THE EXPERT'S VOICE® IN OPEN SOURCE

Covers version 8

Beginning Databases with PostgreSQL

From Novice to Professional

Effectively manage and develop data-driven applications with the powerful PostgreSQL database server.

SECOND EDITION

Neil Matthew
and Richard Stones

Apress®

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

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

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

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, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail 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, or visit http://www.apress.com.

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

Contents at a Glance

Contents

About the Authors

NEIL MATTHEW has been interested in and has programmed computers since 1974. A mathematics graduate from the University of Nottingham, Neil is just plain keen on programming languages and likes to explore new ways of solving computing problems. He has written systems to program in BCPL, FP (Functional Programming), Lisp, Prolog, and a structured BASIC. He even wrote a 6502 microprocessor emulator to run BBC microcomputer programs on UNIX systems.

In terms of UNIX experience, Neil has used almost every flavor since the late 1970s, including BSD UNIX, AT&T System V, Sun Solaris, IBM AIX, and many others. Neil has been using Linux since August 1993, when he acquired a floppy disk distribution of Soft Landing (SLS) from Canada, with kernel version 0.99.11. He has used Linux-based computers for hacking C, C++, Icon, Prolog, Tcl, and Java, at home and at work. Most of Neil's home projects were originally developed using SCO UNIX, but they've all ported to Linux with little or no trouble. He says Linux is mush easier because it supports quite a lot of features from other systems, so that both BSD- and System V-targeted programs will generally compile with little or no change.

As the head of software and principal engineer at Camtec Electronics in the 1980s, Neil programmed in C and C++ for real-time embedded systems. Since then, he has worked on software development techniques and quality assurance. After a spell as a consultant with Scientific Generics, he is currently working as a systems architect with Celesio AG.

Neil is married to Christine and has two children, Alexandra and Adrian. He lives in a converted barn in Northamptonshire, England. His interests include solving puzzles by computer, music, science fiction, squash, mountain biking, and not doing it yourself.

RICK STONES started programming at school, more years ago than he cares to remember, on a 6502-powered BBC micro, which with the help of a few spare parts, continued to function for the next 15 years. He graduated from the University of Nottingham with a degree in Electronic Engineering, but decided software was more fun.

Over the years, he has worked for a variety of companies, from the very small, with just a dozen employees, to the very large, including the IT services giant EDS. Along the way, he has worked on a range of projects, from real-time communications to accounting systems, very large help desk systems, and more recently, as the technical authority on a large EPoS and retail central systems program.

A bit of a programming linguist, Rick has programmed in various assemblers, a rather neat proprietary telecommunications language called SL-1, some FORTRAN, Pascal, Perl, SQL, and smidgeons of Python and C++, as well as C. (Under duress, he even admits that he was once reasonably proficient in Visual Basic, but tries not to advertise this aberration.)

Rick lives in a village in Leicestershire, England, with his wife Ann, children Jennifer and Andrew, and two cats. Outside work, his main interest is classical music, especially early religious music, and he even does his best to find time for some piano practice. He is currently trying to learn to speak German.

About the Technical Reviewer

ROBERT TREAT is a long-time open-source user, developer, and advocate. He has worked with a number of projects, but his favorite is certainly PostgreSQL. His current involvement includes helping maintain the postgresql.org web sites, working on phpPgAdmin, and contributing to the PostgreSQL core whenever he can. He has contributed several articles to the PostgreSQL "techdocs" site, was a presenter at OSCon 2004, worked as the PHP Foundry Admin on sourceforge.net, and has been recognized as a Major Developer for his work within the PostgreSQL community. Outside the free software world, Robert enjoys spending time with his three children, Robert, Dylan, and Emma, and with his high school sweetheart-turned-wife, Amber.

Acknowledgments

We would like to thank the many people who helped to make this book possible.

Neil would like to thank his wife, Christine, for her understanding, and children Alex and Adrian for not complaining too loudly at dad spending so long in The Den writing.

Rick would like to thank his wife, Ann, and children, Jennifer and Andrew, for their very considerable patience during the evenings and weekends while dad was yet again "doing book work."

Special thanks must go to Robert Treat, our technical reviewer. We are indebted to him for his excellent, detailed reviewing of our work and the many helpful comments and suggestions he made.

We would also like to thank Jon Parise for writing the PHP chapter for us, and Meeraj and Gavin for their kind permission to reuse some earlier material.

We are grateful to the entire Apress team for providing a smooth road from writing to production. To Gary Cornell and Jason Gilmore for getting the project off the ground, Sofia Marchant for coping admirably with a project schedule that initially appeared to require time travel, Nancy Wright for the transfer of material from the first edition, Marilyn Smith for first-class copy editing, Katie Stence for production editing, and Jason (again) for his editor role. We've learned a lot more about how books get made, and this one is certainly a better book than it would have been without this team's efforts.

Thanks are also due to the PostgreSQL development team for creating such a strong database system, allowing us to cover a great deal of SQL with an open-source product.

We would also like to thank our employer, Celesio, for support during the production of both editions of this book.

Introduction

Welcome to *Beginning Databases with PostgreSQL*.

Early in our careers, we came to recognize the qualities of open-source software. Not only is it often completely free to use, but it can also be of extremely high quality. If you have a problem, you can examine the source code to see how it works. If you find a bug, you can fix it yourself or pass it on to someone else to fix it for you. We have been working with open-source software since 1978 or so, including using the wonderful GNU tools, including GNU Emacs and GCC. We started using Linux in 1993 and have been delighted to be able to create a complete, free computing environment using a Linux kernel and the GNU tools, together with the X Window System, to provide a graphical user interface. PostgreSQL fits beautifully with this, providing an exceptional database system that adheres to the same open-source principles. (For more on open source and the freedom it can bring, please visit `http://www.opensource.org`.)

Databases are remarkably useful things. Many people find a "desktop database" useful for small applications in the office and around the home. Many web sites are data-driven, with content being extracted from databases behind the web server. As databases are becoming ubiquitous, we feel that there is a need for a book that includes some database theory and teaches good practice.

We have written this book to be a general introduction to databases, with broad coverage of the range of capabilities that modern, relational database systems have and how to use them effectively. With PostgreSQL as their database system, no one has an excuse for not doing things "properly." It supports good database design, is resilient and scalable, and runs on just about every type of computer you can think of, including Linux, UNIX, Windows, Mac OS X, AIX, Solaris, and HP-UX.

Oh, in case you were wondering, PostgreSQL is pronounced "post-gres-cue-el" (not "post-gray-ess-cue-el").

The book is roughly divided into thirds. The first part covers getting started, both with databases in general (what they are and what they are useful for) and with PostgreSQL in particular (how to obtain it, install it, start it, and use it). If you follow along with the examples, by the end of Chapter 5, you will have built your first working database and be able to use several tools to do useful things with it, such as entering data and executing queries.

The second part of the book explores in some depth the heart of relational databases: the query language SQL. Through sample programs and "Try It Out" sections, you will learn many aspects of database programming, ranging from simple data insertions and updates, through powerful types of queries, to extending the database server functionality with stored procedures and triggers. A great deal of the material in this section is database-independent, so knowledge gained here will stand you in good stead if you need to develop with another type of database. Of course, all of the material is illustrated with examples using PostgreSQL and a sample database. Chapters on PostgreSQL system administration and good practice in database design complete this section.

The third part of the book concentrates on harnessing the power of PostgreSQL in your own programs. These chapters cover connecting to a database, executing queries, and dealing with the results using a wide range of programming languages. Whether you are developing a dynamic web site with PHP or Perl, an enterprise application in Java or C#, or a client program in C, you will find a chapter to help you.

This is the second edition of *Beginning Databases with PostgreSQL*; the first edition was published by Wrox Press in 2001. Since then, every chapter has been updated with material to cover the latest version of PostgreSQL, version 8. We have taken the opportunity in this edition to add a new chapter on accessing PostgreSQL from the C# language to complement revised chapters covering C, Perl, PHP, and Java.

CHAPTER 1

Introduction to PostgreSQL

This book is all about one of the most successful open-source software products of recent times, a relational database called PostgreSQL. PostgreSQL is finding an eager audience among database aficionados and open-source developers alike. Anyone who is creating an application with nontrivial amounts of data can benefit from using a database. PostgreSQL is an excellent implementation of a relational database, fully featured, open source, and free to use.

PostgreSQL can be used from just about any major programming language you care to name, including C, C++, Perl, Python, Java, Tcl, and PHP. It very closely follows the industry standard for query languages, SQL92, and is currently implementing features to increase compliance with the latest version of this standard, SQL:2003. PostgreSQL has also won several awards, including the Linux Journal Editor's Choice Award for Best Database three times (for the years 2000, 2003, and 2004) and the 2004 Linux New Media Award for Best Database System.

We are perhaps getting a little ahead of ourselves here. You may be wondering what exactly PostgreSQL is, and why you might want to use it.

In this chapter, we will set the scene for the rest of the book and provide some background information about databases in general, the different types of databases, why they are useful, and where PostgreSQL fits into this picture.

Programming with Data

Nearly all nontrivial computer applications manipulate large amounts of data, and a lot of applications are written primarily to deal with data rather than perform calculations. Some writers estimate that 80% of all application development in the world today is connected in some way to complex data stored in a database, so databases are a very important foundation to many applications.

Resources for programming with data abound. Most good programming books will contain chapters on creating, storing, and manipulating data. Three of our previous books (published by Wrox Press) contain information about programming with data:

- *Beginning Linux Programming, Third Edition* (ISBN 0-7645-4497-7) covers the DBM library and the MySQL database system.
- *Professional Linux Programming* (ISBN 1-861003-01-3) contains chapters on the PostgreSQL and MySQL database systems.
- *Beginning Databases with MySQL* (ISBN 1-861006-92-6) covers the MySQL database system.

Constant Data

Data comes in all shapes and sizes, and the ways that we deal with it will vary according to the nature of the data. In some cases, the data is simple—perhaps a single number such as the value of π that might be built into a program that draws circles. The application itself may have this as a hard-coded value for the ratio of the circumference of a circle to its diameter. We call this kind of data *constant*, as it will never need to change.

Another example of constant data is the exchange rates used for the currencies of some European countries. In so-called "Euro Land," the countries that are participating in the single European currency (euro) fixed the exchange rates between their national currencies to six decimal places. Suppose we developed a Euro Land currency converter application. It could have a hard-coded table of currency names and base exchange rates, the numbers of national units to the euro. These rates will never change. We are not quite finished though, as it is possible for this table of currencies to grow. As countries sign up for the euro, their national currency exchange rate is fixed, and they will need to be added to the table. When that happens, the currency converter needs to be changed, its built-in table changed, and the application rebuilt. This will need to be done every time the currency table changes.

A better method would be to have the application read a file containing some simple currency data, perhaps including the name of the currency, its international symbol, and exchange rate. Then we can just alter the file when the table needs to change, and leave the application alone.

The data file that we use has no special structure; it's just some lines of text that mean something to the particular application that reads it. It has no inherent structure. Therefore we call it a *flat file*. Here's what our currency file might look like:

```
France      FRF   6.559570
Germany     DEM   1.955830
Italy       ITL   1936.270020
Belgium     BEF   40.339901
```

Flat Files for Data Storage

Flat files are extremely useful for many application types. As long as the size of the file remains manageable, so that we can easily make changes, a flat file scheme may be sufficient for our needs.

Many systems and applications, particularly on UNIX platforms, use flat files for their data storage or data interchange. An example is the UNIX password file, which typically has lines that look like this:

```
neil:*:500:100:Neil Matthew:/home/neil:/bin/bash
nick:*:501:100:Rick Stones:/home/rick:/bin/bash
```

These examples consist of a number of elements of information, or *attributes*, together making up a *record*. The file is arranged so that each line represents a single record, and the whole file acts to keep the related records together. Sometimes this scheme is not quite good enough, however, and we need to add extra features to support the job the application must do.

Repeating Groups and Other Problems

Suppose that we decide to extend the currency exchange rate application (introduced earlier in the chapter) to record the language spoken in each country, together with its population and area. In a flat file, we essentially have one record per line, each record made up of several attributes. Each individual attribute in a record is always in the same place; for example, the currency symbol is always the second attribute. So, we could think of looking at the data by columns, where a column is always the same type of information.

To add the language spoken in a particular country, we might think that we just need to add a new column to each of our lines. We hit a snag with this as soon as we realize that some countries have more than one official language. So, in our record for Belgium, we would need to include both Flemish and French. For Switzerland, we would need to add four languages. The flat file would now look something like this:

```
France      FRF  6.559570     French    60424213  547030
Germany     DEM  1.955830     German    82424609  357021
Italy       ITL  1936.270020  Italian   58057477  301230
Belgium     BEF  40.339901    Flemish   French    10348276  30528
Switzerland CHF  1.5255       German    French    Italian   Romansch  7450867  41290
```

This problem is known as *repeating groups*. We have the situation where a perfectly valid item (language) can be repeated in a record, so not only does the record (row) repeat, but the data in that row repeats as well. Flat files do not cope with this, as it is impossible to determine where the languages stop and the rest of the record starts. The only way around this is to add some structure to the file, and then it would not be a flat file anymore.

The repeating groups problem is very common and is the issue that really started the drive toward more sophisticated database management systems. We can attempt to resolve this problem by using ordinary text files with a little more structure. These are still often referred to as flat files, but they are probably better described as structured text files.

Here's another example. An application that stores the details of DVDs might need to record the year of production, director, genre, and cast list. We could design a file that looks a little like a Windows `.ini` file to store this information, like this:

```
[2001: A Space Odyssey]
year=1968
director=Stanley Kubrick
genre=science fiction
starring=Keir Dullea
starring=Leonard Rossiter
…
[Toy Story]
…
```

We have solved the repeating groups problem by introducing some tags to indicate the type of each element in the record. However, now our application must read and interpret a more complex file just to get its data. Updating a record and searching in this kind of structure can be quite difficult. How can we make sure that the descriptions for genre or classification are chosen from a specific subset? How can we easily produce a sorted list of Kubrick-directed films?

As data requirements become increasingly complex, we are forced to write more and more application code for reading and storing our data. If we extend our DVD application to include information useful to a DVD rental store owner—such as membership details, rentals, returns, and reservations—the prospect of maintaining all of that information in flat files becomes very unappealing.

Another common problem is simply that of size. Although the structured text file could be scanned by brute force to answer complex queries such as, "Tell me the addresses of all my members who have rented more than one comedy movie in the last three months," not only will it be very difficult to code, but the performance will be dire. This is because the application has no choice but to process the whole file to look for any piece of information, even if the question relates to just a single entry, such as "Who starred in *2001: A Space Odyssey*?"

What we need is a general-purpose way of storing and retrieving data, not a solution invented many times to fit slightly different, but very similar, problems as in a generic data-handling system.

What we need is a database and a database management system.

What Is a Database Management System?

The Merriam-Webster online dictionary (`http://www.merriam-webster.com`) defines a *database* as a usually large collection of data organized especially for rapid search and retrieval (as by a computer).

A *database management system* (DBMS) is usually a suite of libraries, applications, and utilities that relieve an application developer from the burden of worrying about the details of storing and managing data. It also provides facilities for searching and updating records. DBMSs come in a number of flavors developed over the years to solve particular kinds of data-storage problems.

Database Models

During the 1960s and 1970s, developers created databases that solved the repeating groups problem in several different ways. These methods result in what are termed *models* for database systems. Research performed at IBM provided much of the basis for these models, which are still in use today.

A main driver in early database system designs was efficiency. One of the common ways to make systems more efficient was to enforce a fixed length for database records, or at least have a fixed number of elements per record (columns per row). This essentially avoids the repeating group problem. If you are a programmer in just about any procedural language, you will readily see that in this case, you can read each record of a database into a simple C structure. Real life is rarely that accommodating, so we need to find ways to deal with inconveniently structured data. Database systems designers did this by introducing different database types.

Hierarchical Database Model

The IMS database system from IBM in the late 1960s introduced the hierarchical model for databases. In this model, considering data records to be composed of collections of others solves the repeating groups problem.

The model can be compared to a bill of materials used to describe how a complex manufactured product is composed. For example, let's say a car is composed of a chassis, a body, an engine, and four wheels. Each of these major components is broken down further. An engine comprises some cylinders, a cylinder head, and a crankshaft. These components are broken down further until we get to the nuts and bolts that make up every part in an automobile.

Hierarchical model databases are still in use today, including Software AG's ADABAS. A hierarchical database system is able to optimize the data storage to make it more efficient for particular questions; for example, to determine which automobile uses a particular part.

Network Database Model

The network model introduces the idea of pointers within the database. Records can contain references to other records. So, for example, you could keep a record for each of your company's customers. Each customer has placed many orders with you over time (a repeating group). The data is arranged so that the customer record contains a pointer to just one order record. Each order record contains both the order data for that specific order and a pointer to another order record.

Returning to our currency application, we might end up with record structures that look a little like those shown in Figure 1-1.

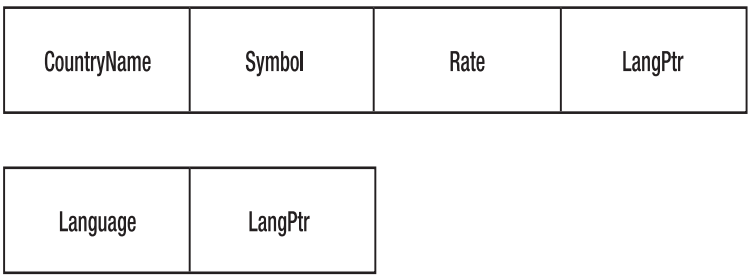


Figure 1-1. *Currency application record types*

Once the data is loaded, we end up with a linked (hence, the name *network* model) list used for the languages, as shown in Figure 1-2. The two different record types shown here would be stored separately, each in its own table.

Of course, to be more efficient in terms of storage, the actual database would not repeat the language names over and over again, but would probably contain a third table of language names, together with an identifier (often a small integer) that would be used to refer to the language name table entry in the other record types. This is called a *key*.

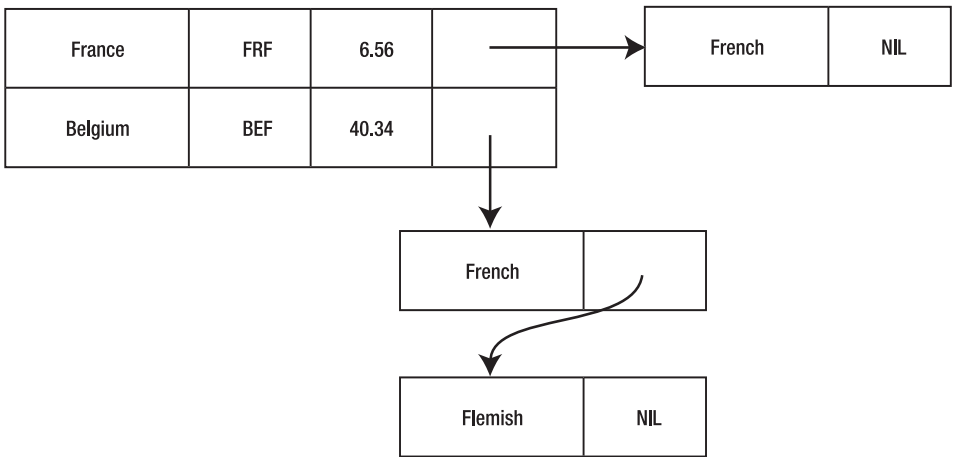


Figure 1-2. *Currency application data structure*

A network model database has some strong advantages. If you need to discover all of the records of one type that are related to a specific record of another type (in this example, the languages spoken in a country), you can find them extremely quickly by following the pointers from the starting record.

There are, however, some disadvantages, too. If you want to list the countries that speak French, you need to follow the links from all of the country records, which for large databases will be extremely slow. This can be fixed by having other linked lists of pointers specifically for languages, but it rapidly becomes very complex and is clearly not a general-purpose solution, since you need to decide in advance how the pointers will be designed. Writing applications that use a network model database can also be very tiresome, as the application typically must take responsibility for setting up and maintaining the pointers as records are updated and deleted.

Relational Database Model

The theory of DBMSs took a gigantic leap forward in 1970 with the publication of "A Relational Model of Data for Large Shared Data Banks," a paper by E. F. Codd (see `http://www.acm.org/classics/nov95/toc.html`). This revolutionary paper introduced the idea of relations and showed how tables could be used to represent facts that relate to real-world objects, and therefore, hold data about them.

By this time, it had also become clear that the initial driving force behind database design, efficiency, was often less important than another concern: data integrity. The relational model emphasizes data integrity much more than either of the earlier models. *Referential integrity* refers to making sure that data in the database makes sense at all times, so that, for example, all orders have customers. (We will have much more to say about integrity in Chapter 12, when we cover database design.)

Records in a table in a relational database are known as *tuples*, and this is the terminology you will see used in some parts of the PostgreSQL documentation. A tuple is an ordered group of components, or attributes, each of which has a defined type.

Several important rules define a relational database management system (RDBMS). All tuples must follow the same pattern, in that they all have the same number and types of components. Here is an example of a set of tuples:

```
{"France", "FRF", 6.56}
{"Belgium", "BEF", 40.34}
```

Each of these tuples has three attributes: a country name (string), a currency (string), and an exchange rate (a floating-point number). In a relational database, all records that are added to this set, or table, must follow the same form, so the following are disallowed:

```
{"Germany", "DEM"}
```

This has too few attributes.

```
{"Switzerland", "CHF", "French", "German", "Italian", "Romansch"}
```

This has too many attributes.

```
{1936.27, "ITL", "Italy"}
```

This has incorrect attribute types (wrong order).

Furthermore, in any table of tuples, there should be no duplicates. This means that in any table in a properly designed relational database, there cannot be any identical rows or records. This might seem to be a rather draconian restriction. For example, in a system that records orders placed by customers, it would appear to disallow the same customer from ordering the same product twice. In the next chapter, we will see that there is an easy way to work around this requirement, by adding an attribute.

Each attribute in a record must be *atomic*; that is, it must be a single piece of data, not another record or a list of other attributes. Also, the type of corresponding attributes in every record in the table must be the same. Technically, this means that they must be drawn from the same set of values or domain. In practical terms, it means they will all be a string, an integer, a floating-point value, or some other type supported by the database system.

The attribute (or attributes) used to distinguish a particular record in a table from all the other records in a table is called a *primary key*. In a relational database, each relation, or table, must have a primary key for each record to make it unique—different from all the others in that table.

One last rule that determines the structure of a relational database is referential integrity. As we noted earlier, this is a desire that all of the records in the database make sense at all times. Database application programmers must be careful to make sure that their code does not break the integrity of the database. Consider what happens when we delete a customer. If we try to remove the customer from the `customer` relation, we also need to delete all of his orders from the `orders` table. Otherwise, we will be left with records about orders that have no valid customer.

We will see much more on the theory and practice of relational databases in later chapters. For now, it is enough to know that the relational model for databases is based on some mathematical concepts of sets and relations, and that there are some rules that need to be observed by systems that are based on this model.

Query Languages

RDBMSs offer ways to add and update data, of course, but their real power stems from their ability to allow users to ask questions about the data stored, in the form of *queries*. Unlike many earlier database designs, which were often structured around the type of question that the data needed to answer, relational databases are much more flexible at answering questions that were not known at the time the database was designed.

Codd's proposals for the relational model use the fact that relations define sets, and sets can be manipulated mathematically. He suggested that queries might use a branch of theoretical logic called the predicate calculus, and that query languages would use this as their base. This would bring unprecedented power for searching and selecting data sets. Modern database systems, including PostgreSQL, hide all the mathematics behind an expressive and easy-to-learn *query language*.

One of the first implementations of a query language was QUEL, used in the Ingres database developed in the late 1970s. Another query language that takes a different approach is QBE (Query By Example). At around the same time a team at IBM's research center developed SQL (Structured Query Language), usually pronounced "sequel."

SQL Standards and Variations

SQL has become very widely adopted as a standard for database query languages and is defined in a series of international standards. The most commonly used definition is ISO/IEC 9075:1992, "Database Language SQL." This is more simply referred to as SQL92. These standards replaced an earlier standard, SQL89. The latest version of the SQL standard is ISO/IEC 9075:2003, more simply referred to as SQL:2003.

At present, most RDBMSs comply with the SQL92 version of the standard, or sometimes ANSI X3.135-1992, which is an identical United States standard differing only in some cover pages. There are three levels of conformance to SQL92: Entry SQL, Intermediate SQL, and Full SQL. By far, the most common conformance level is Entry SQL.

■ **Note** PostgreSQL is very close to SQL92: Entry SQL conformance, with only a few slight differences. The developers keep a close eye on standards compliance, and PostgreSQL becomes more compliant with each release.

Today, just about every useful database system supports SQL to some extent. In theory, SQL acts as a good unifier, since database applications written to use SQL as the interface to the database can be ported to other database systems with little cost in terms of time and effort. Commercial pressures however, dictate that database manufacturers distinguish their products one from another. This has led to SQL variations, not helped by the fact that the standard for SQL does not define commands for many of the database administration tasks that are an essential part of using a database in the real world. So, there are differences between the SQL used by Oracle, SQL Server, PostgreSQL, and other database systems.

SQL Command Types

The SQL language comprises three types of commands:

- **Data Manipulation Language (DML):** This is the part of SQL that you will use 90% of the time. It is made up of the commands for inserting, deleting, updating, and selecting data from the database.
- **Data Definition Language (DDL):** These are the commands for creating tables, defining relationships, and controlling other aspects of the database that are more structural than data related.
- **Data Control Language (DCL):** This is a set of commands that generally control permissions on the data, such as defining access rights. Many database users will never use these commands, because they work in larger company environments where one or more database administrators are employed specifically to manage the database, and usually one of their roles is to control permissions.

A Brief Introduction to SQL

You will see a lot of SQL in this book. Here, we will take a brief look at some examples as an introduction. We will see that we do not need to worry about the formal basis of SQL to be able to use it.

Here is some SQL for creating a new table in a database. This example creates a table for customers:

```
CREATE TABLE customer
(
    customer_id    serial,
    title          char(4),
    fname          varchar(32),
    lname          varchar(32) not null,
    addressline    varchar(64),
    town           varchar(32),
    zipcode        char(10) not null,
    phone          varchar(16),
);
```

We state that the table requires an identifier, which will act as a primary key, and that this is to be generated automatically by the database system. It has type `serial`, which means that every time a `customer` is added, a new, unique `customer_id` will be created in sequence. The customer `title` is a text attribute of four characters, and `zipcode` has ten characters. The other attributes are variable-length strings up to a defined maximum length, some of which must be present (those marked `not null`).

Next, we have some SQL statements that can be used to populate the table we have just created. These are very straightforward:

```
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
    VALUES('Mr','Neil','Matthew','5 Pasture Lane','Nicetown','NT3 7RT','267 1232');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
    VALUES('Mr','Richard','Stones','34 Holly Way','Bingham','BG4 2WE','342 5982');
```

The heart of SQL is the SELECT statement. It is used to create result sets that are groups of records (or attributes from records) that match a particular set of criteria. The criteria can be quite complex if required. These result sets can then be used as the targets for changes with an UPDATE statement or deleted with a DELETE statement.

Here are some examples of SELECT statements:

```
SELECT * FROM customer

SELECT * FROM customer, orderinfo
    WHERE orderinfo.customer_id = customer.customer_id GROUP BY customer_id

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
```

These SELECT statements list all the customers, all the customer orders, and count the orders each customer has made, respectively. We will see the results of these SQL statements in Chapter 2, and learn much more about SELECT in Chapter 4.

■ **Note** SQL command keywords such as SELECT and INSERT are case-insensitive, so they can be written in either uppercase or lowercase. In this book, we have used uppercase to aid readability.

As you read through this book, we will be teaching you SQL, so by the time you get to the end, you will be comfortable with a wide range of SQL statements and how to use them.

Database Management System Responsibilities

As we stated earlier, a DBMS is a suite of programs that allow the construction of databases and applications that use them. The responsibilities of a DBMS include the following:

- **Creating the database:** Some systems will manage one large file and create one or more databases inside it; others may use many operating system files or use a raw disk partition directly. Users need not worry about the low-level structure of these files, as the DBMS provides all of the access developers and users need.
- **Providing query and update facilities:** A DBMS will have a method of asking for data that matches certain criteria, such as all orders made by a particular customer that have not yet been delivered. Before the widespread introduction of the SQL standard, the way that queries like this were expressed varied from system to system.

- **Multitasking:** If a database is used in several applications, or is accessed concurrently by several users at the same time, the DBMS will make sure that each user's request is processed without impacting the others. This means that users need to wait in line only if someone else is writing to the precise item of data that they wish to read (or write). It is possible to have many simultaneous reads of data going on at the same time. In practice, different database systems support different degrees of multitasking, and may even have configurable levels, as we will see in Chapter 9.
- **Maintaining an audit trail:** A DBMS will keep a log of all the changes to the data for a period of time. This can be used to investigate errors, but perhaps even more important, can be used to reconstruct data in the event of a fault in the system, perhaps an unscheduled power down. A data backup and an audit trail of transactions can be used to completely restore the database in case of disk failure.
- **Managing the security of the database**: A DBMS will provide access controls so that only authorized users can manipulate the data held in the database and the structure of the database itself (the attributes, tables, and indices). Typically, there will be a hierarchy of users defined for any particular database, from a superuser who can change anything, through users with permission to add or delete data, down to users who can only read data. The DBMS will have facilities to add and delete users, and specify which features of the database system they are able to use.
- **Maintaining referential integrity:** Many database systems provide features that help to maintain referential integrity—the correctness of the data, as mentioned earlier. They will report an error when a query or update would break the relational model rules.

What Is PostgreSQL?

Now we are in a position to say what PostgreSQL actually is. It is a DBMS that incorporates the relational model for its databases and supports the SQL standard query language.

PostgreSQL also happens to be very capable and very reliable, and it has good performance characteristics. It runs on just about any UNIX platform, including UNIX-like systems, such as FreeBSD, Linux, and Mac OS X. It can also run on Microsoft Windows NT/2000/2003 servers, or even on Windows XP for development. And, as we mentioned at the beginning of this chapter, it's free and open source.

PostgreSQL can be compared favorably to other DBMSs. It contains just about all the features that you would find in other commercial or open-source databases, and a few extras that you might not find elsewhere.

PostgreSQL features (as listed in the PostgreSQL FAQ) include the following:

- Transactions
- Subselects
- Views
- Foreign key referential integrity
- Sophisticated locking

- User-defined types
- Inheritance
- Rules
- Multiple-version concurrency control

Since release 6.5, PostgreSQL has been very stable, with a large series of regression tests performed on each release to ensure its stability. The release of the 7.*x* series brought conformance to SQL92 closer than ever, and an irksome row-size restriction was removed.

The release of PostgreSQL that we used in this book, version 8, added several new features:

- Native Microsoft Windows version
- Table spaces
- Ability to alter column types
- Point-in-time recovery

PostgreSQL has proven to be very reliable in use. Each release is very carefully controlled, and beta releases are subject to at least a month's testing. With a large user community and universal access to the source code, bugs can get fixed very quickly.

The performance of PostgreSQL has been improving with each release, and the latest benchmarks show that, in some circumstances, it compares well with commercial products. Some less fully featured database systems will outperform it at the cost of lower overall functionality. Then again, for simple enough applications, so will a flat-file database!

A Short History of PostgreSQL

PostgreSQL can trace its family tree back to 1977 at the University of California at Berkeley (UCB). A relational database called Ingres was developed at UCB between 1977 and 1985. Ingres was a popular UCB export, making an appearance on many UNIX computers in the academic and research communities. To serve the commercial marketplace, the code for Ingres was taken by Relational Technologies/Ingres Corporation and became one of the first commercially available RDBMSs.

■ **Note** Today, Ingres has become CA-INGRES II, a product from Computer Associates. Interestingly, it has been recently released under an Open Source license.

Meanwhile, back at UCB, work on a relational database server called Postgres continued from 1986 to 1994. Again, this code was taken up by a commercial company and offered for sale as a product. This time it was Illustra, since swallowed up by Informix. Around 1994, SQL features were added to Postgres, and its name was changed to Postgres95.

By 1996, Postgres was becoming very popular, and the developers decided to open up its development to a mailing list, starting what has become a very successful collaboration of volunteers driving Postgres forward. At this time, Postgres underwent its final name change, ditching the dated "95" tag for a more appropriate "SQL," to reflect the support Postgres now has for the query language standard. PostgreSQL was born.

Today, a team of Internet developers develops PostgreSQL in much the same manner as other open-source software such as Perl, Apache, and PHP. Users have access to the source code and contribute fixes, enhancements, and suggestions for new features. The official PostgreSQL releases are made via `http://www.postgresql.org`.

Commercial support is available from several companies. See the list at `http://techdocs.postgresql.org/companies.php`.

The PostgreSQL Architecture

One of PostgreSQL's strengths derives from its architecture. In common with commercial database systems, PostgreSQL can be used in a client/server environment. This has many benefits for both users and developers.

The heart of a PostgreSQL installation is the database server process. It runs on a single server. Applications that need to access the data stored in the database are required to do so via the database process. These client programs cannot access the data directly, even if they are running on the same computer as the server process.

■ **Note** PostgreSQL does not yet have the high-availability features of a few enterprise-class commercial database systems that can spread the load across several servers, giving additional scalability and resilience. There are some PostgreSQL-sanctioned projects underway at `http://gborg.postgresql.org` that aim to add these features, and there are some commercial solutions available.

This separation into client and server allows applications to be distributed. You can use a network to separate your clients from your server and develop client applications in an environment that suits the users. For example, you might implement the database on UNIX and create client programs that run on Microsoft Windows. Figure 1-3 shows a typical distributed PostgreSQL application.

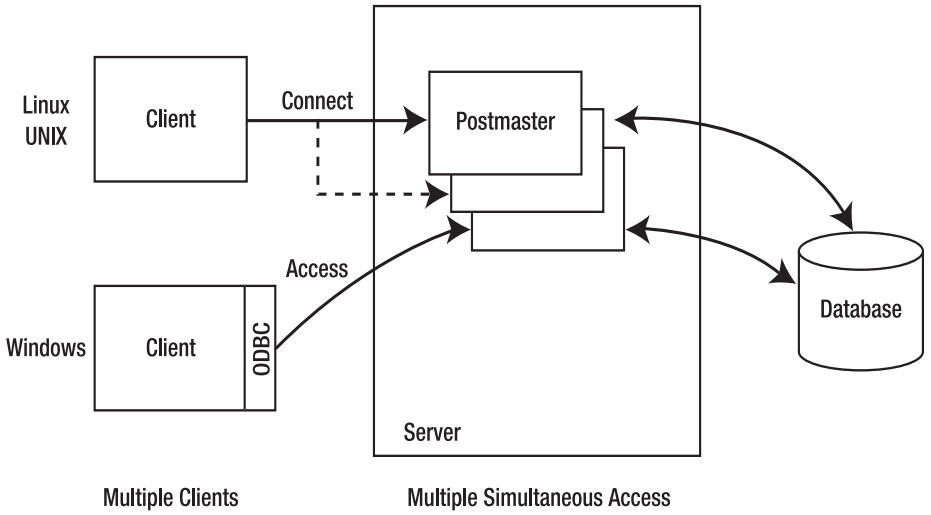


Figure 1-3. *PostgreSQL architecture*

In Figure 1-3, you can see several clients connecting to the server across a network. For PostgreSQL, this needs to be a TCP/IP network—a local area network (LAN) or possibly even the Internet. Each client connects to the main database server process (shown as `postmaster` in Figure 1-3), which creates a new server process specifically for servicing access requests for this client.

Concentrating the data handling in a server, rather than attempting to control many clients accessing the same data stored in a shared directory on a server, allows PostgreSQL to efficiently maintain the data's integrity, even with many simultaneous users.

The client programs connect using a message protocol specific to PostgreSQL. It is possible, however, to install software on the client that provides a standard interface for the application to work to, such as the Open Database Connectivity (ODBC) standard or the Java Database Connectivity (JDBC) standard used by Java programs. The availability of an ODBC driver allows many existing applications to use PostgreSQL as a database, including Microsoft Office products such as Excel and Access. You will see examples of different PostgreSQL connection methods in Chapters 3, 5, and 13 through 18.

The client/server architecture for PostgreSQL allows a division of labor. A server machine well suited to the storage and access of large amounts of data can be used as a secure data repository. Sophisticated graphical applications can be developed for the clients. Alternatively, a web-based front-end can be created to access the data and return results as web pages to a standard web browser, with no additional client software at all. We will return to these ideas in Chapters 5 and 15.

Data Access with PostgreSQL

With PostgreSQL, you can access your data in several ways:

- Use a command-line application to execute SQL statements. We will do this throughout the book.
- Embed SQL directly into your application (using embedded SQL). We will see how to do this for C applications in Chapter 14.
- Use function calls (APIs) to prepare and execute SQL statements, scan result sets, and perform updates from a large variety of different programming languages. Chapter 13 covers C language APIs for PostgreSQL.
- Access the data in a PostgreSQL database indirectly using a driver such as ODBC (see Chapter 3) or the JDBC standard (see Chapter 17), or by using a standard library such as Perl's DBI (see Chapter 16).

What Is Open Source?

As we start the twenty-first century, much is being made of open-source software, of which PostgreSQL is such a good example. But what does *open source* mean exactly?

The term *open source* has a very specific meaning when applied to software. It means that the software is supplied with the source code included. It does not necessarily mean that there are no conditions applied to the software's use. It is still licensed in that you are given permission to use the software in certain ways.

An Open Source license will grant you permission to use the software, modify it, and redistribute it without paying license fees. This means that you may use PostgreSQL in your organization as you see fit.

If you have problems with open-source software, because you have the source code, you can either fix them yourself or give the code to someone else to try to fix. There are now many commercial companies offering support for open-source products, so that you do not have to feel neglected if you choose to use an open-source product.

There are many different variations on Open Source licenses, some more liberal than others. All of them adhere to the principle of source code availability and allowing redistribution.

The most liberal license is the Berkeley Software Distribution (BSD) license, which says in effect, "Do what you will with this software. There is no warranty." The license for PostgreSQL (`http://www.postgresql.org/about/licence`) echoes the BSD license sentiments and takes the form of a copyright statement that says, "Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies." The paragraphs that follow this statement disclaim liability and warranty.

Resources

There are many printed and online sources of further information about databases in general and about PostgreSQL.

For more on the theory of databases, check out the Database Theory section of David Frick's site at `http://www.frick-cpa.com/ss7/default.htm`.

The official PostgreSQL site is `http://www.postgreSQL.org`, where you can find more on the history of PostgreSQL, download copies of PostgreSQL, browse the official documentation, and much more besides (including learning how to pronounce PostgreSQL).

PostgreSQL is also the foundation of the former Red Hat Database, now known as PostgreSQL-Red Hat Edition. You can find more on this version of PostgreSQL and tools developed for it by Red Hat at `http://sources.redhat.com/rhdb/`.

For more information about open-source software and the principle of freedom in software, take a few moments to visit these two sites: `http://www.gnu.org` and `http://www.opensource.org`.

CHAPTER 2

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:

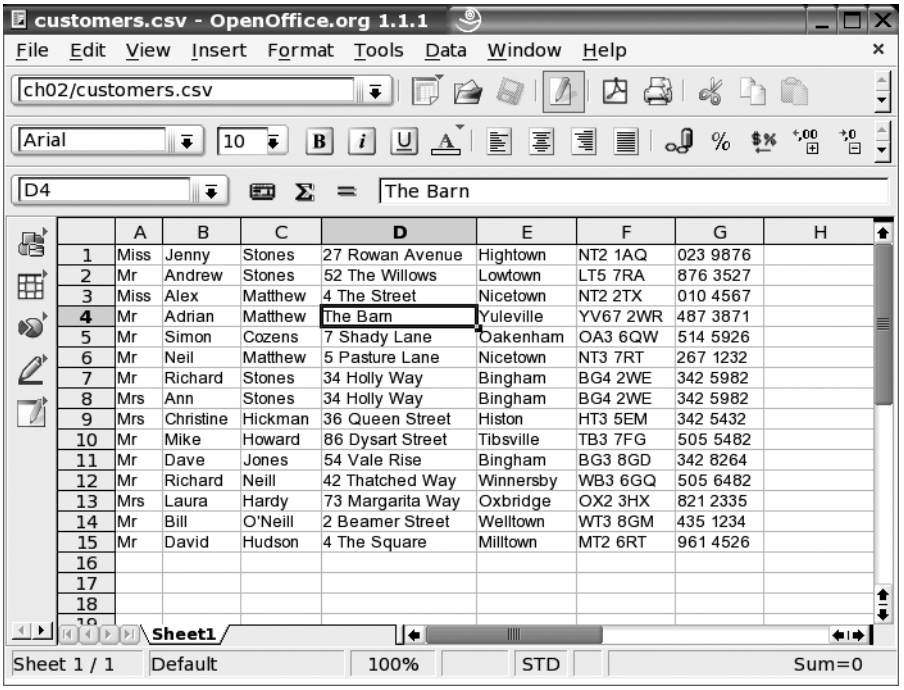
- Spreadsheets: their problems and limitations
- How databases store data
- How to access data in a database
- Basic database design, with multiple tables
- Relationships between tables
- Some basic data types
- The `NULL` token, used to indicate an unknown value

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.



	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	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	
16								
17								
18								

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*.

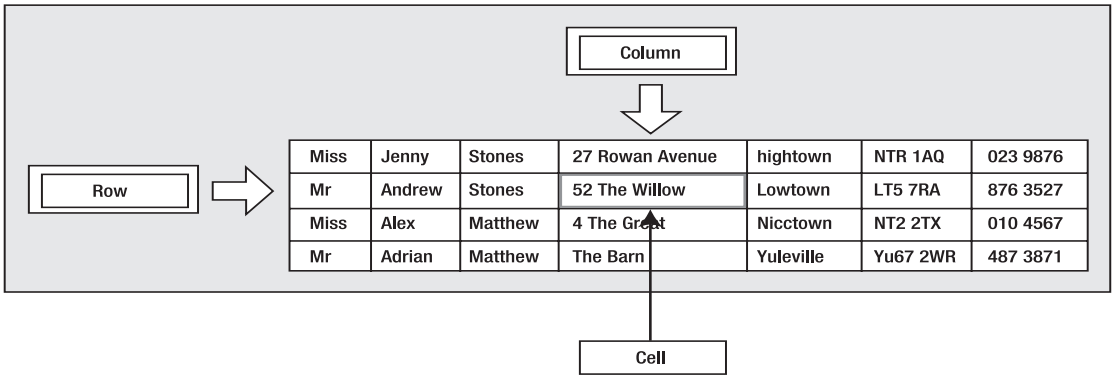


Figure 2-2. *Some spreadsheet terminology*

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:

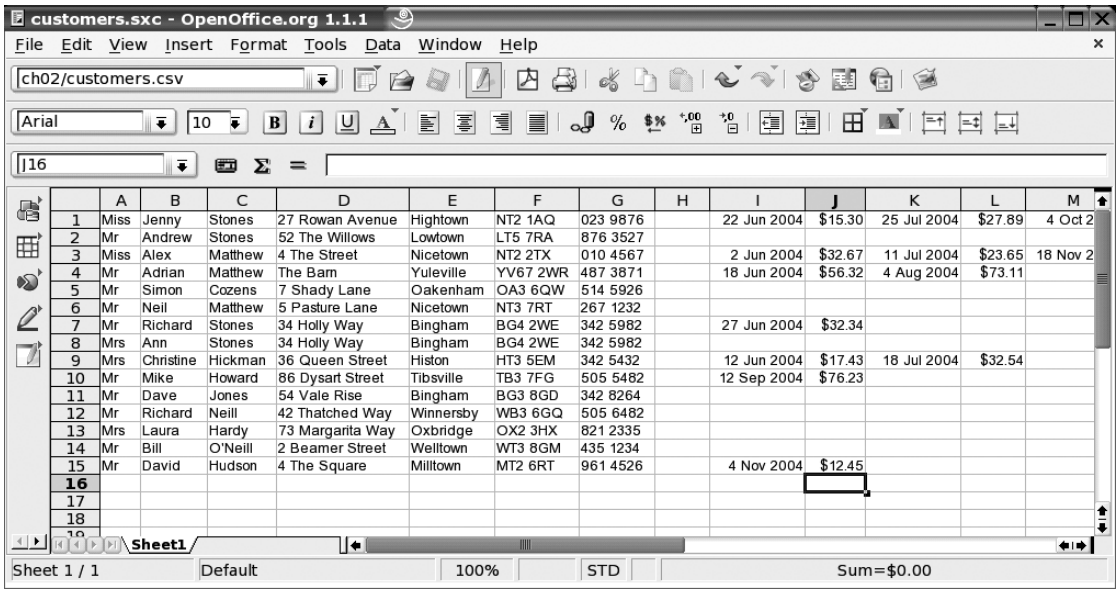
- Don't have too many customers
- Don't have many complex details for each customer
- Don't need to store any other repeating information, such as the various orders each customer has placed
- Don't want several people to be able to update the information simultaneously
- Do ensure the spreadsheet gets 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	Nicetown	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	Nicetown	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	Tibsville	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	Milltown	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.

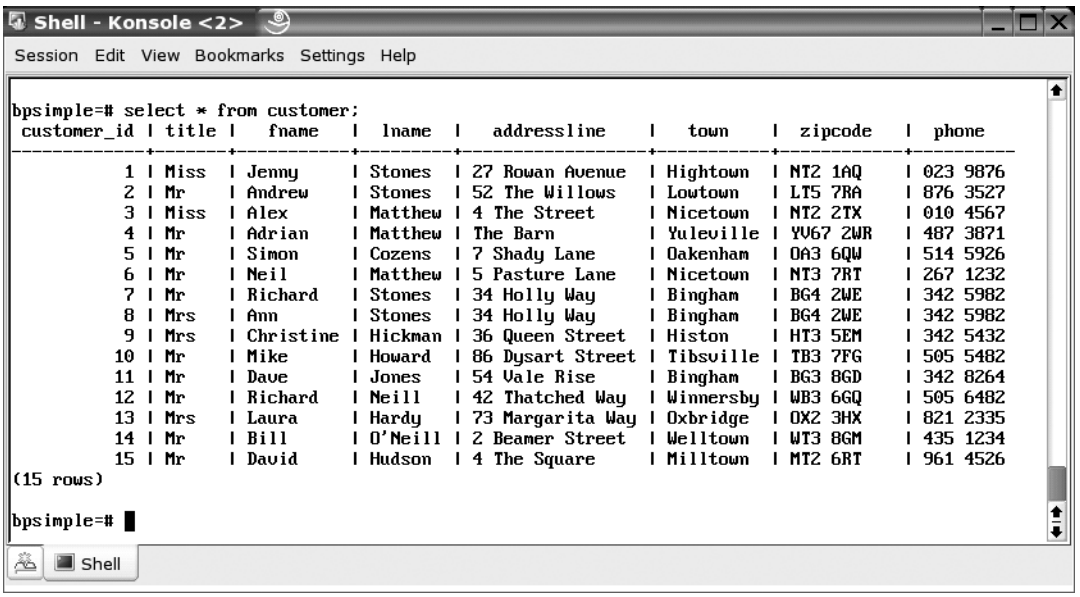


Figure 2-4. *Command-line viewing of customer data from a database*

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.

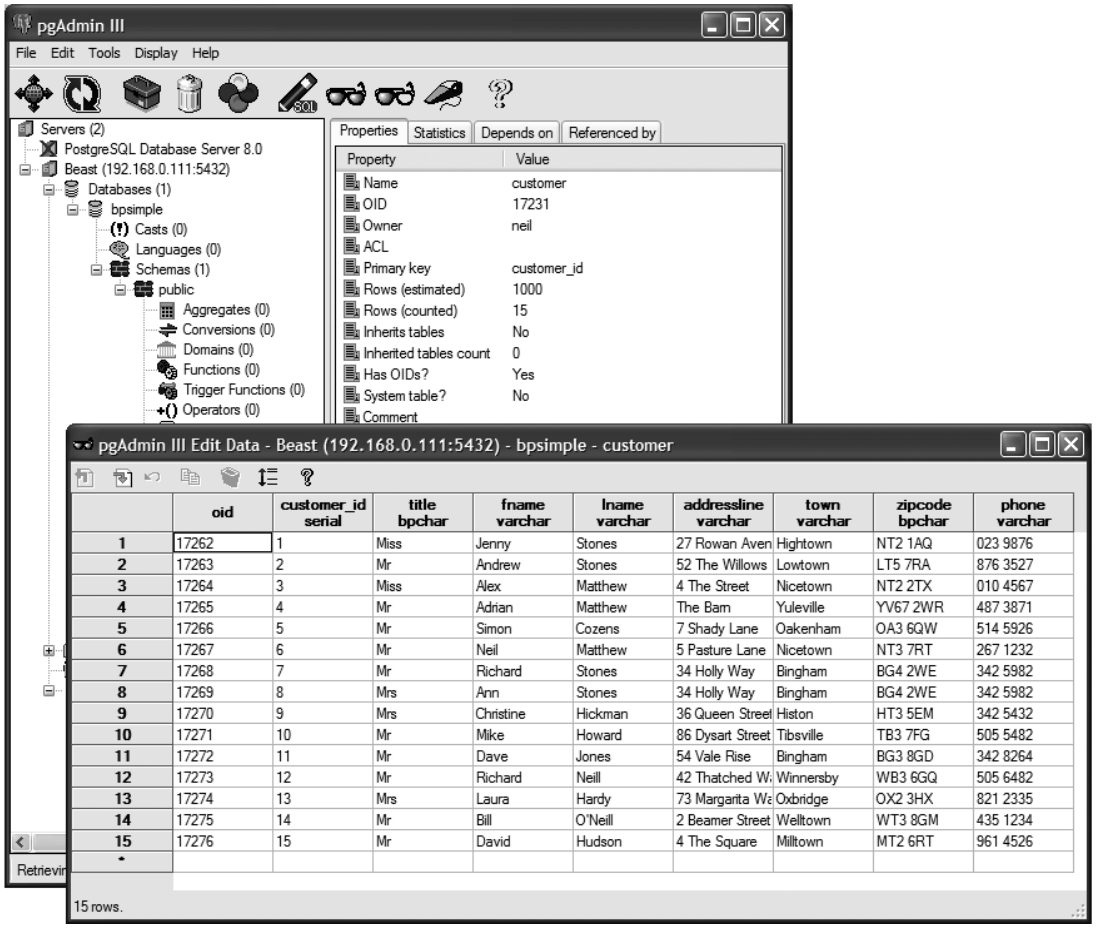
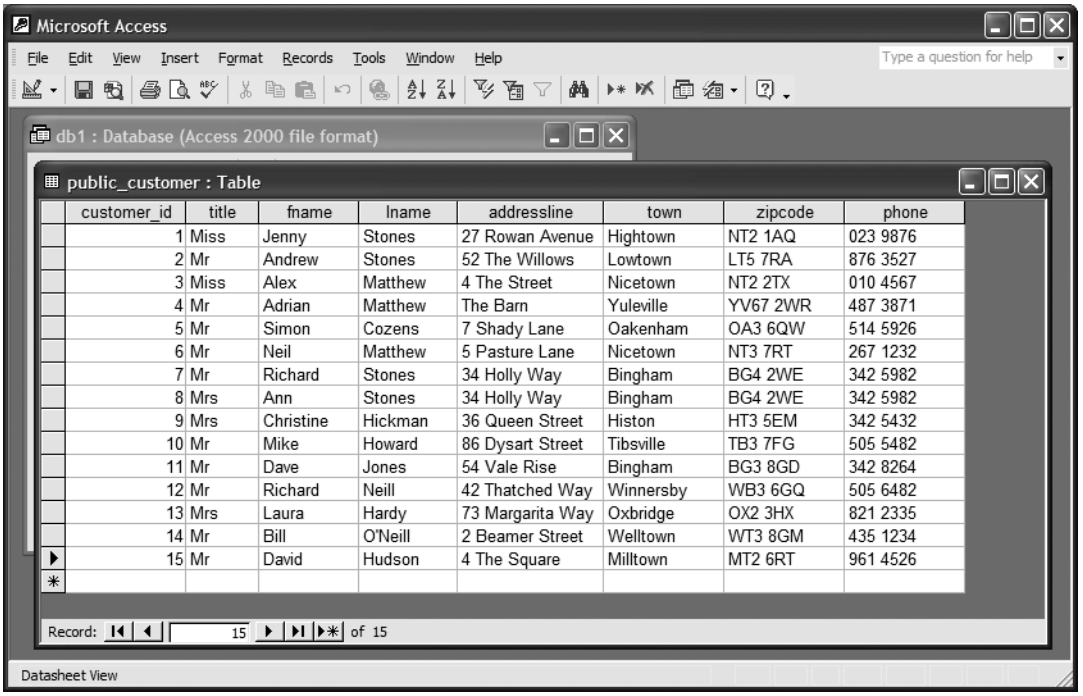


Figure 2-5. *Viewing customer data from a database with pgAdmin III*

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.



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	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	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

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.

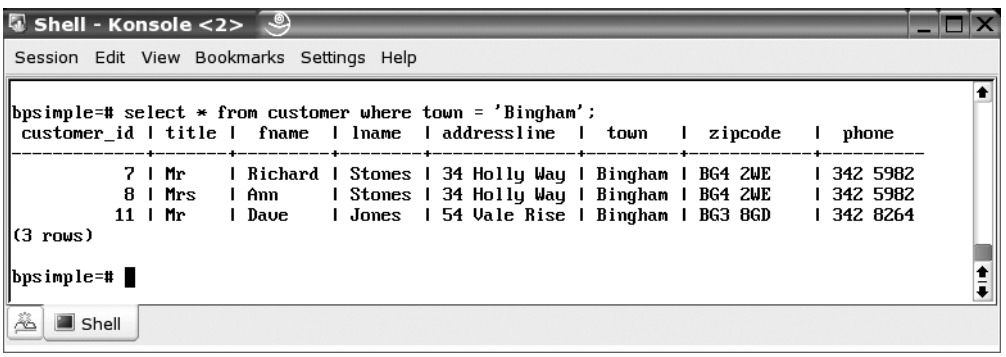


Figure 2-7. *Selecting a subset of the data rows*

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.

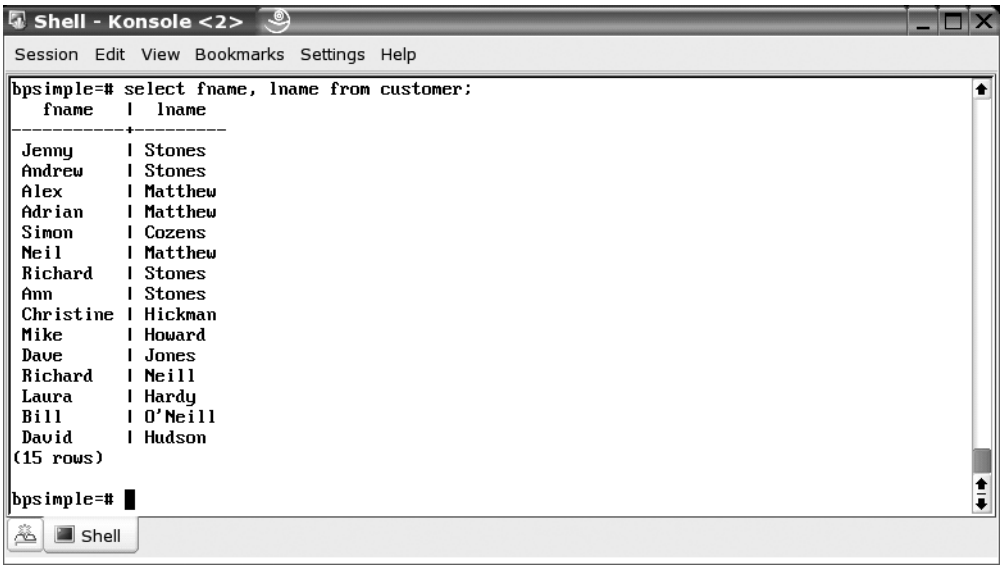


Figure 2-8. *Selecting a subset of the data columns*

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.

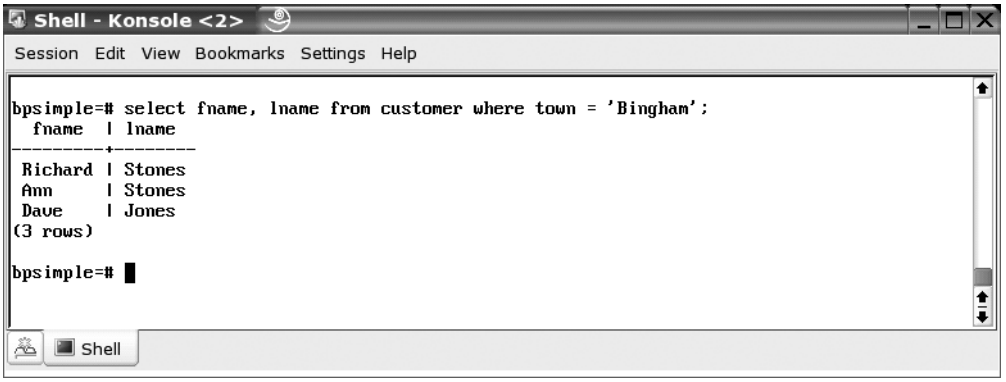


Figure 2-9. *Selecting a subset of both columns and rows*

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.

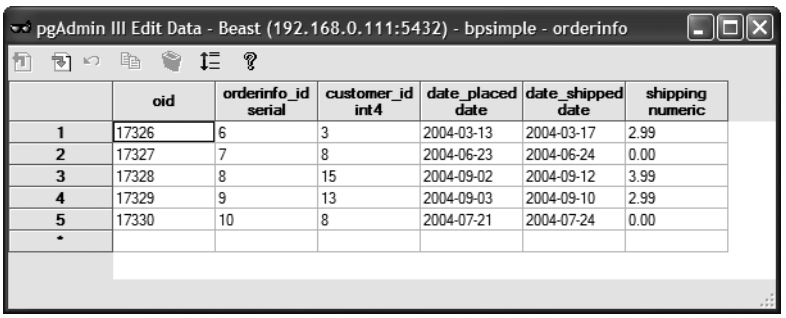
Using Multiple Tables

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.



	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.

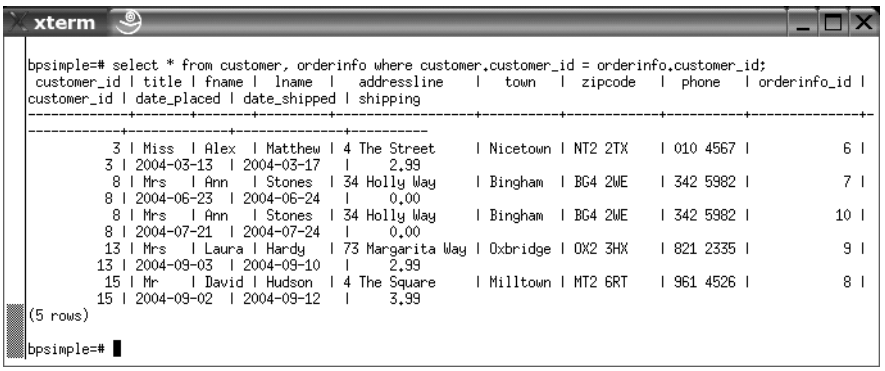


Figure 2-11. *Selecting data from two tables in one operation*

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.

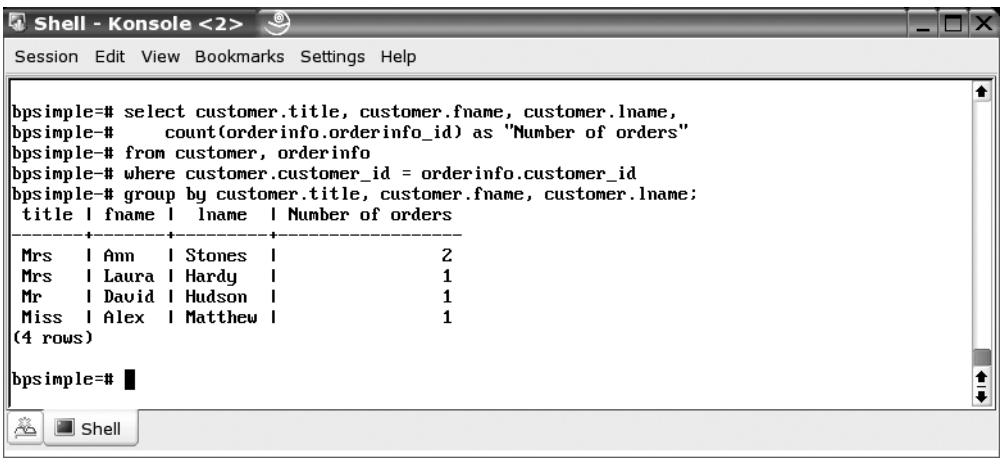


Figure 2-12. *Retrieving order frequency*

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.

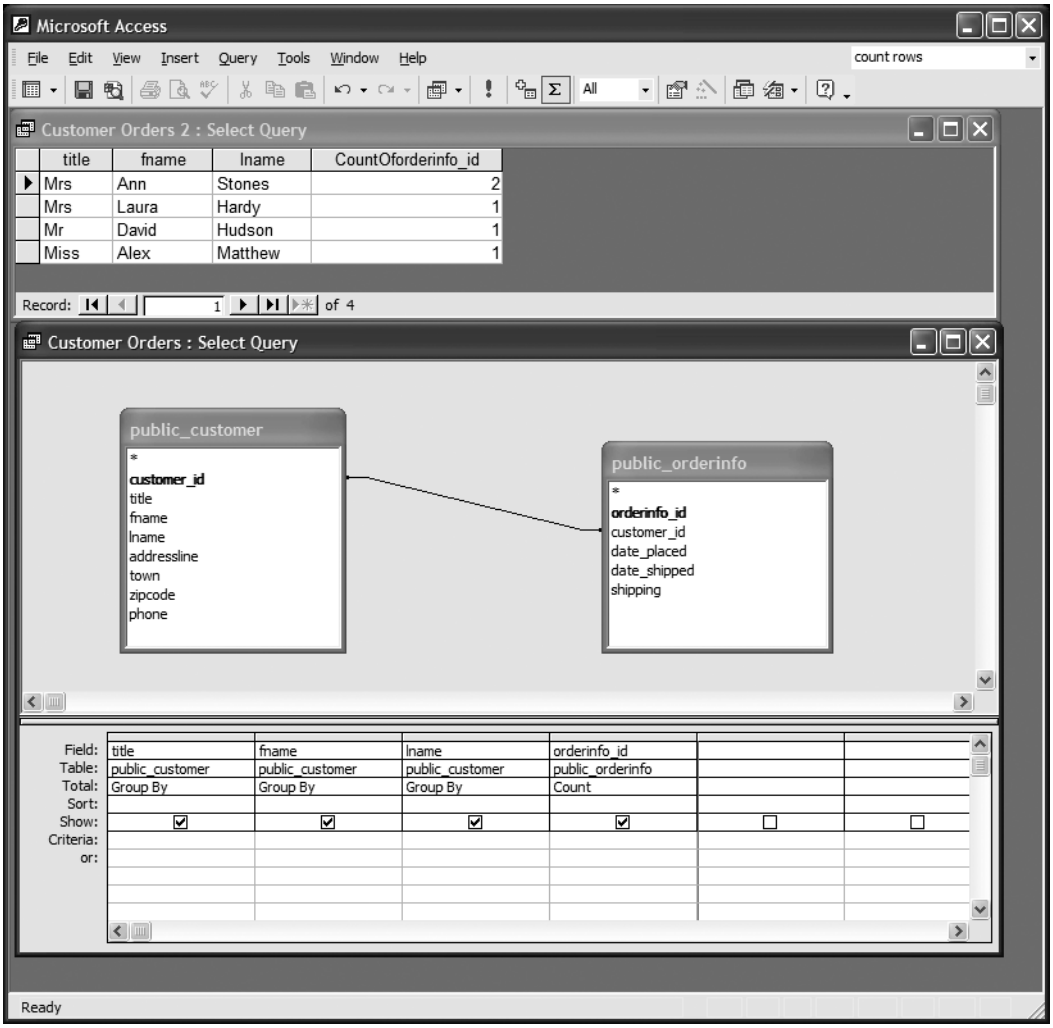


Figure 2-13. *Building a query graphically*

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.

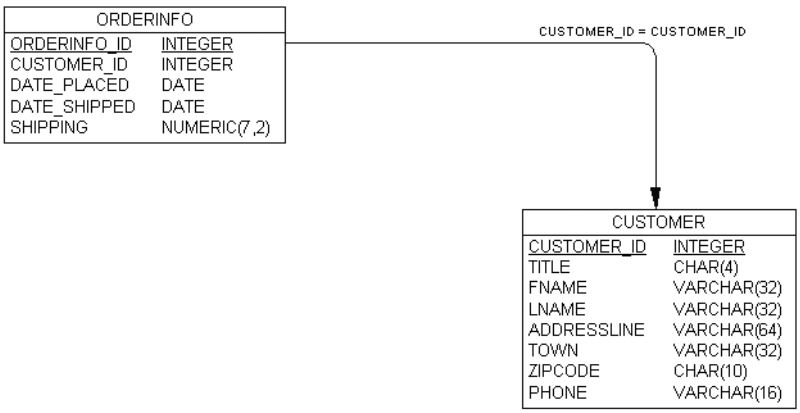


Figure 2-14. *A simple entity relationship diagram*

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.

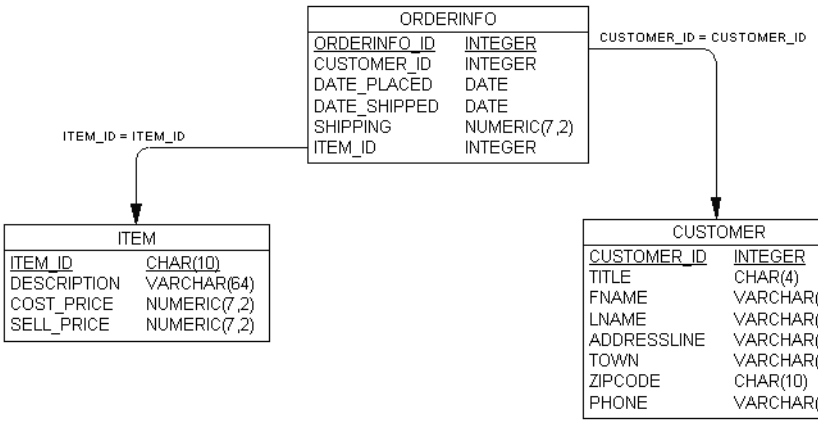


Figure 2-15. *An attempt at relating customers and ordered items*

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.

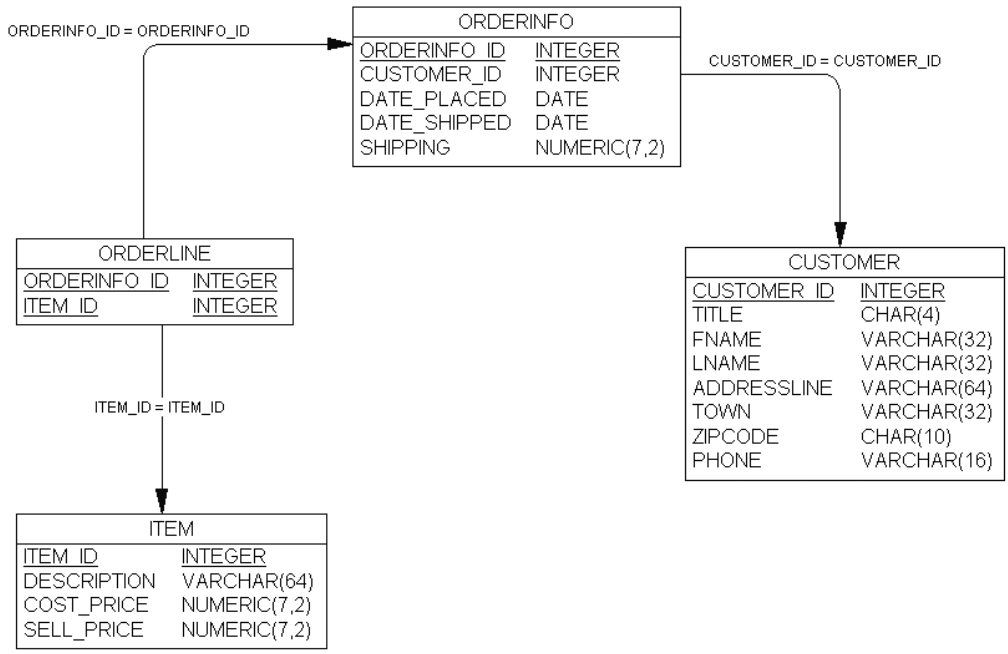


Figure 2-16. *Relating customers and orders*

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 `orderinfo_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 `orderinfo_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 *overfill 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.

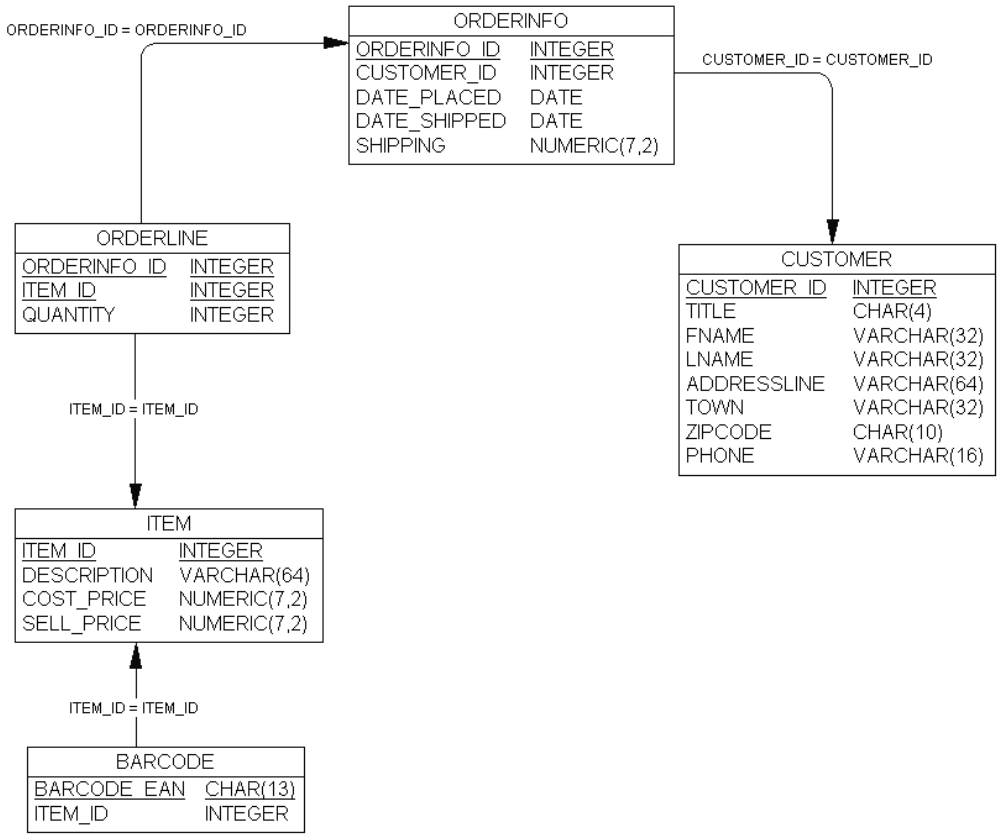


Figure 2-17. *Adding the barcode relationship*

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.

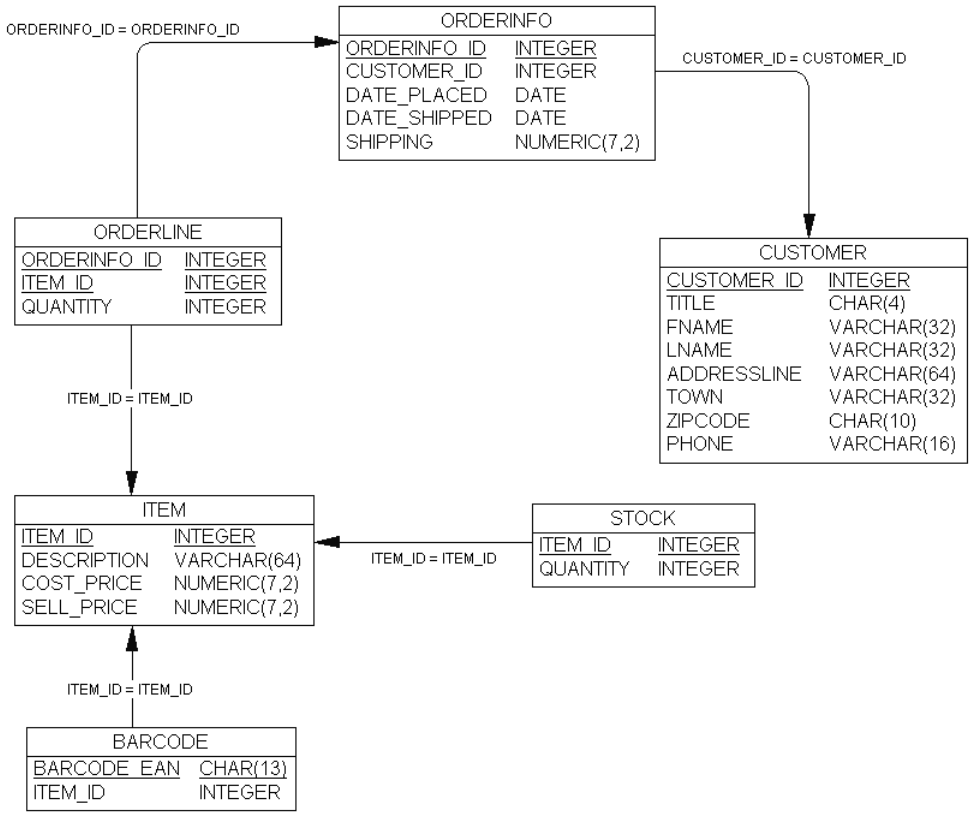


Figure 2-18. *The design of the* `bpsimple` *database*

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 `_id` columns. The figures in this chapter show such fields as `integer`, 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 `char(256)` 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 `varchar` 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, `numeric(7,2)` 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.

CHAPTER 3

Getting Started with PostgreSQL

In this chapter, we will look at installing and setting up PostgreSQL on various operating systems. If you need to install it on a Linux system, precompiled binary packages provide an easy route. If you are running a UNIX or UNIX-like system—such as Linux, FreeBSD, AIX, Solaris, HP-UX, or Mac OS X—it is not difficult to compile PostgreSQL from the source code.

We will also cover how to install and set up PostgreSQL on Windows platforms, using the Windows installer introduced in PostgreSQL version 8.0. Earlier versions can be installed on Windows, but this requires some additional software to create a UNIX-like environment. We therefore recommend version 8.0 or later for Windows systems.

Finally, we will prepare for the examples in the following chapters by creating the sample database discussed in Chapter 2.

In particular, this chapter will cover the following topics:

- Installing PostgreSQL from Linux binaries
- Installing PostgreSQL from the source code
- Setting up PostgreSQL on Linux and UNIX systems
- Installing and setting up PostgreSQL on Windows
- Creating a database with tables and adding data

Installing PostgreSQL on Linux and UNIX Systems

If you are running a Linux system installed from a recent distribution, you may already have PostgreSQL installed or available to you as an installable package on the operating system installation disks. If not, you can use RPM packages to install PostgreSQL on many Linux distributions or flavors. Additionally, you can build and install PostgreSQL from the source code on just about any UNIX-compatible system.

Installing PostgreSQL from Linux Binaries

Probably the easiest way to install PostgreSQL on Linux is by using precompiled binary packages. The binaries for PostgreSQL are available for download as RPM (RPM Package Manager, formerly Red Hat Package Manager) packages for various Linux distributions. At the time of writing this book, RPM packages are available at `http://www.postgresql.org/` for the following operating systems:

- Red Hat 9
- Red Hat Advanced Server 2.1
- Red Hat Enterprise Linux 3.0
- Fedora Core 1, 2 (including 64-bit), and 3

You can find binary packages at `http://www.rpmfind.net` for other Linux distributions, including the following:

- SuSE Linux 8.2 and 9.*x*
- Conectiva Linux
- Mandrake
- Yellow Dog PPC

■ **Note** Debian Linux users can install PostgreSQL using `apt-get`.

Table 3-1 lists the PostgreSQL binary packages. For a functional database and client installation, you need to download and install at least the base, libs, and server packages.

Table 3-1. *PostgreSQL Binary Packages*

Package	Description
`postgresql`	The base package including clients and utilities
`postgresql-libs`	Shared libraries required from clients
`postgresql-server`	Programs to create and run a server
`postgresql-contrib`	Contributed extensions

Table 3-1. *PostgreSQL Binary Packages (Continued)*

Package	Description
`postgresql-devel`	Header files and libraries for development
`postgresql-docs`	Documentation
`postgresql-jdbc`	Java database connectivity for PostgreSQL
`postgresql-odbc`	Open database connectivity for PostgreSQL
`postgresql-pl`	PostgreSQL server support for Perl
`postgresql-python`	PostgreSQL server support for Python
`postgresql-tcl`	PostgreSQL server support for Tcl
`postgresql-test`	PostgreSQL test suite

The exact filenames will have version numbers appended with the package. It is advisable to install a matching set of packages, all with the same revision level. In a package with the version number 8.*x.y*, the *x.y* portion determines the revision level.

Installing the RPMs

To install the RPMs, you can use any of the following techniques:

- Use the RPM Package Manager application. Make sure that you have logged on as the superuser (`root`) to perform the installation.
- Use the graphical package manager of your choice, such as KPackage, to install the RPMs.
- Place all the RPM files in a single directory and, as superuser (`root`), execute the following command to unpack the packages and install all the files they contain into their correct places for your distribution:

```
$ rpm -i *.rpm
```

You can also install from the PostgreSQL packages that are bundled along with your Linux distribution, such as in Red Hat or SuSE Linux. For example, on SuSE Linux 9.*x*, you can install a version of PostgreSQL by running the YaST2 installation tool and selecting the packages listed in Table 3-1, as shown in Figure 3-1.

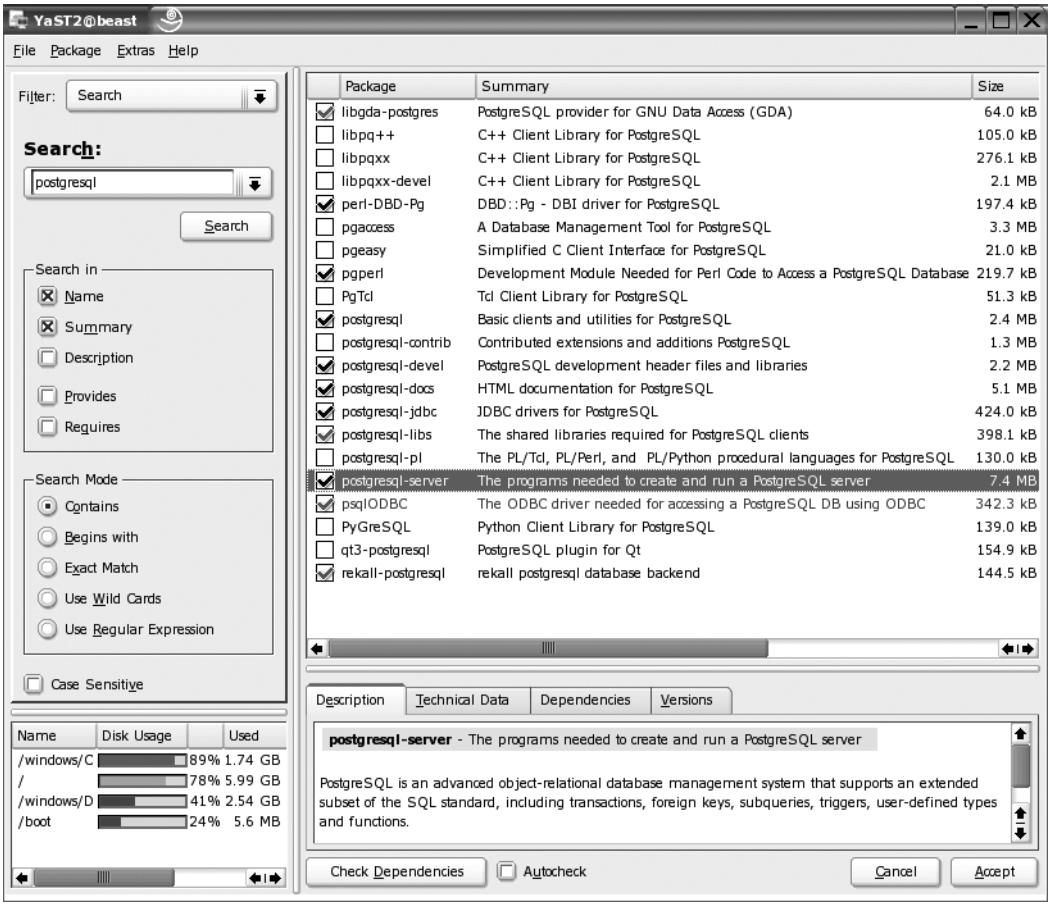


Figure 3-1. *Installing PostgreSQL from SuSE packages with YaST2*

Upgrading to a New PostgreSQL Version

PostgreSQL is under continuous development, so new versions become available from time to time. Installing from RPM packages has the advantage that you can very simply upgrade to the most recent version. To do that, just let `rpm` know that you are performing an upgrade rather than a first-time installation by specifying the `-U` option instead of the `-i` option:

```
$ rpm -U *.rpm
```

However, before performing an upgrade, you should back up the existing data in the database. Any precautions that must be taken when performing an upgrade to the latest release will be noted at the PostgreSQL home site and in the release notes. Backing up existing databases is discussed in detail in Chapter 11 of this book.

■ **Caution** If you are installing a new version of PostgreSQL as an upgrade to an existing installation, be sure to read the release notes for the new version before starting. In some cases, it may be necessary to back up and restore your PostgreSQL databases during an upgrade if, for example, the new version has introduced changes in the way that data is stored.

Anatomy of a PostgreSQL Installation

A PostgreSQL installation consists of a number of applications, utilities, and data directories. The main PostgreSQL application (`postmaster`) contains the server code that services the requests to access data from clients. Utilities such as `pg_ctl` are used to control a master server process that needs to be running all the time the server is active.

PostgreSQL uses a data directory to store all of the files needed for a database. This directory not only stores the tables and records, but also system parameters. A typical installation would have all of the components of a PostgreSQL installation shown in Table 3-2, arranged in subdirectories of one PostgreSQL directory. One common location (and the default when you install from source code, as described in the next section) is `/usr/local/pgsql`.

Table 3-2. *PostgreSQL Installation Anatomy*

Directory	Description
`bin`	Applications and utilities such as `pg_ctl` and `postmaster`
`data`	The database itself, initialized by `initdb`
`doc`	Documentation in HTML format
`include`	Header files for use in developing PostgreSQL applications
`lib`	Libraries for use in developing PostgreSQL applications
`man`	Manual pages for PostgreSQL tools
`share`	Sample configuration files

There is a drawback with the single directory approach: both fixed program files and variable data are stored in the same place, which is often not ideal.

The files that PostgreSQL uses fall into two main categories:

- Files that are written to while the database server is running, including data files and logs. The data files are the heart of the system, storing all of the information for all of your databases. The log file that the database server produces will contain useful information about database accesses and can be a big help when troubleshooting problems. It effectively just grows as log entries are added.

- Files that are not written to while the database server is running, which are effectively read-only files. These files include the PostgreSQL applications like `postmaster` and `pg_ctl`, which are installed once and never change.

For a more efficient and easier to administer setup, you might wish to separate the different categories of files. PostgreSQL offers the flexibility to store the applications, logs, and data in different places, and some Linux distributions have made use of this flexibility to good effect. For example, in SuSE Linux 9.*x*, the PostgreSQL applications are stored with other applications in `/usr/bin`, the log file is in `/var/log/postgresql`, and the data is in `/var/lib/pgsql/data`. This means that it is easy to arrange backups of the critical data separately from the not-so-critical files, such as the log files.

Other distributions will have their own scheme for file locations. You can use `rpm` to list the files that have been installed by a particular package. To do this, use the query option, like this:

```
$ rpm -q -l postgresql-libs
/usr/lib/libecpg.so.4
/usr/lib/libecpg.so.4.1
...
/usr/share/locale/zh_TW/LC_MESSAGES/libpq.mo
$
```

To see where all the files have been installed, you will need to run `rpm` for all of the packages that make up the complete PostgreSQL set. Different distributions may call the packages by slightly different names. For example, SuSE Linux uses the package name `pg_serv` for the server package, so the query option looks like this:

```
$ rpm -q -l pg_serv
/etc/init.d/postgresql
/etc/logrotate.d/postgresql
...
/var/lib/pgsql/data/pg_options
$
```

Alternatively, you can use one of the graphical package manager tools, such as KPackage, which comes with the KDE desktop environment. Figure 3-2 shows an example of viewing a package's contents with KPackage.

The disadvantage of installing from a Linux distribution is that it is not always clear where everything lives. So, if you wish to upgrade to the most recent release, it can be tricky to ensure that you have untangled the original installation. An alternative is to install PostgreSQL from the source code, as described in the next section. If you have no intention of installing from source, you can skip the next section and continue with the PostgreSQL setup described in the "Setting Up PostgreSQL on Linux and UNIX" section.

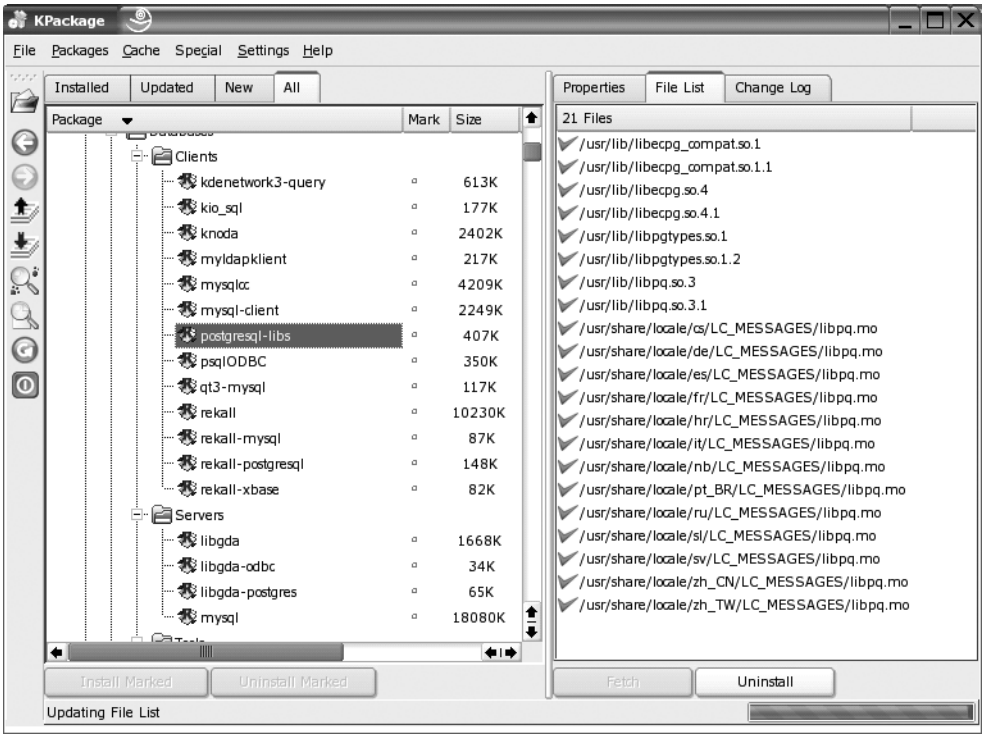


Figure 3-2. *Examining a package's contents with KPackage*

Installing PostgreSQL from the Source Code

As explained in the previous section, you can use RPM packages to install PostgreSQL on many Linux distributions or flavors. Additionally, you can build and install PostgreSQL from the source code on just about any UNIX-compatible system, including Mac OS X.

The source code for PostgreSQL is available at `http://www.postgresql.org`. Here, you will find the code for the latest release and often the source code for beta test versions of the next release. Unless you like to live on the edge, it is probably a good idea to stick to the most recent stable release.

You can find the entire PostgreSQL source code in a single, compressed archive file, either in gzipped-tarball format, with a name something like `postgresql-8.0.0.tar.gz`, or in bzipped-tarball format, with a name like `postgresql-8.0.0.tar.bz2`. At the time of writing, the PostgreSQL tarball was around 13MB in size. To ease the download process in case of an unreliable or slow connection, the source is also available in a set of smaller files:

- `postgresql-8.0.0.base.tar.gz`
- `postgresql-8.0.0.docs.tar.gz`
- `postgresql-8.0.0.opt.tar.gz`
- `postgresql-8.0.0.test.tar.gz`

The exact filenames depend on the current version revision number at the time.

Compiling PostgreSQL is very simple. If you are familiar with compiling open-source products, there will be no surprises for you here. Even if this is your first experience in compiling and installing an open-source product, you should have no difficulty.

To perform the source-code compilation, you will need a Linux or UNIX system with a complete development environment installed. This includes a C compiler and the GNU version of the `make` utility (needed to build the database system). Linux distributions generally ship with a suitable development environment containing the GNU tools from the Free Software Foundation. These include the excellent GNU C compiler (`gcc`), which is the standard compiler for Linux. The GNU tools are available for most other UNIX platforms, too, and we recommend them for compiling PostgreSQL. You can download the latest tools from `http://www.gnu.org`. Once you have a development environment installed, the compilation of PostgreSQL is straightforward.

Extracting the Code

Start the installation as a normal user. Copy your source-code tarball file to an appropriate directory for compiling. This does not need to be—in fact, it should not be—the final resting place of your PostgreSQL installation. One possible choice is a subdirectory in your home directory, since you do not need superuser permissions to compile PostgreSQL; you only need those permissions to install it once it's built. We generally prefer to unpack source code into a directory specifically created for maintaining source code products, `/usr/src`, but you can unpack anywhere you have sufficient disk space for the compilation. You need to allow around 90MB or so.

Unpack the tarball to extract the source code:

```
$ tar zxf postgresql-8.0.0.tar.gz
```

The extraction process will have made a new directory, related to the version of PostgreSQL you are building. Move into that directory:

```
$ cd postgresql-8.0.0
```

■ **Tip** You should find a file, `INSTALL`, in this directory that contains detailed manual build instructions, in the unlikely event that the automated method outlined here fails for some reason.

Configuring the Compilation

The build process uses a configuration script, `configure`, to tailor the build parameters to your specific environment. To accept all defaults, you can simply run `configure` without arguments. Here is an example of running `configure` on a Linux system:

```
$ ./configure
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking which template to use... linux
checking whether to build with 64-bit integer date/time support... no
checking whether NLS is wanted... no
checking for default port number... 5432
checking for gcc... gcc
...
$
```

The `configure` script sets variables that control the way the PostgreSQL software is built, taking into account the type of platform on which you are compiling, the features of your C compiler, and so on. The `configure` script will automatically set locations for the installation. The default locations are for PostgreSQL to be compiled to use `/usr/local/pgsql` as the main directory for its operation, with subdirectories for applications and data.

You can use arguments to `configure` to change the default location settings, to set the network port the database server will use, and to include support for additional server-side programming languages for stored procedures. These options are listed in Table 3-3.

Table 3-3. *PostgreSQL Configure Script Options*

Option	Description
`--prefix=`*`prefix`*	Install in directories under *`prefix`*; defaults to `/usr/local/pgsql`
`--bindir=`*`dir`*	Install application programs in *`dir`*; defaults to *`prefix`*`/bin`
`--with-docdir=`*`dir`*	Install documentation in *`dir`*; defaults to *`prefix`*`/doc`
`--with-pgport=`*`port`*	Set the default TCP port number for serving network connections
`--with-tcl`	Compile server-side support for Tcl stored procedures
`--with-perl`	Compile server-side support for Perl stored procedures
`--with-python`	Compile server-side support for Python stored procedures

To see a full list of options to `configure`, you can use the `--help` argument:

```
$ ./configure --help
`configure' configures PostgreSQL 8.0.0 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE.  See below for descriptions of some of the useful variables.
...
$
```

You do not need to settle on final locations for the database files and the log file at this stage. You can always specify these locations to the server process, when you start it after installation.

Building the Software

Once the compilation is configured, you can build the software using `make`. The PostgreSQL build process uses a sophisticated set of makefiles to control the compilation process. Due to this, we recommend that you use a version of GNU `make` for the build. This is the default on Linux. On other UNIX platforms, you may need to install GNU `make` separately. Often, this will be given the name `gmake` to distinguish it from the version of `make` supplied with the operating system. In the instructions here, `make` refers to GNU `make`.

The next step is to run `make` to compile the software:

```
$ make
...
All of PostgreSQL successfully made. Ready to install.
```

If all goes well, you should see a large number of compilations proceeding. You will be finally rewarded with the message that everything has been made successfully.

When `make` has finished, you need to copy the programs to their final resting places. You use `make` to do this for you, too, but you need to be the superuser first:

```
$ su
# make install
...
PostgreSQL installation complete.
# exit
$
```

Once the software is built and installed, you can query the configuration of a PostgreSQL system with `pg_config`:

```
pg_config --bindir | --includedir | --libdir | --configure | --version
```

The `pg_config` command will report the directory where the PostgreSQL programs are installed (`--bindir`), the location of C include files (`--includedir`) and object code libraries (`--libdir`), and the version of PostgreSQL (`--version`):

```
$ pg_config --version
PostgreSQL 8.0.0
$
```

The build-time configuration can be reported by using `pg_config --configure`. This will report the command-line options passed to the `configure` script when the PostgreSQL server was configured for compilation.

That's just about all there is to installing PostgreSQL. You now have a set of programs that make up the PostgreSQL database server in the right place on your system.

At this point, you are in the same situation as you would have been had you installed from packages. Now it's time to turn our attention to setting up PostgreSQL now that it's installed.

Setting Up PostgreSQL on Linux and UNIX

After you have PostgreSQL installed, whether from RPMs or compiled from the source code, you need to take a few steps to get it up and running. First, you should create a `postgres` user. Then you create a data directory for the database and the initial database structures. At that point, you can start PostgreSQL by starting the `postmaster` process.

Creating the postgres User

The main database process for PostgreSQL, `postmaster`, is quite a special program. It is responsible for dealing with all data access from all users to all databases. It must allow users to access their data but not access other users' data, unless authorized. To do this, it needs to have sole control of all of the data files, so that no normal user can access any of the files directly. The `postmaster` process will control access to the data files by checking the permissions granted to the users that request access and performing the access on their behalf.

Strictly speaking, PostgreSQL needs to run only as a non-`root` user, which could be any normal user; if you install in your home directory, it could be your own user. However, a PostgreSQL installation typically uses the concept of a pseudo user to enforce data access. A user, often called `postgres`, is created for the sole purpose of owning the data files and has no other access rights. A `postgres` pseudo user provides some additional security, as no one can log in as the `postgres` user and gain illicit access. This user identity is used by the `postmaster` program to access the database files on behalf of others.

The first step in establishing a working PostgreSQL system is, therefore, to create this `postgres` user. The precise procedure for making new users differs from system to system. Linux users can (as `root`) simply use `useradd`:

```
# useradd postgres
```

Other UNIX systems may require you to create a home directory, edit the configuration files, or run the appropriate administration tool on your Linux distribution. Refer to your operating system documentation for details about using such administration tools.

Creating the Database Directory

Next, you must create, as `root`, the directory PostgreSQL is going to use for its databases and change its owner to be `postgres`:

```
# mkdir /usr/local/pgsql/data
# chown postgres /usr/local/pgsql/data
```

Here, we are using the default location for the database. You might choose to store the data in a different location, as we discussed earlier, in the "Anatomy of a PostgreSQL Installation" section.

Initializing the Database

You initialize the PostgreSQL database by using the `initdb` utility, specifying where in your file system you want the database files to reside. This will do several things, including creating the data structures PostgreSQL needs to run and creating an initial working database, `template1`.

You need to assume the identity of the `postgres` user to run the `initdb` utility. To do this, the most reliable way is to change your identity in two steps, first becoming `root` with `su` and then becoming `postgres` as follows. (As a normal user, you may not have permission to assume another user's identity, so you must become the superuser first.)

```
$ su
# su - postgres
pg$
```

Now the programs you run will assume the rights of the `postgres` user and will be able to access the PostgreSQL database files. For clarity, we have shown the shell prompt for commands executed as the `postgres` user as `pg$`.

■ **Caution** Do not be tempted to shortcut the process of using the `postgres` user and run these programs as `root`. For security reasons, running server processes as `root` can be dangerous. If there were a problem with the process, it could result in an outsider gaining access to your system via the network. For this reason, `postmaster` will refuse to run as `root`.

Initialize the database with `initdb`:

```
pg$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale en_GB.UTF-8.
The default database encoding has accordingly been set to UNICODE.
...
WARNING: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the -A option the
next time you run initdb.

Success. You can now start the database server using:

    /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
or
    /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
pg$
```

If all goes well, you will have a new, completely empty database in the location you specified with the `-D` option to `initdb`.

Granting Connection Permissions

By default, PostgreSQL will not allow general remote access. To grant permission to connect, you must edit a configuration file, `pg_hba.conf`. This file lives in the database file area (`/usr/local/pgsql/data` in our example), and contains entries that grant or reject permission

for users to connect to the database. By default, local users may connect and remote users cannot. Its format is fairly simple, and the default file shipped with PostgreSQL contains many helpful comments for adding entries. You can grant permission for individual users, hosts, groups of computers, and individual databases, as necessary.

For example, to allow the user `neil` on a machine with IP address 192.168.0.3 to connect to the `bpsimple` database, add the following line to `pg_hba.conf`:

```
host        bpsimple        neil      192.168.0.3/32        md5
```

Note that in versions of PostgreSQL earlier than 8.0, the `pg_hba.conf` file used host address specifications using an IP address and subnet mask, so the preceding example would need to be written as follows:

```
host        bpsimple        neil      192.168.0.3     255.255.255.255       md5
```

Here, we will add an entry to allow any computer on the local network (in this case the subnet 192.168.*x*.*x*) to connect to any database with password authentication. (If you require a different access policy, refer to the comments in the configuration file.) We add a line to the end of `pg_hba.conf` that looks like this:

```
host        all        all      192.168.0.0/16        md5
```

This means that all computers with an IP address that begins 192.168 can access all databases.

Alternatively, if we trust all of the users on all of the machines in a network, we can allow unrestricted access by specifying `trust` as the authentication mechanism, like this:

```
host        all        all      192.168.0.0/16        trust
```

The PostgreSQL `postmaster` server process reads a configuration file, `postgresql.conf` (also in the `data` directory) to set a number of runtime options, including (if not otherwise specified in a `-D` option or the `PGDATA` environment variable) the location of the database data files. The configuration file is well commented, providing guidance if you need to change any settings. There is also a section on runtime configuration in the PostgreSQL documentation.

As an example, we can allow the server to listen for network connections by setting the `listen_addresses` variable in `postgresql.conf`, instead of using the now deprecated `-i` option to `postmaster`, as follows:

```
listen_addresses='*'
```

In fact, setting configuration options in `postgresql.conf` is the recommended approach for controlling the behavior of the `postmaster` process.

Starting the postmaster Process

Now you can start the server process itself. Again, you use the `-D` option to tell `postmaster` where the database files are located. If you want to allow users on a network to access your data, you can specify the `-i` option to enable remote clients (if you haven't enabled `listen_addresses` in `postgresql.conf`, as in the preceding example):

```
pg$ /usr/local/pgsql/bin/postmaster -i -D /usr/local/pgsql/data >logfile 2>&1 &
```

This command starts `postmaster`, redirects the process output to a file (called `logfile` in the `postgres` user's home directory), and merges standard output with standard error by using the shell construction `2>&1`. You can choose a different location for your log file by redirecting output to another file.

The `pg_ctl` utility provided with PostgreSQL offers a simple way of starting, stopping, and restarting (the equivalent of stop followed by start) the `postmaster` process. If PostgreSQL is fully configured using the `postgresql.conf` configuration file, as mentioned in the previous section, it is possible to start, stop, and restart with these commands:

```
pg_ctl start
pg_ctl stop
pg_ctl restart
```

Connecting to the Database

Now you can check that the database is functioning by trying to connect to it. The `psql` utility is used to interact with the database system and perform simple administrative tasks such as creating users, creating databases, and creating tables. We will use it to create and populate the sample database later in the chapter, and it is covered in more detail in Chapter 5. For now, you can simply try to connect to a database. The response you get should show that you have `postmaster` running:

```
pg$ /usr/local/pgsql/bin/psql
psql: FATAL 1: Database "postgres" does not exist in the system catalog.
```

Don't be taken aback by the fatal error it displays. By default, `psql` connects to the database on the local machine and tries to open a database with the same name as the user running the program. We have not created a database called `postgres`, so the attempt fails. It does indicate, however, that `postmaster` is running and able to respond with details of the failure.

To specify a particular database to connect to, use the `-d` option to `psql`. A new PostgreSQL system does contain some databases that are used by the system as the base for new databases you might create. One such database is called `template1`. If you need to, you can connect to this database for administration purposes.

To check network connectivity, you can use `psql` installed on another machine on the network as a client, or any other PostgreSQL compatible application. With `psql`, you specify the host (either the name or IP address) with the `-h` option, and one of the system databases (as you haven't yet created a real database):

```
remote$ psql -h 192.168.0.111 -d template1
Welcome to psql 8.0.0, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

template1=# \q

remote$
```

Configuring Automatic Startup

The final step you need to take is to arrange for the `postmaster` server process to be started automatically every time the machine is rebooted. Essentially, all you need to do is make sure that `postmaster` is run at startup. Again, there is little standardization between Linux and UNIX variants as to how this should be done. Refer to your installation's documentation for specific details.

If you have installed PostgreSQL from a Linux distribution, it is likely that the startup is already configured by the RPM packages you installed. On SuSE Linux, PostgreSQL is automatically started when the system enters multiuser mode, by a script in `/etc/rc.d/init.d` called `postgresql`.

If you are creating a startup script yourself, the easiest thing to do is create a simple shell script that starts `postmaster` with the parameters you need, and add a call to your script from one of the scripts that is run automatically at startup, such as those found in `/etc/rc.d`. Be sure that `postmaster` is run as the user `postgres`. Here is an example of a script that does the job for a default PostgreSQL installation built from source code:

```
#!/bin/sh

# Script to start and stop PostgreSQL

SERVER=/usr/local/pgsql/bin/postmaster
PGCTL=/usr/local/pgsql/bin/pg_ctl
PGDATA=/usr/local/pgsql/data
OPTIONS=-i
LOGFILE=/usr/local/pgsql/data/postmaster.log

case "$1" in
      start)
            echo -n "Starting PostgreSQL..."

            su -l postgres -c "nohup $SERVER $OPTIONS -D $PGDATA >$LOGFILE 2>&1 &"
            ;;
      stop)
            echo -n "Stopping PostgreSQL..."
            su -l postgres -c "$PGCTL -D $PGDATA stop"
            ;;
      *)
            echo "Usage: $0 {start|stop}"
            exit 1
            ;;
esac
exit 0
```

■ **Note** On Debian Linux, you may need to use `su -` in place of `su -l`.

Create an executable script file with this script in it. Call it, for example, `MyPostgreSQL`. Use the `chmod` command to make it executable, as follows:

```
# chmod a+rx MyPostgreSQL
```

Then you need to arrange that the script is called to start and stop PostgreSQL when the server boots and shuts down:

```
MyPostgreSQL start
MyPostgreSQL stop
```

For systems (such as many Linux distributions) that use System V type `init` scripting, you can place the script in the appropriate place. For SuSE Linux, for example, you would place the script in `/etc/rc.d/init.d/MyPostgreSQL`, and make symbolic links to it from the following places to automatically start and stop PostgreSQL as the server enters and leaves multiuser mode:

```
/etc/rc.d/rc2.d/S25MyPostgreSQL
/etc/rc.d/rc2.d/K25MyPostgreSQL
/etc/rc.d/rc3.d/S25MyPostgreSQL
/etc/rc.d/rc3.d/K25MyPostgreSQL
```

Refer to your system's documentation on startup scripts for more specific details.

Stopping PostgreSQL

It is important that the PostgreSQL server process is shut down in an orderly fashion. This will allow it to write any outstanding data to the database and free up shared memory resources it is using.

To cleanly shut down the database, use the `pg_ctl` utility as `postgres` or `root`, like this:

```
# /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data stop
```

If startup scripts are in place, you can use those, as in this example:

```
# /etc/rc.d/init.d/MyPostgreSQL stop
```

The scripts also make sure that the database is shut down properly when the machine is halted or rebooted.

RESOURCES

To make life a little easier when dealing with PostgreSQL, it might be of some use to add the PostgreSQL applications directory to your execution path, and similarly the manual pages. To do this for the standard UNIX shell, place the following commands in your shell startup file (`.profile` or `.bashrc`):

```
PATH=$PATH:/usr/local/pgsql/bin
MANPATH=$MANPATH:/usr/local/pgsql/man
export PATH MANPATH
```

The source code for the current and latest test releases of PostgreSQL can be found at `http://www.postgresql.org`. More resources for PostgreSQL are listed in Appendix G of this book.

Installing PostgreSQL on Windows

Let's begin this section with some good news for Windows users. Although PostgreSQL was developed for UNIX-like platforms, it was written to be portable. It has been possible for some time now to write PostgreSQL client applications for Windows, and from version 7.1 onwards, PostgreSQL could be compiled, installed, and run as a PostgreSQL server on Microsoft Windows NT 4, 2000, XP, and Server 2003.

With PostgreSQL version 8.0, a native Windows version is available, offering a Windows installer for both server and client software, which makes installing on Windows a breeze. Prior to version 8.0, Windows users needed to install some additional software to support some UNIX features on Windows.

■ **Note** PostgreSQL 8.0 is supported on Windows 2000, Windows XP, and Windows Server 2003. It requires features not present in Windows 95, 98, and Me, so it will not run on those versions. It can be persuaded to run on Windows NT, but the installation must be performed by hand, as the PostgreSQL installer does not work correctly for Windows NT.

It may seem that an open-source platform like Linux would be the natural home of an open-source database like PostgreSQL. Indeed, we would not recommend running a production database on a desktop version of Windows, but installing on Windows can have its advantages. For example, having the PostgreSQL utilities like `psql` on the same machine as some client applications can be useful in testing new database installations and troubleshooting connection problems, even if you don't need to run the server on Windows. Running the database server on a development machine can avoid any potential problems with developers needing to share a server instance elsewhere.

Using the Windows Installer

The installer for the Windows version of PostgreSQL is a separate PostgreSQL-related project with its home page at `http://pgfoundry.org/projects/pginstaller`. The latest version of the installer may be downloaded from the home page or one of the PostgreSQL mirror sites.

The installer is packaged as a Microsoft Windows Installer (`.msi`) file inside a ZIP archive. To run the installer, you will need version 2.0 or later of the Windows Installer. A suitable version is included with Windows XP and later. If necessary, the Windows Installer can be downloaded from `http://www.microsoft.com` (search for "Windows Installer redistributable").

The PostgreSQL MSI package has a filename similar to `postgresql-8.0.0.msi`. To start the installation wizard, just double-click this file to start the installer. After choosing a language to use for installation and reading the installation notes presented, you will see the installation options dialog box, as shown in Figure 3-3.

You can also use this dialog box to set the locations for the PostgreSQL applications and the PostgreSQL database files.

Click PostgreSQL, and then click Browse to set the location for the application installation (the default is `C:\Program Files\PostgreSQL\8.0.0`).

Click Data directory, and then click Browse to set the location of the database files (the default is `C:\Program Files\PostgreSQL\8.0.0\data`).

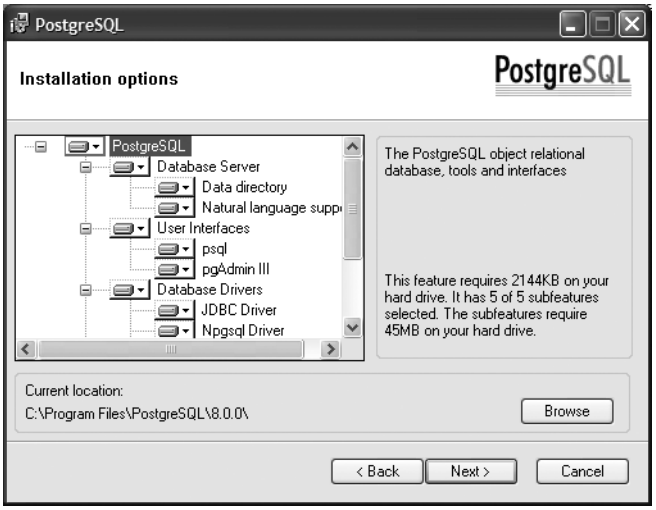


Figure 3-3. *PostgreSQL installation options*

The locations of other components can be similarly set if required; they will default to subfolders of the application installation folder. We recommend leaving them at their default locations.

Here, you can choose which components you need to install, depending on how you will use the machine on which PostgreSQL is being installed: as a database server, a client, a development machine, or a mixture. The installation options are summarized in Table 3-4.

Table 3-4. *PostgreSQL Installation Options*

Option	Meaning
Database Server	The PostgreSQL database
Natural language support	Support for status and error messages in non-English languages
psql	The PostgreSQL command-line interface
pgAdmin III	A graphical PostgreSQL management console
JDBC Driver	The PostgreSQL JDBC driver for Java clients
Npgsql Driver	The PostgreSQL Microsoft .NET driver
ODBC Driver	The PostgreSQL ODBC driver
OLEDB Provider	The PostgreSQL OLEDB Provider
Documentation	HTML format documentation
Development	Support files and utilities for creating PostgreSQL clients

The following are our recommendations for each type of setup:

- To set up a simple database server, it is sufficient to select the Database Server option. This will result in an installation that needs to be managed remotely from another machine. It is helpful to also include the `psql` command-line interface, even for a server-only installation.
- To set up a machine for managing a remote server, we suggest you choose the psql and pgAdmin III options. These can be installed independently of the database server.
- To set up a machine that will run applications that connect to a remote PostgreSQL database, choose the appropriate drivers. As mentioned in Chapter 2 and covered in more detail in Chapter 5, you can use the ODBC driver to connect applications such as Microsoft Access and Excel to PostgreSQL. Java and .NET applications need the JDBC and Npgsql drivers, respectively. The OLEDB Provider allows PostgreSQL to be used with OLEDB clients such as Microsoft Visual Studio.
- To set up a machine that will be used to develop client applications, such as those covered in Chapters 13 and 14, choose the Development option to install appropriate header files and libraries. You will also need a development environment such as Microsoft Visual Studio or Cygwin (`http://www.cygwin.com`) to compile your applications.

For our installation, we selected all of the available options.

The next step in the installation is to configure the database server to run as a service, as shown in Figure 3-4. This is the recommended option, as it will allow the PostgreSQL server to be automatically started when Windows is booted.

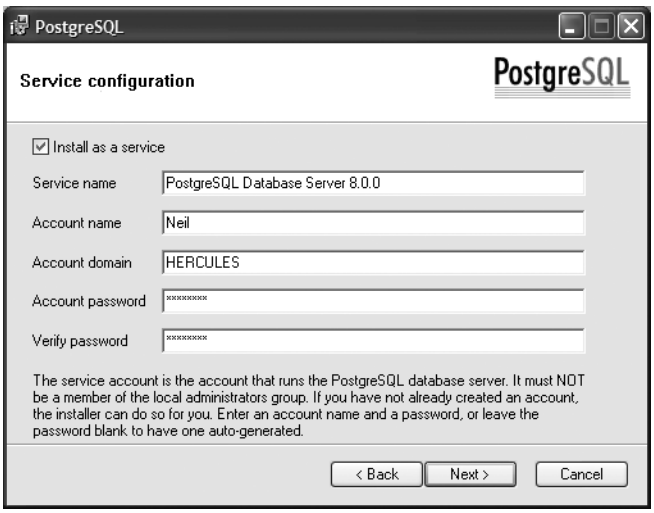


Figure 3-4. *PostgreSQL service configuration*

PostgreSQL must run as a non-administrator user. This avoids any potential security risk from running a service that accepts network connections as a user that has administrative privileges. In the unlikely event of security vulnerabilities in PostgreSQL being discovered and exploited, then only files and data managed by PostgreSQL would be at risk, rather than the entire server. You can either create an account on Windows for the purposes of running PostgreSQL or have the installer do it for you: just give an account name to the installer, and it will create it, if it does not already exist.

Next, initialize the PostgreSQL database, as shown in Figure 3-5. (Note that for an upgrade installation, it will not be necessary to perform the database initialization step, as you would normally want the existing database to be preserved.)

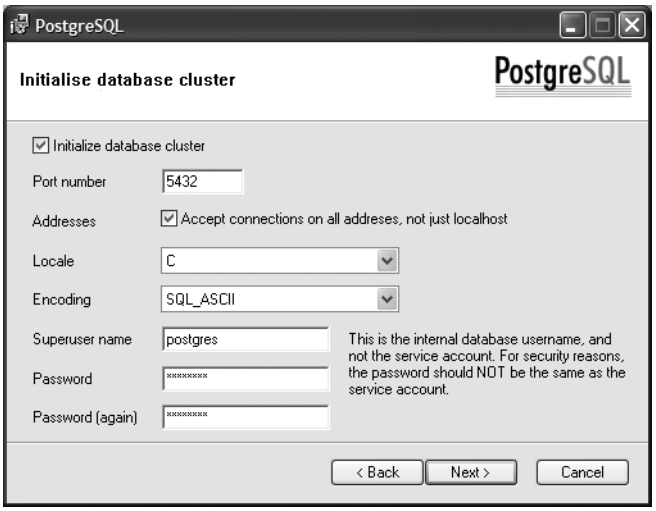


Figure 3-5. *PostgreSQL database initialization*

Here, you specify a superuser account for PostgreSQL. This is a database user that has permissions to create and manage databases within the server. It is different from the Windows account that is used to run the server. PostgreSQL accounts are used by clients connecting to the database, and PostgreSQL itself manages the authentication of these users; they do not need to have Windows accounts on the database server. As noted in the dialog box shown in Figure 3-5, there are security advantages to using a different username and password for the database superuser.

To allow the database server to accept connections from the network, check the Addresses check box. Without this selected, only clients running on the server machine will be able to connect. Although this option will start the server listening on the network, you still have control over who can connect and where from. We will cover client access configuration in the next section.

In our installation, we have left the locale and encoding schemes for the database at their default values. If you are installing a PostgreSQL database in an environment that requires the use of a specific character set or locale, these options can be set here. If you are not familiar with character sets and locales, then the defaults will probably work just fine.

In Chapter 9, we will cover stored procedures, which are functions that you can execute on the server to perform tasks more efficiently than in a client application. PostgreSQL supports stored procedures written in a variety of programming languages, including its own PL/pgSQL,

Perl, Python, and others. To run the examples in Chapter 9, you need to select the PL/pgSQL option in the next installation dialog box, Procedural Languages, shown in Figure 3-6.

Figure 3-6. *PostgreSQL procedural languages*

The next installation dialog box deals with contributed modules, and its settings can safely be left at the defaults. (We do not cover these advanced topics in this book.)

The installation will now proceed and complete. The database server processes should be running. They will be visible as `postmaster.exe` and several `postgres.exe` processes in Task Manager, as shown in Figure 3-7.

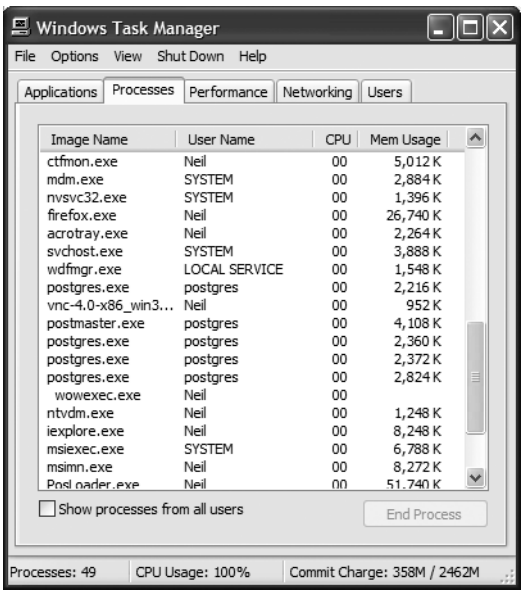


Figure 3-7. *PostgreSQL processes*

The PostgreSQL applications and utilities are installed in a new program group accessible from the Start menu, as shown in Figure 3-8.

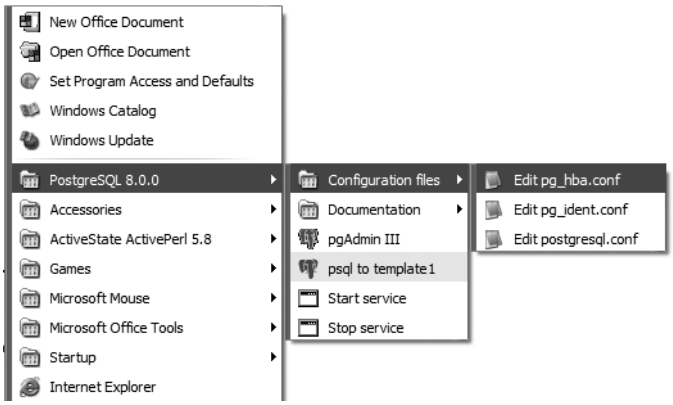


Figure 3-8. *PostgreSQL program menu*

Configuring Client Access

To configure remote hosts and users that can connect to the PostgreSQL service, you need to edit the `pg_hba.conf` file. This file contains many comments that document the options available for remote access. In our sample installation, we want to allow users from any host on the local network to access all of the databases on our server. To do this, we add the following line at the end of the file:

```
host    all    all     192.168.0.0/16    trust
```

This means that all computers with an IP address that begins 192.168 can access all databases.

The simplest way to make this configuration change take effect is to restart the PostgreSQL server.

The `pg_hba.conf` file takes the same form on Windows systems as it does on Linux and UNIX systems. For other examples of configuration access, see the "Granting Connection Permissions" section earlier in this chapter.

Creating the Sample Database

Now that we have PostgreSQL up and running, we are going to create a simple database, which we will call `bpsimple`, to support our customer order tables examples. This database (together with a variant called `bpfinal` created in Chapter 8) is used throughout the book. We'll cover the details of creating databases and creating and populating databases in later chapters. Here, we will just show the steps and SQL scripts, so that we have a database to use for demonstration.

Before we start, one simple way to check if PostgreSQL is running on your system is to look for the `postmaster` process. On Windows systems, look for `postmaster.exe` in the processes tab of Task Manager. On UNIX and Linux systems, run the following command:

```
$ ps -el | grep post
```

If there is a process running called `postmaster` (the name might be abbreviated in the display), then you are running a PostgreSQL server.

Creating User Records

Before we can create a database, we need to tell PostgreSQL about the valid users by creating records for them within the system. Valid users of a PostgreSQL database system can read data, insert data, or update data; create databases of their own; and control access to the data those databases hold. To create user records, we use PostgreSQL's `createuser` utility.

On Linux and UNIX systems, use `su` (from `root`) to become the PostgreSQL user, `postgres`. Then run `createuser` to register the user. The user login name given is recorded as a valid PostgreSQL user. Let's give user rights to the (existing UNIX/Linux) user `neil`:

```
$ su
# su - postgres
pg$ /usr/local/pgsql/bin/createuser neil
Shall the new user be able to create databases? (y/n) y
Shall the new user be able to create new users (y/n) y
CREATE USER
pg$
```

On Windows systems, open a Command Prompt window and change the directory to the location of the PostgreSQL application (the default is `C:\Program Files\PostgreSQL\8.0.0` in our installation). Then run the `createuser.exe` utility:

```
C:\Program Files\PostgreSQL\8.0.0\bin>createuser -U postgres -P neil
Enter password for new user:
Enter it again:
Shall the new user be allowed to create databases? (y/n) y
Shall the new user be allowed to create more new users? (y/n) y
Password:
CREATE USER
```

The `-U` option is used to specify the identity you want to use for creating the new user. It must be a PostgreSQL user with permission to create users, normally the PostgreSQL user you named when you performed the installation. The `-P` option causes `createuser` to prompt for a password for the new user.

Here, we have allowed `neil` to create new databases, and he is allowed to create new users. Some of the examples in the book use another user, `rick`, who also has permission to create databases, but does not have permission to create new users. If you would like to exactly replicate these examples, now is a good time to create this user.

Once you have created a PostgreSQL user with these rights, you will be able to create the `bpsimple` database.

Creating the Database

To create the database on Linux and UNIX systems, change back to your own (non-`root`) login and run the following command:

```
$ /usr/local/pgsql/bin/createdb bpsimple
CREATE DATABASE
$
```

On Windows systems, run the `createdb.exe` command:

```
C:\Program Files\PostgreSQL\8.0.0\bin>createdb -U neil bpsimple
Password:
CREATE DATABASE
```

You should now be able to connect (locally) to the server, using the interactive terminal `psql`. On Linux and UNIX systems, use the following command:

```
$ /usr/local/pgsql/bin/psql -U neil -d bpsimple
Welcome to psql 8.0.0, the PostgreSQL interactive terminal.
...
bpsimple=#
```

On Windows systems, use this command:

```
C:\Program Files\PostgreSQL\8.0.0\bin>psql -U neil -d bpsimple
Password:
Welcome to psql 8.0.0, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

Warning: Console codepage (850) differs from windows codepage (1252)
         8-bit characters will not work correctly. See PostgreSQL
         documentation "Installation on Windows" for details.

bpsimple=#
```

Alternatively, you can select the Windows Start menu item `psql to template1`, and then change the database within `psql`:

```
template1=# \c bpsimple
You are now connected to database "bpsimple".
bpsimple=#
```

You are now logged into PostgreSQL, ready to execute some commands. To exit back to the shell, use the command `\q`.

Next, we will use a set of SQL statements to create and populate the sample database.

Creating the Tables

You can create the tables in your `bpsimple` database by typing in the SQL commands that follow at the `psql` command prompt. However, it's easier to download the code bundle from the Downloads section of the Apress web site (`http://www.apress.com`), unpack it, and then execute the commands using `\i <filename>`. (The `\i` command in `psql` can be used to execute groups of SQL statements and other PostgreSQL commands stored in text files, called *scripts*.) The commands are just plain text, so you can always edit them with your preferred text editor if you want.

To run the `create_tables-bpsimple.sql` script to create the tables, enter the following:

```
bpsimple=# \i create_tables-bpsimple.sql
CREATE TABLE
...
bpsimple=#
```

It is a very good practice to script all database schema (tables, indexes, and procedures) statements. That way, if the database needs to be re-created, you can do that from the scripts. Scripts should also be used whenever the schema needs to be updated.

Here is the SQL for creating our tables (the ones we designed in Chapter 2), which you will find in `create_tables-bpsimple.sql` in the code bundle:

```
CREATE TABLE customer
(
    customer_id      serial                        ,
    title            char(4)                       ,
    fname            varchar(32)                   ,
    lname            varchar(32)           NOT NULL,
    addressline      varchar(64)                   ,
    town             varchar(32)                   ,
    zipcode          char(10)              NOT NULL,
    phone            varchar(16)                   ,
    CONSTRAINT       customer_pk PRIMARY KEY(customer_id)
);


CREATE TABLE item
(
    item_id          serial                        ,
    description      varchar(64)           NOT NULL,
    cost_price       numeric(7,2)                  ,
    sell_price       numeric(7,2)                  ,
    CONSTRAINT       item_pk PRIMARY KEY(item_id)
);
```

```
CREATE TABLE orderinfo
(
    orderinfo_id     serial                        ,
    customer_id      integer              NOT NULL,
    date_placed      date                 NOT NULL,
    date_shipped     date                          ,
    shipping         numeric(7,2)                  ,
    CONSTRAINT       orderinfo_pk PRIMARY KEY(orderinfo_id)
);


CREATE TABLE stock
(
    item_id          integer              NOT NULL,
    quantity         integer              NOT NULL,
    CONSTRAINT       stock_pk PRIMARY KEY(item_id)
);


CREATE TABLE orderline
(
    orderinfo_id     integer              NOT NULL,
    item_id          integer              NOT NULL,
    quantity         integer              NOT NULL,
    CONSTRAINT       orderline_pk PRIMARY KEY(orderinfo_id, item_id)
);


CREATE TABLE barcode
(
    barcode_ean      char(13)             NOT NULL,
    item_id          integer              NOT NULL,
    CONSTRAINT       barcode_pk PRIMARY KEY(barcode_ean)
);
```

Removing the Tables

If, at some later date, you wish to delete all the tables (also known as *dropping* the tables) and start again, you can. The command set is in the `drop_tables.sql` file, and looks like this:

```
DROP TABLE barcode;
DROP TABLE orderline;
DROP TABLE stock;
DROP TABLE orderinfo;
DROP TABLE item;
DROP TABLE customer;
```

```
DROP SEQUENCE customer_customer_id_seq;
DROP SEQUENCE item_item_id_seq;
DROP SEQUENCE orderinfo_orderinfo_id_seq;
```

Be warned, if you drop the tables, you also lose any data in them!

■ **Note** The `drop_tables.sql` script also explicitly drops the special attributes, called *sequences*, that PostgreSQL uses to maintain the automatically incrementing `serial` columns. In PostgreSQL version 8.0 and later, these sequences will be automatically dropped when the relevant table is dropped, but we have retained the commands for compatibility with earlier versions.

If you run this script after creating the tables, then you should run the `create_tables-bpsimple.sql` script again before attempting to populate the tables with data.

Populating the Tables

Last, but not least, we need to add some data to the tables, or *populate* the tables.

The sample data is in the code bundle available from the Apress web site, as `pop_tablename.sql`. If you choose to use your own data, your results will be different from the ones presented in the book. So, until you are confident, it's probably best to stick with our sample data.

The line wraps are simply a necessity of the fitting the commands on the printed page. You can type each command on a single line. You do need to include the terminating semicolon, which tells `psql` where each SQL command ends.

Customer table

```
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Miss','Jenny','Stones','27 Rowan Avenue','Hightown','NT2 1AQ','023 9876');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mr','Andrew','Stones','52 The Willows','Lowtown','LT5 7RA','876 3527');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Miss','Alex','Matthew','4 The Street','Nicetown','NT2 2TX','010 4567');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mr','Adrian','Matthew','The Barn','Yuleville','YV67 2WR','487 3871');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mr','Simon','Cozens','7 Shady Lane','Oakenham','OA3 6QW','514 5926');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mr','Neil','Matthew','5 Pasture Lane','Nicetown','NT3 7RT','267 1232');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mr','Richard','Stones','34 Holly Way','Bingham','BG4 2WE','342 5982');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mrs','Ann','Stones','34 Holly Way','Bingham','BG4 2WE','342 5982');
```

```
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mrs','Christine','Hickman','36 Queen Street','Histon','HT3 5EM','342 5432');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mr','Mike','Howard','86 Dysart Street','Tibsville','TB3 7FG','505 5482');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mr','Dave','Jones','54 Vale Rise','Bingham','BG3 8GD','342 8264');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mr','Richard','Neill','42 Thatched Way','Winersby','WB3 6GQ','505 6482');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mrs','Laura','Hardy','73 Margarita Way','Oxbridge','OX2 3HX','821 2335');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mr','Bill','O\'Neill','2 Beamer Street','Welltown','WT3 8GM','435 1234');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mr','David','Hudson','4 The Square','Milltown','MT2 6RT','961 4526');
```

Item table

```
INSERT INTO item(description, cost_price, sell_price)
VALUES('Wood Puzzle', 15.23, 21.95);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Rubik Cube', 7.45, 11.49);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Linux CD', 1.99, 2.49);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Tissues', 2.11, 3.99);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Picture Frame', 7.54, 9.95);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Fan Small', 9.23, 15.75);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Fan Large', 13.36, 19.95);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Toothbrush', 0.75, 1.45);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Roman Coin', 2.34, 2.45);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Carrier Bag', 0.01, 0.0);
INSERT INTO item(description, cost_price, sell_price)
VALUES('Speakers', 19.73, 25.32);
```

Barcode table

```
INSERT INTO barcode(barcode_ean, item_id) VALUES('6241527836173', 1);
INSERT INTO barcode(barcode_ean, item_id) VALUES('6241574635234', 2);
INSERT INTO barcode(barcode_ean, item_id) VALUES('6264537836173', 3);
INSERT INTO barcode(barcode_ean, item_id) VALUES('6241527746363', 3);
INSERT INTO barcode(barcode_ean, item_id) VALUES('7465743843764', 4);
INSERT INTO barcode(barcode_ean, item_id) VALUES('3453458677628', 5);
```

```
INSERT INTO barcode(barcode_ean, item_id) VALUES('6434564564544', 6);
INSERT INTO barcode(barcode_ean, item_id) VALUES('8476736836876', 7);
INSERT INTO barcode(barcode_ean, item_id) VALUES('6241234586487', 8);
INSERT INTO barcode(barcode_ean, item_id) VALUES('9473625532534', 8);
INSERT INTO barcode(barcode_ean, item_id) VALUES('9473627464543', 8);
INSERT INTO barcode(barcode_ean, item_id) VALUES('4587263646878', 9);
INSERT INTO barcode(barcode_ean, item_id) VALUES('9879879837489', 11);
INSERT INTO barcode(barcode_ean, item_id) VALUES('2239872376872', 11);
```

Orderinfo table

```
INSERT INTO orderinfo(customer_id, date_placed, date_shipped, shipping)
VALUES(3,'03-13-2000','03-17-2000', 2.99);
INSERT INTO orderinfo(customer_id, date_placed, date_shipped, shipping)
VALUES(8,'06-23-2000','06-24-2000', 0.00);
INSERT INTO orderinfo(customer_id, date_placed, date_shipped, shipping)
VALUES(15,'09-02-2000','09-12-2000', 3.99);
INSERT INTO orderinfo(customer_id, date_placed, date_shipped, shipping)
VALUES(13,'09-03-2000','09-10-2000', 2.99);
INSERT INTO orderinfo(customer_id, date_placed, date_shipped, shipping)
VALUES(8,'07-21-2000','07-24-2000', 0.00);
```

Orderline table

```
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(1, 4, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(1, 7, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(1, 9, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(2, 1, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(2, 10, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(2, 7, 2);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(2, 4, 2);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(3, 2, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(3, 1, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(4, 5, 2);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(5, 1, 1);
INSERT INTO orderline(orderinfo_id, item_id, quantity) VALUES(5, 3, 1);
```

Stock table

```
INSERT INTO stock(item_id, quantity) VALUES(1,12);
INSERT INTO stock(item_id, quantity) VALUES(2,2);
INSERT INTO stock(item_id, quantity) VALUES(4,8);
INSERT INTO stock(item_id, quantity) VALUES(5,3);
INSERT INTO stock(item_id, quantity) VALUES(7,8);
INSERT INTO stock(item_id, quantity) VALUES(8,18);
INSERT INTO stock(item_id, quantity) VALUES(10,1);
```

With the PostgreSQL system running, the database created, and the tables made and populated, we are ready to continue our exploration of PostgreSQL features.

Summary

In this chapter, we have taken a look at some of the options for installing PostgreSQL on Linux, UNIX-compatible systems, and Windows. The simplest way is probably to use some form of precompiled binary package. We provided step-by-step instructions for compiling, installing, and confirming a working installation on Linux systems from packages, UNIX-compatible systems from source code, and Windows systems using the Microsoft Windows installer.

Finally, we created a sample database that we will be using throughout the rest of the book to demonstrate the features of the PostgreSQL system. We'll begin in the next chapter by exploring how to access your data.

CHAPTER 4

Accessing Your Data

So far in this book, our encounters with SQL have been rather informal. We have seen some statements that retrieve data in various ways, as well as some SQL for creating and populating tables.

In this chapter, we will take a slightly more formal look at SQL, starting with the `SELECT` statement. In fact, this whole chapter is devoted to the `SELECT` statement. Your first impression might be that a whole chapter on one part of SQL is a bit excessive, but the `SELECT` statement is at the heart of the SQL language. Once you understand `SELECT`, you really have done the hard part of learning SQL.

In the next chapter, we will talk about some of the GUI clients you can use, but for now, we will be using `psql`, a simple command-line tool that ships with PostgreSQL, to access the database.

In this chapter, we'll cover the following topics:

- Using the `psql` command to interact with the PostgreSQL database
- Using some simple `SELECT` statements for retrieving data
- Improving the output readability by overriding column names
- Controlling the order of rows in retrieved data
- Suppressing duplicate rows
- Performing mathematical calculations while retrieving data
- Aliasing table names for convenience
- Using pattern matching to specify what data to retrieve
- Making comparisons using various data types
- Retrieving data from multiple tables in a single `SELECT` statement
- Relating three or more tables in a `SELECT` statement

By now, you should have PostgreSQL up and running. Throughout this chapter, we will be using the sample database we designed in Chapter 2 and created and populated in Chapter 3.

Using psql

Assuming you have followed the instructions in Chapter 3, you should now have a database called `bpsimple`, accessible from your normal PostgreSQL login prompt.

■ **Caution** You should never use the `postgres` user for accessing the PostgreSQL server, except in the special case of database administration.

Starting Up on Linux Systems

If you are on a Linux system, and you created an ordinary user without a password, to start `psql` accessing the `bpsimple` database, you include your username in the connection command. For example, to access the database as user `rick`, you would enter:

```
$ psql -d bpsimple -U rick
```

You should see the following:

```
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

bpsimple=>
```

We are now ready to enter commands. If you created the user with a password, you may be prompted for a password, depending on the exact authentication configuration. We will explain more about authentication in Chapter 11.

Starting Up on Windows Systems

If you are using Windows, begin by opening the Start menu and choosing the command `psql to template1`. You'll be prompted for the `postgres` user password. After successful connecting, switch to the `bpsimple` database and your own username (here, we show user `rick`) using the `\c` command, like this:

```
template1=# \c bpsimple rick
You are now connected to database "bpsimple" as user "rick".
bpsimple=>
```

Notice the prompt changes from =# to => to show that you no longer have permission to create databases.

Alternatively, you could create a shortcut for the menu command. For example, here we connect to a remote server (the -h option) on IP address 192.168.0.3, to database (the -d option) bpsimple, as the user rick (the -U option).

```
"C:\Program Files\PostgreSQL\8.0\bin\psql.exe" -h 192.168.0.3 -d bpsimple -U rick
```

Replace rick with postgres for a connection as the administrative user, and you can omit the -h option if the server is local.

Resolving Startup Problems

If psql complains about pg_shadow, then you have not yet created the supplied username as a database user. If it complains about not knowing the username or lack of permissions, then you may not have granted permissions correctly. Refer to Chapter 3 for details on how to grant permissions.

If you have come unstuck, the easiest way to fix things at this stage is to delete the database and user, and then re-create them. To do so, exit the psql command using the command \q, which will return you to the command-line prompt (Linux) or close the Windows Command Prompt window.

Next, reconnect to the database server as the postgres user. In Linux, do this from the prompt like this:

```
$ psql -d template1
```

On Windows, use the Start menu command psql to template1.

Enter the password you used during the installation process, and you should see this prompt:

```
template1=#
```

Now delete the user database and the user (rick in this example) like this:

```
template1=# DROP DATABASE bpsimple;
DROP DATABASE
template1=# DROP USER rick;
DROP USER
template1=#
```

Next, re-create the user and choose a password (apress4789 in this example):

```
template1=# CREATE USER rick WITH CREATEDB PASSWORD 'apress4789';
CREATE USER
template1=#
```

The CREATEDB option allows users to create their own databases.

Now reconnect to the database, as your newly created user:

```
template1=# \c template1 rick
Password:
template1=#
```

You are now connected to database `template1` as your new user (again, `rick` in this example).

Next, create the `bpsimple` database:

```
template1=> CREATE DATABASE bpsimple;
CREATE DATABASE
template1=>
```

Reattach to the `bpsimple` database as the new user:

```
template1=> \c bpsimple rick
You are now connected to database "bpsimple" as user "rick".
bpsimple=>
```

You will then need to rerun the steps described at the end of the previous chapter, starting from the "Creating the Tables" section, in order to create the sample tables and data we will use in this chapter.

If you see an error message like this when trying to create the database:

```
ERROR:  source database "template1" is being accessed by other users
```

this means that there is some other session attached to the database `template1`—perhaps another `psql` session or a GUI tool such as pgAdmin III. Ensure your current `psql` session is the only one in use, and then try again.

To check if you have the tables created for the `bpsimple` database, enter `\dt`, and then press Return, and you should see output similar to this:

```
bpsimple=> \dt
            List of relations
Schema |     Name     | Type  |  Owner
-------+--------------+-------+----------
public | barcode      | table | rick
public | customer     | table | rick
public | item         | table | rick
public | orderinfo    | table | rick
public | orderline    | table | rick
public | stock        | table | rick
(6 rows)

bpsimple=>
```

The Owner column will show your login name (`rick` in this example).

■ **Note** You may see some tables with names such as `pg_ts_dict`, `pg_ts_parser`, `pg_ts_cfg`, and `pg_ts_cfgmap`. These are additional tables added by some optional user-contributed tools. You can safely ignore them.

You can see the same information in pgAdmin III by navigating to Databases, then bpsimple, then Schemas, then public, and then Tables, as shown in Figure 4-1.

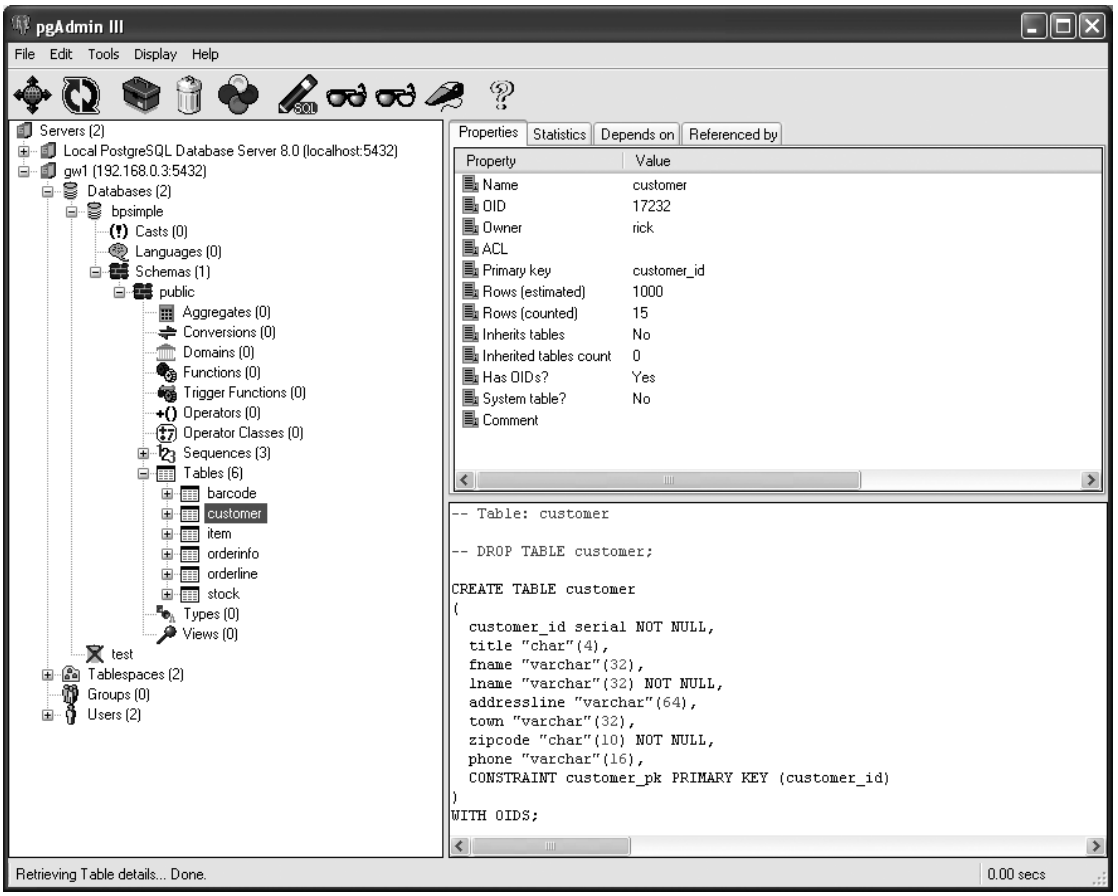


Figure 4-1. *Examining the bpsimple database with pgAdmin III*

We'll cover database and user management in considerable detail in Chapter 11.

Using Some Basic psql Commands

We will use only a few basic `psql` commands in this chapter (we will meet the full set in Chapter 5). For now, the commands you need to know are listed in Table 4-1.

Table 4-1. *Basic psql Commands*

Command	Description
`\?`	Get a help message
`\do`	List operators
`\dt`	List tables
`\dT`	List types
`\h <cmd>`	Get help on a SQL command; replace *<cmd>* with the actual command
`\i <filename>`	Execute commands read from the filename *<filename>*
`\r`	Reset the buffer (discard any typing)
`\q`	Quit `psql`

Each of the commands listed in Table 4-1 must be followed by pressing Return (or Enter). You should also be able to use the arrow keys to recall previous lines and move within lines to edit them. On Linux systems, this feature of `psql` depends on the presence of the GNU `readline` facility, which is usually, but not always, installed.

Now we are ready to start accessing our PostgreSQL database using SQL commands. In the next chapter, we will meet some of the GUI tools that you can use with PostgreSQL, but in this chapter, we will use the `psql` tool.

■ **Note** If you prefer to use the GUI tools, you may want to look ahead to Chapter 5 first. Then you can return to this chapter. You should be able to try all of the examples in this chapter from any GUI tool that allows you to type SQL directly to PostgreSQL, such as pgAdmin III (`http://www.pgadmin.org/`). However, we suggest working through at least this chapter with the command line, because it is often very handy to know the basics of accessing PostgreSQL using only a command-line tool.

Using Simple SELECT Statements

As with all relational databases, we retrieve data from PostgreSQL using the `SELECT` statement. It's probably the most complex statement in SQL, but it really is at the heart of using relational databases effectively.

Let's start our investigation of `SELECT` by simply asking for all the data in a particular table. We do this by using a very basic form of the `SELECT` statement, specifying a list of columns and a `FROM` clause with a table name:

```
SELECT <comma-separated list of columns> FROM <table name>
```

If we can't remember what the exact column names are called, or want to see all the columns, we can just use an asterisk (*) in place of the column list.

■ **Note** In this book, we show SQL keywords that structure the commands in uppercase in the text to make them stand out clearly. SQL is not case-sensitive, although a few implementations do make table names case-sensitive. Data stored in SQL databases is case-sensitive, so the character string `"Newtown"` is different from the character string `"newtown"`.

Try It Out: Select All Columns from a Table

We will start by fetching all the data from the `item` table:

```
SELECT * FROM item;
```

Remember that the semicolon (`;`) is for the benefit of `psql`, to tell it you have finished typing. Strictly speaking, it is not part of SQL. If you prefer, you can terminate SQL statements typed into `psql` with `\g`, which has the same effect as the semicolon. If you are using a different tool to send SQL to PostgreSQL, you may not need either of these terminators.

Enter the command, and you'll see PostgreSQL's response:

```
bpsimple=> SELECT * FROM item;
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       1 | Wood Puzzle   |      15.23 |      21.95
       2 | Rubic Cube    |       7.45 |      11.49
       3 | Linux CD      |       1.99 |       2.49
       4 | Tissues       |       2.11 |       3.99
       5 | Picture Frame |       7.54 |       9.95
       6 | Fan Small     |       9.23 |      15.75
       7 | Fan Large     |      13.36 |      19.95
       8 | Toothbrush    |       0.75 |       1.45
       9 | Roman Coin    |       2.34 |       2.45
      10 | Carrier Bag   |       0.01 |       0.00
      11 | Speakers      |      19.73 |      25.32
(11 rows)
bpsimple=>
```

How It Works

We simply asked PostgreSQL for all the data from all the columns in the `item` table, using an * for the column names. PostgreSQL gave us just that, but neatly arranged with column headings and a pipe (`|`) symbol to separate each column. It even told us how many rows we retrieved.

But suppose we didn't want all the columns? In general, you should ask PostgreSQL, or indeed any relational database, to retrieve only the data you actually want. Each column of

each row that is retrieved adds a little extra work. There is no point in making the server do work unnecessarily; it's always nice to keep things clean and efficient.

You will also find that, once you start having SQL embedded in other languages (see Chapter 14), specifying exact columns will protect you against changes to the database schema. For example, if you use * to retrieve all columns and an additional column had been inserted in a table since the code was tested, you may find that you are processing data from a different column than the one you intended. If a column that you are using is deleted, then the SQL in your program will fail, since the column can no longer be retrieved; however, that is a much easier bug to find and correct than some application code accessing the wrong column while processing data. If you specify the columns by name, you have the option of searching all your code to see if the column names appear, before making changes to the database, and preventing bugs from ever occurring.

Let's try restricting the columns we retrieve. As we saw in the syntax earlier, we do this by specifying each column we want, separated by a comma. If we don't want the columns in the order we specified when we created the database table, that's fine—we can specify the columns in any order we like, and they will be returned in that order.

Try It Out: Select Named Columns in a Specific Order

To retrieve the name of the town and last name of all our customers, we must specify the name of the columns for town and last name, and, of course, the table from which to retrieve them. Here is the statement we need and PostgreSQL's response:

```
bpsimple=> SELECT town, lname FROM customer;
   town    |  lname
-----------+---------
 Hightown  | Stones
 Lowtown   | Stones
 Nicetown  | Matthew
 Yuleville | Matthew
 Oakenham  | Cozens
 Nicetown  | Matthew
 Bingham   | Stones
 Bingham   | Stones
 Histon    | Hickman
 Tibsville | Howard
 Bingham   | Jones
 Winersby  | Neill
 Oxbridge  | Hardy
 Welltown  | O'Neill
 Milltown  | Hudson
(15 rows)

bpsimple=>
```

How It Works

PostgreSQL returns all the data rows from the table we specified, but only from the columns we requested. It also returns the column data in the order in which we specified the columns in the `SELECT` statement.

Overriding Column Names

You will notice that the output uses the database column name as the heading for the output. Sometimes, this is not very easy to read, particularly when the column in the output isn't an actual database column, so it has no name. There is a very simple syntax for specifying the display name (a column name *alias*) to use with each column, which is to add `AS "<display name>"` after each column in the `SELECT` statement. You can specify the names of all columns you select or just a few. Where you don't specify the name, PostgreSQL just uses the column name.

For example, to change the preceding output to add meaningful names, we would use this:

```
SELECT town, lname AS "Last Name" FROM customer;
```

We will see an example of this in use in the next section. It's worth noting that, in the SQL92 standard, the `AS` clause is optional; however, as of release 8.0, PostgreSQL still requires the `AS` keyword.

Controlling the Order of Rows

So far, we have retrieved the data from the columns we wanted, but the data is not always in the most suitable order for viewing. The data we are seeing may look as though it is in the order we inserted it into the database, but that is probably simply because we have not been updating the data by inserting and deleting rows.

As we mentioned in Chapter 2, unlike in a spreadsheet, the order of rows in a database is unspecified. The database server is free to store rows in the most effective way, which is not usually the most natural way for viewing the data. The output you see is not sorted in any meaningful order, and the next time you ask for the same data, it could be displayed in a different order. Generally, the data will be returned in the order it is stored in the database internally. No SQL database, PostgreSQL included, is obliged to return the data in a particular order, unless you specifically request it to be ordered when you retrieve it.

We can control the order in which data is displayed from a `SELECT` statement by adding an `ORDER BY` clause to the `SELECT` statement, which specifies the order we would like the data to be returned. The syntax is as follows:

```
SELECT <comma-separated list of columns> FROM <table name> ORDER BY <column name>
[ASC | DESC]
```

The slightly strange-looking syntax at the end means that after the column name, we can write either `ASC` (short for ascending) or `DESC` (short for descending). By default, ascending order is used. The data is then returned to us ordered by the column we specified, sorted in the direction we requested.

Try It Out: Order the Data

In this example, we will sort the data by town, and we will also override the default column name for the `lname` column, similar to what we saw in the previous section, to make the output slightly easier to read.

Here is the command we require and PostgreSQL's response:

```
bpsimple=> SELECT town, lname AS "Last Name" FROM customer ORDER BY town;
   town    | Last Name
-----------+-----------
 Bingham   | Stones
 Bingham   | Stones
 Bingham   | Jones
 Hightown  | Stones
 Histon    | Hickman
 Lowtown   | Stones
 Milltown  | Hudson
 Nicetown  | Matthew
 Nicetown  | Matthew
 Oakenham  | Cozens
 Oxbridge  | Hardy
 Tibsville | Howard
 Welltown  | O'Neill
 Winnersby | Neill
 Yuleville | Matthew
(15 rows)

bpsimple=>
```

Notice that since we want the data in ascending order, we can omit the `ASC`, as ascending is the default sort order. As you can see, the data is now sorted in ascending order by town.

How It Works

This time, we made two changes to our previous statement. We added an `AS` clause to change the name of the second column to `Last Name`, which makes it easier to read, and we also added an `ORDER BY` clause to specify the order in which PostgreSQL should return the data to us.

Sometimes, we need to go a little further and order the data by more than a single column. For example, in the previous output, although the data is ordered by `town`, there is not much order in the `Last Name`. We can see, for example, that `Jones` is listed after `Stones` under all the customers found in the town `Bingham`.

We can more precisely order the output by specifying more than one column in the `ORDER BY` clause. If we want to, we can even specify that the order is ascending for one column and descending for another column.

Try It Out: Order the Data Using ASC and DESC

Let's try our `SELECT` again, but this time, we will sort the town names into descending order, and then sort the last names into ascending order where rows share the same town name.

The statement we need and PostgreSQL's response are as follows:

```
bpsimple=> SELECT town, lname AS 'Last Name' FROM customer
ORDER BY town DESC, lname ASC;
   town    | Last Name
-----------+-----------
 Yuleville | Matthew
 Winnersby | Neill
 Welltown  | O'Neill
 Tibsville | Howard
 Oxbridge  | Hardy
 Oakenham  | Cozens
 Nicetown  | Matthew
 Nicetown  | Matthew
 Milltown  | Hudson
 Lowtown   | Stones
 Histon    | Hickman
 Hightown  | Stones
 Bingham   | Jones
 Bingham   | Stones
 Bingham   | Stones
(15 rows)
bpsimple=>
```

How It Works

As you can see, PostgreSQL first orders the data by `town` in descending order, which was the first column we specified in our `ORDER BY` clause. It then orders those entries that have multiple last names for the same town in an ascending order. This time, although `Bingham` is now last in the rows retrieved, the last names of our customers in that town are ordered in an ascending fashion.

Usually, the columns by which you can order the output are restricted, not unreasonably, to columns you have requested in the output. PostgreSQL, at least in the current version, does not enforce this standard restriction, and it will accept a column in the `ORDER BY` clause that is not in the selected column list. However, this is nonstandard SQL, and we suggest that you avoid relying on this feature.

Suppressing Duplicates

You may have noticed that there are several duplicate rows in the previous output. For example, the following town and last name rows appear twice:

```
Nicetown  | Matthew
Bingham   | Stones
```

What's going on here? In the original data, there are indeed two customers in Nicetown named Matthew and two customers in Bingham named Stones. For reference, here are the rows, showing the first names as well:

```
Nicetown  | Matthew | Alex
Nicetown  | Matthew | Neil
Bingham   | Stones  | Richard
Bingham   | Stones  | Ann
```

When PostgreSQL listed two rows for Nicetown and Matthew, and two rows for Bingham and Stones, it was quite correct. There are two customers in each of those towns with the same last names. They look exactly the same because we did not ask for the columns that distinguish them.

The default behavior is to list all the rows, but that is not always what we want. For example, we might want just a list of towns where we have customers, perhaps to determine where we should build distribution centers. Based on our knowledge so far, we might reasonably try this:

```
bpsimple=> SELECT town FROM customer ORDER BY town;
   town
-----------
 Bingham
 Bingham
 Bingham
 Hightown
 Histon
 Lowtown
 Milltown
 Nicetown
 Nicetown
 Oakenham
 Oxbridge
 Tibsville
 Welltown
 Winersby
 Yuleville
(15 rows)

bpsimple=>
```

PostgreSQL has listed all the towns, once for each time a town appeared in the `customer` table. This is correct, but it's probably not quite the listing we would like. What we actually need is a list where each town appeared just once; in other words, a list of distinct towns.

In SQL, you can suppress duplicate rows by adding the keyword `DISTINCT` to the `SELECT` statement. The syntax is as follows:

```
SELECT DISTINCT <comma-separated list of columns> FROM <table name>
```

As with pretty much all the clauses on `SELECT`, you can combine this with other clauses, such as renaming columns or specifying an order.

Try It Out: Use DISTINCT

Let's get a list of all the towns that appear in our `customer` table, without duplicates. We can try the following code to get the response:

```
bpsimple=> SELECT DISTINCT town FROM customer;
   town
-----------
 Bingham
 Hightown
 Histon
 Lowtown
 Milltown
 Nicetown
 Oakenham
 Oxbridge
 Tibsville
 Welltown
 Winersby
 Yuleville
(12 rows)

bpsimple=>
```

How It Works

The `DISTINCT` keyword tells PostgreSQL to remove all duplicate rows. Notice that the output is now ordered by town. This is because of the way PostgreSQL has chosen to implement the `DISTINCT` clause for your data. In general, you cannot assume the data will be sorted in this way. If you want the data sorted in a particular way, you must add an `ORDER BY` clause to specify the order.

Notice that the `DISTINCT` clause is not associated with a particular column. You can suppress only rows that are duplicated in all the columns you select, not suppress duplicates of a particular column. For example, suppose that we used this form:

```
SELECT DISTINCT town, fname FROM customer;
```

We would again get 15 rows, because there are 15 different town and first name combinations.

A word of warning is in order here: Although it might look like a good idea to always use `DISTINCT` with `SELECT`, in practice, this is a bad idea for two reasons. First, by using `DISTINCT`, you are asking PostgreSQL to do significantly more work in retrieving your data and checking for duplicates. Unless you know there will be duplicates that need to be removed, you shouldn't use the `DISTINCT` clause. The second reason is a bit more pragmatic. Occasionally, `DISTINCT` will mask errors in your data or SQL that would have been easy to spot if duplicate rows had been displayed.

■ **Caution** Use `DISTINCT` only where you actually need it, because it requires more work and may mask errors.

Performing Calculations

We can also perform simple calculations on data in the rows we retrieve before sending them to the output.

Suppose we wanted to display the cost price of items in our `item` table. We could just execute `SELECT` as shown here:

```
bpsimple=> SELECT description, cost_price FROM item;
  description  |  cost_price
---------------+-------------
 Wood Puzzle   |       15.23
 Rubic Cube    |        7.45
 Linux CD      |        1.99
 Tissues       |        2.11
 Picture Frame |        7.54
 Fan Small     |        9.23
 Fan Large     |       13.36
 Toothbrush    |        0.75
 Roman Coin    |        2.34
 Carrier Bag   |        0.01
 Speakers      |       19.73
(11 rows)

bpsimple=>
```

Suppose we wanted to see the price in cents. We can do a simple calculation in SQL, like this:

```
bpsimple=> SELECT description, cost_price * 100 FROM item;
  description  |  ?column?
---------------+--------------
 Wood Puzzle   |  1523.00
 Rubic Cube    |   745.00
 Linux CD      |   199.00
 Tissues       |   211.00
 Picture Frame |   754.00
 Fan Small     |   923.00
 Fan Large     |  1336.00
 Toothbrush    |    75.00
 Roman Coin    |   234.00
 Carrier Bag   |     1.00
 Speakers      |  1973.00
(11 rows)

bpsimple=>
```

It seems a little weird, with the decimal points and the strange column name, so let's get rid of the decimal points by using a SQL function, and also explicitly name the resulting column. We use the `cast` function to change the type of the column, which, in conjunction with an `AS` clause to name the column, gives us the better-looking output:

```
bpsimple=> SELECT description, cast((cost_price * 100) AS int AS "Cost Price"
FROM item;
description   |  Cost Price
--------------+--------------
Wood Puzzle   |       1523
Rubic Cube    |        745
Linux CD      |        199
Tissues       |        211
Picture Frame |        754
Fan Small     |        923
Fan Large     |       1336
Toothbrush    |         75
Roman Coin    |        234
Carrier Bag   |          1
Speakers      |       1973
(11 rows)

bpsimple=>
```

We'll talk more about the `cast` function later in this chapter, in the "Setting the Time and Date Style" section.

Choosing the Rows

So far in this chapter, we have always worked with all the rows of data, or at least all the distinct rows. It's time to look at how we can choose the specific rows we want to see. You probably won't be surprised to learn that we do this with yet another clause on the `SELECT` statement: the `WHERE` clause.

The simplified syntax for `WHERE` is as follows:

```
SELECT <comma-separated list of columns> FROM <table name> WHERE <conditions>
```

There are many possible conditions, which can also be combined by the keywords `AND`, `OR`, and `NOT`.

The standard comparison operators used in conditions are listed in Table 4-2. The comparison operators can be used on most types, both numeric and string, although there are some special conditions when working with dates, which we will see later in this chapter.

Table 4-2. *Standard Comparison Operators*

Operator	Description
<	Less than
<=	Less than or equal to
=	Equal to
>=	Greater than or equal to
>	Greater than
<>	Not equal to

We will start with a simple condition, choosing to retrieve rows for people who live in the town Bingham:

```
bpsimple=> SELECT town, lname, fname FROM customer WHERE town = 'Bingham';
town    | lname  |  fname
--------+--------+---------
Bingham | Stones | Richard
Bingham | Stones | Ann
Bingham | Jones  | Dave
(3 rows)

bpsimple=>
```

That was pretty straightforward, wasn't it? Notice the single quotes round the string `Bingham`, which are needed to make it clear that this is a string. Also notice that because `Bingham` is being matched against data in the database, it is case-sensitive. If we had used `... town = 'bingham'`, no data would have been returned.

We can have multiple conditions, combined using `AND`, `OR`, and `NOT`, with parentheses to make the expression clear. We can also use conditions on columns that don't appear in the list of columns we have selected. You will remember that this generally isn't true for clauses such as `ORDER BY`.

Try It Out: Use Operators

Let's try a more complicated set of conditions. Suppose we want to see the names of our customers who do not have a title of Mr., but do live in either Bingham or Nicetown. Here is the statement we need and PostgreSQL's response:

```
bpsimple=> SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr'
bpsimple-> AND (town = 'Bingham' OR town = 'Nicetown');
title  | fname |  lname  |   town
-------+-------+---------+----------
 Miss  | Alex  | Matthew | Nicetown
 Mrs   | Ann   | Stones  | Bingham
(2 rows)

bpsimple=>
```

How It Works

Although it might look a little complex at first glance, this statement is actually quite simple. The first part is just our usual `SELECT`, listing the columns we want to see in the output. After the `WHERE` clause, we initially check that the title is not `Mr`. Then, using `AND`, we check that another condition is true. This second condition is that the town is either `Bingham` or `Nicetown`. Notice that we need to use parentheses to make it clear how the clauses are to be grouped.

You should be aware that PostgreSQL, or any other relational database, is not under any obligation to process the clauses in the order you write them in the SQL statement. All that is promised is that the result will be the correct answer to the SQL "question." Generally, relational databases have sophisticated optimizers, which look at the request, and then determine the

optimal way to satisfy it. Optimizers are not perfect, and you will very occasionally come across statements that run better when rewritten in different ways. For reasonably simple statements like this one, we can safely assume the optimizer will do a good job.

■ **Tip** If you want to know how PostgreSQL will process a SQL statement, you can get it to tell you by prefixing the SQL with `EXPLAIN`. Rather than execute the statement, PostgreSQL will tell you how the statement would be processed.

Using More Complex Conditions

One of the things that we frequently need to do when working with strings is to allow partial matching. For example, we may be looking for a person named Robert, but the name may have been shortened in the database to Rob or Bob. There are some special operations in SQL that make working with strings, either partial ones or lists of strings, easier.

The first new condition is `IN`, which allows us to check against a list of items, rather than using a string of `OR` conditions. Consider the following:

```
SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr' AND
(town = 'Bingham' OR town = 'Nicetown');
```

We can rewrite this as follows:

```
SELECT title, fname, lname, town FROM customer WHERE title <> 'Mr' AND
town IN ('Bingham', 'Nicetown');
```

We will get the same result, although it's possible the output rows could be in a different order, since we did not use an `ORDER BY` clause. There is no particular advantage in using the `IN` clause in this case, except for the simplification of the expression. When we meet subqueries in Chapter 7, we will use `IN` again, as it offers more advantages in those situations.

The next new condition is `BETWEEN`, which allows us to check a range of values by specifying the endpoints. Suppose we wanted to select the rows with `customer_id` values between `5` and `9`. Rather than write a sequence of `OR` conditions, or an `IN` with many values, we can simply write this:

```
bpsimple=> SELECT customer_id, town, lname FROM customer WHERE customer_id
BETWEEN 5 AND 9;
customer_id  |   town   |  lname
-------------+----------+---------
           5 | Oahenham | Cozens
           6 | Nicetown | Matthew
           7 | Bingham  | Stones
           8 | Bingham  | Stones
           9 | Histon   | Hickman
(5 rows)
bpsimple=>
```

It's also possible to use BETWEEN with strings; however, you need to be careful, because the answer may not always be exactly what you were expecting, and you must know the case, since, as mentioned earlier, string comparisons are case-sensitive.

Try It Out: Use Complex Conditions

Let's try a BETWEEN statement, comparing strings. Suppose we wanted a distinct list of all the towns that start with letters between *B* and *N*. All of the towns in the customer table start with a capital letter, so we might write the following:

```
bpsimple=> SELECT DISTINCT town FROM customer WHERE town BETWEEN 'B' AND 'N';
   town
----------
 Bingham
 Hightown
 Histon
 Lowtown
 Milltown
(5 rows)

bpsimple=>
```

If you look at these results closely, you'll see that this SQL doesn't work as expected. Where is Newtown? It certainly starts with an *N*, but it hasn't been listed.

Why It Didn't Work

The reason this statement didn't work is that PostgreSQL, as per the SQL standard, pads the string you give it with blanks until it is the same length as the string it is checking against. So when the comparison arrived at Newtown, PostgreSQL compared *N* (*N* followed by six spaces) with Newtown, and because whitespace appears in the ASCII table before all the other letters, it decided the Newtown came after *N* , so it shouldn't be included in the list.

How to Make It Work

It's actually quite easy to make this work as expected. Either we need to prevent the behavior of adding blanks to the search string by adding some additional *z* characters after the *N* or search using the next letter in the alphabet, *O*, in the BETWEEN clause. Of course, if there were a town called *O*, we would then erroneously retrieve it, so you need to be careful using this method. It's generally better to use *z* rather than *Z*, because *z* appears after *Z* in the ASCII table. Thus, our SQL should read as follows:

```
SELECT DISTINCT town FROM customer WHERE town BETWEEN 'B' AND 'Nz';
```

Notice that we didn't add a z after the B, because the B string being padded with blanks does work to find all towns that start with a *B*, since it is the starting point, rather than the endpoint. Also if there were a town that started with the letters *Nzz*, we would again fail to find it, because we would then compare Nz against Nzz, and decide that Nzz came after Nz, because the third string location would have been padded to a space, which comes before the z in the third place of the string we are comparing against.

This type of matching has rather subtle behavior, so if you do use `BETWEEN` with strings, always think carefully about exactly what is being matched.

Pattern Matching

The string-comparison operations we have seen until now are fine as far as they go, but they don't help very much with real-world string pattern matching. The SQL condition for pattern matching is `LIKE`.

Unfortunately, `LIKE` uses a different set of string-comparison rules from all other programming languages we know. However, as long as you remember the rules, it's easy enough to use. When comparing strings with `LIKE`, you use a percent sign (`%`) to mean any string of characters, and you use an underscore (`_`) to match a single character. For example, to match towns beginning with the letter *B*, we would write this:

```
... WHERE town LIKE  'B%'
```

To match first names that end with `e`, we would write this:

```
... WHERE fname LIKE '%e';
```

To match first names that are exactly four characters long, we would use four underscore characters, like this:

```
... WHERE fname LIKE '_ _ _ _';
```

We can also combine the two types in a single string.

Try It Out: Pattern Matching

Let's find all the customers who have first names that have an *a* as the second character. Here is the SQL statement to achieve this:

```
bpsimple=> SELECT fname, lname FROM customer WHERE fname LIKE '_a%';
fname | lname
------+--------
Dave  | Jones
Laura | Hardy
David | Hudson
(3 rows)

bpsimple=>
```

How It Works

The first part of the pattern, `_a`, matches strings that start with any single character, then have a lowercase *a*. The second part of the pattern, `%`, matches any remaining characters. If we didn't use the trailing `%`, only strings exactly two characters long would have been matched.

Limiting the Results

In the examples we have been using so far, the number of result rows returned has always been quite small, because we have only a few sample rows in our sample database. In a real-world database we could easily have many thousands of rows that match the selection criteria. If we are working on our SQL, refining our statements, we almost certainly do not want to see many thousands of rows scrolling past on our screen. A few sample rows to check our logic would be quite sufficient.

PostgreSQL has an extra clause on the `SELECT` statement, `LIMIT`, which is not part of the SQL standard but is very useful when we want to restrict the number of rows returned.

If you append `LIMIT` and a number to your `SELECT` clause, only rows up to the number you specified will be returned, starting from the first row. A slightly different way to use `LIMIT` is in conjunction with the `OFFSET` clause, which specifies a starting position.

It's easier to show it in action than to describe it. Here we display only the first five matching rows:

```
bpsimple=> SELECT customer_id, town FROM customer LIMIT 5;
customer_id  |   town
-------------+-----------
           1 | Hightown
           2 | Lowtown
           3 | Nicetown
           4 | Yuleville
           5 | Oahenham
(5 rows)

bpsimple=>
```

The following skips the first two result rows, then returns the next five rows:

```
bpsimple=> SELECT customer_id, town FROM customer LIMIT 5 OFFSET 2;
 customer_id |   town
-------------+-----------
           3 | Nicetown
           4 | Yuleville
           5 | Oahenham
           6 | Nicetown
           7 | Bingham
(5 rows)

bpsimple=>
```

It's also possible to use `OFFSET` on its own, like this:

```
bpsimple=> SELECT customer_id, town FROM customer OFFSET 12;
 customer_id |   town
-------------+----------
          13 | Oxbridge
          14 | Welltown
          15 | Milltown
(3 rows)

bpsimple=>
```

If you want to combine `LIMIT` with other `SELECT` clauses, the `LIMIT` clause should always appear after the normal `SELECT` statement, followed only by the `OFFSET` clause, if you use it.

Checking for NULL

So far, we do not know a way of checking to see if a column contains a `NULL` value. We can check if it equals a value or string, or if it doesn't equal a value or string, but that's not sufficient.

You will remember from Chapter 2 that `NULL` is a special column value that means either unknown or not relevant. We need to look at checking for `NULL` separately, because it requires special consideration to ensure that the results are as expected.

Suppose we have an integer column `tryint` in a table `testtab` that we know stores `0`, `1`, or `NULL`. We can check if it is `0` by writing this:

```
SELECT * FROM testtab WHERE tryint = 0;
```

We can check if it is `1` by writing this:

```
SELECT * FROM testtab WHERE tryint = 1;
```

We need another check to see if the value is `NULL`. PostgreSQL supports the standard SQL syntax for checking whether a value is `NULL`. We do this with the use of `IS NULL`, like this:

```
SELECT * FROM testtab WHERE tryint IS NULL;
```

Notice that we use the keyword `IS` rather than an `=` sign.

We can also test to see if the value is something other than `NULL` by adding a `NOT` to invert the test:

```
SELECT * FROM testtab WHERE tryint IS NOT NULL;
```

Why do we suddenly need this extra bit of syntax? You are probably familiar with two-valued logic, where everything is either `true` or `false`. What is happening here is that we have stumbled into *three-value logic*, with `true`, `false`, and unknown.

Unfortunately, this property of `NULL`, being unknown, has some other effects outside the immediate concern of checking for `NULL`.

Suppose we ran our statement on a table where some values of `tryint` were `NULL`:

```
SELECT * FROM testtab WHERE tryint = 1;
```

What does our `tryint = 1` mean when `tryint` is actually `NULL`? We are asking the question, "Is unknown equal to 1?" This is interesting, because we can't know that the statement is `false`, but neither can we know it to be `true`. So the answer must be unknown, hence the rows where `NULL` appears are not matched. If we reversed the test, and compared `tryint != 1`, the rows with `NULL` would also not be found, because that condition would not be `true` either. This can be confusing, because we have apparently used two tests, with opposite conditions, and still not retrieved all the rows from the table.

It's important to be aware of these issues with `NULL`, because it's all too easy to forget about `NULL` values. If you start getting slightly unexpected results when using conditions on a column that can have `NULL` values, verify whether the rows consisting of the `NULL` value are the cause of your problems.

Checking Dates and Times

PostgreSQL has two basic types for handling date and time information: `timestamp`, which holds both a date and a time; and `date`, which holds day, month, and year information. PostgreSQL has some built-in functions that help us work with dates and time, which are traditionally rather difficult units to manipulate. Here, we'll concentrate on those that are most commonly used. (You can find all of the built-in functions listed in the online documentation.)

Before we start, we need to tackle one of those apparently trivial problems that can so easily cause confusion: how do we specify a date?

When we write the date 1/2/2005, what do we mean? Europeans generally mean the first day of February 2005, but Americans usually mean the second day of January 2005. This is because Europeans generally read dates as *DD*/*MM*/*YYYY*, but Americans expect *MM*/*DD*/*YYYY*. Just to add to the confusion, the ISO standard 8601 (officially adopted in Europe as European Standard EN 28601) specifies the logical (but rarely seen in everyday use) date format *YYYY-MM-DD*.

PostgreSQL lets you change the way dates are handled to suit your local needs, so before we get into checking dates and time, it's probably sensible to look at how you control this aspect of PostgreSQL's behavior.

Setting the Time and Date Style

Unfortunately, PostgreSQL's method of setting the way the date and time are handled is, at first sight at least, a little strange.

There are two things you can control:

- The order in which days and months are handled, United States or European style
- The display format; for example, numbers only or more textual date output

The unfortunate part of the story is that PostgreSQL uses the same variable to handle both settings, which frankly, can be a bit confusing. The good news is that PostgreSQL defaults to using the unambiguous ISO-8601 style date and time output, which is always written in the form *YYYY-MM-DD hh*:*mm*:*ss.ssTZD*. This gives you the year, month, day, hours, minutes, seconds, decimal part of a second to two places, and a time zone designator, which indicates a

plus or minus number of hours difference between local time and UTC. For example, a full date and time would look like `2005-02-01 05:23:43.23+5`, which equates to February 1, 2005, at 23 minutes and 43.23 seconds past five o'clock in the morning, and the time zone used is five hours ahead of UTC. If you specify the time in UTC, with no time zone, the standard says you should use a *Z* (pronounced "Zulu") to indicate this, although it seems common to omit the *Z*.

For input in the form *NN*/*NN*/*NNNN*, PostgreSQL defaults to expecting the month before the day (United States style). For example, 2/1/05 is the first of February. Alternatively, you can use a format like February 1, 2005, or the ISO style 2005-02-01. If that behavior is all you need, you are in luck—you don't need to know any more about controlling how PostgreSQL accepts and displays dates, and you can skip ahead to the next section on date and time functions.

The default style is actually controlled by the `postgresql.conf` file, in the data directory, which has a line of the form `datestyle = 'iso, mdy'`. So, you could change the default globally, if you wish.

If you need more control over how dates are handled, PostgreSQL does allow this, but it can be a little tricky. The confusing thing is that there are two independent features to control, and you set them both using `datestyle`. Do remember, however, that this is all to do with presentation. Internally, PostgreSQL stores dates in a way totally independent of any representation that users expect when data is input or retrieved.

The syntax as a command to `psql` is as follows:

```
SET datestyle TO 'value';
```

To set the order in which months and days are handled, you set the `datestyle` value to either `US`, for month-first behavior (02/01/1997, for February 1) or `European` for day-first behavior (01/02/1997 for February 1).

To change the display format, you also set `datestyle`, but to one of four different values:

- `ISO` for the ISO-8601 standard, using - as the field separator, as in 1997-02-01
- `SQL` for the traditional style, as in 02/01/1997
- `Postgres` for the default PostgreSQL style, as in Sat Feb 01
- `German` for the German style, as in 01.02.1997

■ **Note** In the current release, the `Postgres` format defaults to displaying in `SQL` style for both date and timestamps.

You set the `datestyle` variable to the value pair by separating the values with a comma. So, for example, to specify that we want dates shown in SQL style, but using the European convention of day before month, we use this setting:

```
SET datestyle TO 'European, SQL';
```

Rather than set the date-handling style locally in session, you can set it for an entire installation, or set the default for a session. If you want to set the default style for date input for a complete installation, you can set the environment variable `PGDATESTYLE` before starting the

postmaster master server process. Setting the same options using the environment variable in Linux, we would use the following:

```
PGDATESTYLE="European, SQL"
export PGDATESTYLE
```

A much better way of changing the default date handling is to edit the configuration file postgresql.conf (found in the data subdirectory of your installation), and set an option such as datestyle='European, SQL' or datestyle = 'iso, mdy', depending on your preferences. You will need to restart the PostgreSQL server after making this change for it to become effective.

If you want to set the date style for individual users, you use the same environment variable, but set for the local user, before psql is invoked. A local setting will override any global setting you have made.

Before we demonstrate how this all works, it's very handy to remember the special PostgreSQL function cast, which allows you to convert one format into another. We saw it briefly earlier in this chapter, when we looked at doing calculations in the SELECT statement, but there is much more to it than the simple conversion to integer we saw earlier. Although PostgreSQL does a pretty good job of getting types correct, and you shouldn't need conversion functions often, they can be very useful occasionally. The conversion we need to use to investigate our date and time values is the following to get a date:

```
cast('string' AS date)
```

Alternatively, to get a value that includes a time:

```
cast('string' AS timestamp)
```

We are also going to use a little trick to demonstrate this function more easily. Almost every time you use the SELECT statement, you will be fetching data from a table. However, you can use SELECT to get data that isn't in a table at all, as in this example:

```
bpsimple=> SELECT 'Fred';
?column?
--------------
 Fred
(1 row)

bpsimple=>
```

PostgreSQL is warning us that we haven't selected any columns, but it is quite happy to accept the SELECT syntax without a table name, and just returns the string we specified.

We can use the same feature, in conjunction with cast, to see how PostgreSQL treats dates and time, without needing to create a temporary table to experiment with.

Try It Out: Set Date Formats

We start off with the environment variable PGDATESTYLE unset, so you can see the default behavior, and then set the date style, so you can see how things progress. We enter a date in ISO format, so this is always a safe option, and PostgreSQL outputs in the same format. There is no ambiguity in either of these statements:

```
bpsimple=> SELECT cast('2005-02-1' AS date);
?column?
------------
2005-02-01
(1 row)
bpsimple=>
```

Changing the style to US and SQL format, we still enter the date in ISO style, which is unambiguous; it's still the first of February, but now the output shows the more conventional, but possibly ambiguous, United States style *MM*/*DD*/*YYYY* output:

```
bpsimple=> SET datestyle TO 'US, SQL';
SET VARIABLE
bpsimple=> SELECT cast('2005-02-1' AS date);
?column?
------------
02/01/2005
(1 row)
bpsimple=>
```

Now is a good time to also ask psql what the internal variable datestyle is set to:

```
bpsimple=> SHOW datestyle;
 DateStyle
-----------
 SQL, MDY
(1 row)
bpsimple=>
```

In older versions of PostgreSQL, the output was more verbose, so you may see something slightly different, depending on which version of PostgreSQL you are using.

Now let's try some more formats:

```
bpsimple=> SET datestyle TO 'European, SQL';
SET
bpsimple=> SELECT cast('2005 02 1' AS date);
    date
------------
 01/02/2005
(1 row)

bpsimple=>
```

With the output set to European, the display changes to *DD*/*MM*/*YYYY*.

Let's go back to ISO input and output:

```
bpsimple=> SET datestyle TO 'European, ISO';
SET
bpsimple=> SELECT cast('2005-02-1' AS date);
    date
------------
 2005-02-01
(1 row)

bpsimple=>
```

The `European` setting has no effect, because ISO is the same for all locales.

Now let's consider times. We use the `timestamp` type, which displays time:

```
bpsimple=> SELECT cast('2005-02-1' AS timestamp);
        ?column?
------------------------
 2005-02-01 00:00:00
(1 row)

bpsimple=>
```

We didn't specify any hours or minutes, so they all default to zero.

Let's try PostgreSQL-style output:

```
bpsimple=> SET datestyle TO 'European, Postgres';
SET
bpsimple=> SELECT cast('2005-02-1' AS timestamp);
        timestamp
--------------------------
 Tue 01 Feb 00:00:00 2005
(1 row)

bpsimple=>
```

This produces output that is unambiguous and rather more reader-friendly.

How It Works

As you can see, we can vary the way both dates and times are displayed, as well as how ambiguous input strings, such as `01/02/2005`, are interpreted.

Time zones are much simpler than date formats. Providing that your local environment variable `TZ` or configuration option in `postgresql.conf` is correctly set, PostgreSQL will manage time zones without further ado.

Using Date and Time Functions

Now that we have seen how dates work, we can look at a couple of useful functions you might need when comparing dates:

- date_part(*units required, value to use*) allows you to extract a particular component of a date, such as the month.
- now simply gets the current date and time, and is equivalent to the more standard "magic variable" current_timestamp.

Suppose we wanted to select the rows from our orderinfo table where the date the order was placed is in September. We know September is the ninth month; therefore, we just ask the following:

```
bpsimple=> SELECT * FROM orderinfo WHERE date_part('month',date_placed)=9;
 orderinfo_id | customer_id | date_placed | date_shipped | shipping
--------------+-------------+-------------+--------------+----------
            3 |          15 | 02-09-2004  | 12-09-2004   |     3.99
            4 |          13 | 03-09-2004  | 10-09-2004   |     2.99
(2 rows)

bpsimple=>
```

PostgreSQL extracts the appropriate rows for us. Note the date is being displayed in ISO format. We can extract the following parts from a date or timestamp:

- Year
- Month
- Day
- Hour
- Minute
- Second

We can also compare dates, using the same operators that we can use with numbers: <>, <=, <, >, >=, and =. Here is an example:

```
bpsimple=> SELECT * FROM orderinfo WHERE date_placed >= cast('2004 07 21' AS date);
 orderinfo_id | customer_id | date_placed | date_shipped | shipping
--------------+-------------+-------------+--------------+----------
            3 |          15 | 02-09-2004  | 12-09-2004   |     3.99
            4 |          13 | 03-09-2004  | 10-09-2004   |     2.99
            5 |           8 | 21-07-2004  | 24-07-2004   |     0.00
(3 rows)

bpsimple=>
```

Notice that we need to convert our string to a date, using the cast operation, and that we stick to the unambiguous ISO style dates.

The second function, now, simply gives us the current date and time, which would be handy if, for example, we were adding a new row for an order being placed while the customer was on the phone, or in real time over the Internet.

```
bpsimple=> SELECT now(), current_timestamp;
               now                |            timestamptz
----------------------------------+----------------------------------
 Sat 16 Oct 13:46:05.99938 2004 BST | Sat 16 Oct 13:46:05.99938 2004 BST
(1 row)

bpsimple=>
```

We can also do simple calculations using dates. For example, to discover the number of days between an order being placed and shipped, we could use a query like this:

```
bpsimple=> SELECT date_shipped - date_placed FROM orderinfo;
 ?column?
---------
        4
        1
       10
        7
        3
(5 rows)

bpsimple=>
```

This returns the number of days between the two dates stored in the database.

■ **Note** More extensive details on PostgreSQL's handling of dates, times, time zones, and related conversion functions can be found in the online documentation.

Working with Multiple Tables

By now, you should have a good idea of how we can select data from a table, picking which columns we want, which rows we want, and how to control the order of the data. We have also seen how to perform simple calculations, make type conversions, and handle the rather special date and time formats.

It's now time to move on to one of the most important features of SQL, and indeed, relational databases in general: relating data in one table to data in another table automatically. The good news is that it's all done with the `SELECT` statement, and everything that you have learned so far about `SELECT` is just as true with many tables as it was with a single table.

Relating Two Tables

Before we look at the SQL for using many tables at the same time, let's have a quick recap of the material we saw in Chapter 2 about relating tables.

You will remember that we have a customer table, which stores details of our customers, and an orderinfo table, which stores details of the orders they have placed. This allows us to store details of each customer only once, no matter how many orders they placed. We linked the two tables together by having a common piece of data, the customer_id, stored in both tables.

If we think about this as a picture, we could imagine a row in the customer table, which has a customer_id, being related to none, one, or many rows in the orderinfo table, where the same customer_id value appears, as illustrated in Figure 4-2.

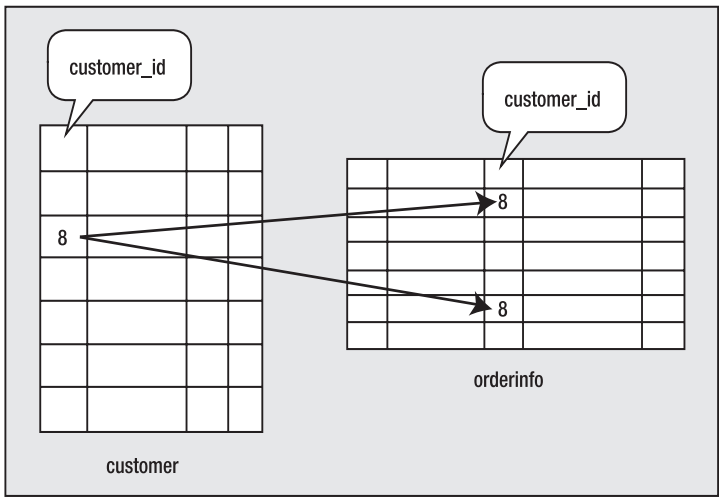


Figure 4-2. *The customer table and orderinfo table relationship*

We could say that the value 8 for customer_id in the row in the customer table relates to two rows in the orderinfo table, where the column customer_id also appears. Of course, we didn't need to have the two columns with the same name, but given that they both store the customer's ID, it would have been very confusing, and inconsistent, to give them different names.

Suppose we wanted to find all the orders that had been placed by our customer Ann Stones. Logically, what we do is first look in our customer table to find this customer:

```
bpsimple=> SELECT customer_id FROM customer WHERE fname = 'Ann'
AND lname = 'Stones';
customer_id
-------------
           8
(1 row)

bpsimple=>
```

Now that we know the customer_id, we can check for orders from this customer:

```
bpsimple=> SELECT * FROM orderinfo WHERE customer_id = 8;
 orderinfo_id | customer_id | date_placed | date_shipped | shipping
--------------+-------------+-------------+--------------+----------
            2 |           8 | 23-06-2004  | 24-06-2004   |     0.00
            5 |           8 | 21-07-2004  | 24-07-2004   |     0.00
(2 rows)

bpsimple=>
```

This worked, but it took two steps, and we had to remember the `customer_id` between steps. As we explained in Chapter 2, SQL is a declarative language; that is, you tell SQL what you want to achieve, rather than explicitly defining the steps of how to get to the solution. What we have just done is to use SQL in a procedural way. We specified two discrete steps to get to our answer and discover the orders placed by a single customer. Wouldn't it be more elegant to do it all in one step?

Indeed, in SQL we can do this all in a single step, by specifying that we want to know the orders placed by `Ann Stones` and that the information is in the `customer` table and `orderinfo` table, which are related by the `customer_id` column that appears in both tables.

The new bit of SQL syntax we need to do this is an extension to the `WHERE` clause:

```
SELECT <column list> FROM <table list> WHERE <join condition>
AND <row-selection conditions>
```

That looks a little complex, but actually it's quite easy. Just to make our first example a little simpler, let's assume we know the customer ID is `8`, and just fetch the order date(s) and customer first name(s). We need to specify the columns we want, the customer first name, the date the order was placed, that the two tables are related by `customer_id` column, and that we want only rows where the `customer_id` is `8`.

You will immediately realize we have a slight problem. How do we tell SQL which `customer_id` we want to use: the one in the `customer` table or the one in the `orderinfo` table? Although we are about to check that they are equal, this might not always be the case, so how do we handle columns whose name appears in more than one table? We simply specify the column name using the extended syntax: *`tablename.columnname`*. We can then unambiguously describe every column in our database.

In general, PostgreSQL is quite forgiving, and if a column name appears in only one table in the `SELECT` statement, we don't need to explicitly use the table name as well. In this case, we will use `customer.fname`, even though `fname` would have been sufficient, because it's a little easier to read, especially when you are learning SQL. The first part of our statement, therefore, needs to be:

```
SELECT customer.fname, orderinfo.date_placed FROM customer, orderinfo
```

That indicates to PostgreSQL the columns and tables we wish to use.

Now we need to specify our conditions. We have two different conditions: that the `customer_id` is `8` and that the two tables are related, or joined, using `customer_id`. Just as we saw earlier with multiple conditions, we do this by using the keyword `AND` to specify multiple conditions that must all be `true`:

```
WHERE customer.customer_id = 8 AND customer.customer_id = orderinfo.customer_id;
```

Notice that we need to tell SQL a specific customer_id column, using the *tablename.columnname* syntax, even though, in practice, it would not matter which of the two tables' customer_id column were checked against 8, since we also specify that they must have the same value. Putting it all together, the statement we need is as follows:

```
bpsimple=> SELECT customer.fname, orderinfo.date_placed
 FROM customer, orderinfo
 WHERE customer.customer_id = 8
 AND customer.customer_id = orderinfo.customer_id;

 fname | date_placed
-------+-------------
 Ann   | 2004-06-23
 Ann   | 2004-07-21
(2 rows)

bpsimple=>
```

It's much more elegant than multiple steps, isn't it? Perhaps more important, by specifying the entire problem in a single statement, we allow PostgreSQL to fully optimize the way the data is retrieved.

Try It Out: Relate Tables

Now we know the principle, let's try our original question and find all the orders placed by Ann Stones, assuming we don't know the customer_id.

We now only know a name, rather than a customer ID; therefore, our SQL is slightly more complex. We must specify the customer by name:

```
bpsimple=> SELECT customer.fname, orderinfo.date_placed
FROM customer, orderinfo
WHERE customer.fname = 'Ann' AND customer.lname = 'Stones'
AND customer.customer_id = orderinfo.customer_id;
 fname | date_placed
-------+-------------
 Ann   | 2004-06-23
 Ann   | 2004-07-21
(2 rows)

bpsimple=>
```

How It Works

Just as we saw in our earlier example, we specify the columns we want, (customer.fname, orderinfo.date_placed), the tables involved (customer, orderinfo), the selection conditions (customer.fname = 'Ann' AND customer.lname = 'Stones'), and how the two tables are related (customer.customer_id = orderinfo.customer_id).

SQL does the rest for us. It doesn't matter if the customer has placed no orders, one order, or many orders. SQL is perfectly happy to execute the SQL query, provided it's valid, even if there are no rows that match the condition.

Let's now look at another example. Suppose we want to list all the products we have, with their barcodes. You will remember that barcodes are held in the `barcode` table, and items are stored in the `item` table. The two tables are related by having an `item_id` column in each table. You may also remember that the reason we split this out into two tables is that many products, or items, actually have multiple barcodes.

Using our newfound expertise in joining tables, we know that we need to specify the columns we want, the tables, and how they are related, or joined together. Being confident, we also decide to order the result by the cost price of the item:

```
bpsimple=> SELECT description, cost_price, barcode_ean FROM item, barcode
  WHERE barcode.item_id = item.item_id ORDER BY cost_price;
  description  | cost_price |  barcode_ean
---------------+------------+---------------
 Toothbrush    |       0.75 | 6241234586487
 Toothbrush    |       0.75 | 9473625532534
 Toothbrush    |       0.75 | 9473627464543
 Linux CD      |       1.99 | 6264537836173
 Linux CD      |       1.99 | 6241527746363
 Tissues       |       2.11 | 7465743843764
 Roman Coin    |       2.34 | 4587263646878
 Rubic Cube    |       7.45 | 6241574635234
 Picture Frame |       7.54 | 3453458677628
 Fan Small     |       9.23 | 6434564564544
 Fan Large     |      13.36 | 8476736836876
 Wood Puzzle   |      15.23 | 6241527836173
 Speakers      |      19.73 | 9879879837489
 Speakers      |      19.73 | 2239872376872
(14 rows)

bpsimple=>
```

This looks reasonable, except several items seem to appear more than once, and we don't remember stocking two different speakers. Also, we don't remember stocking that many items. What's going on here?

Let's count the number of items we stock, using our newfound SQL skills:

```
bpsimple=> SELECT * FROM item;
```

PostgreSQL responds with the data, showing 11 rows. (More experienced SQL users would use the more efficient `SELECT count(*) FROM item`; this function is introduced in Chapter 7.)

We stock only 11 items, but our earlier query found 14 rows. Did we make a mistake?

No, all that's happened is that for some items, such as `Toothbrush`, there are many different barcodes against a single product. PostgreSQL simply repeated the information from the `item` table against each barcode, so that it listed all the barcodes and the item each one belonged to.

You can check this out by also selecting the item ID, by adding it to the SELECT statement, like this:

```
bpsimple=> SELECT item.item_id, description, cost_price, barcode_ean
  FROM item, barcode
  WHERE barcode.item_id = item.item_id ORDER BY cost_price;
 item_id |  description  | cost_price |  barcode_ean
---------+---------------+------------+---------------
       8 | Toothbrush    |       0.75 | 6241234586487
       8 | Toothbrush    |       0.75 | 9473625532534
       8 | Toothbrush    |       0.75 | 9473627464543
       3 | Linux CD      |       1.99 | 6264537836173
       3 | Linux CD      |       1.99 | 6241527746363
       4 | Tissues       |       2.11 | 7465743843764
       9 | Roman Coin    |       2.34 | 4587263646878
       2 | Rubic Cube    |       7.45 | 6241574635234
       5 | Picture Frame |       7.54 | 3453458677628
       6 | Fan Small     |       9.23 | 6434564564544
       7 | Fan Large     |      13.36 | 8476736836876
       1 | Wood Puzzle   |      15.23 | 6241527836173
      11 | Speakers      |      19.73 | 9879879837489
      11 | Speakers      |      19.73 | 2239872376872
(14 rows)

bpsimple=>
```

Notice that we have specified precisely which table item_id comes from, since it appears in the item table as well as the barcode table.

It is now clear what exactly is going on. If the data you get returned from a SELECT statement looks a little odd, it's often a good idea to add all the id type columns to the SELECT statement, just to see what is happening.

Aliasing Table Names

Earlier in the chapter, we saw how we could change column names in the output using AS to give more descriptive names. It's also possible to alias table names, if you wish. This is handy in a few special cases, where you need two names for the same table, but more commonly, it is used to save on typing. You will also see it used frequently in GUI tools, where it makes SQL generation a little easier.

To alias a table name, you simply put the alias name immediately after the table name in the FROM part of the SQL clause. Once you have done this, you can use the alias name, rather than the real table name, in the rest of the SQL statement.

Suppose we had this simple SQL statement:

```
SELECT lname FROM customer;
```

As we saw earlier, you can explicitly name the column by preceding it with the table name, like this:

```
SELECT customer.lname FROM customer;
```

If we alias the `customer` table to `cu`, we could instead prefix the column with `cu` like this:

```
SELECT cu.lname FROM customer cu;
```

Notice that we have added a `cu` directly after the table name, as well as prefixing the column with `cu`.

When a single table is involved, aliasing table names is not very interesting. With multiple tables, it starts to be a bit more useful. Consider our earlier query:

```
SELECT customer.fname, orderinfo.date_placed FROM customer, orderinfo WHERE
customer.fname = 'Ann' AND customer.lname = 'Stones' AND customer.customer_id =
orderinfo.customer_id;
```

With aliases for table names, we could write this as follows:

```
SELECT cu.fname, oi.date_placed FROM customer cu, orderinfo oi
WHERE cu.fname = 'Ann'
AND cu.lname = 'Stones' AND cu.customer_id = oi.customer_id;
```

Aliases table names can be useful both to make the SQL clearer and to avoid typing long table names many times in a complex query.

Relating Three or More Tables

Now that we know how to relate two tables together, can we extend the idea to three or even more tables? Yes, we can. SQL is a very logical language, so if we can do something with *N* items, we can almost always do it with *N*+1 items. Of course, the more tables you include, the more work PostgreSQL needs to do, so queries with many tables can be rather slow, especially if many of the tables have very large numbers of rows.

Suppose we wanted to relate customer information to actual item IDs ordered?

If you look at our schema in Figure 4-3, you will see we need to use three tables to get from the customer to the actual ordered items: `customer`, `orderinfo`, and `orderline`. Redrawing our earlier diagram with three tables, it would look like Figure 4-4.

Here, we can see that customer `123` matches several rows in the `orderinfo` table—those with `orderinfo` IDs of `579`, `426`, `723`, and `114`—and each of these, in turn, relates to one or more rows in the `orderline` table. Notice that there is no direct relationship between `customer` and `orderline`. We must use the `orderinfo` table, since that contains the information that binds the customers to their orders.

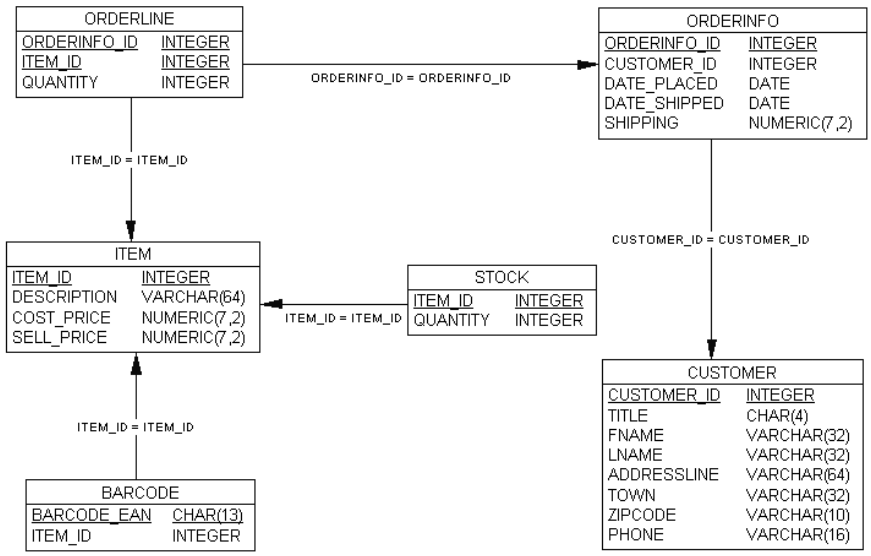


Figure 4-3. *Database schema*

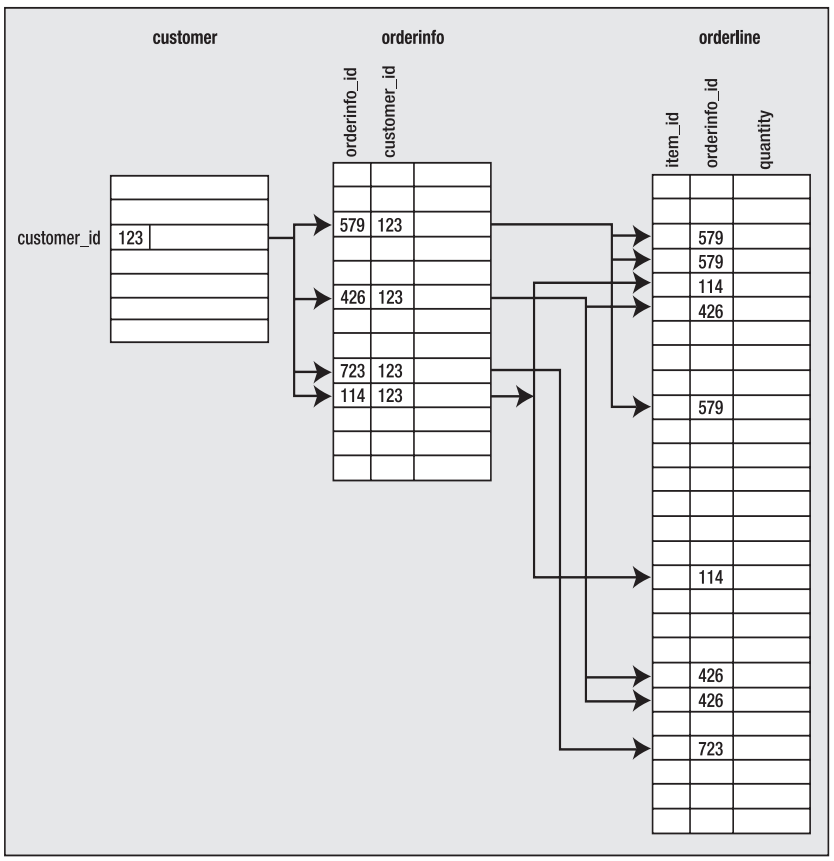


Figure 4-4. *Three related tables*

Try It Out: Join Multiple Tables

Let's first build a three-table join to discover the `item_id`s for Ann Stones's orders. We start with the columns we need:

```
SELECT customer.fname, customer.lname, orderinfo.date_placed,
  orderline.item_id, orderline.quantity
```

Then we list the tables involved:

```
FROM customer, orderinfo, orderline
```

Then we specify how the `customer` and `orderinfo` tables are related:

```
WHERE customer.customer_id = orderinfo.customer_id
```

We must also specify how the `orderinfo` and `orderline` tables are related:

```
orderinfo.orderinfo_id = orderline.orderinfo_id
```

Now our conditions:

```
customer.fname = 'Ann' AND customer.lname = 'Stones';
```

Putting them all together, and spreading the typing over several lines (notice the `bpsimple ->` continuation prompt), we get this:

```
bpsimple=> SELECT customer.fname, customer.lname, orderinfo.date_placed,
bpsimple-> orderline.item_id,orderline.quantity
bpsimple-> FROM customer, orderinfo, orderline
bpsimple-> WHERE
bpsimple-> customer.customer_id = orderinfo.customer_id AND
bpsimple-> orderinfo.orderinfo_id = orderline.orderinfo_id AND
bpsimple-> customer.fname = 'Ann' AND
bpsimple-> customer.lname = 'Stones';
 fname | lname  | date_placed | item_id | quantity
-------+--------+-------------+---------+----------
 Ann   | Stones | 2004-06-23  |       1 |        1
 Ann   | Stones | 2004-06-23  |       4 |        2
 Ann   | Stones | 2004-06-23  |       7 |        2
 Ann   | Stones | 2004-06-23  |      10 |        1
 Ann   | Stones | 2004-07-21  |       1 |        1
 Ann   | Stones | 2004-07-21  |       3 |        1
(6 rows)

bpsimple=>
```

Notice that whitespace outside strings is not significant to SQL, so we can add extra spaces and line breaks to make the SQL easier to read. The `psql` program just issues a continuation prompt, `bpsimple->`, and waits till it sees a semicolon before it tries to interpret what we have been typing.

Having seen how easy it is to go from two tables to three tables, let's take our query a step further and list all the items by description that our customer Ann Stones has ordered. To do this,

we need to use an extra table, the item table, to get at the item description. The rest of the query however, is pretty much as before:

```
bpsimple=> SELECT customer.fname, customer.lname, orderinfo.date_placed,
bpsimple-> item.description, orderline.quantity
bpsimple-> FROM customer, orderinfo, orderline, item
bpsimple-> WHERE
bpsimple-> customer.customer_id = orderinfo.customer_id AND
bpsimple-> orderinfo.orderinfo_id = orderline.orderinfo_id AND
bpsimple-> orderline.item_id = item.item_id AND
bpsimple-> customer.fname = 'Ann' AND
bpsimple-> customer.lname = 'Stones';
 fname | lname  | date_placed | description | quantity
-------+--------+-------------+-------------+----------
 Ann   | Stones | 2004-06-23  | Wood Puzzle |        1
 Ann   | Stones | 2004-06-23  | Tissues     |        2
 Ann   | Stones | 2004-06-23  | Fan Large   |        2
 Ann   | Stones | 2004-06-23  | Carrier Bag |        1
 Ann   | Stones | 2004-07-21  | Wood Puzzle |        1
 Ann   | Stones | 2004-07-21  | Linux CD    |        1
(6 rows)

bpsimple=>
```

How It Works

Once you have seen how three-table joins work, it's not difficult to extend the idea to more tables. We added the item description to the list of columns to be shown, added the item table to the list of tables to select from, and added the information about how to relate the item table to the tables we already had, orderline.item_id = item.item_id. You will notice that Wood Puzzle is listed twice, since it was purchased on two different occasions.

In this SELECT, we have displayed at least one column from each of the tables we used in our join. There is actually no need to do this. If we had just wanted the customer name and item description, we could have simply chosen not to retrieve the columns we didn't need.

A version retrieving fewer columns is just as valid, and may be marginally more efficient than our earlier attempt:

```
SELECT customer.fname, customer.lname, item.description
FROM customer, orderinfo, orderline, item
WHERE
    customer.customer_id = orderinfo.customer_id AND
    orderinfo.orderinfo_id = orderline.orderinfo_id AND
    orderline.item_id = item.item_id AND
    customer.fname = 'Ann' AND
    customer.lname = 'Stones';
```

To conclude this example, let's go back to something we learned early in the chapter: how to remove duplicate information using the DISTINCT keyword.

Try It Out: Add Extra Conditions

Suppose we want to discover what type of items Ann Stones bought. All we want listed are the descriptions of items purchased, ordered by the description. We don't even want to list the customer name, since we know that already (we are using it to select the data). We need to select only the `item.description`, and we also need to use the `DISTINCT` keyword, to ensure that `Wood Puzzle` is listed only once, even though it was bought several times:

```
bpsimple=> SELECT DISTINCT  item.description
bpsimple-> FROM customer, orderinfo, orderline, item
bpsimple-> WHERE
bpsimple-> customer.customer_id = orderinfo.customer_id AND
bpsimple-> orderinfo.orderinfo_id = orderline.orderinfo_id AND
bpsimple-> orderline.item_id = item.item_id AND
bpsimple-> customer.fname = 'Ann' AND
bpsimple-> customer.lname = 'Stones'
bpsimple-> ORDER BY item.description;
 description
-------------
 Carrier Bag
 Fan Large
 Linux CD
 Tissues
 Wood Puzzle
(5 rows)

bpsimple=>
```

How It Works

We simply take our earlier SQL, remove the columns we no longer need, add the `DISTINCT` keyword after `SELECT` to ensure each row appears only once, and add our `ORDER BY` condition after the `WHERE` clause.

That's one of the great things about SQL: once you have learned a feature, it can be applied in a general way. `ORDER BY`, for example, works with many tables in just the same way as it works with a single table.

The SQL92 SELECT Syntax

You may have noticed that the `WHERE` clause actually has two slightly different jobs. It specifies the conditions to determine which rows we wish to retrieve (`customer.fname = 'Ann'`) but also specifies how multiple tables relate to each other (`customer.customer_id = orderinfo.customer_id`).

This didn't really cause anyone any problems for many years, until the SQL standards committee tried to extend the syntax to help handle the increasingly complex jobs to which SQL was being put. When the SQL92 standard was released, a new form of the `SELECT` statement syntax was added to separate these two subtly different uses. This new syntax (sometimes referred to as the SQL92/99 syntax, or the ANSI syntax) was surprisingly slow to catch on with

many SQL databases. Microsoft was an early adopter in SQL Server 6.5, and PostgreSQL added support in version 7.1, but it took Oracle till version 9 to support the new syntax.

The new syntax uses the `JOIN … ON` syntax to specify how tables relate, leaving the `WHERE` clause free to concentrate on which rows to select. The new syntax moves the linking of tables into the `FROM` section of the `SELECT` statement, away from the `WHERE` clause. So the syntax changes from this:

```
SELECT <column list> FROM <table list>
    WHERE <join condition> <row-selection conditions>
```

to this:

```
SELECT <column list> FROM <table> JOIN <table> ON <join condition>
    WHERE <row-selection conditions>
```

It's easier than it looks—really! Suppose we wanted to join the `customer` and `orderinfo` tables, which share a common key of `customer_id`. Instead of writing the following:

```
FROM customer, orderinfo WHERE customer.customer_id = orderinfo.customer_id
```

we would write this:

```
FROM customer JOIN orderinfo ON customer.customer_id = orderinfo.customer_id
```

This is slightly more long-winded, but it is both clearer and an easier syntax to extend, as we will see when we look at outer joins in Chapter 7.

Extensions to more than two tables are straightforward. Consider our earlier query:

```
SELECT customer.fname, customer.lname, item.description
FROM customer, orderinfo, orderline, item
WHERE
    customer.customer_id = orderinfo.customer_id AND
    orderinfo.orderinfo_id = orderline.orderinfo_id AND
    orderline.item_id = item.item_id AND
    customer.fname = 'Ann' AND
    customer.lname = 'Stones';
```

In the SQL92 syntax, this becomes:

```
SELECT customer.fname, customer.lname, item.description
FROM customer
    JOIN orderinfo ON customer.customer_id = orderinfo.customer_id
    JOIN orderline ON orderinfo.orderinfo_id = orderline.orderinfo_id
    JOIN item ON orderline.item_id = item.item_id
WHERE
    customer.fname = 'Ann' AND
    customer.lname = 'Stones';
```

Both versions of the `SELECT` statement produce identical results.

However, many users seem to have stuck with the earlier syntax, which is still valid and slightly more succinct for many SQL statements. We present the newer SQL92 version here, so you will be familiar with the syntax, but generally in this book, we will stick with the older-style joins, except where we meet outer joins in Chapter 7.

Summary

This has been a fairly long chapter, but we have covered quite a lot. We have discussed the `SELECT` statement in some detail, discovering how to choose columns and rows, how to order the output, and how to suppress duplicate information. We also learned a bit about the `date` type, and how to configure PostgreSQL's behavior in interpreting and displaying dates, as well as how to use dates in condition statements.

We then moved on to the heart of SQL: the ability to relate tables together. After our first bit of SQL that joined a pair of tables, we saw how easy it was to extend this to three and even four tables. We finished off by reusing some of the knowledge we gained early in the chapter to refine our four-table selection to home in on displaying exactly the information we were searching for, and removing all the extra columns and duplicate rows.

The good news is that we have now seen all the everyday features of the `SELECT` statement, and once you understand the `SELECT` statement, much of the rest of SQL is reasonably straightforward. We will be coming back to the `SELECT` statement in Chapter 7 to look at some more advanced features that you will need from time to time, but you will find that much of SQL you need to use in the real world has been covered in this chapter.

CHAPTER 5

PostgreSQL Command-Line and Graphical Tools

A PostgreSQL database is generally created and administered with the command-line tool, `psql`, which we have used in earlier chapters to get started. Command-line tools similar to `psql` are common with commercial databases. Oracle has one such tool called SQL*Plus, for example.

While command-line tools are generally complete, in the sense that they contain ways to perform all the functions that you need, they can be a little user-unfriendly. On the other hand, they make no great demands in terms of graphics cards, memory, and so on.

In this chapter, we will begin by taking a closer look at `psql`. Next, we will cover how to set up an ODBC data source to use a PostgreSQL database, which is necessary for some of the tools described in this chapter. Then we will meet some of the graphical tools available for working with PostgreSQL databases. Some of the tools can also be used for administering databases, which is the topic of Chapter 11. In this chapter, we will concentrate on general database tasks.

In particular, we'll examine the following tools in this chapter:

- `psql`
- ODBC
- pgAdmin III
- phpPgAdmin
- Rekall
- Microsoft Access
- Microsoft Excel

psql

The `psql` tool allows us to connect to a database, execute queries, and administer a database, including creating a database, adding new tables and entering or updating data, using SQL commands.

Starting psql

As we have already seen, we start `psql` by specifying the database to which we wish to connect. We need to know the host name of the server and the port number the database is listening on (if it is not running on the default of 5432), plus a valid username and password to use for the connection. The default database will be the one on the local machine with the same name as the current user login name.

To connect to a named database on a server, we invoke `psql` with a database name, like this:

```
$ psql -d bpsimple
```

We can override defaults for the database name, username, server host name, and listening port by setting the environment variables `PGDATABASE`, `PGUSER`, `PGHOST`, and `PGPORT`, respectively. These defaults may also be overridden by using the `-d`, `-U`, `-h`, and `-p` command-line options to `psql`.

■ **Note** We can run `psql` only by connecting to a database. This presents a "chicken-and-egg" problem for creating our first database. We need a user account and a database to connect to. We created a default user, `postgres`, when we installed PostgreSQL in Chapter 3, so we can use that to connect to create new users and databases. To create a database, we connect to a special database included with all PostgreSQL installations, `template1`. Once connected to `template1`, we can create a database, and then either quit and restart `psql` or use the `\c` internal `psql` command to reconnect to the new database.

When `psql` starts up, it will read a startup file, `.psqlrc`, if one exists and is readable in the current user's home directory. This file is similar to a shell script startup file and may contain `psql` commands to set the desired behavior, such as setting the format options for printing tables and other options. We can prevent the startup file from being read by starting `psql` with the `-X` option.

Issuing Commands in psql

Once running, `psql` will prompt for commands with a prompt that consists of the name of the database we are connected to, followed by `=>`. For users with full permissions on the database, the prompt is replaced with `=#`.

`psql` commands are of two different types:

- **SQL commands:** We can issue any SQL statement that PostgreSQL supports to `psql`, and it will execute it.
- **Internal commands:** These are `psql` commands used to perform operations not directly supported in SQL, such as listing the available tables and executing scripts. All internal commands begin with a backslash and cannot be split over multiple lines.

■ **Tip** You can ask for a list of all supported SQL commands by executing the internal command `\h`. For help on a specific command, use `\h <sql_command>`. The internal command `\?` gives a list of all internal commands.

SQL commands to `psql` may be spread over multiple lines. When this occurs, `psql` will change its prompt to `->` or `-#` to indicate that more input is expected, as in this example:

```
$ psql -d bpsimple
...
bpsimple=# SELECT *
bpsimple-# FROM customer
bpsimple-# ;
...
$
```

To tell `psql` that we have completed a long SQL command that might spread across multiple lines, we need to end the command with a semicolon. Note that the semicolon is not a required part of the SQL command, but is just there to let `psql` know when we are finished. For example, in the `SELECT` statement shown here, we may have wanted to add a `WHERE` clause on the next line.

We can tell `psql` that we will never split our commands over more than one line by starting `psql` with the `-S` option. In that case, we do not need to add the semicolon to the end of our commands. The `psql` prompt will change to `^>` to remind us that we are in single-line mode. This will save us a small amount of typing and may be useful for executing some SQL scripts.

Working with the Command History

On PostgreSQL platforms that support history recording, each command that we ask `psql` to execute is recorded in a history, and we can recall previous commands to run again or edit. Use the arrow keys to scroll through the command history and edit commands. This feature is available unless you have turned it off with the `-n` command-line option (or it has not been compiled in the build for your platform).

We can view the query history with the `\s` command or save it to a file with `\s <file>`. The last query executed is kept in a query buffer. We can see what is in the query buffer with `\p`, and we can clear it with `\r`. We can edit the query buffer contents with an external editor with `\e`. The editor will default to `vi` (on Linux and UNIX), but you can specify your own favorite editor by setting the `EDITOR` environment variable before starting `psql`. We can send the query buffer to the server with `\g`, which gives a simple way to repeat a query.

Scripting psql

We can collect a group of `psql` commands (both SQL and internal) in a file and use it as a simple script. The `\i` internal command will read a set of `psql` commands from a file.

This feature is especially useful for creating and populating tables. We used it in Chapter 3 to create our sample database, `bpsimple`. Here is part of the `create_tables-bpsimple.sql` script file that we used:

```
CREATE TABLE customer
(
    customer_id             serial                          ,
    title                   char(4)                         ,
    fname                   varchar(32)                     ,
    lname                   varchar(32)           NOT NULL,
    addressline             varchar(64)                     ,
    town                    varchar(64)                     ,
    zipcode                 char(10)              NOT NULL,
    phone                   varchar(16)                     ,
    CONSTRAINT              customer_pk PRIMARY KEY(customer_id)
);

CREATE TABLE item
(
    item_id                 serial                          ,
    description             varchar(64)           NOT NULL,
    cost_price              numeric(7,2)                    ,
    sell_price              numeric(7,2)                    ,
    CONSTRAINT              item_pk PRIMARY KEY(item_id)
);
```

We give script files a `.sql` extension by convention, and execute them with the `\i` internal command:

```
bpsimple=#\i create_tables-bpsimple.sql
CREATE TABLE
CREATE TABLE
...
bpsimple=#
```

Here, the script is located in the directory where we started `psql`, but we can execute a script stored elsewhere by giving the full path to it.

Another use of script files is for simple reports. If we want to keep an eye on the growth of a database, we could put a few commands in a script file and arrange to run it every once in a while. To report the number of customers and orders taken, create a script file called `report.sql` that contains the following lines and execute it in a `psql` session:

```
SELECT count(*) FROM customer;
SELECT count(*) FROM orderinfo;
```

Alternatively, we can use the `-f` command line option to get `psql` to execute the file and then exit:

```
$ psql -f report.sql bpsimple
  count
--------
     15
(1 row)
```

```
  count
--------
      5
(1 row)
$
```

If a password is required to access the database, `psql` will prompt for one. We can specify a different database user with the `-U` option to `psql`.

We can redirect query output to a file by using the `-o` command-line option, or to a file or filter program with the `\o` internal command from within a session. For example, from within a `psql` session, we can create a text file called `customers.txt` containing all of our customers by issuing the following commands:

```
bpsimple=# \o customers.txt
bpsimple=# SELECT * FROM customer;
bpsimple=# \o
```

The final command, `\o` without a filename parameter, stops the redirecting of query output and closes the output file.

Examining the Database

We can explore the structure of our database using a number of internal `psql` commands. The *structure* includes the names and definition of the tables that make up the database, any functions (stored procedures and triggers) that may have been defined, the users that have been created, and so on.

The `\d` command lists all of the relations—tables, sequences, and views, if any—in our database. Here is an example:

```
bpsimple=# \d customer
                                      Table "public.customer"
   Column    |         Type          |          Modifiers
-------------+-----------------------+-----------------------------------------
 customer_id | integer               | not null default nextval(...)
 title       | character(4)          |
 fname       | character varying(32) |
 lname       | character varying(32) | not null
 addressline | character varying(64) |
 town        | character varying(32) |
 zipcode     | character(10)         | not null
 phone       | character varying(16) |
Indexes:
    "customer_pk" PRIMARY KEY, btree (customer_id)

bpsimple=#
```

The `\dt` command restricts the listing to tables only. See Table 5-2 in the "Internal Commands Quick Reference" section for more internal `psql` commands.

psql Command-Line Quick Reference

The command syntax for `psql` is:

```
psql [options] [dbname [username]]
```

The `psql` command-line options and their meanings are shown in Table 5-1. To see the complete list of options to `psql`, use the following command:

```
$ psql --help
```

Table 5-1. *`psql` Command-Line Options*

Option	Meaning
`-a`	Echo all input from script
`-A`	Unaligned table output mode; same as `-P format=unaligned`
`-c <query>`	Run only single query (or internal command) and exit
`-d <dbname>`	Specify database name to connect to (default: `$PGDATABASE` or current login name)
`-e`	Echo queries sent to server
`-E`	Display queries that internal commands generate
`-f <filename>`	Execute queries from file, then exit
`-F <string>`	Set field separator (default: |); same as `-P fieldsep=<string>`
`-h <host>`	Specify database server host (default: `$PGHOST` or local machine)
`-H`	Set HTML table output mode; same as `-P format=html`
`--help`	Show help, then exit
`-l`	List available databases, then exit
`-n`	Disable `readline`; prevents line editing
`-o <filename>`	Send query output to *filename* (use the form `|pipe` to send output to a filter program)
`-p <port>`	Specify database server port (default: `$PGPORT` or compiled-in default, usually `5432`)
`-P var[=arg]`	Set printing option *var* to *arg* (see `\pset` command)
`-q`	Run quietly (no messages, only query output)
`-R <string>`	Set record separator (default: `newline`); same as `-P recordsep=<string>`
`-s`	Set single-step mode (confirm each query)

Table 5-1. *psql Command-Line Options (Continued)*

Option	Meaning
-S	Set single-line mode (end of line terminates query rather than semicolon)
-t	Print rows only; same as -P tuples_only
-T <*text*>	Set HTML table tag options (width, border); same as -P tableattr=<*text*>
-U <*username*>	Specify database username (default: $PGUSER or current login)
-v *name=value*	Set psql variable *name* to *value*
--version	Show version information and exit; also –V
-W	Prompt for password (should happen automatically, if a password is required)
-x	Turn on expanded table output; same as -P expanded
-X	Do not read startup file (~/.psqlrc)

psql Internal Commands Quick Reference

The supported internal psql commands are shown in Table 5-2. In many versions of PostgreSQL, some of these commands have more legible longer forms (such as \list for \l).

Table 5-2. *psql Internal Commands*

Command	Meaning
\?	List all available psql internal commands
\a	Toggle between unaligned and aligned mode
\c[onnect] [*dbname*|- [*user*]]	Connect to new database; use - as the database name to connect to the default database if you need to give a username
\C <*title*>	Set table title for output; same as \pset title
\cd <*dir*>	Change the working directory
\copy ...	Perform SQL COPY with data stream to the client machine
\copyright	Show PostgreSQL usage and distribution terms
\d <*table*>	Describe table (or view, index, sequence)
\d{t|i|s|v}	List tables/indices/sequences/views
\d{p|S|l}	List permissions/system tables/lobjects
\da	List aggregates

Table 5-2. *psql Internal Commands (Continued)*

Command	Meaning
\db	List tablespaces
\dc	List conversions
\dC	List casts
\dd [*object*]	List comment for table, type, function, or operator
\dD	List domains
\df	List functions
\dg	List groups
\dl	List large objects; also \lo list
\dn	List schemas
\do	List operators
\dT	List data types
\du	List users
\e [*file*]	Edit the current query buffer or *file* with external editor
\echo <*text*>	Write text to standard output
\encoding <*encoding*>	Set client encoding
\f <*sep*>	Change field separator
\g [*file*]	Send query to back-end (and results in *file*, or |pipe)
\h [*cmd*]	Help on syntax of SQL commands; use * for detail on all commands
\H	Toggle HTML mode
\i <*file*>	Read and execute queries from *file*
\l	List all databases
\lo_export, \lo_import, \lo_list, \lo_unlink	Perform large object operations
\o [*file*]	Send all query results to *file*, or |pipe
\p	Show the content of the current query buffer
\pset <*opt*>	Set table output option, which can be one of the following: format, border, expanded, fieldsep, footer, null, recordsep, tuples_only, title, tableattr, pager
\q	Quit psql
\qecho <*txt*>	Write text to query output stream (see \o)
\r	Reset (clear) the query buffer
\s [*file*]	Print history or save it in *file*

Table 5-2. *psql Internal Commands (Continued)*

Command	Meaning
\set <*var*> <*value*>	Set internal variable
\t	Show only rows (toggles between modes)
\T <*tags*>	Set HTML table tags; same as \pset tableattr
\timing	Toggle timing of commands
\unset <*var*>	Unset (delete) internal variable
\w <*file*>	Write current query buffer to *file*
\x	Toggle expanded output
\z	List access permissions for tables, views, and sequences
\! [*cmd*]	Escape to shell or execute a shell command

ODBC Setup

Several of the tools discussed in this chapter, as well as some of the programming language interfaces discussed in later chapters, use the ODBC standard interface to connect to PostgreSQL. ODBC defines a common interface for databases and is based on X/Open and ISO/IEC programming interfaces. In fact, ODBC stands for Open Database Connectivity and is not (as is often believed) limited to Microsoft Windows clients. Programs written in many languages—like C, C++, Ada, PHP, Perl, and Python—can make use of ODBC. OpenOffice, Gnumeric, Microsoft Access, and Microsoft Excel are just a few examples of applications that can use ODBC.

To use ODBC on a particular client machine, we need both an application written for the ODBC interface and a driver for the particular database that we want to use. PostgreSQL has an ODBC driver called psqlodbc, which we can install on our clients. Often, clients will be running on machines that are different from the server, and possibly different from each other, requiring us to compile the ODBC driver on several client platforms. For example, we might have the database server on Linux and our client applications running on Windows and Mac OS X.

The source code and a binary installation for Windows are available from the psqlODBC project home page at http://gborg.postgresql.org/project/psqlodbc/.

■ **Note** The standard Windows installation of PostgreSQL also contains a version of the ODBC driver that can be installed on a Windows server at the same time as the database.

Installing the ODBC Driver

On Microsoft Windows, ODBC drivers are made available through the Control Panel's Administrative Tools Data Sources applet, as shown in Figure 5-1.

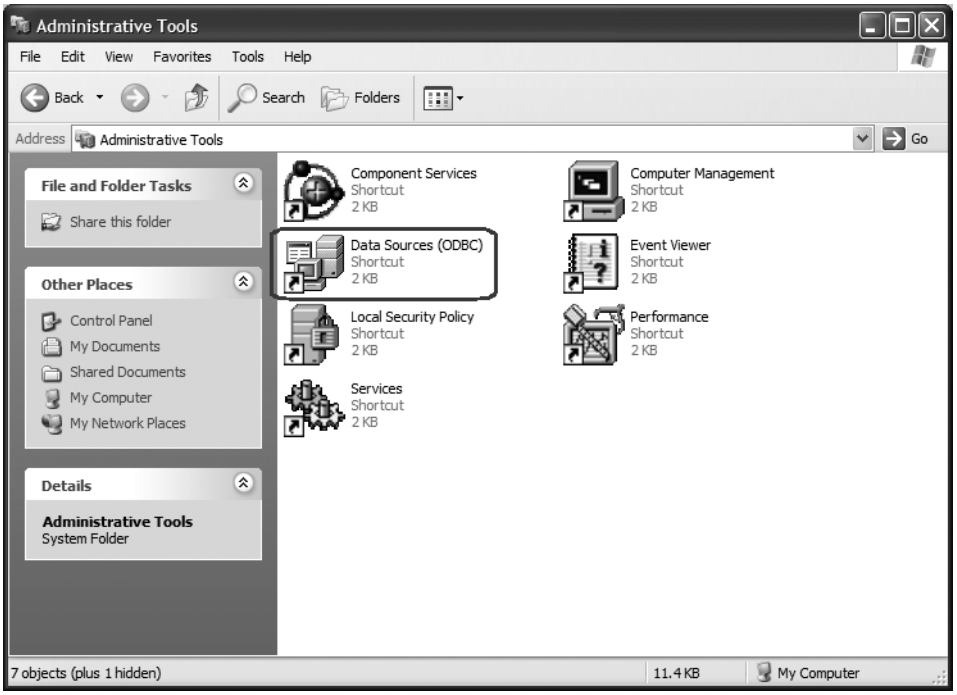


Figure 5-1. *The ODBC Data Sources applet*

The Drivers tab of this applet lists the installed ODBC drivers, as shown in Figure 5-2.

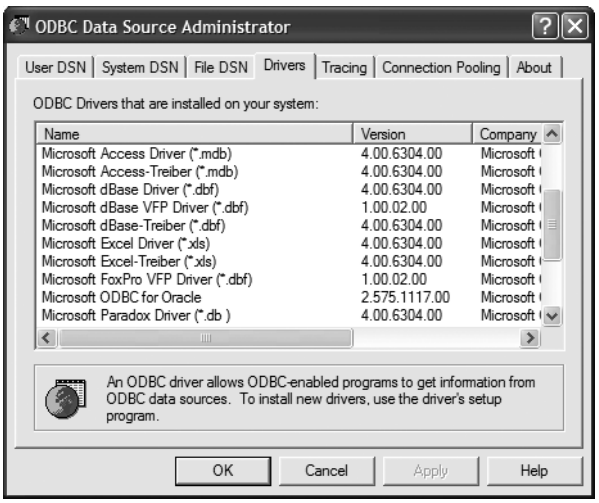


Figure 5-2. *Installed ODBC drivers*

To install the PostgreSQL ODBC driver, we need to perform two steps:

1. Download a suitable version of the driver from `http://gborg.postgresql.org/project/psqlodbc`. If you have a version of Windows that includes the Microsoft Windows Installer, the MSI version of the drivers is the recommended choice, as it is much smaller; otherwise, download the full installation. At the time of writing, both versions of the driver are located in compressed archive files named `psqlodbc-07_03_0200.zip`.
2. Extract the driver installation file from the downloaded archive. It will be named either `psqlodbc.msi` or `psqlodbc.exe`. Double-click the installation file and follow the instructions to install the PostgreSQL ODBC driver.

After performing these two steps, we can confirm that we have successfully installed the driver by again selecting the Drivers tab in the ODBC applet and noting that PostgreSQL now appears in the list, as shown in Figure 5-3.

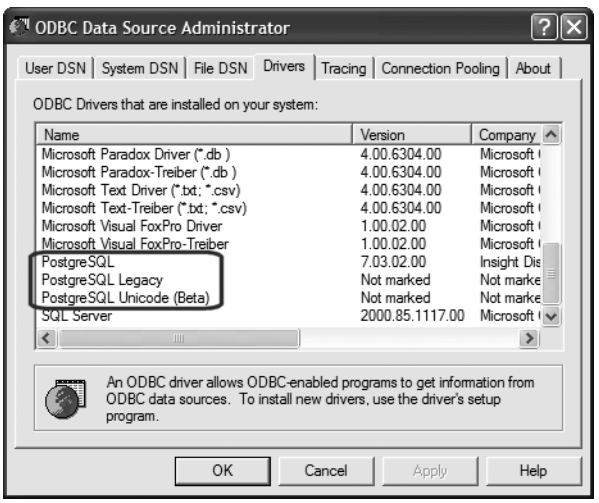


Figure 5-3. *PostgreSQL ODBC driver installed*

Creating a Data Source

Now we will be able to use ODBC-aware applications to connect to PostgreSQL databases. To make a specific database available, we need to create a data source, as follows:

1. Select User DSN in the ODBC applet to create a data source that will be available to the current user. (If you select System DSN, you can create data sources that all users can see.)
2. Click Add to begin the creation process. A dialog box for selecting which driver the data source will use appears, as shown in Figure 5-4.

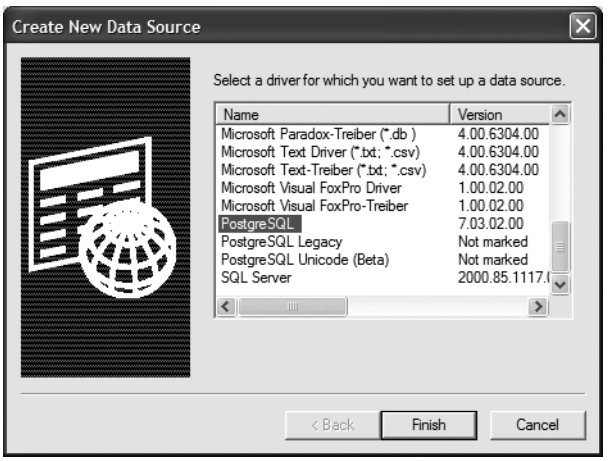


Figure 5-4. *Creating a PostgreSQL data source*

3. Select the PostgreSQL driver and click Finish.

4. We now have a PostgreSQL driver entry that must be configured. A Driver Setup box will appear for us to enter the details of this data source. As shown in Figure 5-5, give the data source a name and a description, and set the network configuration. Here, we are creating an ODBC connection to a copy of our `bpsimple` database running on a Linux server using the IP address of the server. (If you are running a fully configured naming service such as DNS or WINS, you can use a machine name for the server.) We also specify the username and password to be used at the server to access the database we have chosen.

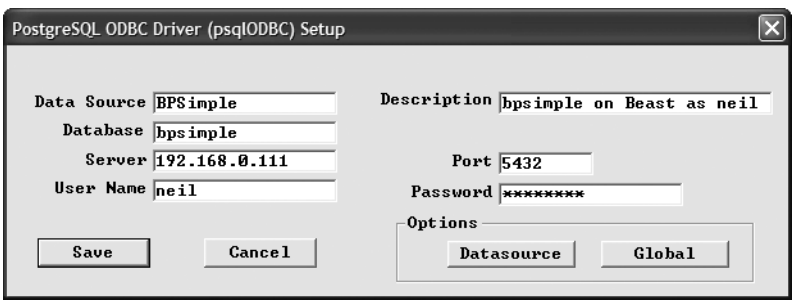


Figure 5-5. *Configuring a PostgreSQL data source*

■ **Tip** Additional options are available under the Global and DataSource options in the ODBC Driver Setup dialog box. If you will be using ODBC applications to update data or insert new data into the PostgreSQL database, you may need to configure the data source to support this. To do this, click the DataSource button and make sure that the Read Only box is not checked in the dialog box that appears.

5. Click Save to complete the setup.

We are now ready to access our PostgreSQL database from ODBC applications such as Microsoft Access and Excel, as we will discuss later in this chapter. Next, we will look at some open-source alternatives, starting with pgAdmin III.

pgAdmin III

pgAdmin III is a full-featured graphical interface for PostgreSQL databases. It is free software, community-maintained at `http://www.pgadmin.org`. According to the web site, `pgAdmin` is "a powerful administration and development platform for the PostgreSQL database, free for any use." It runs on Linux, FreeBSD, and Windows 2000/XP. Versions for Sun and Mac OS X are being developed.

pgAdmin III offers a variety of features. With it, we can do the following:

- Create and delete tablespaces, databases, tables, and schemas
- Execute SQL with a query window
- Export the results of SQL queries to files
- Back up and restore databases or individual tables
- Configure users, groups, and privileges
- View, edit, and insert table data

Let's look at how to get up and running with this versatile tool.

Installing pgAdmin III

With the release of pgAdmin III, the developers have made installation of the program much simpler. Previous versions require the PostgreSQL ODBC driver to be installed to provide access to the database, but this dependency has been removed. If you have used an earlier version of pgAdmin, we recommend that you consider upgrading.

■ **Note** The standard Windows installation of PostgreSQL includes a version of pgAdmin III that can be installed on a Windows server along with the database or on a client without a database.

Binary packages for Microsoft Windows 2000/XP, FreeBSD, Debian Linux, Slackware Linux, and Linux distributions that use the RPM package format (such as Red Hat and SuSE Linux) are available to download from `http://www.pgadmin.org/pgadmin3/download.php`.

Download the appropriate package for the system you want to run pgAdmin III and install it. The Windows package contains an installer to execute, packaged in a compressed archive ZIP file. After installation, you should have a new program (pgAdmin III) in the Windows Start menu.

Using pgAdmin III

Before we can use pgAdmin III in earnest, we need to make sure that we can create objects in the database we want to maintain. This is because pgAdmin III augments the database with objects of its own that are stored on the server. To perform all of the maintenance functions with pgAdmin III, we need to log on as a user that has complete privileges for the database—a superuser, in other words. If we choose a user without superuser status, we will get an error.

■ **Tip** We will be looking at users and permissions in Chapter 11. If your PostgreSQL database installation was performed on Windows with the default settings, you should have a user `postgres` that is used to control the database, and you can try to log on as that user. If you installed on Linux or UNIX following the steps in Chapter 3, you will have created a suitable user; we used `neil`.

We can manage several database servers at once with pgAdmin III, so our first task is to create a server connection. Select Add Server from the File menu to bring up a dialog box very similar to the one we used to create an ODBC connection earlier. Figure 5-6 shows a connection being made to a PostgreSQL database on a Linux server.

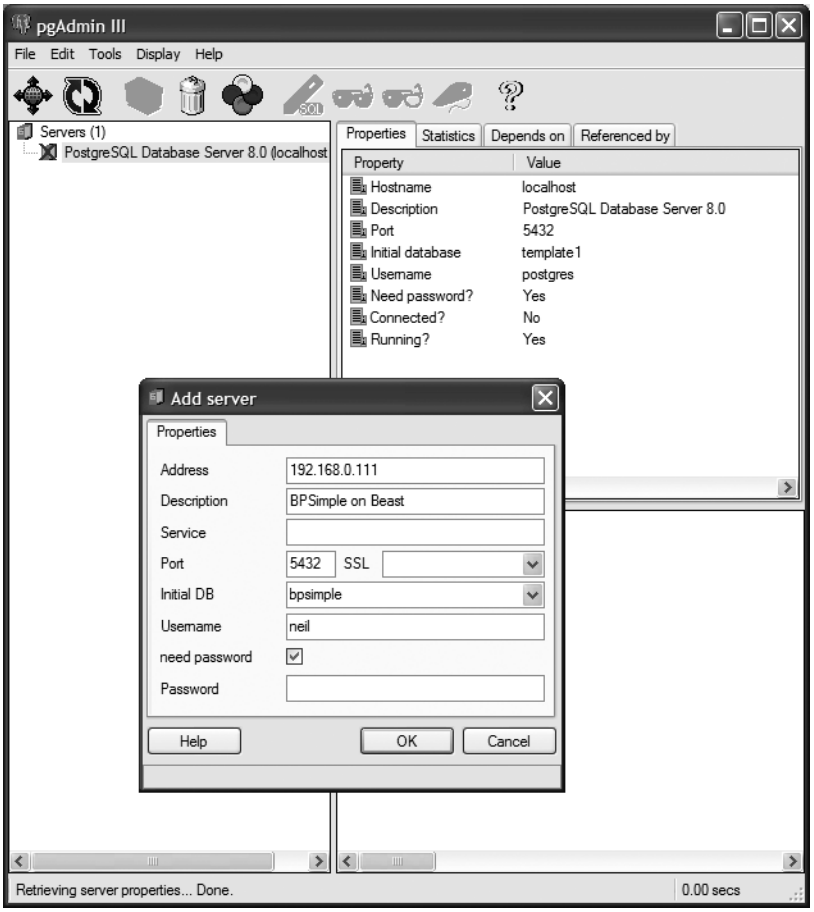


Figure 5-6. *Adding a server connection in pgAdmin III*

Once the server connection has been created, we can connect to the database server and browse the databases, tables, and other objects that the server is providing. Figure 5-7 shows an example of pgAdmin III exploring the tables of the `bpsimple` database and examining the `lname` attribute of the `customer` table.

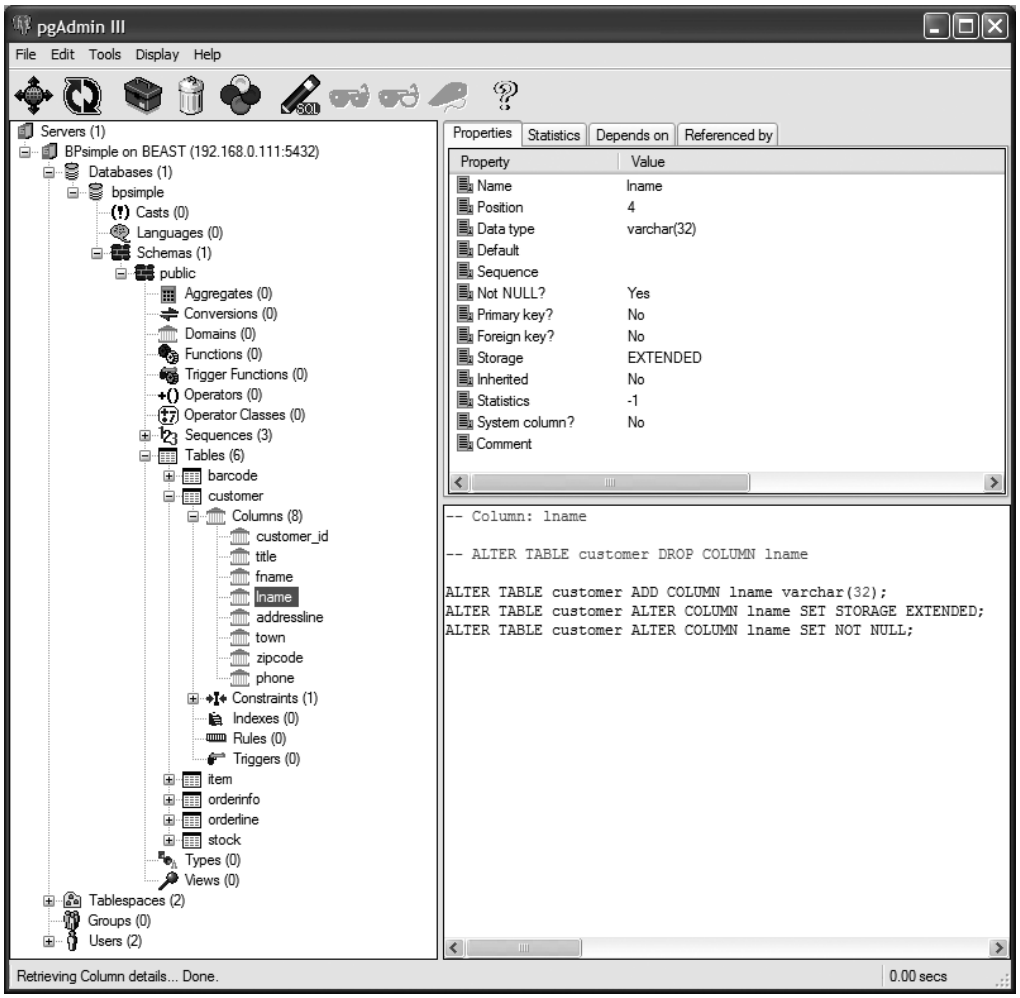


Figure 5-7. *Examining table properties with pgAdmin III*

One feature of pgAdmin III that is potentially very useful is its backup and restore functionality. This provides a simple interface to the PostgreSQL `pg_dump` utility, which we will cover in Chapter 11. We can back up and restore individual tables or an entire database. There are options to control how and where the backup file is created and what method will be used to restore the database, if necessary (for example, by using the `\copy` command or SQL `INSERT` statements).

To open the Backup dialog box, right-click the object (database or table) to back up and select Backup. Figure 5-8 shows the Backup dialog box for the `bsimple` database.

Figure 5-8. *The pgAdmin III Backup dialog box*

We will cover more of pgAdmin III's features for managing databases in Chapter 11.

phpPgAdmin

A web-based alternative for managing PostgreSQL databases is phpPgAdmin. This is an application (written in the PHP programming language) that is installed on a web server and provides a browser-based interface for administration of database servers. The project home page is at `http://phppgadmin.sourceforge.net/`.

With phpPgAdmin, we can perform many tasks with our databases, including the following:

- Manage users and groups
- Create tablespaces, databases, and schemas
- Manage tables, indexes, constraints, triggers, rules, and privileges
- Create views, sequences, and functions
- Create and run reports
- Browse table data
- Execute arbitrary SQL
- Export table data in many formats: SQL, COPY (data suitable for the SQL `COPY` command), XML, XHTML, comma-separated values (CSV), tab-delimited, and `pg_dump`
- Import SQL scripts, COPY data, XML files, CSV files, and tab-delimited files

Installing phpPgAdmin

Installing phpPgAdmin is very straightforward. The program is available as a download package in several formats, including ZIP and compressed tarball (`.tar.gz`). The package needs to be extracted into a folder served by a web server that supports the PHP programming language. A popular choice for this is the Apache web server configured with the `mod_php` extension. More information about Apache and PHP can be found at `http://www.apache.org` and `http://www.php.net`, respectively. Many Linux distributions provide a suitably configured Apache installation.

The only configuration that phpPgAdmin requires is the setting of some variables in its configuration file, `conf/conf.inc.php`. The following extract shows the lines in this file that need to be configured to set up phpPgAdmin to manage a database on another server.

```
// Display name for the server on the login screen
$conf['servers'][0]['desc'] = 'Beast';

// Hostname or IP address for server.  Use '' for UNIX domain socket.
$conf['servers'][0]['host'] = '192.168.0.111';

// Database port on server (5432 is the PostgreSQL default)
$conf['servers'][0]['port'] = 5432;

// Change the default database only if you cannot connect to template1
$conf['servers'][0]['defaultdb'] = 'template1';
```

Using phpPgAdmin

To demonstrate the cross-platform potential of phpPgAdmin, Apache, and PostgreSQL, Figures 5-9 and 5-10 show a browser running on an Apple Mac, accessing an Apache web server with phpPgAdmin installed running on Windows XP (at address 192.168.0.3), managing a database on a Linux server called Beast at address 192.168.0.111. Figure 5-11 depicts the customer table data being viewed. The URL for the browser is `http://192.168.0.3/phpPgAdmin/index.php`.

Figure 5-9. *phpPgAdmin login*

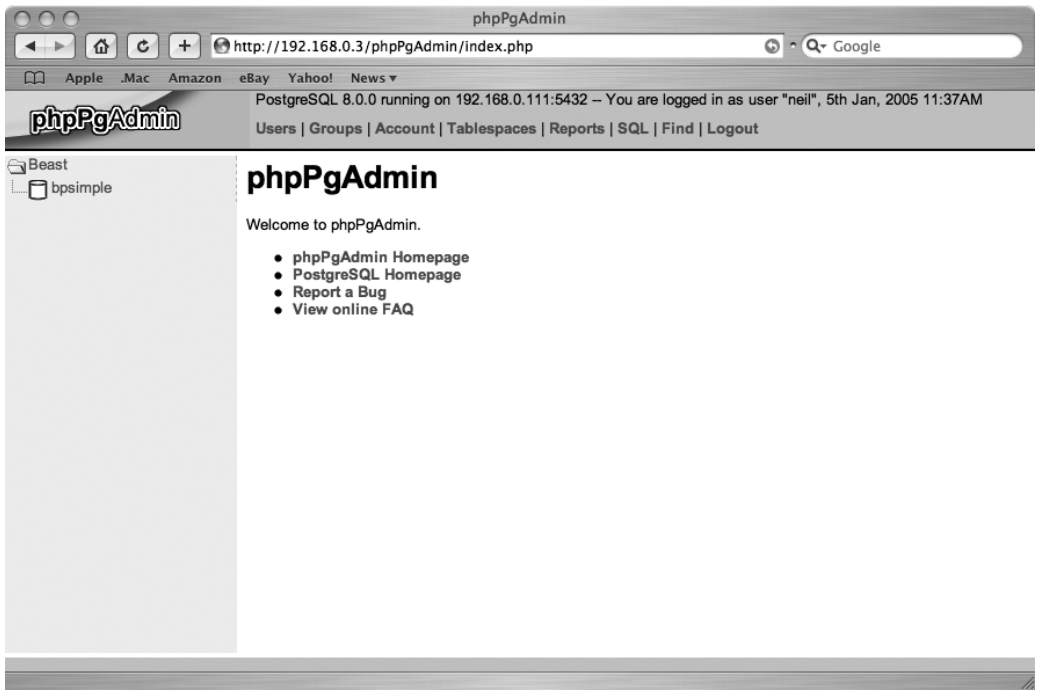


Figure 5-10. *phpPgAdmin main page*

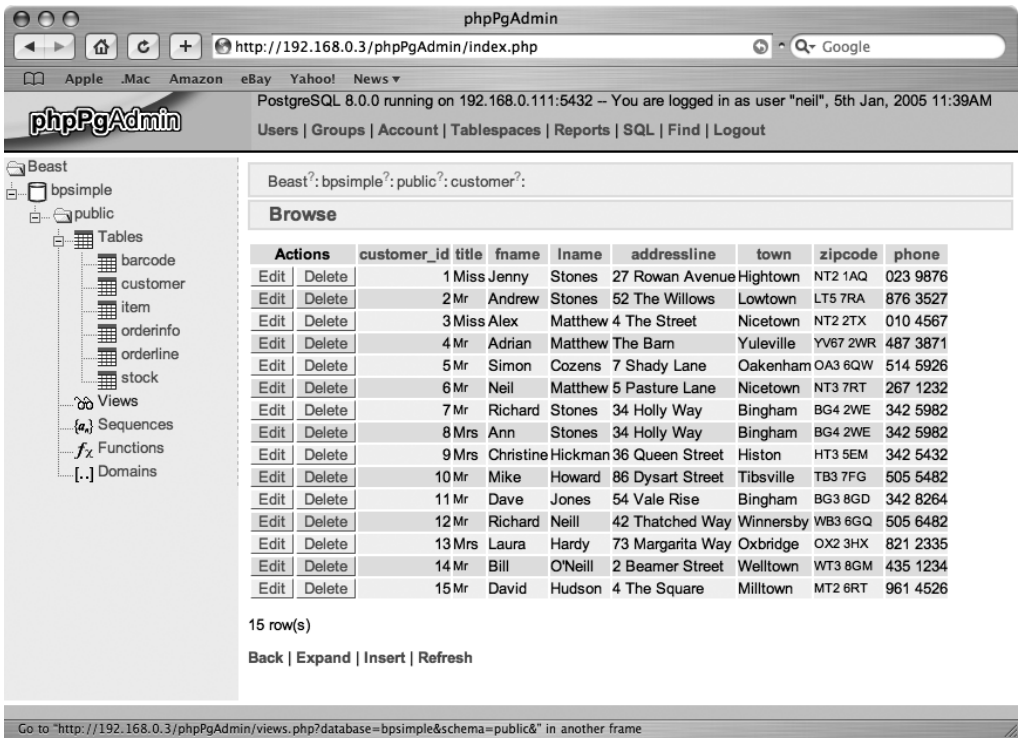


Figure 5-11. *phpPgAdmin browsing table data*

One feature of phpPgAdmin that is potentially very useful is its data import functionality. If we have some data that we would like to import into a PostgreSQL table, phpPgAdmin can help. One way of importing data is to make it available as a comma-separated values (CSV) file. Applications such as Microsoft Excel are able to export data in this format.

Let's consider a simple example. Suppose that from an Excel spreadsheet, we have saved some rows for the `item` table in the `bpsimple` database, in a CSV with headings format. This means that there are column names present in the first row, followed by the data, like this:

```
description,cost_price,sell_price
Wood Puzzle,15.23,21.95
Rubik Cube,7.45,11.49
Linux CD,1.99,2.49
```

We start the import process by selecting the table we want to import into, clicking Import, and selecting the import file type (CSV in this example) and the import filename, as shown in Figure 5-12. We can then click Import, and (assuming we have permission) the new rows will be incorporated into our database table.

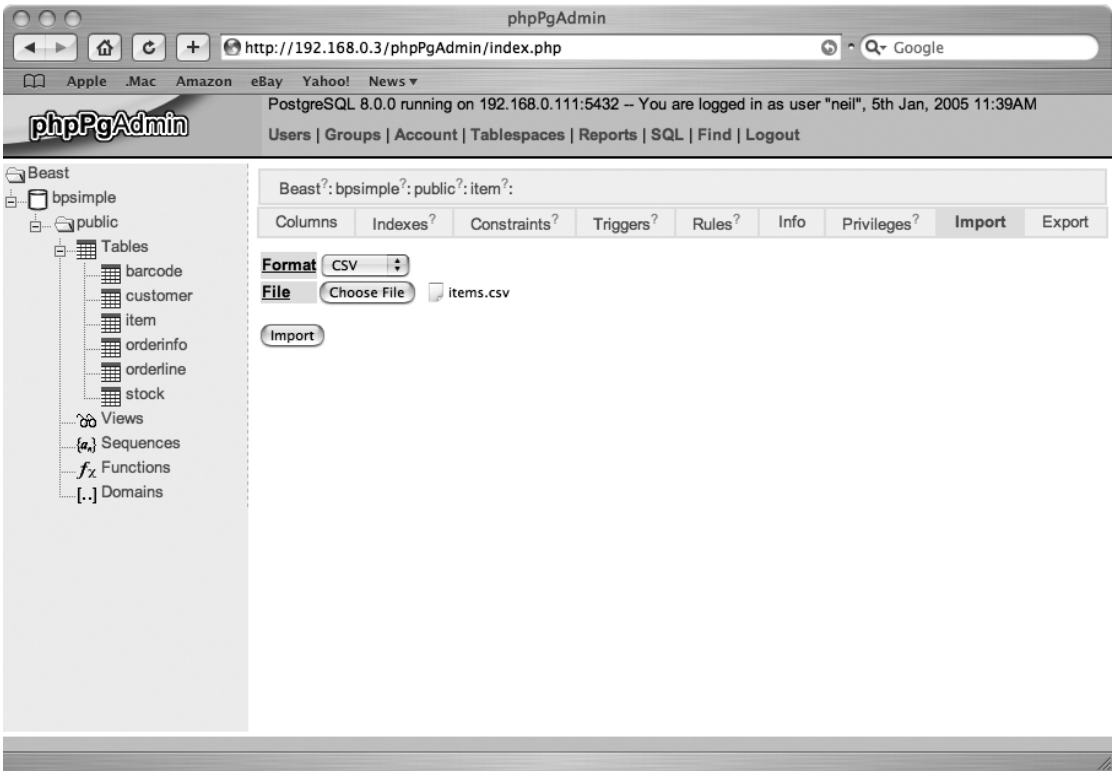


Figure 5-12. *Importing data with phpPgAdmin*

Rekall

Rekall is a multiplatform database front-end originally developed by theKompany (`http://www.thekompany.com/`) as a tool to extract, display, and update data from several different database types. It works with PostgreSQL, MySQL, and IBM DB2 using native drivers, and other databases using ODBC.

While Rekall does not include the PostgreSQL-specific administration features found in pgAdmin III and phpPgAdmin, it does add some very useful user functionality. In particular, it contains a visual query designer and a form builder for creating data-entry applications. Furthermore, Rekall uses the Python programming language for scripting, allowing sophisticated database applications to be constructed.

Rekall has been made available under a dual-license scheme. There is a commercial version and also a community-developed open-source version released under the GNU Public License (GPL). Both are available at `http://www.rekallrevealed.org/`. The open-source version

can be built and installed on Linux and other systems that are running the KDE desktop environment or have the appropriate KDE libraries available. Rekall is beginning to be provided as part of Linux distributions, including SuSE Linux 9.1. It connects to PostgreSQL using a native driver. The commercial version of Rekall adds a Microsoft Windows version and support for ODBC database connections.

Rekall is very easy to use and comes with an on-line handbook, called *Rekall Unbound*, which provides information on every aspect of Rekall's features. It can be accessed either from Rekall's help manual or through the KDE Konqueror web browser at the URL `help:/rekall`.

Here, we will take a quick look at the open-source version of Rekall, running on SuSE Linux 9.1.

Connecting to a Database

Connections to databases are created using a connection wizard that prompts for a host, database name, and user credentials. Several options are available for each connection, but the defaults work just fine. Figure 5-13 shows a database connection initiated in Rekall.

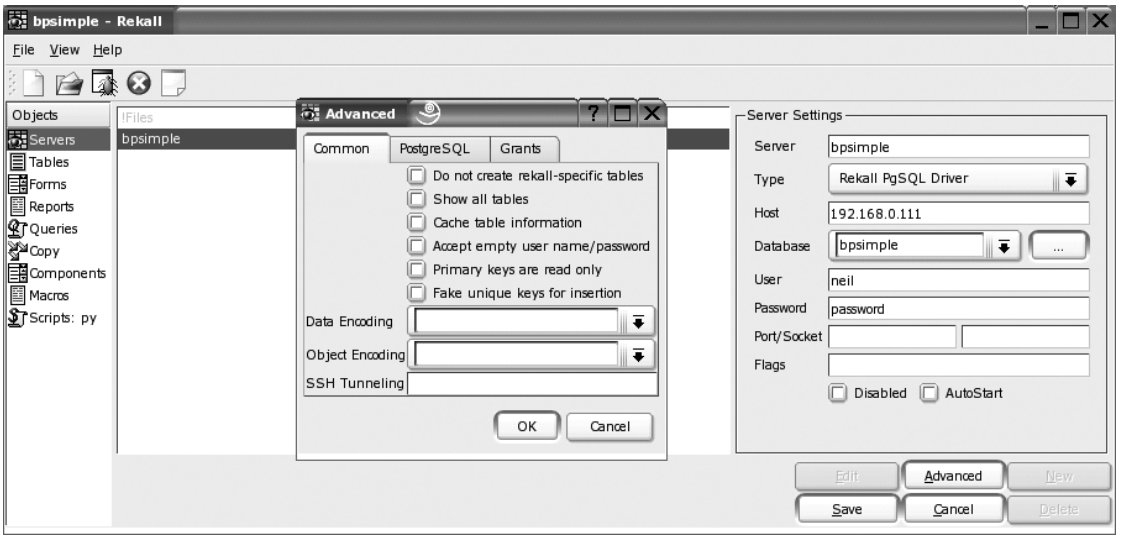
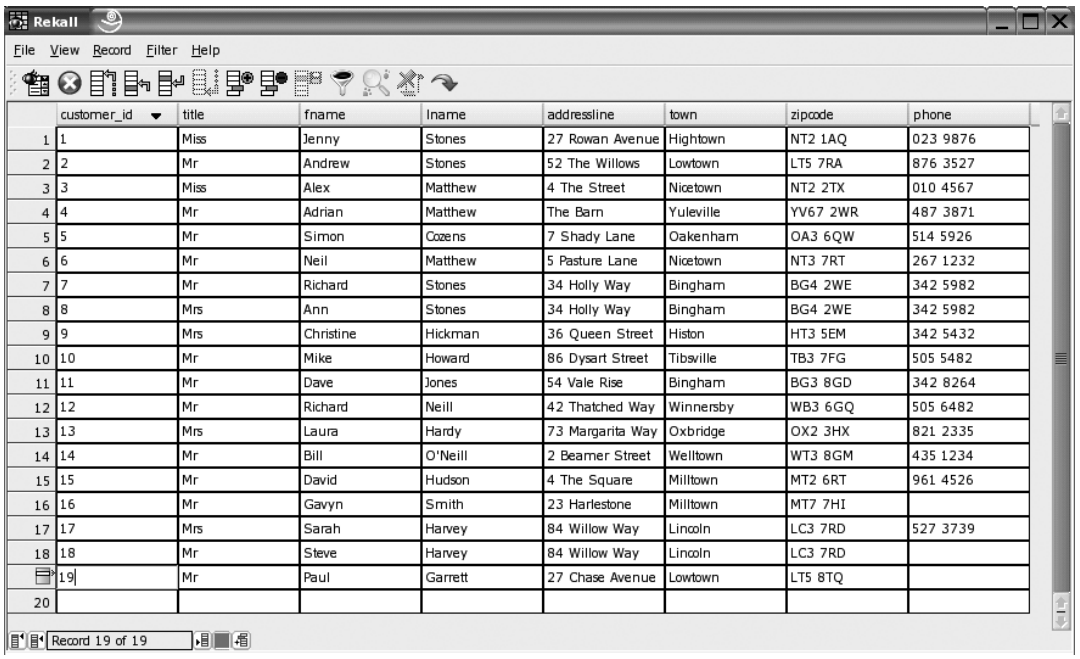


Figure 5-13. *A Rekall database connection*

Once we are connected to a database, we can browse the tables, view, and edit data. This process is depicted in Figure 5-14.

For many of its operations, Rekall provides a data view and a design view. Switching to the design view reveals the structure of the object. So, when browsing a table, the design view shows us the definition of the table and its columns. We can use the design view to create new objects, such as tables, forms, and queries. For forms and queries, the data view allows us to use the form to enter data or view the results of the query.



	customer_id	title	fname	lname	addressline	town	zipcode	phone
1	1	Miss	Jenny	Stones	27 Rowan Avenue	Hightown	NT2 1AQ	023 9876
2	2	Mr	Andrew	Stones	52 The Willows	Lowtown	LT5 7RA	876 3527
3	3	Miss	Alex	Matthew	4 The Street	Nicetown	NT2 2TX	010 4567
4	4	Mr	Adrian	Matthew	The Barn	Yuleville	YV67 2WR	487 3871
5	5	Mr	Simon	Cozens	7 Shady Lane	Oakenham	OA3 6QW	514 5926
6	6	Mr	Neil	Matthew	5 Pasture Lane	Nicetown	NT3 7RT	267 1232
7	7	Mr	Richard	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
8	8	Mrs	Ann	Stones	34 Holly Way	Bingham	BG4 2WE	342 5982
9	9	Mrs	Christine	Hickman	36 Queen Street	Histon	HT3 5EM	342 5432
10	10	Mr	Mike	Howard	86 Dysart Street	Tibsville	TB3 7FG	505 5482
11	11	Mr	Dave	Jones	54 Vale Rise	Bingham	BG3 8GD	342 8264
12	12	Mr	Richard	Neill	42 Thatched Way	Winnersby	WB3 6GQ	505 6482
13	13	Mrs	Laura	Hardy	73 Margarita Way	Oxbridge	OX2 3HX	821 2335
14	14	Mr	Bill	O'Neill	2 Beamer Street	Welltown	WT3 8GM	435 1234
15	15	Mr	David	Hudson	4 The Square	Milltown	MT2 6RT	961 4526
16	16	Mr	Gavyn	Smith	23 Harlestone	Milltown	MT7 7HI	
17	17	Mrs	Sarah	Harvey	84 Willow Way	Lincoln	LC3 7RD	527 3739
18	18	Mr	Steve	Harvey	84 Willow Way	Lincoln	LC3 7RD	
19	19	Mr	Paul	Garrett	27 Chase Avenue	Lowtown	LT5 8TQ	
20								

Figure 5-14. *Browsing a table with Rekall*

Creating Forms

The support for forms in Rekall is extensive. We can create a new form very quickly using a form wizard, which simply asks which columns from which table should be included on the form. A graphical designer allows us to lay out the form if the default is not suitable. We can add buttons to the form to provide navigation (next record, delete records, and so on), and there is an optional navigation toolbar that can be added to forms. Figure 5-15 shows a form for the `customer` table. This form is nearly the default produced by Rekall; only the text labels for the data have been changed.

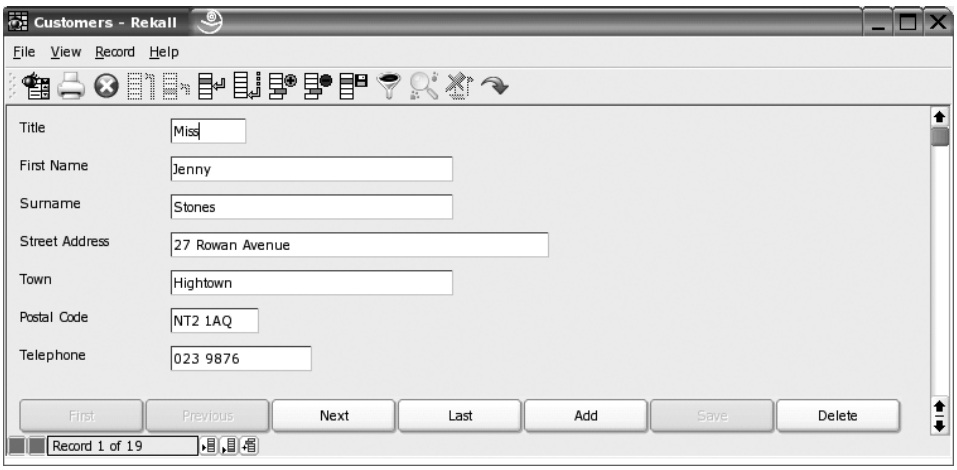


Figure 5-15. *A simple data-entry form in Rekall*

Each of the buttons on the form is scriptable. The default form contains actions written in Python for performing an appropriate action, such as saving the record. By adding our own code to these script actions, we can create a more sophisticated form, perhaps adding entry validation.

Building Queries

The graphical query designer in Rekall allows us to create, save, and execute potentially quite complex queries by essentially drawing a picture of the relationships we need to express. We will be dealing with some fairly complex queries as we progress through the book. For now, to give a taste of what Rekall can do, let's look at a couple of examples that show one of the queries we used in Chapter 4 being constructed and the results being displayed.

In Figure 5-16, we are using a three-table join to find out which items our customer Ann Stones has ordered from us. This query was created by double-clicking tables to add them to the query and dragging columns from one table to another to indicate the required joins. The only typing required was to specify the first name and surname of the customer of interest.

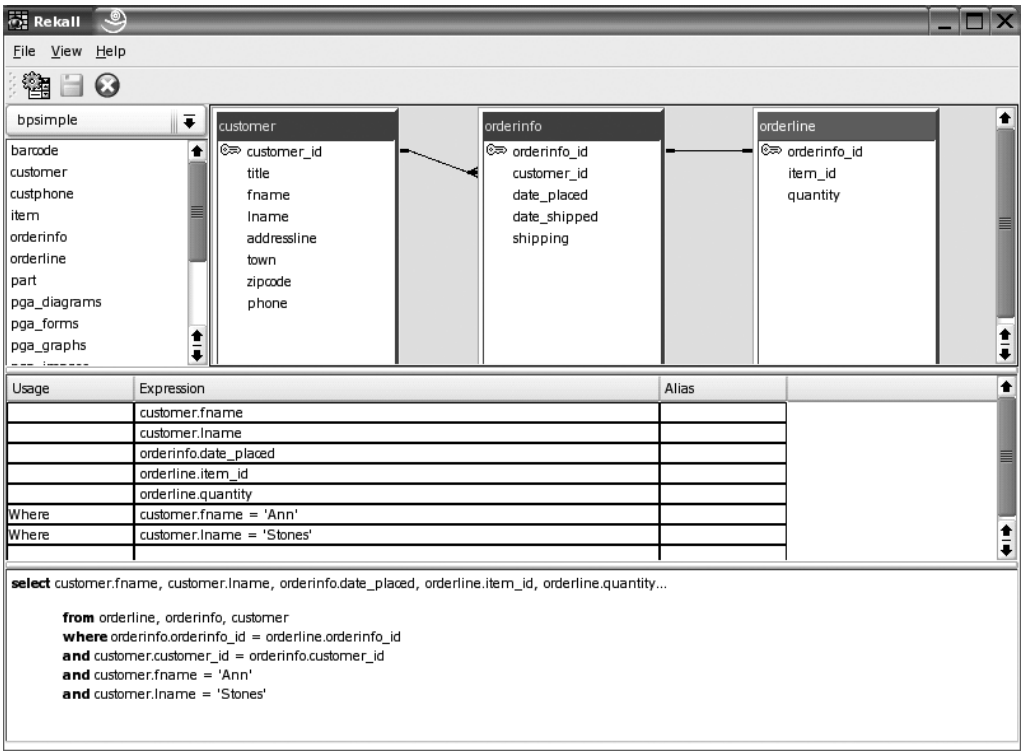


Figure 5-16. *A complex query in Rekall*

When we switch to the data view, we see the results of the query being executed, and we get the same results as in Chapter 4. This task is shown in Figure 5-17.

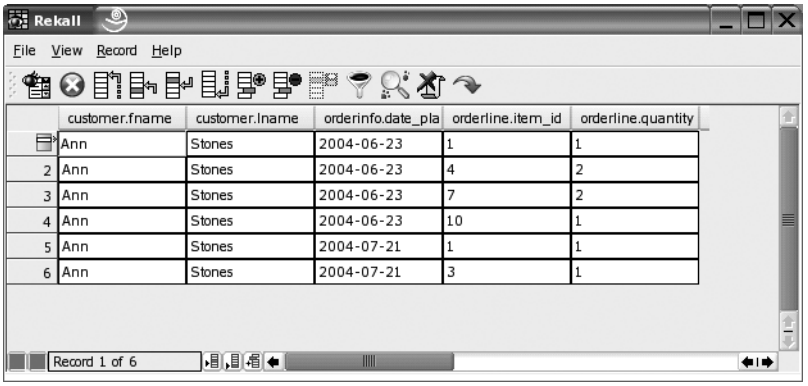


Figure 5-17. *Query results in Rekall*

Microsoft Access

Although it may seem an odd idea at first sight, we can use Microsoft Access with PostgreSQL. If Access is already a database system, why would we want to use PostgreSQL to store data? And, as there are a number of tools available that work with PostgreSQL, why do we need to use Microsoft Access?

First, when developing a database system, we need to consider requirements for matters such as data volumes, the possibility of multiple concurrent users, security, robustness, and reliability. You may decide on PostgreSQL because it fits better with your security model, your server platforms, and your data-growth predictions.

Second, although PostgreSQL running on a UNIX or Linux server may be the ideal environment for your data, it might not be the best, or most familiar, environment for your users and their applications. There is a case for allowing users to use tools such as Access or other third-party applications to create reports or data-entry forms for PostgreSQL databases. Since PostgreSQL has an ODBC interface, this is not only possible but remarkably easy.

Once you have established the link from Access to PostgreSQL, you can use all of the features of Access to create easy-to-use PostgreSQL applications. In this section, we will look at creating an Access database that uses data stored on a remote PostgreSQL server, and writing a simple report based on that data. (We assume that you are reasonably familiar with creating Access databases and applications.)

Using Linked Tables

Access allows us to import a table into a database in a number of different ways, one of which is by means of a *linked table*. This is a table that is represented in Access as a query. The data is retrieved from another source when it is needed, rather than being copied into the database. This means that when the data changes in the external database, the change is also reflected in Access.

In the `bpsimple` database, we have a table called `item` that records a unique identifier for each product we sell, a description of that product, a cost price, and a selling price. As an example, let's go through the steps to create a simple Access database to update and report on the product information stored in our sample database system.

1. In Access, create a new blank database. Click Tables in the list on the left side of the window, as shown in Figure 5-18.

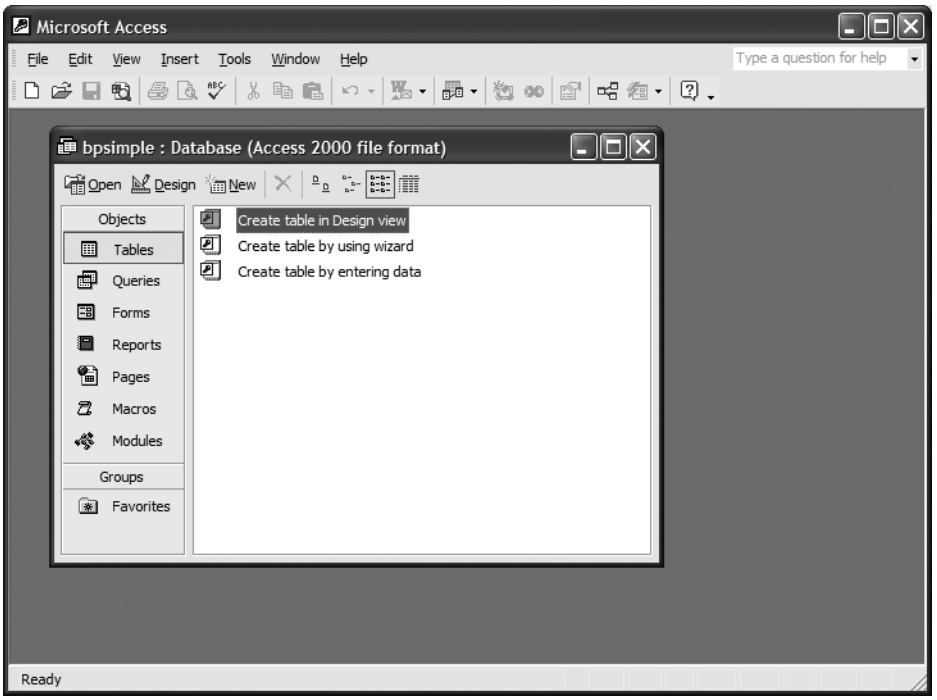


Figure 5-18. *Creating a blank Access database*

2. Click New to bring up the New Table dialog box and select the Link Table option, as shown in Figure 5-19.

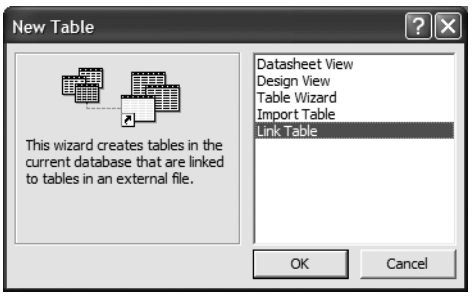


Figure 5-19. *Adding a link table*

3. In the Link dialog box that appears, choose files of type ODBC Databases to bring up the ODBC data source selection dialog box. Select Machine Data Source and the appropriate PostgreSQL database connection, as shown in Figure 5-20. We created a suitable database connection in the "ODBC Setup" section earlier in this chapter,

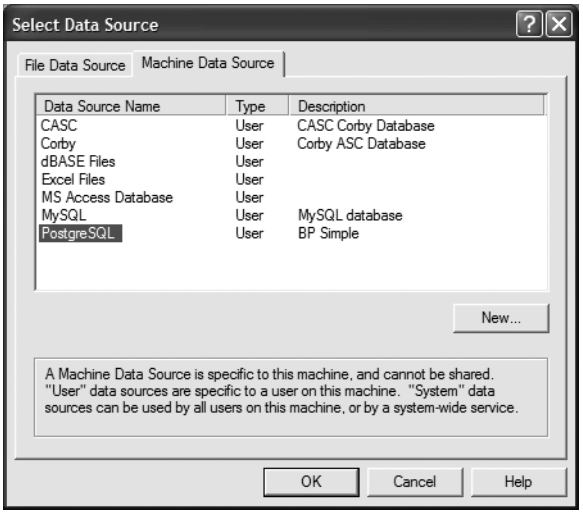


Figure 5-20. *Selecting an ODBC data source*

4. When the connection is made, you are presented with a list of available tables in the remote database. You can choose to link one or more tables from this list. For our example, we will select `public.item` to link the `item` table to our Access database, as shown in Figure 5-21.

Figure 5-21. *Selecting the tables to link*

■ **Note** Before Access can link a table, it needs to know which of the fields in the table it can use to uniquely identify each record. In other words, it needs to know which columns form the primary key. For this table, the `item_id` column is the primary key, so Access will select that. For tables that do not have a defined primary key, Access will prompt you to select a column to use. If a table has a composite key, you can select more than one column.

Now we will see that the Access database has a new table, also called `item`, that we can browse and edit, just as if the data were held in Access. This is depicted in Figure 5-22.

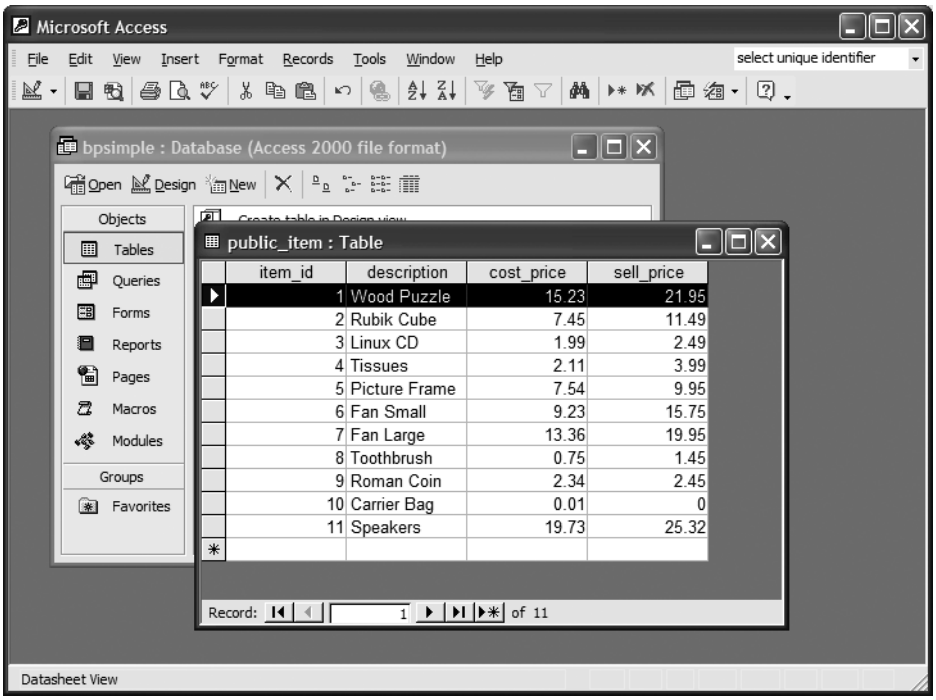


Figure 5-22. *Browsing a link table*

That's just about all there is to linking a PostgreSQL database table to Access.

■ **Note** You might see slightly different screens than the ones in the figures shown here, depending on your version of Windows and Access. If you see an additional column in your table called `oid`, this is the internal PostgreSQL object identifier and can be ignored. To prevent the `object_id` column being shown, be sure to uncheck the OID column options in the ODBC data source configuration.

Entering Data and Creating Reports

We can use the table browser in Access to examine data in the PostgreSQL table and to add more rows. Figure 5-23 shows an Access data-entry form being used to add items to the `item` table. We can use the programming features of Access to create more sophisticated data-entry applications that perform validation on entries or prevent the modification of existing data.

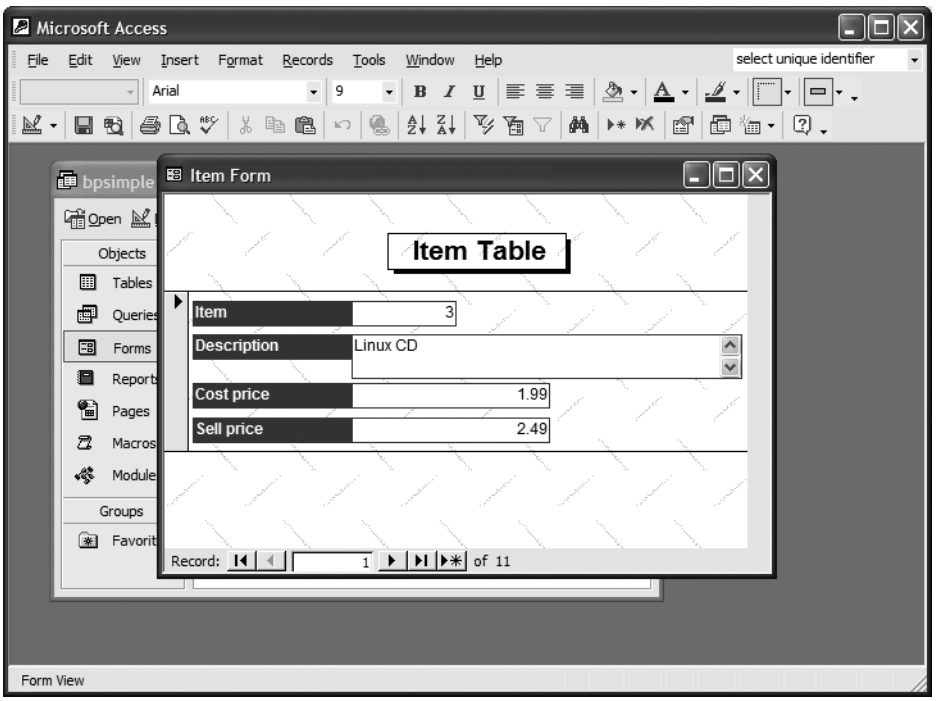


Figure 5-23. *A simple Access data-entry form*

Creating reports is just as easy. Use the Access report designer to generate reports based on the data stored in PostgreSQL tables, just as you would any other Access table. We can include derived columns in the report to answer questions about the data in the table. For example, Figure 5-24 shows an Access report that displays the markup (that is, the difference between the `sell_price` and the `cost_price`) that we are applying to the products in the `item` table.

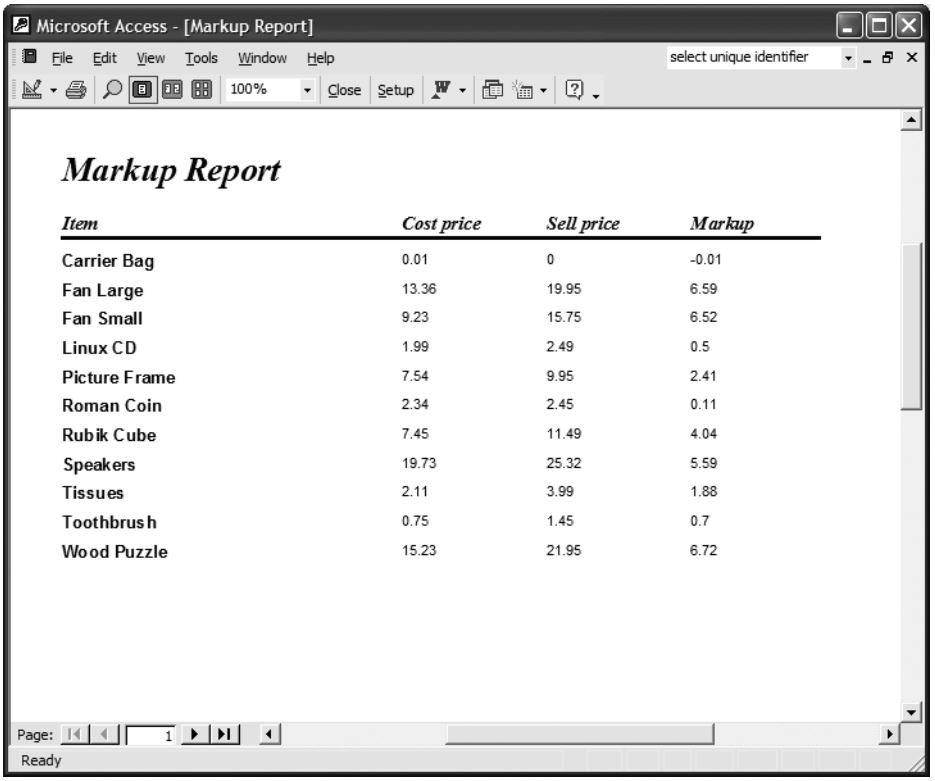


Figure 5-24. *A simple Access report*

Combining Microsoft Access and PostgreSQL increases the number of options you have for creating database applications. The scalability and reliability of PostgreSQL with the familiarity and ease of use of Microsoft Access may be just what you need.

Microsoft Excel

As with Microsoft Access, you can employ Microsoft Excel to add functionality to your PostgreSQL installation. This is similar to the way you can work with Access; you include data in your spreadsheets that is taken from (or rather, linked to) a remote data source. When the data changes, you can refresh the spreadsheet and have the spreadsheet reflect the new data. Once you have made a spreadsheet based on PostgreSQL data, you can use Excel's features, such as charting, to create graphical representations of your data.

Let's extend our report example from Access to make a chart showing the markup we have applied to the products in the `item` table.

1. We need to tell Excel that some portion of a spreadsheet needs to be linked to an external database table. Starting from a blank spreadsheet, choose the menu option to import external data with a new database query, as shown in Figure 5-25.

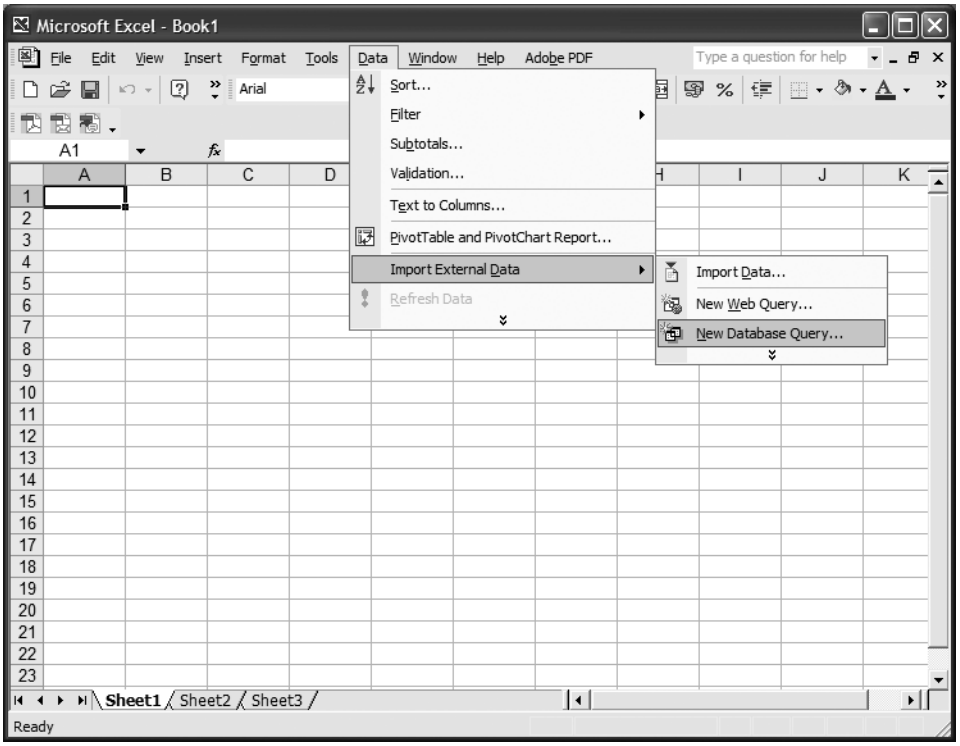


Figure 5-25. *Importing data into Excel*

2. We are presented with an ODBC data source selection dialog box to select our data source, as with Access (see Figure 5-20). Select the appropriate PostgreSQL database connection.

3. When the connection to the database is made, you can choose which table you want to use, and which columns you want to appear in the spreadsheet. For this example, we select the item identifier, the description, and both prices from the `item` table, as shown in Figure 5-26.

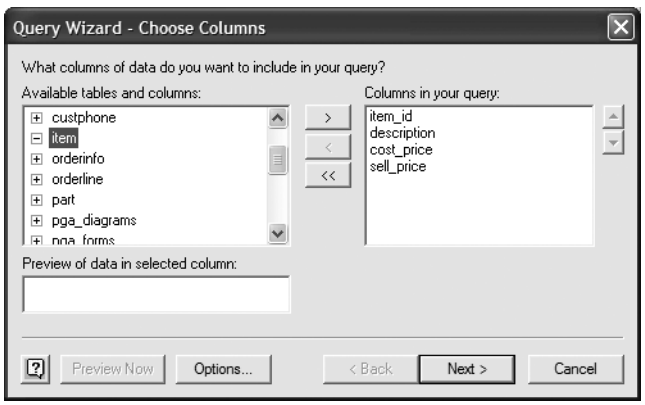


Figure 5-26. *Choosing columns to import into Excel*

4. If you want to restrict the number of rows that appear in your spreadsheet, you can do this by specifying selection criteria at the next dialog box. In Figure 5-27, we select those products with a selling price greater than $2.

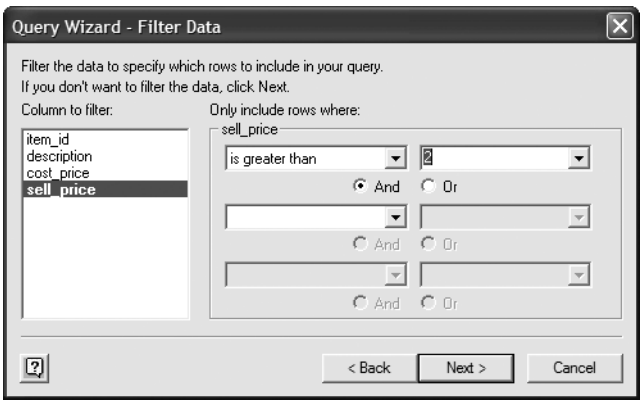


Figure 5-27. *Restricting rows to import*

5. Finally, you can choose to have the data sorted by a particular column or group of columns, in either sort direction. For this example, we choose to sort by the selling price, in ascending order, as shown in Figure 5-28.

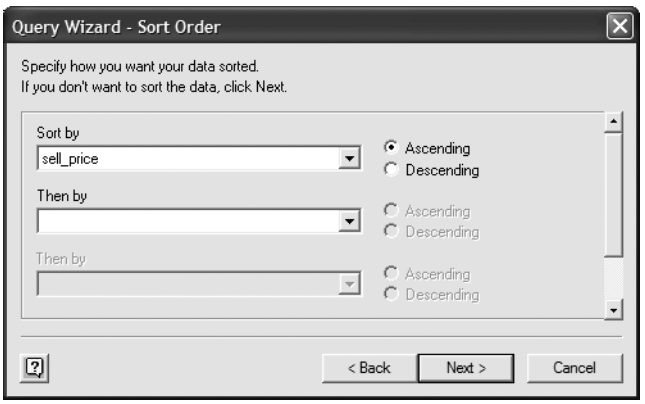


Figure 5-28. *Defining the sort criteria for imported data*

6. Choose to return the data to Excel in the next dialog box.

7. Now, you get the chance to specify where in your spreadsheet you want the data to appear. It is probably a good idea to have data from a PostgreSQL table appear on a worksheet by itself. This is because you need to make sure that you provide for the number of rows increasing as the database grows. You will refresh the spreadsheet data and will need space for the data to expand. However, for this example, we simply allow the data to occupy the top-left area of the sheet, as shown in Figure 5-29.

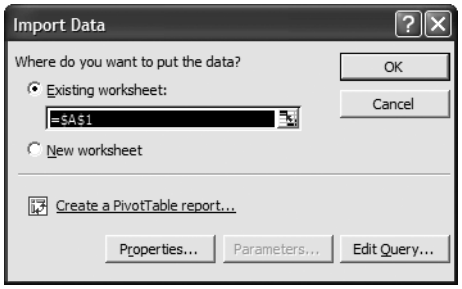
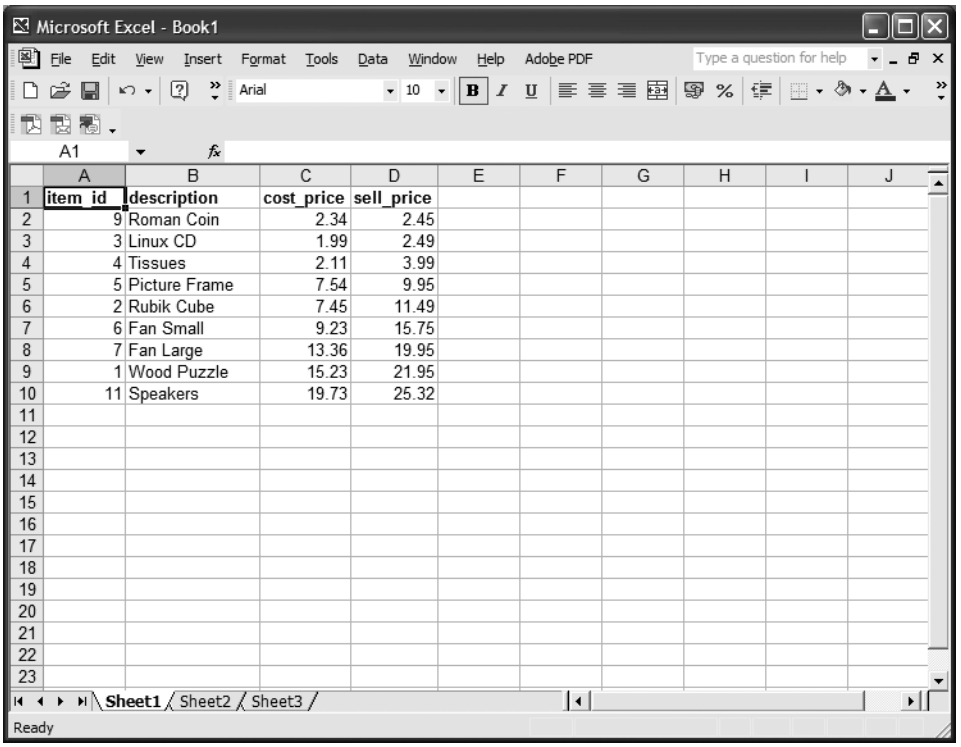


Figure 5-29. *Choosing an import location*

Now we can see the data present in our worksheet, as shown in Figure 5-30.



	A	B	C	D	E	F	G	H	I	J
1	item_id	description	cost_price	sell_price						
2	9	Roman Coin	2.34	2.45						
3	3	Linux CD	1.99	2.49						
4	4	Tissues	2.11	3.99						
5	5	Picture Frame	7.54	9.95						
6	2	Rubik Cube	7.45	11.49						
7	6	Fan Small	9.23	15.75						
8	7	Fan Large	13.36	19.95						
9	1	Wood Puzzle	15.23	21.95						
10	11	Speakers	19.73	25.32						
11										
12										
13										
14										
15										
16										
17										
18										
19										
20										
21										
22										
23										

Figure 5-30. *Viewing imported data in Excel*

We could use this spreadsheet to perform calculations on the data. For example, we might calculate the sales margin being earned from each product by setting up an additional column with an appropriate formula.

■ **Caution** When the data changes in the database, Excel will not automatically update its version of the rows. To make sure that the data you are viewing in Excel is accurate, you must refresh the data. This is simply done by selecting the Refresh Data option on the Data menu.

We can also employ some of Excel's features to add value to our PostgreSQL application. In the example shown in Figure 5-31, we have added a chart showing the markup on each product. It is simply built by using the Excel Chart Wizard and selecting the PostgreSQL data area of the sheet as the source data for the chart. When the data in the PostgreSQL database changes and we refresh the spreadsheet, the chart will automatically update.

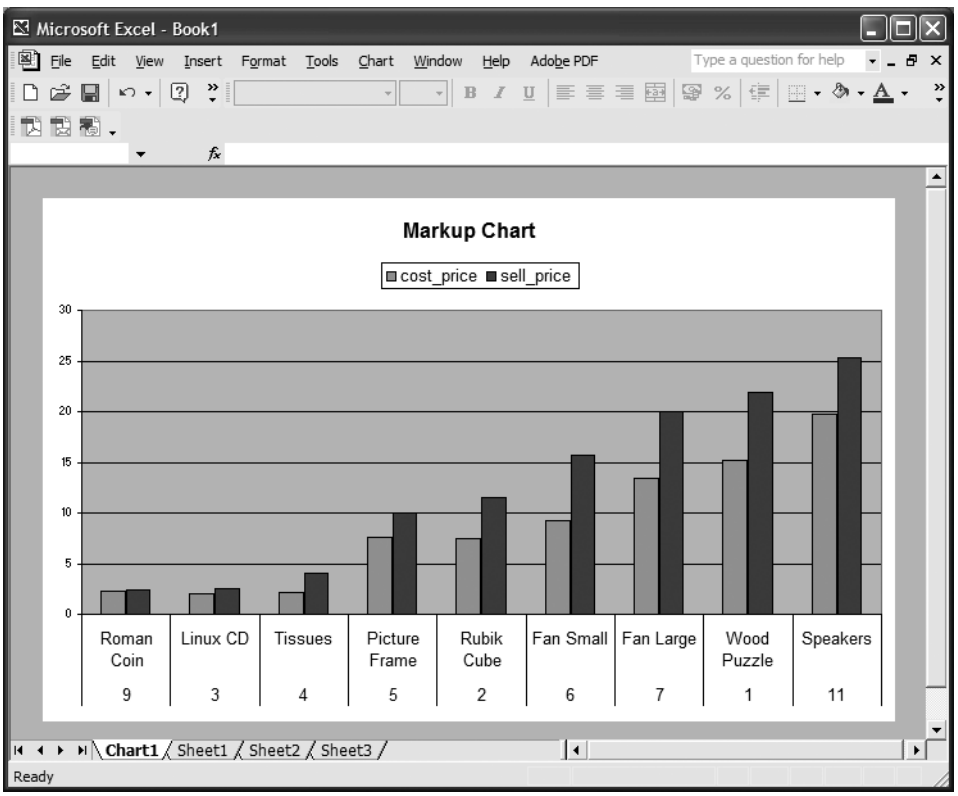


Figure 5-31. *An Excel chart using PostgreSQL data*

Resources for PostgreSQL Tools

A good place to start to look for tools to use with PostgreSQL is pgFoundry, the PostgreSQL project's web site at `http://pgfoundry.org`. The GBorg site at `http://gborg.postgresql.org/` also currently hosts many PostgreSQL-related projects. It is probable that these PostgreSQL project web sites will become merged and be accessible at `http://projects.postgresql.org`.

You can find a list of graphical tools that support PostgreSQL at `http://techdocs.postgresql.org/guides/GUITools`.

A session monitor for PostgreSQL, called `pgmonitor`, is in development and can be found at `http://gborg.postgresql.org/project/pgmonitor`. This is a Tcl/Tk program that allows you to monitor activity on your database. It needs to run on the database server, but it can display on a client machine if you are running the X Window System on UNIX or a UNIX-like operating system.

Summary

In this chapter, we looked at some of the tools we have at our disposal for getting the most out of PostgreSQL. The standard distribution comes with the command-line tool, `psql`, which is capable of carrying out most of the operations we need for creating and maintaining databases.

Database administration can be carried out on a client machine using the very capable pgAdmin III tool or over a network using the browser-based phpPgAdmin tool.

We can view data, design queries graphically, and create data-entry forms using Rekall, for free on Linux, and through a commercial product on Windows.

We can use Microsoft Office products, including Excel and Access, to manipulate and report on data held in a PostgreSQL database. This allows us to combine the scalability and reliability of the PostgreSQL system running on a UNIX or Linux platform with the easy use of familiar tools.

Now that we've reviewed some of the PostgreSQL tools, in the next chapter, we will return to the topic of using SQL to handle the data in a PostgreSQL database, focusing on inserting, updating, and deleting data.

CHAPTER 6

Data Interfacing

So far, we have looked at why a relational database, and PostgreSQL in particular, is a powerful tool for organizing and retrieving data. In the previous chapter, we examined some of the graphical tools, such as pgAdmin III, that can also be used for administering PostgreSQL. We have even looked at how to use Microsoft Access with PostgreSQL and add more functionality to it by using Microsoft Excel. Of course, none of these tools would be much use to us without any data in the database. In Chapter 3, we populated our `bpsimple` database using some SQL scripts.

In this chapter, we will move beyond the basics and learn more about handling data. We are going to look in detail at how to insert data into a PostgreSQL database, update data already in the database, and delete data from a database.

As we progress through this chapter, we will cover the following topics:

- Adding data to the database with `INSERT`
- Inserting data into `serial` columns
- Inserting `NULL` values
- Loading data from text files using the `\copy` command
- Loading data directly from another application
- Updating data in the database with `UPDATE`
- Deleting data from the database with `DELETE`

Adding Data to the Database

Surprisingly perhaps, after the complexities of the `SELECT` statement that we saw in Chapter 4, adding data into a PostgreSQL database is quite straightforward. We add data to PostgreSQL using the `INSERT` statement. We can add data to only a single table at any one time, and generally we do that one row at a time.

Using Basic INSERT Statements

The basic SQL `INSERT` statement has a very simple syntax:

```
INSERT INTO tablename VALUES (list of column values);
```

We specify a list of comma-separated column values, which must be in the same order as the columns in the table.

■ **Caution** Although this syntax is very appealing because of its simplicity, it is also rather dangerous as it relies on knowledge of the table structure—specifically, the order of the columns—which might change if the database is modified to support additional data. Therefore, we urge you to avoid this syntax, and instead use the safer syntax shown later, in the "Using Safer INSERT Statements" section. In the safer syntax, the column names are specified as well as the data values. We present the simple syntax here, because you will see it in common use, but we recommend that you avoid using it.

Try It Out: Use INSERT Statements

Let's add some new rows to the `customer` table. The first thing we must do is to discover the correct column order. This is the same order in which they were listed in the original `CREATE TABLE` command. If we don't have access to that table-creation SQL, which is unfortunately all too common, then we can use the `psql` command-line tool to describe the table, using the `\d` command. Suppose we wanted to look at the definition of the `customer` table in our database (as presented in Chapter 3). We would use the `\d` command to ask for its description to be shown. Let's do that now:

```
bpsimple=# \d customer
                                        Table "public.customer"
   Column    |         Type          |                              Modifiers

-------------+-----------------------+-----------------------------------------
------------------------
 customer_id | integer               | not null default nextval('public.customer
_customer_id_seq'::text)
 title       | character(4)          |
 fname       | character varying(32) |
 lname       | character varying(32) | not null
 addressline | character varying(64) |
 town        | character varying(32) |
 zipcode     | character(10)         | not null
 phone       | character varying(16) |
Indexes:
    "customer_pk" primary key, btree (customer_id)

bpsimple=#
```

The display is slightly confused by the wrapping introduced to get it on the page, but it does show us the column order for our `customer` table. You will notice that the `customer_id` column isn't described exactly as we specified in the `CREATE TABLE` statement we saw in Chapter 3. This is because of the way PostgreSQL implements our `serial` definition of

customer_id. For now, we just need to remember that it is an integer field. We will explain how PostgreSQL implements serial columns in Chapter 8.

To insert character data, we must enclose it in single quotes ('). Numbers do not need any special treatment. For NULLs, we just write NULL, or, as we will see later in a more complex form of the INSERT statement, simply provide no data for that column.

Now that we know the column order, we can write our INSERT statement like this:

```
bpsimple=# INSERT INTO customer VALUES(16, 'Mr', 'Gavyn', 'Smith',
bpsimple-# '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');
INSERT 17331 1

bpsimple=#
```

The exact number you see after the INSERT will almost certainly be different in your case. The important thing is that PostgreSQL has inserted the new data. The first number is actually an internal PostgreSQL identification number, called an *OID*, which is normally hidden.

■ **Note** The OID (Object IDentification) number is a unique, normally invisible number assigned to every row in PostgreSQL. When you initialize the database, a counter is created. This counter is used to uniquely number every row. Here, the INSERT command has been executed, 17331 is the OID assigned to the new row, and 1 is the number of rows inserted. This OID number is not part of standard SQL, and it will not normally be sequential within a table, so we urge you to be aware of its existence but never to use it in applications. Starting in release 8.0, PostgreSQL has the option to avoid creating OIDs on tables, so even their very existence is not reliable.

We can easily check that the data has been inserted correctly by using a SELECT statement to retrieve it, like this:

```
bpsimple=# SELECT * FROM customer WHERE customer_id > 15;
customer_id | title | fname | lname |  addressline  |   town    | zipcode | phone
------------+-------+-------+-------+---------------+-----------+---------+---------
         16 | Mr    | Gavyn | Smith | 23 Harlestone | Milltown | MT7 7HI | 746 3725
(1 row)

bpsimple=#
```

Depending on the size of your terminal window, the display may be wrapped, but you should be able to see that the data was correctly inserted.

Suppose that we want to insert another row, where the last name is O'Rourke. What do we do with the single quote that is already in the data? If a single quote must appear in a character string, we precede it with a backslash (\). The backslash is called an *escape* character, and it indicates that the following character has no special meaning and is part of the data. So, to insert Mr. O'Rourke's data, we escape the quote in his name using a single backslash (\), like this:

```
INSERT INTO customer VALUES(17, 'Mr', 'Shaun', 'O\'Rourke',
    '32 Sheepy Lane', 'Milltown', 'MT9 8NQ', '746 3956');
```

Check that the data has been inserted:

```
bpsimple=# SELECT * FROM customer WHERE customer_id > 15;
customer_id | ti | fname |   lname  |  addressline   |   town   | zipcode |  phone
------------+----+-------+----------+----------------+----------+---------+---------
         16 | Mr | Gavyn | Smith    | 23 Harlestone  | Milltown | MT7 7HI | 746 3725
         17 | Mr | Shaun | O'Rourke | 32 Sheepy Lane | Milltown | MT9 8NQ | 746 3956
(2 rows)

bpsimple=#
```

■ **Note** In some cases, to fit the output on the page, we've needed to make some slight changes. For example, here, we've abbreviated `title` to `ti`. These adjustments are just for legibility, and we've made sure that the point of each example is clear.

How It Works

We used the `INSERT` statement to add data to the `customer` table, specifying column values in the same order as they were created in the table. To add a number to a column, just write the number. To add a string, enclose it in single quotes. To insert a single quote into the string, we must precede the single quote with a backslash character (`\`). If we ever need to insert a backslash character, then we would write a pair, like this `\\`.

Suppose we want to insert another row, where the address is something strange, such as `Midtown Street A\33`. What do we do with the single backslash that is already in the data? We would escape the single backslash by using two backslashes, like this:

```
INSERT INTO customer VALUES(18, 'Mr', 'Jeff', 'Baggott',
    'Midtown Street A\\33', 'Milltown', 'MT9 8NQ', '746 3956');
```

This is how it looks:

```
bpsimple=# SELECT * FROM customer WHERE addressline='Midtown Street A\\33';
c_id  | ti | fname |  lname  |     addressline     |   town   |  zipcode |  phone
------+----+-------+---------+---------------------+----------+----------+----------
   18 | Mr | Jeff  | Baggott | Midtown Street A\33 | Milltown | MT9 8NQ  | 746 3956
(1 row)

bpsimple=#
```

Using Safer INSERT Statements

While `INSERT` statements like the ones we just tried out work, it is not always convenient to specify every single column or to get the data order exactly the same as the table column order. This adds an element of risk in that we may accidentally write an `INSERT` statement with the column data in the wrong order. This would result in the addition of incorrect data to our database.

In the previous example, suppose we had erroneously exchanged the position of the fname and lname columns. The data would have been inserted successfully, because both columns are text columns, and PostgreSQL would have been unable to detect our mistake. If we had later asked for a list of the last names of our customers, Gavyn would have appeared as a valid customer last name, rather than Smith, as we intended.

Poor-quality data, or just plain incorrect data, is a major problem in databases, and we generally take as many precautions as we can to ensure that only correct data gets in. Simple mistakes might be easy to spot in our sample database with just tens of rows, but in a database with tens of thousands of customers, spotting mistakes—particularly within data with unusual names—would be very difficult indeed.

Fortunately, there is a slight variation of the INSERT statement that is both easier to use and much safer as well:

```
INSERT INTO tablename(list of column names)
VALUES (list of column values corresponding to the column names);
```

In this variant of the INSERT statement, we must list the column names and data values for those columns in the same order, which can be different from the order we used when we created the table. Using this variant, we no longer need to know the order in which the columns were defined in the database. We also have a nice, clear, almost side-by-side list of column names and the data we are about to insert into them.

Try It Out: Insert Values Corresponding to Column Names

Let's add another row to the database, this time explicitly naming the columns, like this:

```
INSERT INTO customer(customer_id, title, fname, lname, addressline, ...)
VALUES(19, 'Mrs', 'Sarah', 'Harvey', '84 Willow Way', ...)
```

We can enter an INSERT statement over several lines, making it easier to read, and check that we have the column names and data values in the same order.

Let's execute an example, typing it in over several lines so it is easier to read:

```
bpsimple=# INSERT INTO
bpsimple-# customer(customer_id, title, lname, fname, addressline, town,
bpsimple-#           zipcode, phone)
bpsimple-# VALUES(19, 'Mrs', 'Harvey', 'Sarah',  '84 Willow Way', 'Lincoln',
bpsimple-#           'LC3 7RD', '527 3739');
INSERT 22592 1

bpsimple=#
```

Notice how much easier it is to compare the names of the fields with the values being inserted into them. We deliberately swapped the fname and lname column positions, just to show it could be done. You can use any column order you like; all that matters is that the values match the order in which you list the columns.

You will also notice the psql prompt changes on subsequent lines, and it remains changed until we terminate the command with a semicolon.

■ **Tip** We strongly recommend that you always use the named column form of the `INSERT` statement, because the explicit naming of columns makes it much safer to use.

Inserting Data into Serial Columns

At this point, it is time to confess to a minor sin we have been committing with the `customer_id` column. Up to this point in the chapter, we have not covered how to insert data into some columns of a table while leaving others alone. With the second form of the `INSERT` statement, using named columns, we can do this and see how it is particularly important when inserting data into tables with `serial` type columns.

You will remember from Chapter 2 that we met the rather special data type `serial`, which is effectively an integer, but automatically increments to give us an easy way of creating unique ID numbers for each row. So far in this chapter, we have been inserting data into rows, providing a value for the `customer_id` column, which is a `serial` type data field.

Let's take a look at the data in our `customer` table so far:

```
bpsimple=# SELECT customer_id, fname, lname, addressline FROM customer;
 customer_id |   fname   |  lname   |     addressline
-------------+-----------+----------+---------------------
           1 | Jenny     | Stones   | 27 Rowan Avenue
           2 | Andrew    | Stones   | 52 The Willows
           3 | Alex      | Matthew  | 4 The Street
           4 | Adrian    | Matthew  | The Barn
           5 | Simon     | Cozens   | 7 Shady Lane
           6 | Neil      | Matthew  | 5 Pasture Lane
           7 | Richard   | Stones   | 34 Holly Way
           8 | Ann       | Stones   | 34 Holly Way
           9 | Christine | Hickman  | 36 Queen Street
          10 | Mike      | Howard   | 86 Dysart Street
          11 | Dave      | Jones    | 54 Vale Rise
          12 | Richard   | Neill    | 42 Thatched Way
          13 | Laura     | Hardy    | 73 Margarita Way
          14 | Bill      | O'Neill  | 2 Beamer Street
          15 | David     | Hudson   | 4 The Square
          16 | Gavyn     | Smith    | 23 Harlestone
          17 | Shaun     | O'Rourke | 32 Sheepy Lane
          18 | Jeff      | Baggott  | Midtown Street A\33
          19 | Sarah     | Harvey   | 84 Willow Way
(19 rows)

bpsimple=#
```

Certainly, all looks well. However, there is a slight problem because, by forcing values into the `customer_id` column, we have inadvertently confused PostgreSQL's internal `serial` counter.

Suppose we try inserting another row, this time allowing the `serial` type to provide our automatically incrementing `customer_id` value:

```
bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town,
bpsimple-# zipcode, phone)
bpsimple-# VALUES('Mr', 'Steve', 'Clarke',  '14 Satview way', 'Lincoln',
bpsimple-# 'LC4 3ED', '527 7254');
ERROR:  duplicate key violates unique constraint "customer_pk"

bpsimple=#
```

Clearly, something has gone wrong, since we did not provide any duplicate values. What has happened is that earlier in the chapter, when we were providing values for `customer_id`, we bypassed PostgreSQL's automatic allocation of IDs in the `serial` column and caused the automatic allocation system to get out of step with the actual data in the table.

■ **Caution** Avoid providing values for `serial` data columns when inserting data.

The out-of-step sequence problem is reasonably rare, but most commonly occurs as a result of one of the following:

- You have dropped and re-created the table, but did not drop and re-create the sequence (PostgreSQL version 8.0 and later does this automatically).
- You mixed styles of adding data—allowing PostgreSQL to pick values for some `serial` columns and explicitly specifying values for some `serial` columns yourself.

In this case, the latter occurred. Having gotten ourselves into a bit of a mess, how do we recover? The answer is that we need to give PostgreSQL a helping hand, and put its internal sequence number back in step with the actual data.

Accessing Sequence Numbers

When the `customer` table was created, the `customer_id` column was defined as having type `serial`. You may have noticed that PostgreSQL then gave us some informational messages, saying that it was creating a `customer_customer_id_seq` sequence. Also, when we ask PostgreSQL to describe the table using `\d`, we see the column is specially defined:

```
customer_id integer not null default nextval('customer_customer_id_seq'::text)
```

PostgreSQL has created a special counter for the column, a *sequence*, which it can use to generate unique IDs. Notice that the sequence is always named `<tablename>_<columnname>_seq`. The default behavior for the column has been automatically specified by PostgreSQL to be the result of the function `nextval('customer_customer_id_seq')`. When we failed to provide data for the column in our `INSERT` statement, this is the function that was being automatically executed by PostgreSQL for us. By inserting or providing data to this column, we have upset this automatic mechanism, since the function will not get called if data is provided. Fortunately, we are not reduced to deleting all the data from the table and starting again, because PostgreSQL allows us to directly manipulate the sequence number.

When inserting data like this, you can usually find the value of a sequence number using the `currval` function:

```
currval('sequence name');
```

PostgreSQL will tell you the current value of the sequence number:

```
bpsimple=# SELECT currval('customer_customer_id_seq');
 currval
---------
      16
(1 row)

bpsimple=#
```

■ **Note** Strictly speaking, `currval` tells you the value from the last call to `nextval`, so for this to work, either a new row will need to have been inserted or `nextval` called explicitly in this `psql` session.

As you can see, PostgreSQL thinks that the current number for the last row in the table is 16, but, in fact, the last row is 19. When we try to insert data into the `customer` table, leaving the `customer_id` column to PostgreSQL, it attempts to provide a value for the column by calling the `nextval` function:

```
nextval('sequence number');
```

This function first increments the provided sequence number, and then returns the result. We can try this directly:

```
bpsimple=# SELECT nextval('customer_customer_id_seq');
 nextval
---------
      17
(1 row)

bpsimple=#
```

Of course, we could get to the correct value for the sequence by repeatedly calling `nextval`, but that would not be much use if the value were too large or too small. Instead, we can use the `setval` function:

```
setval('sequence number', new value);
```

First, we need to discover what the sequence value should be. This is accomplished by selecting the maximum value of the column that is already in the database. To do this, we will use the `max(`*`column name`*`)` function, which simply tells us the maximum value stored in a column:

```
bpsimple=# SELECT max(customer_id) FROM customer;
 max
-----
  19
(1 row)

bpsimple=#
```

PostgreSQL will respond with the largest number that it found in the `customer_id` column in the `customer` table. (We will discuss the `max(`*`column name`*`)` function in more detail in the next chapter.) Now we set the sequence, using the function `setval(`*`sequence, value`*`)`, which allows us to set a sequence to any value we choose. The current largest value in the table is 19, and the sequence number is always incremented before its value is used. Therefore, the sequence should normally have the same number as the current biggest value in the table:

```
bpsimple=# SELECT setval('customer_customer_id_seq', 19);
 setval
--------
     19
(1 row)

bpsimple=#
```

Now that the sequence number is correct, we can insert our data, allowing PostgreSQL to provide the value for the `serial` column `customer_id`:

```
bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town,
bpsimple-# zipcode, phone) VALUES('Mr', 'Steve', 'Clarke',  '14 Satview
bpsimple-# way', 'Lincoln', 'LC4 3ED', '527 7254');
INSERT 21459 1
bpsimple=#
```

Success! PostgreSQL is now back in step, and it will continue to create `serial` values correctly.

PostgreSQL versions 7.3 and later allow you to use the `DEFAULT` keyword in `INSERT` statements to indicate that a column's declared default value should be inserted, which is especially useful in keeping sequence values in line. Where we have been adding rows using explicit `customer_id` values, we can write statements like this instead:

```
INSERT INTO customer(customer_id, title, fname, lname,
    addressline, town, zipcode, phone)
VALUES(DEFAULT, 'Mrs', 'Sarah', 'Harvey',
    '84 Willow Way', 'Lincoln', 'LC3 7RD', '527 3739');
```

Here, the default value of the `customer_id` is the next value in the sequence, as `customer_id` is a `serial` column.

We will return to the topic of default column values in Chapter 8.

Inserting NULL Values

We briefly mentioned in Chapter 2 that NULL values could be inserted into columns using the INSERT statement. Let's look at this in a little more detail.

If you are using the first form of the INSERT statement, where you insert data into the columns in the order they were defined when the table was created, you simply write NULL in the column value. Note that you must not use quotes, as this is not a string. You should also remember that NULL is a special undefined value in SQL, not the same as an empty string.

Consider our previous example:

```
INSERT INTO customer VALUES(16, 'Mr', 'Gavyn', 'Smith',
    '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');
```

Suppose that we did not know the first name. The table definition allows NULL in the fname column, so adding data without knowing the first name is perfectly valid. If we had written this:

```
INSERT INTO customer VALUES(16, 'Mr', '', 'Smith',
    '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');
```

it would not be what we intended, because we would have added an empty string as the first name, perhaps implying that Mr. Smith has no first name. What we intended was to use a NULL, because we do not know the first name.

The correct INSERT statement would have been as follows:

```
INSERT INTO customer VALUES(16, 'Mr', NULL, 'Smith',
    '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');
```

Notice the lack of quotes around NULL. If quotes had been used, fname would have been set to the string 'NULL', rather than the *value* NULL.

Using the second (safer) form of the INSERT statement, where columns are explicitly named, it is much easier to insert NULL values where we neither list the column nor provide a value for it, like this:

```
INSERT INTO customer(title, lname, addressline, town, zipcode, phone)
VALUES('Mr', 'Smith', '23 Harlestone', 'Milltown', 'MT7 7HI', '746 3725');
```

Notice that the fname column is neither listed nor is a value defined for it. Alternatively, we could have listed the column, and then written NULL in the value list.

This will not work if we try to add a NULL value in a column that is defined as not allowing NULL values. Suppose we try to add a customer with no last name (lname) column:

```
bpsimple=# INSERT INTO customer(title, fname, addressline, town, zipcode,
bpsimple-# phone) VALUES('Ms', 'Gill', '27 Chase Avenue', 'Lowtown',
bpsimple-# 'LT5 8TQ', '876 1962');
ERROR:  null value in column "lname" violates not-null constraint
bpsimple=#
```

Notice that we did not provide a value for lname, so the INSERT was rejected, because the customer table is defined to not allow NULL in that column:

```
bpsimple=# \d customer
                                        Table "public.customer"
   Column    |         Type          |                              Modifiers

-------------+-----------------------+-----------------------------------------
------------------------
 customer_id | integer               | not null default nextval('public.customer
_customer_id_seq'::text)
 title       | character(4)          |
 fname       | character varying(32) |
 lname       | character varying(32) | not null
 addressline | character varying(64) |
 town        | character varying(32) |
 zipcode     | character(10)         | not null
 phone       | character varying(16) |
Indexes:
    "customer_pk" primary key, btree (customer_id)

bpsimple=#
```

We will see in Chapter 8 how we can more generally define explicit default values to be used in columns when data is inserted with no value, by specifying a default value for a column.

Using the \copy Command

Although INSERT is the standard SQL way of adding data to a database, it is not always the most convenient. Suppose we had a large number of rows to add to the database, but already had the actual data available, perhaps in a spreadsheet. One way to get started on inserting data into the database would be to use a spreadsheet export, so we would probably export the spreadsheet as a comma-separated values (CSV) file. We can then use a text editor like Emacs, or at least one with a macro facility, to convert all our data into INSERT statements.

Consider the following data:

```
Miss,Jenny,Stones,27 Rowan Avenue,Hightown,NT2 1AQ,023 9876
Mr,Andrew,Stones,52 The Willows,Lowtown,LT5 7RA,876 3527
Miss,Alex,Matthew,4 The Street,Nicetown,NT2 2TX,010 4567
```

We might transform it into a series of INSERT statements, so it looks like this:

```
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Miss','Jenny','Stones','27 Rowan Avenue','Hightown',
      'NT2 1AQ','023 9876');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
 VALUES('Mr','Andrew','Stones','52 The Willows','Lowtown',
       'LT5 7RA','876 3527');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
 VALUES('Miss','Alex','Matthew','4 The Street','Nicetown',
       'NT2 2TX','010 4567');
```

Then save it in a text file with a .sql extension.

We could then use the \i command in psql to execute the statements in the file. This is how the pop_customer.sql file works (we used this in Chapter 3 to initially populate our database). Notice here that we allowed PostgreSQL to generate the unique customer_id value.

This isn't very convenient, however. It would be much nicer if data could be moved between flat files and the database in a more general way. There are a couple of ways of doing this in PostgreSQL. Rather confusingly, both are called the copy command. There is a PostgreSQL SQL command called COPY, which can save and restore data to flat files, but its use is limited to the database administrator, as files are read and written on the server to which normal users would not necessarily have access. More useful is the general-purpose \copy command, which implements almost all the functionality of COPY, but can be used by everyone, and data is read and written on the client machine. The SQL-based COPY command is, therefore, almost totally redundant.

■ **Note** The COPY command does have one advantage: it is significantly quicker than \copy, because it executes directly in the server process. The \copy command executes in the client process, potentially having to pass all the data across a network. COPY can also be slightly more reliable when errors occur. Unless you have very large amounts of data, however, the difference will not be that noticeable.

The \copy command has this basic syntax for importing data:

```
\copy tablename FROM 'filename'
    [USING DELIMITERS 'a single character to use as a delimiter']
    [WITH NULL AS 'a string that means NULL']
```

It looks a little imposing, but it is quite simple to use. The sections in square braces, [], are optional, so you only need to use them if required. Do notice, however, that the filename needs to be enclosed in single quotes.

The option USING DELIMITERS '*a single character to use as a delimiter*' allows you to specify how each column is separated in the input file. By default, a tab character is assumed to separate columns in the input data. In our case, we will assume that we have started with a CSV file that we have exported from a spreadsheet. In practice, the CSV format is often not a good choice because the comma character can appear in the data, and address data is particularly prone to containing comma characters. Unfortunately, spreadsheets often do not offer sensible alternatives to CSV file exports, so you may need to work with what you've got. Given the choice, a pipe character, |, is often useful as a delimiter, as it very rarely appears in user data.

The option WITH NULL AS '*a string that means NULL*' allows you to specify a string that should be interpreted as NULL. By default, \N is assumed. Notice that in the \copy command, you must include single quotes around the string, because that tells PostgreSQL that it is a string, although quotes will not be expected in the actual data. So, if you want the string NOTHING to be loaded as a NULL value in the database, you would specify the option WITH NULL AS 'NOTHING'. Then if we did not know Mr. Hudson's first name, for example, the data should look like this:

```
15,Mr,NOTHING,Hudson,4 The Square, Milltown,MT2 6RT,961 4526
```

When inserting data directly, it is very important to watch out that the data is "clean." You need to ensure that no columns are missing, all quote characters have been correctly escaped

with a backslash, there are no binary characters present, and so on. PostgreSQL will catch most of these mistakes for you, and load only valid data, but untangling several thousand rows of data that have almost been completely loaded is a slow, unreliable, and unrewarding job. It is well worth going to the effort to clean the data as much as possible *before* attempting to "bulk load" it with the `\copy` command.

Try It Out: Load Data Using \copy

Let's create some additional customer data in a `cust.txt` file that looks like this:

```
21,Miss,Emma,Neill,21 Sheepy Lane,Hightown,NT2 1YQ,023 4245
22,Mr,Gavin,Neill,21 Sheepy Lane,Hightown,NT2 1YQ,023 4245
23,Mr,Duncan,Neill,21 Sheepy Lane,Hightown,NT2 1YQ,023 4245
```

You can create the simple `cust.txt` file using any text editor, or use the file included with this book's downloadable code (available from the Downloads section of the Apress web site, at `http://www.apress.com`). Conveniently, there are no `NULL`s to worry about, so we just need to specify the comma as the column separator. To load this data, execute this command:

```
\copy customer from 'cust.txt' using delimiters ','
```

Notice there is no semicolon (`;`) at the end of this command, since it is a `\` command directly to `psql`, not SQL. `psql` responds with the rather brief `\.`, which tells us that all is well.

Then execute the following:

```
SELECT * FROM customer;
```

We will see that the additional rows have been added.

There is, however, a slight problem lurking. Remember the sequence number that can get out of step? Unfortunately, using `\copy` to load data is one way this can happen. Let's check what has happened to our sequence number:

```
bpsimple=# SELECT max(customer_id) FROM customer;
 max
-----
  23
(1 row)

bpsimple=# SELECT currval('customer_customer_id_seq');
 currval
---------
      21
(1 row)

bpsimple=#
```

Oops! The maximum value stored in `customer_id` is currently 23, so the next ID allocated should be 24, but the sequence is going to try to allocate 22 as the next value. Never mind—it's easy to correct:

```
bpsimple=# SELECT setval('customer_customer_id_seq', 23);
 setval
--------
     23
(1 row)

bpsimple=#
```

How It Works

We used the `\copy` command to directly load data that had been exported from a spreadsheet in CSV format into our `customer` table. We subsequently had to correct the sequence number that generates `customer_id` numbers for the `serial` column `customer_id` in the table, which takes significantly less effort than that we would have needed to expend to convert our CSV format data into a series of `INSERT` statements.

Loading Data Directly from Another Application

If the data already resides in a desktop database, such as Microsoft Access, there is an even easier way to load the data into PostgreSQL. We can simply attach the PostgreSQL table to the Access database via ODBC and insert data into a PostgreSQL table.

Often, when you are doing this, you will find that your existing data is not quite what you need, or that it needs some reworking before being inserted into its final destination table. Even if the data is in the correct format, it is often a good idea not to attempt to insert it directly into the database, but rather to first move it to a loading table, and then transfer it from this loading table to the real table. Using an intermediate loading table is a common method in real-world applications for inserting data into a database, particularly when the quality of the original data is uncertain. The data is first loaded into the database in a holding table, checked, corrected if necessary, and then moved into the final table.

Usually, you will write a custom application or stored procedure to check and correct the data, a topic covered in detail in Chapter 10. Once the data is ready to load into the final table though, there is a useful variant of the `INSERT` command that allows us to move data between tables, transferring multiple rows in one command. It is the only time an `INSERT` statement affects multiple rows with a single statement. This is the `INSERT INTO` statement.

The syntax for inserting data from one table into another is as follows:

```
INSERT INTO tablename(list of column names) SELECT normal select statement
```

Try It Out: Load Data Between Tables

Suppose we have a holding table, `tcust`, that has some additional customer data to be loaded into our master `customer` table. We will make our holding table definition look like this:

```
CREATE TABLE tcust
(
    title                       char(4)                 ,
    fname                       varchar(32)             ,
    lname                       varchar(32)             ,
    addressline                 varchar(64)             ,
    town                        varchar(32)             ,
    zipcode                     char(10)                ,
    phone                       varchar(16)
);
```

Notice that there are no primary keys or constraints of any kind. It is normal when cross-loading data into a loading table to make it as easy as possible to get the data into that table. Removing the constraints makes this easier. Also notice that all the required columns are there, except the `customer_id` sequence number, which PostgreSQL can create for us as we load the data.

Suppose we have loaded some data into `tcust` (via ODBC, `\copy`, or some other method), validated, and corrected it. (A suitable script for creating and populating the `tcust` table is included in this book's downloadable code, available from the Apress web site.) Then a `SELECT` output looks like this:

```
bpsimple=# SELECT * FROM tcust;
 title | fname |  lname  |  addressline   |  town   |  zipcode   |  phone
-------+-------+---------+----------------+---------+------------+----------
 Mr    | Peter | Bradley | 72 Milton Rise | Keynes  | MK41 2HQ   |
 Mr    | Kevin | Carney  | 43 Glen Way    | Lincoln | LI2 7RD    | 786 3454
 Mr    | Brian | Waters  | 21 Troon Rise  | Lincoln | LI7 6GT    | 786 7243
(3 rows)

bpsimple=#
```

The first thing to notice is that we have not yet managed to find a phone number for Mr. Bradley. This may or may not be a problem. Let's decide that, for now, we don't wish to load this row, but we do wish to load all the other customers. In a real-world scenario, we may be trying to load hundreds of new customers, and it is quite probable that we will want to load groups of them as the data for each group is validated or cleaned.

The first part of the `INSERT` is quite easy to write. We will use the full syntax of `INSERT`, specifying precisely the columns we wish to load. This is normally the sensible choice:

```
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
```

Notice that we do not specify that we are loading the `customer_id`. You will remember that by leaving this blank, we allow PostgreSQL to automatically create values for us, which is always the safer way to allow `serial` values to be created.

We now need to write the SELECT part of the statement, which will feed this INSERT statement. Remember that we do not wish to insert the information for Mr. Bradley yet, because his phone number is set to NULL, as we are still trying to find it. We could, if we wanted to, load Mr. Bradley's data, since the phone column *will* accept NULL values. What we are doing here is applying a slightly more stringent real-world usage rule to the data than is required by the low-level database rules. We write a SELECT statement like this:

```
SELECT title, fname, lname, addressline, town, zipcode, phone FROM tcust
WHERE phone IS NOT NULL;
```

Of course, this is a perfectly valid statement on its own. Let's test it:

```
bpsimple=# SELECT title, fname, lname, addressline, town, zipcode, phone
bpsimple-# FROM tcust WHERE phone IS NOT NULL;
 title | fname | lname  |  addressline  |  town  |  zipcode   |  phone
-------+-------+--------+---------------+--------+------------+----------
 Mr    | Kevin | Carney | 43 Glen Way   | Lincoln | LI2 7RD   | 786 3454
 Mr    | Brian | Waters | 21 Troon Rise | Lincoln | LI7 6GT   | 786 7243
(2 rows)

bpsimple=#
```

That looks correct. It finds the rows we need, and the columns are in the same order as the INSERT statement. So, we can now put the two statements together and execute them, like this:

```
bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town,
bpsimple-# zipcode, phone) SELECT title, fname, lname, addressline, town,
bpsimple-# zipcode, phone FROM tcust WHERE phone IS NOT NULL;
INSERT 0 2
bpsimple=#
```

Notice that psql tells us that two rows have been inserted. Now, being extra cautious, let's fetch those rows from the customer table, just to be absolutely sure they were loaded correctly:

```
bpsimple=# SELECT customer_id, fname, lname, addressline FROM customer WHERE
bpsimple-# town = 'Lincoln';
 customer_id   fname | lname  |  addressline
-------------+-------+--------+----------------
          19 | Sarah | Harvey | 84 Willow Way
          20 | Steve | Clarke | 14 Satview way
          24 | Brian | Waters | 21 Troon Rise
          25 | Kevin | Carney | 43 Glen Way
(4 rows)

bpsimple=#
```

We actually get more than two rows, because we already had customers from Lincoln. We can see, however, that our data has been inserted correctly, and customer_id values were created for us.

Now that some of the data from tcust has been loaded into the live customer table, we would normally delete those rows from tcust. For the purposes of the example, we are going to leave that data alone for now and delete it in a later example.

How It Works

We specified the columns we wanted to load in the customer table, and then selected the same set of data, in the same order from the tcust table. We did not specify that we would load the customer_id column, so PostgreSQL used its sequence numbers to generate unique IDs for us.

An alternative method, which you may find easier, particularly if there is a lot of data to load, is to add an additional column to the temporary table, perhaps a column isvalid of type boolean. You then load all the data into the temporary table, and set all the isvalid values to false, using the UPDATE statement that we will meet more formally in the next section of this chapter:

```
UPDATE tcust SET isvalid = 'false';
```

We have not specified a WHERE clause; therefore, all rows have the isvalid column set to false. We can then continue work on the data, modifying it where necessary. When we are happy that a row is correct and complete, we set the isvalid column to true. We can then load the corrected data, selecting only the rows where isvalid is true:

```
bpsimple=# INSERT INTO customer(title, fname, lname, addressline, town,
bpsimple-#  zipcode, phone)
bpsimple-# SELECT title, fname, lname, addressline, town, zipcode, phone
bpsimple-#  FROM tcust WHERE isvalid = true;
```

Once these rows are loaded, we can remove them from the tcust table, like this:

```
DELETE FROM tcust WHERE isvalid = true;
```

Then continue to work on the remaining data in the tcust table. (We will discuss the DELETE statement near the end of this chapter.)

Updating Data in the Database

Now we know how to get data into the database by using INSERT, and how to retrieve it again, using SELECT. Unfortunately, data does not tend to stay static for very long. People move to different addresses, change phone numbers, and so on. We need a way of updating the data in the database. In PostgreSQL, as in all SQL-based databases, this is done with the UPDATE statement.

Using the UPDATE Statement

The UPDATE statement is remarkably simple. Its syntax is as follows:

```
UPDATE tablename SET columnname = value  WHERE condition
```

If we want to set several columns at the same time, we simply specify them as a comma-separated list, like this:

```
UPDATE customer SET town = 'Leicester', zipcode = 'LE4 2WQ' WHERE some condition
```

We can update as many columns simultaneously as we like, provided that each column appears only once. You will notice that you can use only a single table name. This is due to the syntax of SQL. In the rare event that you need to update two separate, but related, tables, you must write two separate UPDATE statements. You can put those UPDATE statements into a *transaction* to ensure that either both updates are performed or no updates are performed. We will look at transactions more closely in Chapter 9.

Try It Out: Use the UPDATE Statement

Suppose we have now tracked down the phone number of Mr. Bradley (missing from our tcust table), and want to update the data into our live customer table. The first part of the UPDATE statement is easy:

```
UPDATE tcust SET phone = '352 3442'
```

Now we need to specify the row to update, which is simply:

```
WHERE fname = 'Peter' and lname = 'Bradley';
```

With UPDATE statements, it is always a good idea to check the WHERE clause. Let's do that now:

```
bpsimple=# SELECT fname, lname, phone FROM tcust
bpsimple-# WHERE fname = 'Peter' AND lname = 'Bradley';
 fname |  lname  | phone
-------+---------+-------
 Peter | Bradley |
(1 row)

bpsimple=#
```

We can see that the single row we want to update is being selected, so we can go ahead and put the two halves of the statement together and execute it:

```
bpsimple=# UPDATE tcust SET phone = '352 3442'
bpsimple-# WHERE fname = 'Peter' AND lname = 'Bradley';
UPDATE 1
bpsimple=#
```

PostgreSQL tells us that one row has been updated. We could, if we wanted, reexecute our SELECT statement to check that all is well.

How It Works

We built our UPDATE statement in two stages. First, we wrote the UPDATE command part that would actually change the column value, and then we wrote the WHERE clause to specify which rows to update. After testing the WHERE clause, we executed the UPDATE statement, which changed the row as required.

Why were we so careful to test the WHERE clause and warn about not executing the first part of the UPDATE statement? The answer is because it is perfectly valid to have an UPDATE statement with no WHERE clause. By default, UPDATE will then update *all* the rows in the table, which is almost never what was intended. It can also be quite hard to correct.

tcust is just temporary experimental data, so let's use it to test an UPDATE with no WHERE clause:

```
bpsimple=# UPDATE tcust SET phone = '999 9999';
UPDATE 3
bpsimple=#
```

Notice that psql has told us that three rows have been updated. Now look at what we have:

```
bpsimple=# SELECT fname, lname, phone FROM tcust;
 fname |  lname  |  phone
-------+---------+----------
 Kevin | Carney  | 999 9999
 Brian | Waters  | 999 9999
 Peter | Bradley | 999 9999
(3 rows)

bpsimple=#
```

This is almost certainly not what we wanted!

■ **Caution** Always test the WHERE clause of UPDATE statements before executing them. A simple error in a WHERE clause can result in many, or even all, of the rows in the table being updated with the same values.

If you do intend to update many rows, rather than retrieve all the data, you can simply check how many rows you are matching using the count(*) syntax, which we will meet in more detail in the next chapter. For now, all you need to know is that replacing the column names in a SELECT statement with count(*) will tell you how many rows were matched, rather than returning the data in the rows. In fact, that's about all there is to the count(*) statement, but it does turn out to be quite useful in practice. Here is an example of our SELECT statement to check how many rows are matched by the WHERE clause:

```
bpsimple=# SELECT count(*) from tcust
bpsimple-# WHERE fname = 'Peter' AND lname = 'Bradley';
 count
-------
     1
(1 row)

bpsimple=#
```

This tells us that the WHERE clause is sufficiently restrictive to specify a single row. Of course, with different data, even specifying both fname and lname may not be sufficient to uniquely identify a row.

Updating from Another Table

PostgreSQL has an extension that allows updates from another table, using the syntax:

```
UPDATE tablename FROM tablename WHERE condition
```

This is an extension to the SQL standard.

Try It Out: Update with FROM

For the purpose of checking out the UPDATE with FROM option, let's create a table named custphone that contains the customer names and their phone numbers. The table looks like this:

```
CREATE TABLE custphone
(
    customer_id                 serial,
    fname                       varchar(32),
    lname                       varchar(32) NOT NULL,
    phone_num                   varchar(16)

);
```

Let's also insert some data into the newly created custphone table that holds the customers and their phone numbers:

```
bpsimple=# INSERT INTO custphone(fname, lname, phone_num)
bpsimple-# VALUES('Peter', 'Bradley', '352 3442');
INSERT 22593 1
bpsimple=#
```

Now we need to specify the row to be updated in the tcust table:

```
bpsimple=# UPDATE tcust SET phone = custphone.phone_num FROM custphone
bpsimple-# WHERE tcust.fname = 'Peter' AND tcust.lname = 'Bradley';
UPDATE 1

bpsimple=#
```

How It Works

We created a new table that contains the phone numbers of the customers. Then we inserted data into the new table. Finally, we executed the UPDATE statement, which changed the row as required.

While UPDATE uses subqueries to control the rows that are updated, the FROM clause allows the inclusion of columns from other tables in the SET clause. In fact, the FROM clause isn't even required. This is because PostgreSQL creates a reference to any table used in a query by default.

Deleting Data from the Database

The last thing we need to learn about in this chapter is deleting data from tables. Prospective customers may never actually place an order, orders get canceled, and so on, so we often need to delete data from the database.

Using the DELETE Statement

The normal way of deleting data is to use the DELETE statement. This has syntax similar to the UPDATE statement:

```
DELETE FROM tablename WHERE condition
```

Notice that there are no columns listed, since DELETE works on rows. If you want to remove data from a column, you must use the UPDATE statement to set the value of the column to NULL or some other appropriate value.

Now that we have copied our data for our two new customers from tcust to our live customer table, we can go ahead and delete those rows from our tcust table.

Try It Out: Delete Data

We know just how dangerous omitting the WHERE clause in statements that change data can be. We can appreciate that accidentally deleting data is even more serious, so we will start by writing and checking our WHERE clause using a SELECT statement:

```
bpsimple=# SELECT fname, lname FROM tcust WHERE town = 'Lincoln';
 fname | lname
-------+--------
 Kevin | Carney
 Brian | Waters
(2 rows)

bpsimple=#
```

That's good—it retrieves the two rows we were expecting.

Now we can prepend the DELETE statement on the front and, after a last visual check that it looks correct, execute it:

```
bpsimple=# DELETE FROM tcust WHERE town = 'Lincoln';
DELETE 2
bpsimple=#
```

■ **CAUTION** Deleting from the database is that easy, so be very careful!

How It Works

We wrote and tested a WHERE clause to choose the rows that we wanted to delete from the database. We then executed a DELETE statement that deleted them.

Just like UPDATE, DELETE can work on only a single table at any one time. If we ever need to delete related rows from more than one table, we will use a *transaction*, which we will meet in Chapter 9.

Using the TRUNCATE Statement

There is one other way of deleting data from a table. It deletes *all* of the data from a table, and unless it is contained within a PostgreSQL version 7.4 or later transaction, it will give you no way of recovering the data. The command is TRUNCATE, and its syntax is as follows:

```
TRUNCATE TABLE tablename
```

This is a command to be used with caution, and only when you are very sure that you want to permanently delete all the data in a table. In some ways, it is similar to dropping and re-creating the table, except it is much easier to use and doesn't reset the sequence number.

Try It Out: Use The TRUNCATE Statement

Suppose we have now finished with our tcust table, and want to delete all the data in it. We could DROP the table, but then if we needed it again, we would need to re-create it. Instead, we can TRUNCATE it, to delete all the rows in the table:

```
bpsimple=# TRUNCATE TABLE tcust;
TRUNCATE TABLE
bpsimple=# SELECT count(*) FROM tcust;
 count
-------
     0
(1 row)

bpsimple=#
```

All the rows are now deleted.

How It Works

TRUNCATE simply deletes all the rows from the specified table.

If you have a large table, perhaps with many thousands of rows, and want to delete all the rows from it, by default, PostgreSQL does not physically remove the rows, but scans through them all, marking each one as deleted. This helps in restoring the data in case the transaction is rolled back. Even though on the command line we might not have explicitly asked for a transaction, all commands automatically get executed inside a transaction. The action of scanning and marking many thousands of rows of a table slows down execution. The TRUNCATE statement deletes the contents of the table very efficiently without scanning the data. So, on very large tables, it executes much more efficiently than DELETE.

■ **Tip** There are two ways to delete all the rows from a table: `DELETE` without a `WHERE` clause and `TRUNCATE`. `TRUNCATE`, although not in SQL92, is a very common SQL statement for efficiently deleting all rows from a table.

You should stick to using `DELETE` almost all of the time, as it is a much safer way of deleting data. Also `DELETE` works in some cases where `TRUNCATE` does not, such as on tables with foreign keys. However, in special cases where you want to efficiently and irrevocably delete all rows from a table, `TRUNCATE` is the solution.

Summary

In this chapter, we looked at the three other parts of data manipulation along with `SELECT`: the ability to add data with the `INSERT` command, modify data with the `UPDATE` command, and remove data with the `DELETE` command.

We learned about the two forms of the `INSERT` command, with data explicitly included in the `INSERT` statement or `INSERT` from data `SELECT`ed from another table. We saw how it is safer to use the longer form of the `INSERT` statement, where all columns are listed, so there is less chance of mistakes. We also met `INSERT`'s cousin command, the rather useful PostgreSQL extension `\copy`, which allows data to be inserted into a table directly from a local file.

We looked at how you need to be careful with the sequence counters for `serial` fields, and how to check the value of a sequence, and if necessary, change it. We saw that, in general, it is better to allow PostgreSQL to generate sequence numbers for you, by not providing data for `serial` type columns.

We saw how the very simple `UPDATE` and `DELETE` statements work, and how to use them with `WHERE` clauses, just as with the `SELECT` statement. We also mentioned that you should always test `UPDATE` and `DELETE` statements with `WHERE` clauses using a `SELECT` statement, as mistakes here can cause problems that are difficult to rectify.

Finally, we looked at the `TRUNCATE` statement, a very efficient way of deleting all rows from a table. Since this is an irrevocable deletion, unless managed by transactions, it should be used with caution.