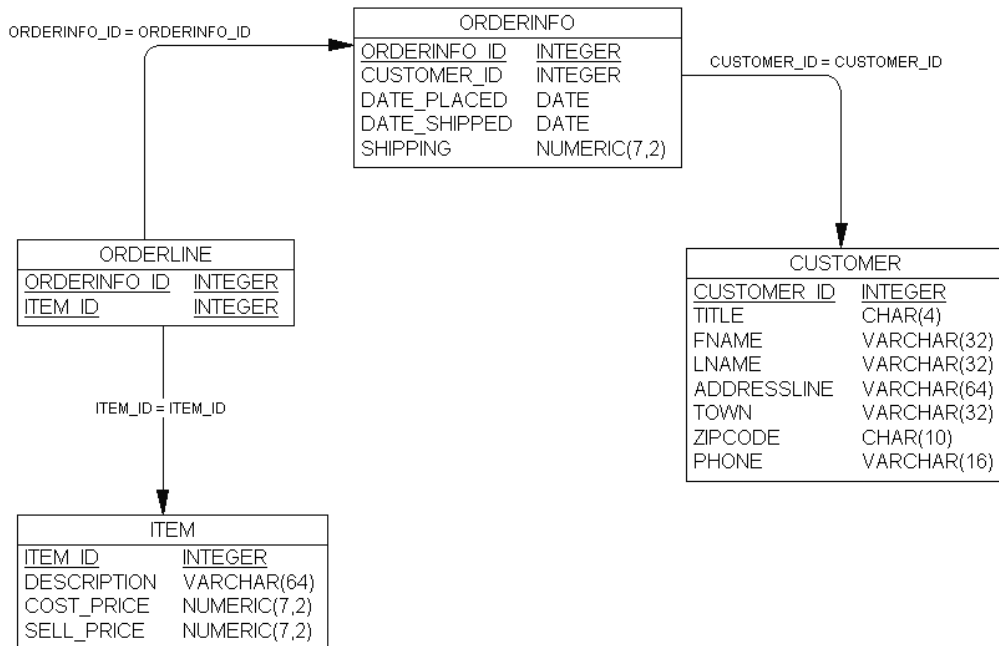
**Figure 2-15.** An attempt at relating customers and ordered items

Much like the customer and orderinfo tables, we separate the information into two tables, and then join them together. We have, however, created a subtle problem here.

If you think carefully about the relationship between an order and an item that may be ordered, you will realize that not only could each orderinfo entry relate to many items, but each item could also appear in many orders, if different customers order the same item.

We will consider this problem further in Chapter 12, but for now, you will be pleased to know that there is a standard solution to this difficulty. You create a third table between the two tables, which implements a *many-to-many relationship*. This is actually easier to do than it is to explain, so let's just go ahead and create a table, orderline, to link the orders with the items, as shown in Figure 2-16.

We have created a table that has rows corresponding to each line of an order. For any single line, we can determine the order it was from using the orderinfo\_id column and the item referenced using the item\_id column. A single item can appear in many order lines, and a single order can contain many order lines. Each order line refers to only a single item, and it can appear in only a single order.



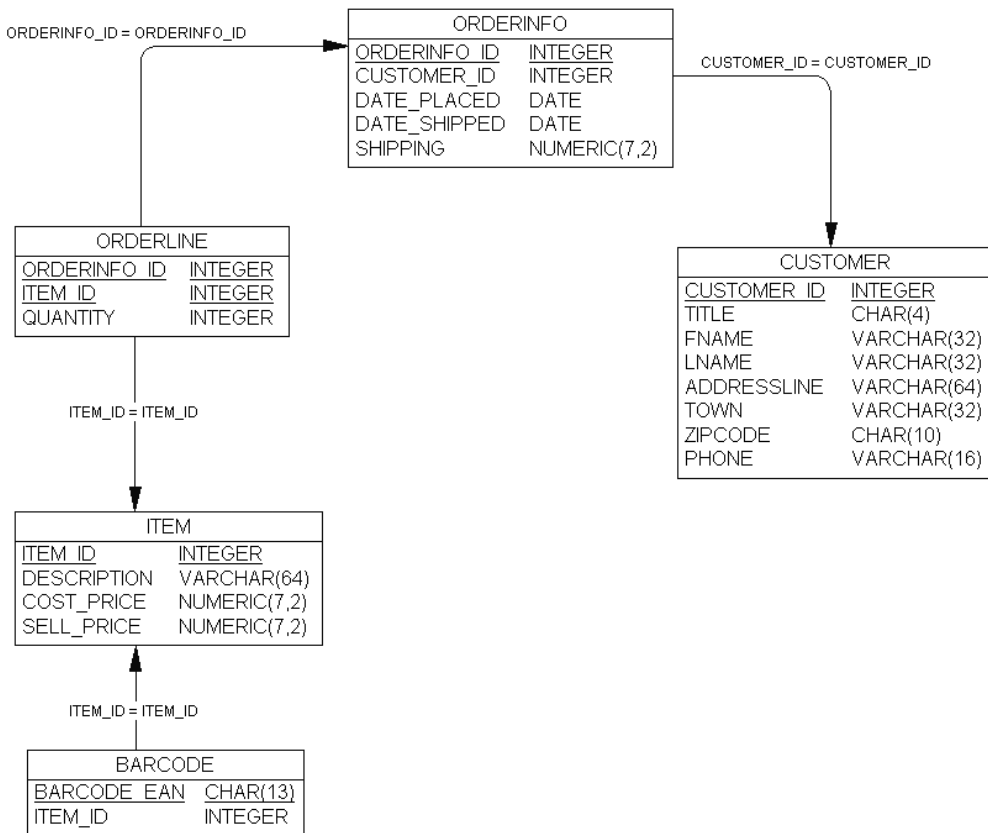
**Figure 2-16.** *Relating customers and orders*

You will also notice that we did not need to add a unique id column to identify each row. That is because the combination of `orderidno_id` and `item_id` is always unique. There is one very subtle problem lurking, however. What happens if a customer orders two of an item in a single order? We cannot just enter another row in `orderline`, because we just said that the combination of `orderidno_id` and `item_id` is always unique. Do we need to add yet another special table to cater to orders that contain more than one of any item? Fortunately, we don't need to do this. There is a much simpler approach. We just need to add a quantity column to the `orderline` table, and all will be well (see Figure 2-17, in the following section).

## Completing the Initial Design

We have just two more pieces of information we need to store before we have the main structure of the first cut of our database design in place. We want to store the barcode that goes with each product, and we also want to store the quantity we have in stock for each item.

It's possible that each product will have more than one barcode, because when manufacturers significantly change the packaging of a product, they often also change the barcode. For example, you have probably seen packs that offer “20% extra for free” (often referred to in the trade as *overflow packs*). Manufacturers will generally change the barcode of these promotion packs, but essentially the product is unchanged. Therefore, we may have a many barcodes-to-one item relationship. We add an additional table to hold the barcodes, as shown in Figure 2-17.

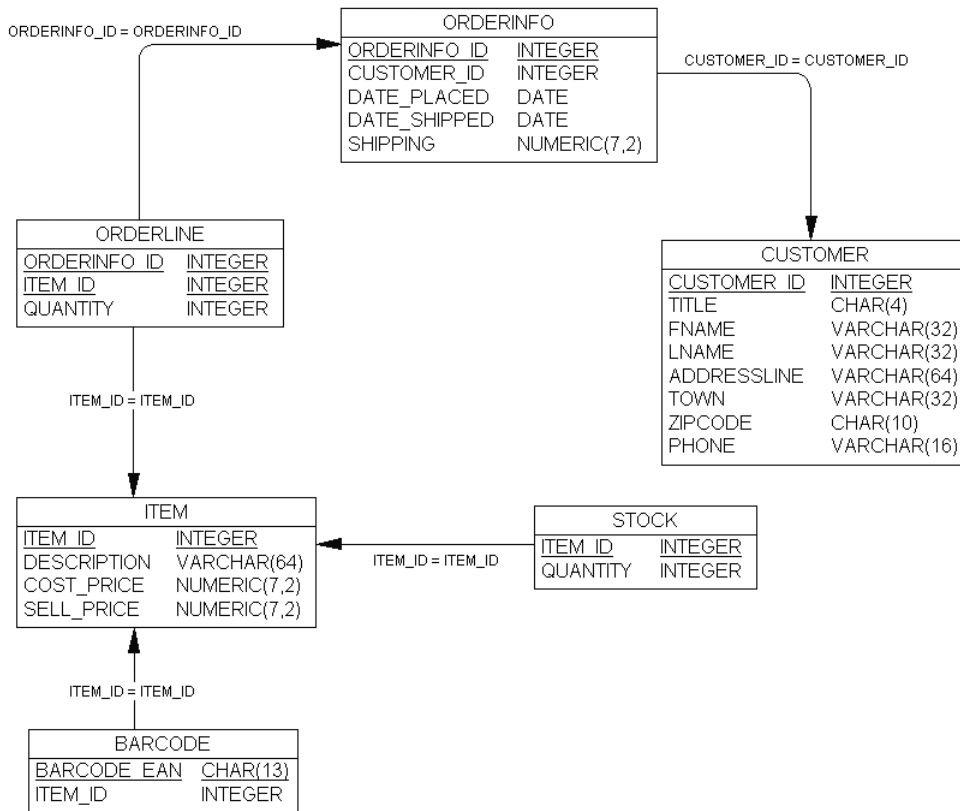


**Figure 2-17.** Adding the barcode relationship

Notice that the arrow points from the barcode table to the item table, because there may be many barcodes for each item. Also notice that the barcode\_ean column is the primary key, since there must be a unique row for each barcode, and a single item could have several barcodes, but no barcode can ever belong to more than one item. (EAN is a European standard for product barcodes.)

The last addition we need to make to our database design is to hold the stock quantity for each item. If most items were in stock, and the stock information were fairly basic, we could simply store a stock quantity directly in the item table. However, this won't work if we offer many items, but only a few are normally in stock, and we need to store a lot of information about the stocked items. For example, in a warehouse operation, we may need to store location information, batch numbers, and expiration dates. If we had an item file with 500,000 items in it, but only held the top 1,000 items in stock, this would be very wasteful. There is a standard way of resolving this problem, using what is called a *supplementary table*. We will take this approach to store stock information for our sample database, as shown in Figure 2-18.





**Figure 2-18.** The design of the *bpsimple* database

We create a new table to store the supplementary information (stock quantity, in this example), and then create only the rows that are required for items that are in stock, linking the information back to the main table. Notice the stock table uses `item_id` as a unique key, and it holds information that relates directly to items, using `item_id` to join to the relevant row in the `item` table. The arrow points to the `item` table, because that is the master table, even though it is not a many-to-one relationship in this case. As in the other tables, the underlining indicates the table's primary key (the information guaranteed to be unique).

As it stands, the design is clearly overly complex, since the additional information we are keeping is so small. We will leave the schema design the way it is to show how it is done, and later in the book, we will demonstrate how to access data when there is additional information in supplementary tables like this one. For those who like sneaking a look ahead, we will use what's called an *outer join*.

---

**Note** In Chapter 8, we will see how we can enforce in the database the rules about relationships between tables, and in Chapter 12 we will revisit the design of databases in more detail. When we get to Chapter 8, we will discover some more advanced techniques to better manage the consistency of our database, and we will enhance our design into a *bpfinal* schema.

---

## Basic Data Types

In our sample database, we've used some basic, generic data types, as summarized in Table 2-1. These can be translated into actual PostgreSQL types when we create the real tables in the next chapter.

**Table 2-1.** *Data Types in the Sample Database*

Data Type	Description
integer	A whole number.
serial	An integer, but automatically set to a unique number for each row that is added. This is the type we would use for the <code>_id</code> columns. The figures in this chapter show such fields as <code>integer</code> , because that's the underlying type in the database.
char	A character array of fixed size, with the size shown in parentheses after the type. For these column types, PostgreSQL will always store exactly the specified number of characters. If we use a <code>char(256)</code> to store just one character, there will still be (at least) 256 bytes held in the database and returned when the data is retrieved.
varchar	This is also a character array, but as its name suggests, it is of variable length. Generally, the space used in the database will be much the same as the actual size of the data stored. When you ask for a <code>varchar</code> field to be returned, it returns just the number of characters you stored. The maximum length is given in the parentheses after the type.
date	This allows you to store year, month, and day information. There are other related types that allow us to store time information as well as date information. We will meet these later in Chapter 8.
numeric	This allows you to store numbers with a specified number of digits (the first number in the parentheses) and using a fixed number of decimal places (the second number in the parentheses). Hence, <code>numeric(7,2)</code> would store exactly seven digits, two of them after the decimal place.

As noted earlier in the chapter, since the need to add a special unique column is so common in databases, there is a built-in solution in most databases: a data type known as `serial`. This special type is effectively an integer that automatically increments as rows are added to the table, assigning a new, unique number as each row is added. When we add a new row to a table that has a `serial` column, we don't specify a value for that column, but allow the database to automatically assign the next number. Most databases, when they assign `serial` values, don't take into account any rows that are deleted. The number assigned will just go on incrementing for each new row. We will look at how to handle out-of-sequence problems with `serial` data types in Chapter 6.

In Chapter 8, we will look at PostgreSQL's other data types, which will give us a chance to reexamine some of these data type choices. Appendix B provides a summary of the PostgreSQL data types.

## Dealing with the Unknown: NULLs

In the `orderinfo` table in our sample database design, we have a `date ordered` and a `date shipped` column, both of type `date`. What do we do when an order has been received but not yet shipped? What should we store in the `date shipped` column? We could store a special date, a *sentinel value*, that lets us know that we have not yet shipped the order. On UNIX-type systems, we might use January 1, 1970, which is traditionally the date from which UNIX systems count. That date is well before the date of any orders we expect to store in the database, so we would always know that this special date means not yet shipped.

However, having special values scattered in tables shows poor design and is rather error-prone. For example, if a new programmer starts on the project and doesn't realize there is a special date, the programmer might try calculating the average time between the order and shipping date, and come up with some very strange answers if there are a few shipped dates set before the order was placed.

Fortunately, all relational database systems support a very special value called `NULL`, which usually means unknown at this time. Notice that it doesn't mean zero, or empty string, or anything that can be represented by the data type of the field. An unknown value is very different from zero or a blank string. Indeed, `NULL` is not really a value at all.

The concept of a `NULL` is often confusing to novice database users. (The Romans also had trouble with things that are not there, so there is no zero in Roman numerals.) In database terminology, `NULL` generally means a value is unknown, but it also has one or two additional and rather subtle variations on that meaning.

It's important to take care of `NULL`s, because they can pop up at odd times and cause you surprises, usually unpleasant ones. So in our `orderinfo` table, we could set `date shipped` to `NULL` before an order is shipped, where the meaning "unknown at this time" is exactly what we require.

There is another subtly different use for `NULL` (not so common), which is to mean "not relevant for this row." Suppose you were doing a survey of people and one of the questions was about the color of spectacles. For people who don't wear spectacles, this is clearly a nonsensical question. This is a case where `NULL` might be used in the column to record that the information is not relevant for this particular row.

One feature of `NULL` is that if you compare two `NULL`s, the answer is always unknown. This sometimes confuses people, but if you think about the meaning of `NULL` as unknown, it's perfectly logical that testing for equality on two unknowns gives the answer unknown. SQL has a special way of checking for `NULL`s, by asking `IS NULL`. This allows you to find and test `NULL` values if you need to do so. `IS NULL` is discussed further in Chapter 4.

`NULL` type values do behave in a slightly different way from more conventional values. Therefore, it is possible to specify when you design a table that some columns cannot hold `NULL` values. It is normally a good idea to specify the columns as `NOT NULL`, when you are sure that `NULL` should never be accepted, such as for primary key columns. Some database designers advocate an almost complete ban on `NULL`, but they do have their uses, so we normally advocate allowing `NULL` values on selected columns, where there is a genuine possibility that unknown values are required. `NOT NULL` is discussed further in Chapter 8.

## Reviewing the Sample Database

In this chapter, we have been designing, in a rather ad-hoc manner, a simple database, named `bpsimple`, to look after customers, orders, and items, such as might be used in a small shop (see Figure 2-18, earlier in this chapter). As the book progresses, we will be using this database to demonstrate SQL and other PostgreSQL features. We will also be discovering the limitations of our existing design, and looking at how it can be improved in some areas.

The simplified database we are using has many elements of what a real retail database might look like; however, it also has many simplifications. For example, an item might have a full description for the stock file, a short description that appears on the till when it is sold, and yet another description that appears on shelf edge labels. The address information we are storing for customers is very simplified. We cannot cope with long addresses, where there is a village name or a state. We also cannot handle overseas orders.

It is often more feasible to start with a reasonably solid base and expand, rather than try to cater to every possible requirement in your initial design. This database is adequate for our initial needs.

In the next chapter, we will look at installing PostgreSQL, creating the tables for our sample database, and populating them with some sample data.

## Summary

In this chapter, we considered how a single database table is much like a single spreadsheet, with four important differences:

- All items in a column must have the same type.
- The number of columns must be the same for all rows in a table.
- It must be possible to uniquely identify each row.
- There is no implied row order in a database table, as there would be in a spreadsheet.

We have seen how we can extend our database to multiple tables, which lets us manage many-to-one relationships in a simple way. We gave some informal rules of thumb to help you understand how a database design needs to be structured. We will come back to the subject of database design in a much more rigorous fashion in later chapters.

We have also seen how to work around many-to-many relationships that turn up in the real world, breaking them down into a pair of one-to-many relationships by adding an extra table.

Finally, we worked on extending our initial database design so we have a demonstration database design, or schema, to work with as the book progresses.

In the next chapter, we will see how to get the PostgreSQL up and running on various platforms.