

PostgreSQL, Second Edition

by Korrry Douglas; Susan Douglas

Publisher: Sams

Pub Date: July 26, 2005

Print ISBN-10: 0-672-32756-2

Print ISBN-13: 978-0-672-32756-8

Pages: 1032

The Real Value in Free Software

These days, it seems that most discussion of open-source software centers around the idea that you should not have to tie your future to the whim of some giant corporation. People say that open-source software is better than proprietary software because it is developed and maintained by the users instead of a faceless company out to lighten your wallet.

I think that the real value in free software is education. I have never learned anything by reading my own code^[1]. On the other hand, it's a rare occasion when I've looked at code written by someone else and haven't come away with another tool in my toolkit. People don't think alike. I don't mean that people disagree with each other; I mean that people solve problems in different ways. Each person brings a unique set of experiences to the table. Each person has his own set of goals and biases. Each person has his own interests. All of these things will shape the way you think about a problem. Often, I'll find myself in a heated disagreement with a colleague only to realize that we are each correct in our approach. Just because I'm right, doesn't mean that my colleague can't be right as well.

^[1] Maybe I should say that I have never learned anything new by reading my own code. I've certainly looked at code that I've written and wondered what I was thinking at the time, learning that I'm not nearly as clever as I had remembered. Oddly enough, those who have read my code have reached a similar conclusion.

Open-source software is a great way to learn. You can learn about programming. You can learn about design. You can learn about debugging. Sometimes, you'll learn how not to design, code, or debug; but that's a valuable lesson, too. You can learn small things, like how to cache file descriptors on systems where file descriptors are a scarce and expensive resource, or how to use the `select()` function to implement fine-grained timers. You can learn big things, like how a query optimizer works or how to write a parser, or how to develop a good memory-management strategy.

PostgreSQL is a great example. I've been using databases for the last two decades. I've used most of the major commercial databases: Oracle, Sybase, DB2, and MS SQL Server. With each commercial database, there is a wall of knowledge between my needs and the vendor's need to protect his intellectual property. Until I started exploring open-source databases, I had an incomplete understanding of how a database works. Why was this particular feature implemented that way? Why am I getting poor performance when I try this? That's a neat feature; I wonder how they did that? Every commercial database tries to expose a small piece of its inner workings. The `explain` statement will show you why the database makes its optimization decisions. But, you only get to see what the vendor wants you to see. The vendor isn't trying to hide things from you (in most cases), but without complete access to the source code, they have to pick and choose how to expose information in a meaningful way. With open-source software, you can dive deep into the source code and pull out all the information you need. While writing this book, I've spent a lot of time reading through the PostgreSQL source code. I've added a lot of my own code to reveal more information so that I could explain things more clearly. I can't do that with a commercial database.

There are gems of brilliance in most open-source projects. In a well-designed, well-factored project, you will find designs and code that you can use in your own projects. Many open-source projects are starting to split their code into reusable libraries. The Apache Portable Runtime is a good example. The Apache Web server runs on many diverse platforms. The Apache development team saw the need for a layer of abstraction that would provide a portable interface to system functions such as shared memory and network access. They decided to factor the portability layer into a library separate from their main project. The result is the Apache Portable Runtime—a library of code that can be used in other open-source projects (such as PostgreSQL).

Some developers hate to work on someone else's code. I love working on code written by another developer—I always learn something from the experience. I strongly encourage you to dive into the PostgreSQL source code. You will learn from it. You might even decide to contribute to the project.

—Korry Douglas

Introduction

PostgreSQL is a relational database with a long history. In the late 1970s, the University of California at Berkeley began development of PostgreSQL's ancestor—a relational database known as Ingres. Relational Technologies turned Ingres into a commercial product. Relational Technologies became Ingres Corporation and was later acquired by Computer Associates. Around 1986, Michael Stonebraker from UC Berkeley led a team that added object-oriented features to the core of Ingres; the new version became known as Postgres. Postgres was again commercialized; this time by a company named Illustra, which became part of the Informix Corporation. Andrew Yu and Jolly Chen added SQL support to Postgres in the mid-'90s. Prior versions had used a different, Postgres-specific query language known as Postquel. In 1996, many new features were added, including the MVCC transaction model, more adherence to the SQL92 standard, and many performance improvements. Postgres once again took on a new name: PostgreSQL.

Today, PostgreSQL is developed by an international group of open-source software proponents known as the PostgreSQL Global Development group. PostgreSQL is an open-source product—it is not proprietary in any way. Red Hat has recently commercialized PostgreSQL, creating the Red Hat Database, but PostgreSQL itself will remain free and open source.

PostgreSQL Features

PostgreSQL has benefited well from its long history. Today, PostgreSQL is one of the most advanced database servers available. Here are a few of the features found in a standard PostgreSQL distribution:

- **Object-relational**— In PostgreSQL, every table defines a class. PostgreSQL implements inheritance between tables (or, if you like, between classes). Functions and operators are polymorphic.
- **Standards compliant**— PostgreSQL syntax implements most of the SQL92 standard and many features of SQL99. Where differences in syntax occur, they are most often related to features unique to PostgreSQL.
- **Open source**— An international team of developers maintains PostgreSQL. Team members come and go, but the core members have been enhancing PostgreSQL's performance and feature set since at least 1996. One advantage to PostgreSQL's open-source nature is that talent and knowledge can be recruited as needed. The fact that this team is international ensures that PostgreSQL is a product that can be used productively in any natural language, not just English.
- **Transaction processing**— PostgreSQL protects data and coordinates multiple concurrent users through full transaction processing. The transaction model used by PostgreSQL is based on multi-version concurrency control (MVCC). MVCC provides much better performance than you would find with other products that coordinate multiple users through table-, page-, or row-level locking.
- **Referential integrity**— PostgreSQL implements complete referential integrity by supporting foreign and primary key relationships as well as triggers. Business rules can be expressed within the database rather than relying on an external tool.
- **Multiple procedural languages**— Triggers and other procedures can be written in any of several procedural languages. Server-side code is most commonly written in PL/pgSQL, a procedural language similar to Oracle's PL/SQL. You can also develop server-side code in Tcl, Perl, even bash (the open-source Linux/Unix shell).
- **Multiple-client APIs**— PostgreSQL supports the development of client applications in many languages. This book describes how to interface to PostgreSQL from C, C++, ODBC, Perl, PHP, Tcl/Tk, and Python.
- **Unique data types**— PostgreSQL provides a variety of data types. Besides the usual numeric, string, and data types, you will also find geometric types, a Boolean data type, and data types designed specifically to deal with network addresses.
- **Extensibility**— One of the most important features of PostgreSQL is that it can be extended. If you don't find something that you need, you can usually add it yourself. For example, you can add new data types, new functions and operators, and even new procedural and client languages. There are many contributed packages available on the Internet. For example, Refractions Research, Inc. has developed a set of geographic data types that can be used to efficiently model spatial (GIS) data.

What Versions Does This Book Cover?

The first edition of this book covered versions 7.1 through 7.3. In this edition, we've updated the basics and added coverage for the new features introduced in versions 7.4 and 8.0. Throughout the book, I'll be sure to let you know which features work only in new releases, and, in a few cases, I'll explain features that have been deprecated (that is, features that are obsolete). You can use this book to install, configure, tune, program, and manage PostgreSQL versions 7.1 through 8.0.

Fortunately, the PostgreSQL developers try very hard to maintain forward compatibility—new features tend not to break existing applications. This means that all the features discussed in this book should still be available and substantially similar in later versions of PostgreSQL. I have tried to avoid talking about features that have not been released at the time of writing—where I have mentioned future developments, I will point them out.

Who Is This Book For?

If you are already using PostgreSQL, you should find this book a useful guide to some of the features that you might be less familiar with. The first part of the book provides an introduction to SQL and PostgreSQL for the new user. You'll also find information that shows how to obtain and install PostgreSQL on a Unix/Linux host, as well as on Microsoft Windows.

If you are developing an application that will store data in PostgreSQL, the second part of this book will provide you with a great deal of information relating to PostgreSQL programming. You'll find information on both server-side and client-side programming in a variety of languages.

Every database needs occasional administrative work. The final part of the book should be of help if you are a PostgreSQL administrator, or a developer or user that needs to do occasional administration. You will also find information on how to secure your data against inappropriate use.

Finally, if you are trying to decide which database to use for your current project (or for future projects), this book should provide all the information you need to evaluate whether PostgreSQL will fit your needs.

What Topics Does This Book Cover?

PostgreSQL is a huge product. It's not easy to find the right mix of topics when you are trying to fit everything into a single book. This book is divided into three parts.

The first part, "[General PostgreSQL Use](#)," is an introduction and user's guide for PostgreSQL. [Chapter 1](#), "Introduction to PostgreSQL and SQL," covers the basics—how to obtain and install PostgreSQL (if you are running Linux, chances are you already have PostgreSQL and it may be installed). The first chapter also provides a gentle introduction to SQL and discusses the sample database we'll be using throughout the book. [Chapter 2](#), "Working with Data in PostgreSQL," describes the many data types supported by a standard PostgreSQL distribution; you'll learn how to enter values (literals) for each data type, what kind of data you can store with each type, and how those data types are combined into expressions. [Chapter 3](#), "PostgreSQL SQL Syntax and Use," fills in some of the details we glossed over in the first two chapters. You'll learn how to create new databases, new tables and indexes, and how PostgreSQL keeps your data safe through the use of transactions. [Chapter 4](#), "Performance," describes the PostgreSQL optimizer. I'll show you how to get information about the decisions made by the optimizer, how to decipher that information, and how to influence those decisions.

[Part II](#), "Programming with PostgreSQL," is all about PostgreSQL programming. In [Chapter 5](#), "Introduction to PostgreSQL Programming," we start off by describing the options you have when developing a database application that works with PostgreSQL (and there are a lot of options). [Chapter 6](#), "Extending PostgreSQL," briefly describes how to extend PostgreSQL by adding new functions, data types, and operators. [Chapter 7](#), "PL/pgSQL," describes the PL/pgSQL language. PL/pgSQL is a server-based procedural language. Code that you write in PL/pgSQL executes within the PostgreSQL server and has very fast access to data. Each chapter in the remainder of the programming section deals with a client-based API. You can connect to a PostgreSQL server using a number of languages. I show you how to interface to PostgreSQL using C, C++, ecpg, ODBC, JDBC, Perl, PHP, Tcl/Tk, Python, and Microsoft's .NET. [Chapters 8 through 18](#) all follow the same pattern: you develop a series of client applications in a given language. The first client application shows you how to establish a connection to the database (and how that connection is represented by the language in question). The next client adds error checking so that you can intercept and react to unusual conditions. The third client in each chapter demonstrates how to process SQL commands from within the client. The final client wraps everything together and shows you how to build an interactive query processor using the language being discussed. Even if you program in only one or two languages, I would encourage you to study the other chapters in this section. I think you'll find that looking at the same application written in a variety of languages will help you understand the philosophy followed by the PostgreSQL development team, and it's a great way to start learning a new language. [Chapter 19](#), "Other Useful Programming Tools," introduces you to a few programming tools (and interfaces) that you might find useful: PL/Java and PL/Perl. I'll also show you how to use PostgreSQL inside of bash shell scripts.

The final part of this book ([Part III](#), "PostgreSQL Administration") deals with administrative issues. The final six chapters of this book show you how to perform the occasional duties required of a PostgreSQL administrator. In the first two chapters, [Chapter 20](#), "Introduction to PostgreSQL Administration," and [Chapter 21](#), "PostgreSQL Administration," you'll learn how to start up, shut down, back up, and restore a server. In [Chapter 22](#), "Internationalization and Localization," you will learn how PostgreSQL supports internationalization and localization. PostgreSQL understands how to store and process a variety of single-byte and multi-byte character sets including Unicode, ASCII, and Japanese, Chinese, Korean, and Taiwan EUC. In [Chapter 23](#), "Security," I'll show you how to secure your data against unauthorized uses (and unauthorized users). In [Chapter 24](#), "[Replicating PostgreSQL with Slony](#)," you'll learn how to replicate data with PostgreSQL's Slony replication system. [Chapter 25](#), "Contributed Modules," introduces a few open-source projects that work well with PostgreSQL. I'll show you how to query a PostgreSQL database using XML, how to configure and use TSEARCH2 (a full-text indexing and search system), and how to install and use PgAdmin III, a graphical user interface specifically designed for PostgreSQL.

What's New in the Second Edition?

The first edition of this book hit the shelves in February 2003—at that time, the PostgreSQL developers had just released version 7.3.2. Release 7.4 was unleashed in November 2003. In January 2005, the PostgreSQL developers released version 8.0—a major release full of new features. We timed the second edition of this book to coincide with the release of version 8.0 (the book will appear in bookstores a few months after 8.0 hits the streets). In this edition, we've added coverage for all of the (major) new features in 7.3, 7.4, and 8.0, including

- Installing, securing, and managing PostgreSQL on Windows hosts
- Tablespaces

- Schemas
- New quoting mechanisms for string values
- New data types (`ANYARRAY`, `ANYELEMENT`, `VOID`)
- The standards-conforming `INFORMATION_SCHEMA`
- Nested transactions (`SAVEPOINT`'s)
- The new PostgreSQL buffer manager
- Auto-vacuum
- Prepared-statement execution (the `PREPARE/EXECUTE` model)
- Set-returning functions
- Exception handling in PL/pgSQL
- `libpqxx`, the new PostgreSQL interface for C++ clients
- New features in `ecpg` (the embedded SQL processor for C)
- New features in the ODBC, JDBC (Java), Perl, Python, PHP, and Tcl/Tk client interfaces
- `npgsql`—the PostgreSQL .NET data provider
- Other useful programming tools (PL/Java, `pgpash`, `pgcurl`, etc.)
- Point-in-time recovery
- Replication
- Using PostgreSQL with XML
- Full-text search

We hope you enjoy this book and find it useful. The PostgreSQL developers have done an incredible job of enhancing what was already a world-class database product. Now dig in.

Part I : General PostgreSQL Use

- 1 Introduction to PostgreSQL and SQL
- 2 Working with Data in PostgreSQL
- 3 PostgreSQL SQL Syntax and Use
- 4 Performance

Chapter 1. Introduction to PostgreSQL and SQL

PostgreSQL is an open-source, client/server, relational database. PostgreSQL offers a unique mix of features that compare well to the major commercial databases such as Sybase, Oracle, and DB2. One of the major advantages to PostgreSQL is that it is open source—you can see the source code for PostgreSQL. PostgreSQL is not owned by any single company. It is developed, maintained, broken, and fixed by a group of volunteer developers around the world. You don't have to buy PostgreSQL—it's free. You won't have to pay any maintenance fees (although you can certainly find commercial sources for technical support).

PostgreSQL offers all the usual features of a relational database plus quite a few unique features. PostgreSQL offers inheritance (for you object-oriented readers). You can add your own data types to PostgreSQL. (I know, some of you are probably thinking that you can do that in your favorite database.) Most database systems allow you to give a new name to an existing type. Some systems allow you to define composite types. With PostgreSQL, you can add new fundamental data types. PostgreSQL includes support for geometric data types such as `point`, `line segment`, `box`, `polygon`, and `circle`. PostgreSQL uses indexing structures that make geometric data types fast. PostgreSQL can be extended—you can build new functions, new operators, and new data types in the language of your choice. PostgreSQL is built around client/server architecture. You can build client applications in a number of different languages, including C, C++, Java, Python, Perl, TCL/Tk, and others. On the server side, PostgreSQL sports a powerful procedural language, PL/pgSQL (okay, the language is sportier than the name). You can add procedural languages to the server. You will find procedural languages supporting Perl, TCL/Tk, and even the `bash` shell.

A Sample Database

Throughout this book, I'll use a simple example database to help explain some of the more complex concepts. The sample database represents some of the data storage and retrieval requirements that you might encounter when running a video rental store. I won't pretend that the sample database is useful for any real-world scenarios; instead, this database will help us explore how PostgreSQL works and should illustrate many PostgreSQL features.

To begin with, the sample database (which is called `movies`) contains three kinds of records: customers, tapes, and rentals.

Whenever a customer walks into our imaginary video store, you will consult your database to determine whether you already know this customer. If not, you'll add a new record. What items of information should you store for each customer? At the very least, you will want to record the customer's name. You will want to ensure that each customer has a unique identifier—you might have two customers named "Danny Johnson," and you'll want to keep them straight. A name is a poor choice for a unique identifier—names might not be unique, and they can often be spelled in different ways. ("Was that Danny, Dan, or Daniel?") You'll assign each customer a unique customer ID. You might also want to store the customer's birth date so that you know whether he should be allowed to rent certain movies. If you find that a customer has an overdue tape rental, you'll probably want to phone him, so you better store the customer's phone number. In a real-world business, you would probably want to know much more information about each customer (such as his home address), but for these purposes, you'll keep your storage requirements to a minimum.

Next, you will need to keep track of the videos that you stock. Each video has a title and a duration—you'll store those. You might own several copies of the same movie and you will certainly have many movies with the same duration, so you can't use either one for a unique identifier. Instead, you'll assign a unique ID to each video.

Finally, you will need to track rentals. When a customer rents a tape, you will store the customer ID, tape ID, and rental date.

Notice that you won't store the customer name with each rental. As long as you store the customer ID, you can always retrieve the customer name. You won't store the movie title with each rental, either—you can find the movie title by its unique identifier.

At a few points in this book, we might make changes to the layout of the sample database, but the basic shape will remain the same.

Basic Database Terminology

Before we get into the interesting stuff, it might be useful to get acquainted with a few of the terms that you will encounter in your PostgreSQL life. PostgreSQL has a long history—you can trace its history back to 1977 and a program known as Ingres. A lot has changed in the relational database world since 1977. When you are breaking ground with a new product (as the Ingres developers were), you don't have the luxury of using standard, well-understood, and well-accepted terminology—you have to make it up as you go along. Many of the terms used by PostgreSQL have synonyms (or at least close analogies) in today's relational marketplace. In this section, I'll show you a few of the terms that you'll encounter in this book and try to explain how they relate to similar concepts in other database products.

- Schema

A schema is a named collection of tables. (see table). A schema can also contain views, indexes, sequences, data types, operators, and functions. Other relational database products use the term catalog.

- Database

A database is a named collection of schemas. When a client application connects to a PostgreSQL server, it specifies the name of the database that it wants to access. A client cannot interact with more than one database per connection but it can open any number of connections in order to access multiple databases simultaneously.

- Command

A command is a string that you send to the server in hopes of having the server do something useful. Some people use the word statement to mean command. The two words are very similar in meaning and, in practice, are interchangeable.

- Query

A query is a type of command that retrieves data from the server.

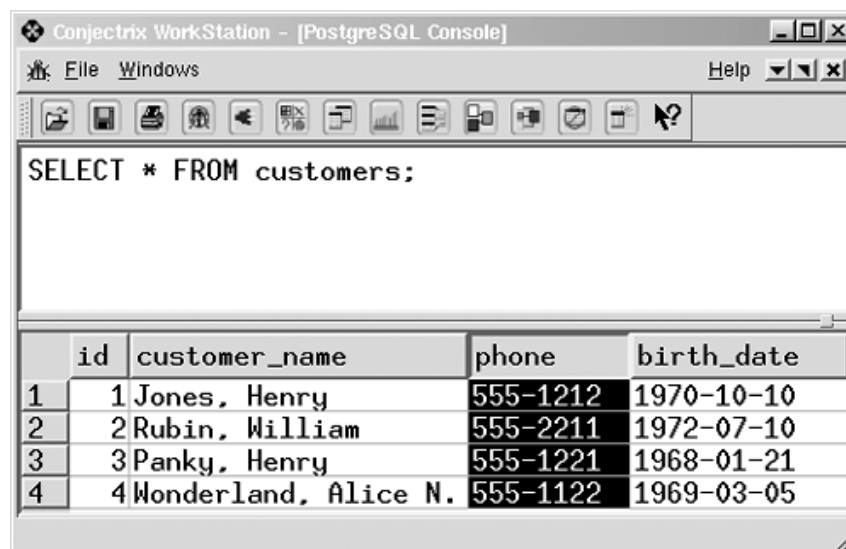
- Table (relation, file, class)

A table is a collection of rows. A table usually has a name, although some tables are temporary and exist only to carry out a command. All the rows in a table have the same shape (in other words, every row in a table contains the same set of columns). In other database systems, you may see the terms relation, file, or even class—these are all equivalent to a table.

- Column (field, attribute)

A column is the smallest unit of storage in a relational database. A column represents one piece of information about an object. Every column has a name and a data type. Columns are grouped into rows, and rows are grouped into tables. In Figure 1.1, the shaded area depicts a single column.

Figure 1.1. A column (highlighted).



```
SELECT * FROM customers;
```

	id	customer_name	phone	birth_date
1	1	Jones, Henry	555-1212	1970-10-10
2	2	Rubin, William	555-2211	1972-07-10
3	3	Panky, Henry	555-1221	1968-01-21
4	4	Wonderland, Alice N.	555-1122	1969-03-05

The terms field and attribute have similar meanings.

- Row (record, tuple)

A row is a collection of column values. Every row in a table has the same shape (in other words, every row is composed of the same set of columns). If you are trying to model a real-world application, a row represents a real-world object. For example, if you are running an auto dealership, you might have a `vehicles` table. Each row in the `vehicles` table represents a car (or truck, or motorcycle, and so on). The kinds of information that you store are the same for all `vehicles` (that is, every car has a color, a vehicle ID, an engine, and so on). In Figure 1.2, the shaded area depicts a row.

Figure 1.2. A row (highlighted).

	id	customer_name	phone	birth_date
1	1	Jones, Henry	555-1212	1970-10-10
2	2	Rubin, William	555-2211	1972-07-10
3	3	Panky, Henry	555-1221	1968-01-21
4	4	Wonderland, Alice N.	555-1122	1969-03-05

You may also see the terms record or tuple—these are equivalent to a row.

- Composite type

Starting with PostgreSQL version 8, you can create new data types that are composed of multiple values. For example, you could create a composite type named `address` that holds a street address, city, state/province, and postal code. When you create a table that contains a column of type `address`, you can store all four components in a single field. We discuss composite types in more detail in [Chapter 2](#), "Working with Data in PostgreSQL."

- Domain

A domain defines a named specialization of another data type. Domains are useful when you need to ensure that a single data type is used in several tables. For example, you might define a domain named `accountNumber` that contains a single letter followed by four digits. Then you can create columns of type `accountNumber` in a general ledger accounts table, an accounts receivable customer table, and so on.

- View

A view is an alternative way to present a table (or tables). You might think of a view as a "virtual" table. A view is (usually) defined in terms of one or more tables. When you create a view, you are not storing more data, you are instead creating a different way of looking at existing data. A view is a useful way to give a name to a complex query that you may have to use repeatedly.

- Client/server

PostgreSQL is built around a client/server architecture. In a client/server product, there are at least two programs involved. One is a client and the other is a server. These programs may exist on the same host or on different hosts that are connected by some sort of network. The server offers a service; in the case of PostgreSQL, the server offers to store, retrieve, and change data. The client asks a server to perform work; a PostgreSQL client asks a PostgreSQL server to serve up relational data.

- Client

A client is an application that makes requests of the PostgreSQL server. Before a client application can talk to a server, it must connect to a `postmaster` (see `postmaster`) and establish its identity. Client applications provide a user interface and can be written in many languages. [Chapters 8](#) through [19](#) will show you how to write a client application.

- Server

The PostgreSQL server is a program that services commands coming from client applications. The PostgreSQL server has no user interface—you can't talk to the server directly, you must use a client application.

- Postmaster

Because PostgreSQL is a client/server database, something has to listen for connection requests coming from a client application. That's what the `postmaster` does. When a connection request arrives, the `postmaster` creates a new server process in the host operating system.

- Transaction

A transaction is a collection of database operations that are treated as a unit. PostgreSQL guarantees that all the operations within a transaction complete or that none of them complete. This is an important property—it ensures that if something goes wrong in the middle of a transaction, changes made before the point of failure will not be reflected in the database. A transaction usually starts with a `BEGIN` command and ends with a `COMMIT` or `ROLLBACK` (see the next entries).

- Commit

A commit marks the successful end of a transaction. When you perform a commit, you are telling PostgreSQL that you have completed a unit of operation and that all the changes that you made to the database should become permanent.

- Rollback

A rollback marks the unsuccessful end of a transaction. When you roll back a transaction, you are telling PostgreSQL to discard any changes that you have made to the database (since the beginning of the transaction).

- Index

An index is a data structure that a database uses to reduce the amount of time it takes to perform certain operations. An index can also be used to ensure that duplicate values don't appear where they aren't wanted. I'll talk about indexes in [Chapter 4](#), "Performance."

- Tablespace

A tablespace defines an alternative storage location where you can create tables and indexes. When you create a table (or index), you can specify the name of a tablespace—if you don't specify a tablespace, PostgreSQL creates all objects in the same directory tree. You can use tablespaces to distribute the workload across multiple disk drives.

- Result set

When you issue a query to a database, you get back a result set. The result set contains all the rows that satisfy your query. A result set may be empty.

Prerequisites

Before I go much further, let's talk about installing PostgreSQL. [Chapters 21](#), "PostgreSQL Administration," and [23](#), "Security," discuss PostgreSQL installation in detail, but I'll show you a typical installation procedure here.

When you install PostgreSQL, you can start with prebuilt binaries or you can compile PostgreSQL from source code. In this chapter, I'll show you how to install PostgreSQL on a Linux host starting from prebuilt binaries. If you decide to install PostgreSQL from source code, many of the steps are the same. I'll show you how to build PostgreSQL from source code in [Chapter 21](#).

In older versions of PostgreSQL, you could run the PostgreSQL server on a Windows host but you had to install a Unix-like infrastructure (Cygwin) first: PostgreSQL wasn't a native Windows application. Starting with PostgreSQL version 8.0, the PostgreSQL server has been ported to the Windows environment as a native-Windows application. Installing PostgreSQL on a Windows server is very simple; simply download and run the installer program. You do have a few choices to make, and we cover the entire procedure in [Chapter 21](#).

Installing PostgreSQL Using an RPM

The easiest way to install PostgreSQL is to use a prebuilt RPM package. RPM is the Red Hat Package Manager. It's a software package designed to install (and manage) other software packages. If you choose to install using some method other than RPM, consult the documentation that comes with the distribution you are using.

PostgreSQL is distributed as a collection of RPM packages—you don't have to install all the packages to use PostgreSQL. [Table 1.1](#) lists the RPM packages available as of release 7.4.5.

Table 1.1. PostgreSQL RPM Packages as of Release 7.4.5

Package	Description
postgresql	Clients, libraries, and documentation
postgresql-server	Programs (and data files) required to run a server
postgresql-devel	Files required to create new client applications
postgresql-jdbc	JDBC driver for PostgreSQL
postgresql-tcl	Tcl client and PL/Tcl
postgresql-python	PostgreSQL's Python library
postgresql-test	Regression test suite for PostgreSQL
postgresql-libs	Shared libraries for client applications
postgresql-docs	Extra documentation not included in the postgresql base package
postgresql-contrib	Contributed software

Don't worry if you don't know which of these you need; I'll explain most of the packages in later chapters. You can start working with PostgreSQL by downloading the `postgresql`, `postgresql-libs`, and `postgresql-server` packages. The actual files (at the www.postgresql.org website) have names that include a version number: `postgresql-7.4.5-2PGDG.i686.rpm`, for example.

I strongly recommend creating an empty directory, and then downloading the PostgreSQL packages into that directory. That way you can install all the PostgreSQL packages with a single command.

After you have downloaded the desired packages, use the `rpm` command to perform the installation procedure. You must have superuser privileges to install PostgreSQL.

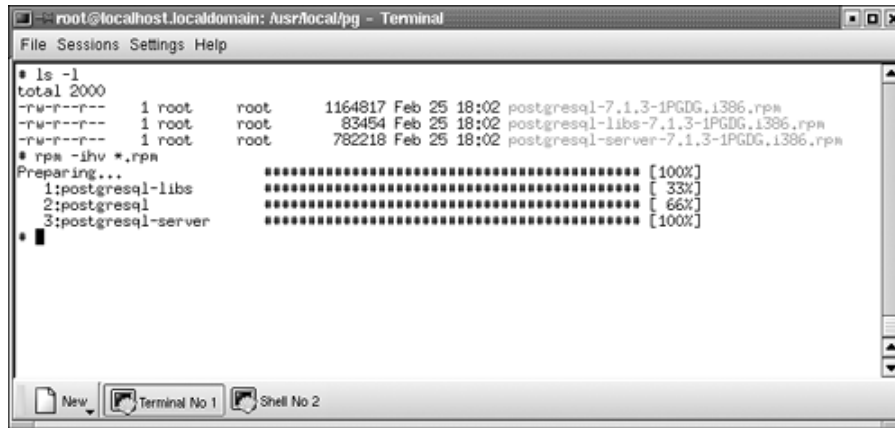
To install the PostgreSQL packages, `cd` into the directory that contains the package files and issue the following command:

```
# rpm -ihv *.rpm
```

The `rpm` command installs all the packages in your current directory. You should see results similar to what is shown in [Figure 1.3](#).

Figure 1.3. Using the `rpm` command to install PostgreSQL.

[\[View full size image\]](#)

A terminal window titled 'root@localhost.localdomain: /usr/local/pg - Terminal'. It shows the output of 'ls -l' for PostgreSQL RPMs, followed by 'rpm -ihv *.rpm' which displays progress bars for 'postgresql-libs', 'postgresql', and 'postgresql-server' being installed at 100%.

The RPM installer should have created a new user (named `postgres`) for your system. This user ID exists so that all database files accessed by PostgreSQL can be owned by a single user.

Each RPM package is composed of many files. You can view the list of files installed for a given package using the `rpm -ql` command:

```
# rpm -ql postgresql-server
/etc/rc.d/init.d/postgresql
/usr/bin/initdb
/usr/bin/initlocation
...
/var/lib/pgsql/data
# rpm -ql postgresql-libs
/usr/lib/libecpg.so.3
/usr/lib/libecpg.so.3.2.0
/usr/lib/libpqeasys.so.2
...
/usr/lib/libpq.so.2.1
```

At this point (assuming that everything worked), you have installed PostgreSQL on your system. Now it's time to create a database to play, er, work in.

While you have superuser privileges, issue the following commands:

```
# su - postgres
bash-2.04$ echo $PGDATA
/var/lib/pgsql/data
bash-2.04$ initdb
```

The first command (`su - postgres`) changes your identity from the OS superuser (`root`) to the PostgreSQL superuser (`postgres`). The second command (`echo $PGDATA`) shows you where the PostgreSQL data files will be created. The final command creates the two prototype databases (`template0` and `template1`).

You should get output that looks like that shown in [Figure 1.4](#).

Figure 1.4. Creating the prototype databases using `initdb`.

[\[View full size image\]](#)

A terminal window titled "Terminal" with a menu bar (File, Sessions, Settings, Help). The prompt is "bash-2.04#". The user enters "su - postgres", then "echo \$PGDATA" which outputs "/var/lib/pgsql/data", and finally "initdb". The terminal shows the initialization of the database system with username "postgres". It lists the creation of directories: /var/lib/pgsql/data/base, /var/lib/pgsql/data/global, and /var/lib/pgsql/data/pg_xlog. It also shows the creation of the template1 database. A notice states that the database is initialized with the en_US collation order. Debug messages show the shutdown and startup of the database system. The process concludes with "Success. You can now start the database server using:" followed by two commands: "/usr/bin/postmaster -D /var/lib/pgsql/data" and "/usr/bin/pg_ctl -D /var/lib/pgsql/data -l logfile start". The prompt returns to "bash-2.04#".

```
# su - postgres
bash-2.04# echo $PGDATA
/var/lib/pgsql/data
bash-2.04# initdb
This database system will be initialized with username "postgres".
This user will own all the data files and must also own the server process.

Fixing permissions on existing directory /var/lib/pgsql/data
Creating directory /var/lib/pgsql/data/base
Creating directory /var/lib/pgsql/data/global
Creating directory /var/lib/pgsql/data/pg_xlog
Creating template1 database in /var/lib/pgsql/data/base/1
NOTICE: Initializing database with en_US collation order.
This locale setting will prevent use of index optimization for
LIKE and regexp searches. If you are concerned about speed of
such queries, you may wish to set LC_COLLATE to "C" and
re-initdb. For more information see the Administrator's Guide.
DEBUG: database system was shut down at 2002-02-25 18:40:27 EST
DEBUG: CheckPoint record at (0, 8)
DEBUG: Redo record at (0, 8); Undo record at (0, 8); Shutdown TRUE
DEBUG: NextTransactionId: 514; NextDid: 16384
DEBUG: database system is in production state
Creating global relations in /var/lib/pgsql/data/global
DEBUG: database system was shut down at 2002-02-25 18:40:28 EST
DEBUG: CheckPoint record at (0, 108)
DEBUG: Redo record at (0, 108); Undo record at (0, 0); Shutdown TRUE
DEBUG: NextTransactionId: 514; NextDid: 17199
DEBUG: database system is in production state
Initializing pg_shadow.
Enabling unlimited row width for system tables.
Creating system views.
Loading pg_description.
Setting lastsysoid.
Vacuuming database.
Copying template1 to template0.

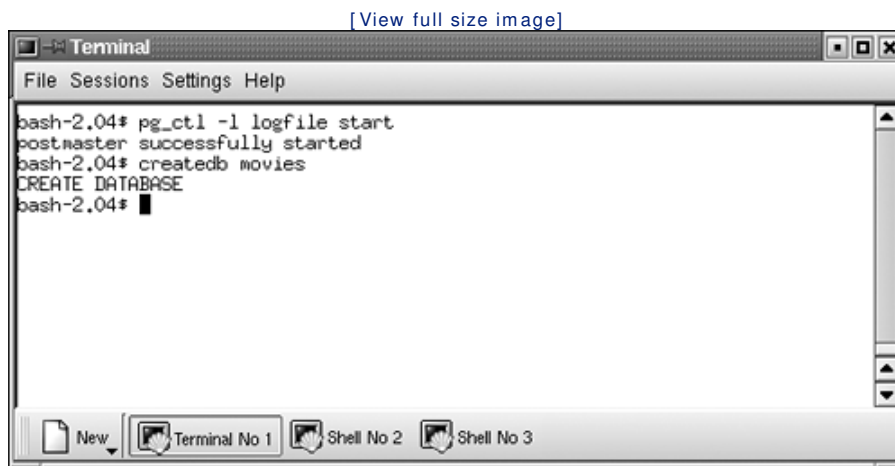
Success. You can now start the database server using:

        /usr/bin/postmaster -D /var/lib/pgsql/data
or
        /usr/bin/pg_ctl -D /var/lib/pgsql/data -l logfile start

bash-2.04#
```

You now have two empty databases named `template0` and `template1`. You really should not create new tables in either of these databases—a template database contains all the data required to create other databases. In other words, `template0` and `template1` act as prototypes for creating other databases. Instead, let's create a database that you can play in. First, start the `postmaster` process. The `postmaster` is a program that listens for connection requests coming from client applications. When a connection request arrives, the `postmaster` starts a new server process. You can't do anything in PostgreSQL without a `postmaster`. [Figure 1.5](#) shows you how to get the `postmaster` started.

Figure 1.5. Creating a new database with `createdb`.

A terminal window titled "Terminal" with a menu bar (File, Sessions, Settings, Help). The prompt is "bash-2.04#". The user enters "pg_ctl -l logfile start", which outputs "postmaster successfully started". Then the user enters "createdb movies", which outputs "CREATE DATABASE". The prompt returns to "bash-2.04#".

```
bash-2.04# pg_ctl -l logfile start
postmaster successfully started
bash-2.04# createdb movies
CREATE DATABASE
bash-2.04#
```

After starting the `postmaster`, use the `createdb` command to create the `movies` database (this is also shown in [Figure 1.5](#)). Most of the examples in this book take place in the `movies` database.

Notice that I used the `pg_ctl` command to start the `postmaster`^[1].

[1] You can also arrange for the `postmaster` to start whenever you boot your computer, but the exact instructions vary depending on which operating system you are using. See the section titled "Arranging for PostgreSQL Startup and Shutdown" in [Chapter 21](#)

The `pg_ctl` program makes it easy to start and stop the `postmaster`. To see a full description of the `pg_ctl` command, enter the command `pg_ctl --help`. You will get the output shown in [Figure 1.6](#).

Figure 1.6. pg_ctl options.

[\[View full size image\]](#)

```

bash-2.04# pg_ctl --help
pg_ctl is a utility to start, stop, restart, and report the status
of a PostgreSQL server.

Usage:
pg_ctl start [-u] [-D DATADIR] [-s] [-l FILENAME] [-o "OPTIONS"]
pg_ctl stop [-W] [-D DATADIR] [-s] [-n SHUTDOWN-MODE]
pg_ctl restart [-u] [-D DATADIR] [-s] [-n SHUTDOWN-MODE] [-o "OPTIONS"]
pg_ctl status [-D DATADIR]

Common options:
-D DATADIR          Location of the database storage area
-s                  Only print errors, no informational messages
-u                  Wait until operation completes
-W                  Do not wait until operation completes
(The default is to wait for shutdown, but not for start or restart.)

If the -D option is omitted, the environment variable PGDATA is used.

Options for start or restart:
-l FILENAME          Write (or append) server log to FILENAME. The
                    use of this option is highly recommended.
-o OPTIONS            Command line options to pass to the postmaster
                    (PostgreSQL server executable)
-p PATH-TO-POSTMASTER Normally not necessary

Options for stop or restart:
-n SHUTDOWN-MODE     May be 'smart', 'fast', or 'immediate'

Shutdown modes are:
smart                Quit after all clients have disconnected
fast                 Quit directly, with proper shutdown
immediate            Quit without complete shutdown; will lead
                    to recovery run on restart

Report bugs to <pgsql-bugs@postgresql.org>.
bash-2.04#

```

If you use a recent RPM file to install PostgreSQL, the two previous steps (`initdb` and `pg_ctl start`) can be automated. If you find a file named `postgresql` in the `/etc/rc.d/init.d` directory, you can use that shell script to initialize the database and start the postmaster. The `/etc/rc.d/init.d/postgresql` script can be invoked with any of the command-line options shown in [Table 1.2](#).

Table 1.2. `/etc/rc.d/init.d/postgresql` Options

Option	Description
<code>start</code>	Start the <code>postmaster</code>
<code>stop</code>	Stop the <code>postmaster</code>
<code>status</code>	Display the process ID of the <code>postmaster</code> if it is running
<code>restart</code>	Stop and then start the <code>postmaster</code>
<code>reload</code>	Force the <code>postmaster</code> to reread its configuration files without performing a full restart

At this point, you should use the `createuser` command to tell PostgreSQL which users are allowed to access your database. Let's allow the user 'bruce' into our system (see [Figure 1.7](#)).

Figure 1.7. Creating a new PostgreSQL user.

[\[View full size image\]](#)

```
Terminal
File Sessions Settings Help

bash-2.04# pg_ctl --help
pg_ctl is a utility to start, stop, restart, and report the status
of a PostgreSQL server.

Usage:
pg_ctl start [-u] [-D DATADIR] [-s] [-l FILENAME] [-o "OPTIONS"]
pg_ctl stop [-W] [-D DATADIR] [-s] [-n SHUTDOWN-MODE]
pg_ctl restart [-u] [-D DATADIR] [-s] [-n SHUTDOWN-MODE] [-o "OPTIONS"]
pg_ctl status [-D DATADIR]

Common options:
-D DATADIR          Location of the database storage area
-s                  Only print errors, no informational messages
-u                  Wait until operation completes
-W                  Do not wait until operation completes
(The default is to wait for shutdown, but not for start or restart.)

If the -D option is omitted, the environment variable PGDATA is used.

Options for start or restart:
-l FILENAME          Write (or append) server log to FILENAME. The
                    use of this option is highly recommended.
-o OPTIONS            Command line options to pass to the postmaster
                    (PostgreSQL server executable)
-p PATH-TO-POSTMASTER Normally not necessary

Options for stop or restart:
-n SHUTDOWN-MODE      May be 'smart', 'fast', or 'immediate'

Shutdown modes are:
smart                 Quit after all clients have disconnected
fast                  Quit directly, with proper shutdown
immediate             Quit without complete shutdown; will lead
                    to recovery run on restart

Report bugs to <pgsql-bugs@postgresql.org>.
bash-2.04# createuser bruce
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
CREATE USER
bash-2.04#
```

That's it! You now have a PostgreSQL database up and running.

Connecting to a Database

Assuming that you have a copy of PostgreSQL up and running, it's pretty simple to connect to the database. Here is an example:

```
$ psql -d movies
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

movies=# \q
```

The `psql` program is a text-based interface to a PostgreSQL database. When you are running `psql`, you won't see a graphical application—no buttons or pictures or other bells and whistles, just a text-based interface. Later, I'll show you another client application that does provide a graphical interface (`pgaccess`).

`psql` supports a large collection of command-line options. To see a summary of the options that you can use, type `psql --help`:

Code View: [Scroll](#) / [Show All](#)

```
$ psql --help
This is psql, the PostgreSQL interactive terminal.

Usage:
  psql [options] [dbname [username]]

Options:
  -a          Echo all input from script
  -A          Unaligned table output mode (-P format=unaligned)
  -c <query>  Run only single query (or slash command) and exit
  -d <dbname> Specify database name to connect to (default: korrry)
  -e          Echo queries sent to backend
  -E          Display queries that internal commands generate
  -f <filename> Execute queries from file, then exit
  -F <string> Set field separator (default: "|") (-P fieldsep=)
  -h <host>    Specify database server host (default: domain socket)
  -H          HTML table output mode (-P format=html)
  -l          List available databases, then exit
  -n          Disable readline
  -o <filename> Send query output to filename (or |pipe)
  -p <port>    Specify database server port (default: hardwired)
  -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) (-P recordsep=)
  -s          Single step mode (confirm each query)
  -S          Single line mode (newline terminates query)
  -t          Print rows only (-P tuples_only)
  -T text      Set HTML table tag options (width, border) (-P tableattr=)
  -U <username> Specify database username (default: Administrator)
  -v name=val  Set psql variable 'name' to 'value'
  -V          Show version information and exit
  -W          Prompt for password (should happen automatically)
  -x          Turn on expanded table output (-P expanded)
  -X          Do not read startup file (~/.psqlrc)
```

For more information, type `\?` (for internal commands) or `\help` (for SQL commands) from within `psql`, or consult the `psql` section in the PostgreSQL manual, which accompanies the distribution and is also available at <http://www.postgresql.org>. Report bugs to bugs@postgresql.org.

The most important options are `-U <user>`, `-d <dbname>`, `-h <host>`, and `-p <port>`.

The `-U` option allows you to specify a username other than the one you are logged in as. For example, let's say that you are logged in to your host as user `bruce` and you want to connect to a PostgreSQL database as user `sheila`. This `psql` command makes the connection (or at least tries to):

```
$ whoami
bruce
$ psql -U sheila -d movies
```

Impersonating Another User

The `-U` option may or may not allow you to impersonate another user. Depending on how your PostgreSQL administrator has configured database security, you might be prompted for `sheila`'s password; if you don't know the proper password,

you won't be allowed to impersonate her. (Chapter 23 discusses security in greater detail.) If you don't provide `psql` with a username, it will assume the username that you used when you logged in to your host.

You use the `-d` option to specify to which database you want to connect. If you don't specify a database, PostgreSQL will assume that you want to connect to a database whose name is your username. For example, if you are logged in as user `bruce`, PostgreSQL will assume that you want to connect to a database named `bruce`.

The `-d` and `-U` are not strictly required. The command line for `psql` should be of the following form:

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

If you are connecting to a PostgreSQL server that is running on the host that you are logged in to, you probably don't have to worry about the `-h` and `-p` options. If, on the other hand, you are connecting to a PostgreSQL server running on a different host, use the `-h` option to tell `psql` which host to connect to. You can also use the `-p` option to specify a TCP/IP port number—you only have to do that if you are connecting to a server that uses a nonstandard port (PostgreSQL usually listens for client connections on TCP/IP port number 5432). Here are a few examples:

```
$ # connect to a server waiting on the default port on host 192.168.0.1
$ psql -h 192.168.0.1

$ # connect to a server waiting on port 2000 on host arturo
$ psql -h arturo -p 2000
```

If you prefer, you can specify the database name, hostname, and TCP/IP port number using environment variables rather than using the command-line options. Table 1.3 lists some of the `psql` command-line options and the corresponding environment variables.

Table 1.3. `psql` Environment Variables

Command-Line Option		Environment Variable Meaning
<code>-d <dbname></code>	<code>PGDATABASE</code>	Name of database to connect to
<code>-h <host></code>	<code>PGHOST</code>	Name of host to connect to
<code>-p <port></code>	<code>PGPORT</code>	Port number to connect to
<code>-U <user></code>	<code>PGUSER</code>	PostgreSQL Username

A (Very) Simple Query

At this point, you should be running the `psql` client application. Let's try a very simple query:

```
$ psql -d movies
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

movies=# SELECT user;
 current_user
-----
    korry
(1 row)

movies=# \q
$
```

Let's take a close look at this session. First, you can see that I started the `psql` program with the `-d movies` option—this tells `psql` that I want to connect to the `movies` database.

After greeting me and providing me with a few crucial hints, `psql` issues a prompt: `movies=#`. `psql` encodes some useful information into the prompt, starting with the name of the database that I am currently connected to (`movies` in this case). The character that follows the database name can vary. A `=` character means that `psql` is waiting for me to start a command. A `-` character means that `psql` is waiting for me to complete a command (`psql` allows you to split a single command over multiple lines. The first line is prompted by a `=` character; subsequent lines are prompted by a `-` character). If the prompt ends with a `(` character, you have entered more opening parentheses than closing parentheses.

You can see the command that I entered following the prompt: `SELECT user`; Each SQL command starts with a verb—in this case, `SELECT`. The verb tells PostgreSQL what you want to do and the rest of the command provides information specific to that command. I am executing a `SELECT` command. `SELECT` is used to retrieve information from the database. When you execute a `SELECT` command, you have to tell PostgreSQL what information you are interested in. I want to retrieve my PostgreSQL user ID so I `SELECT user`. The final part

of this command is the semicolon (;)—each SQL command must end with a semicolon.

After I enter the `SELECT` command (and press the `Return` key), `psql` displays the results of my command:

```
current_user
-----
korrry
(1 row)
```

When you execute a `SELECT` command, `psql` starts by displaying a row of column headers. I have selected only a single column of information so I see only a single column header (each column header displays the name of the column). Following the row of column headers is a single row of separator characters (dashes). Next comes zero or more rows of the data that I requested. Finally, `psql` shows a count of the number of data rows displayed.

I ended this session using the `\q` command.

Tips for Interacting with PostgreSQL

The `psql` client has a lot of features that will make your PostgreSQL life easier.

Besides PostgreSQL commands (`SELECT`, `INSERT`, `UPDATE`, `CREATE TABLE`, and so on), `psql` provides a number of internal commands (also known as meta-commands). PostgreSQL commands are sent to the server, meta-commands are processed by `psql` itself. A meta-command begins with a backslash character (`\`). You can obtain a list of all the meta-commands using the `\?` meta-command:

Code View: [Scroll](#) / [Show All](#)

```
movies=# \?
\A          toggle between unaligned and aligned mode
\c[onnect] [dbname]- [user]]
            connect to new database (currently 'movies')
\C <title>  table title
\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
\dd [object] list comment for table, type, function, or operator
\df         list functions
\do         list operators
\dT         list data types
\e [file]   edit the current query buffer or [file]
            with external editor
\echo <text> write text to stdout
\encoding <encoding> set client encoding
\f <sep>     change field separator
\g [file]   send query to backend (and results in [file] or |pipe)
\h [cmd]    help on syntax of sql commands, * for all commands
\H          toggle HTML mode (currently off)
\i <file>   read and execute queries from <file>
\l          list all databases
\lo_export, \lo_import, \lo_list, \lo_unlink
            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
            <opt> = {format|border|expanded|fieldsep|
            null|recordsep|tuples_only|title|tableattr|pager}
\q          quit psql
\qecho <text> write text to query output stream (see \o)
\r          reset (clear) the query buffer
\s [file]   print history or save it in [file]
\set <var> <value> set internal variable
\t          show only rows (currently off)
\T <tags>   HTML table tags
\unset <var> unset (delete) internal variable
\w <file>   write current query buffer to a <file>
\x          toggle expanded output (currently off)
\z          list table access permissions
\! [cmd]    shell escape or command
movies=#
```

The most important meta-commands are `\?` (meta-command help), and `\q` (quit). The `\h` (SQL help) meta-command is also very useful. Notice that unlike SQL commands, meta-commands don't require a terminating semicolon, which means that meta-commands must be entered entirely on one line. In the next few sections, I'll show you some of the other meta-commands.

Creating Tables

Now that you have seen how to connect to a database and issue a simple query, it's time to create some sample data to work with.

Because you are pretending to model a movie-rental business (that is, a video store), you will create tables that model the data that you might need in a video store. Start by creating three tables: `tapes`, `customers`, and `rentals`.

The `tapes` table is simple: For each videotape, you want to store the name of the movie, the duration, and a unique identifier (remember that you may have more than one copy of any given movie, so the movie name is not sufficient to uniquely identify a specific tape).

Here is the command you should use to create the `tapes` table:

```
CREATE TABLE tapes (  
    tape_id    CHARACTER(8) UNIQUE,  
    title      CHARACTER VARYING(80),  
    duration   INTERVAL  
);
```

Let's take a close look at this command.

The verb in this command is `CREATE TABLE`, and its meaning should be obvious—you want to create a table. Following the `CREATE TABLE` verb is the name of the table (`tapes`) and then a comma-separated list of column definitions, enclosed within parentheses.

Each column in a table is defined by a name and a data type. The first column in `tapes` is named `tape_id`. Column names (and table names) must begin with a letter or an underscore character^[2] and should be 31 characters or fewer^[3]. The `tape_id` column is created with a data type of `CHARACTER(8)`. The data type you define for a column determines the set of values that you can put into that column. For example, if you want a column to hold numeric values, you should use a numeric data type; if you want a column to hold date (or time) values, you should use a date/time data type. `tape_id` holds alphanumeric values (a mixture of numbers and letters), so I chose a character data type, with a length of eight characters.

^[2] You can begin a column or table name with nonalphabetic characters, but you must enclose the name in double quotes. You have to quote the name not only when you create it, but each time you reference it.

^[3] You can increase the maximum identifier length beyond 31 characters if you build PostgreSQL from a source distribution. If you do so, you'll have to remember to increase the identifier length each time you upgrade your server, or whenever you migrate to a different server.

The `tape_id` column is defined as `UNIQUE`. The word `UNIQUE` is not a part of the data type—the data type is `CHARACTER(8)`. The keyword `'UNIQUE'` specifies a column constraint. A column constraint is a condition that must be met by a column. In this case, each row in the `tapes` table must have a unique `tape_id`. PostgreSQL supports a variety of column constraints (and table constraints). I'll cover constraints in [Chapter 2](#).

The `title` is defined as `CHARACTER VARYING(80)`. The difference between `CHARACTER(n)` and `CHARACTER VARYING(n)` is that a `CHARACTER(n)` column is fixed length—it will always contain a fixed number of characters (namely, `n` characters). A `CHARACTER VARYING(n)` column can contain a maximum of `n` characters. I'll mention here that `CHARACTER(n)` can be abbreviated as `CHAR(n)`, and `CHARACTER VARYING(n)` can be abbreviated as `VARCHAR(n)`. I chose `CHAR(8)` as the data type for `tape_id` because I know that a `tape_id` will always contain exactly eight characters, never more and never less. Movie titles, on the other hand, are not all the same length, so I chose `VARCHAR(80)` for those columns. A fixed length data type is a good choice when the data that you store is in fact fixed length; and in some cases, fixed length data types can give you a performance boost. A variable length data type saves space (and often gives you better performance) when the data that you are storing is not all the same length and can vary widely.

The `duration` column is defined as an `INTERVAL`—an `INTERVAL` stores a period of time such as 2 weeks, 1 hour 45 minutes, and so on.

I'll be discussing PostgreSQL data types in detail in [Chapter 2](#). Let's move on to creating the other tables in this example database.

The `customers` table is used to record information about each customer for the video store.

```
CREATE TABLE customers (  
    customer_id    INTEGER UNIQUE,  
    customer_name  VARCHAR(50),  
    phone          CHAR(8),  
    birth_date     DATE,  
    balance        NUMERIC(7,2)
```

```
);
```

Each customer will be assigned a unique `customer_id`. Notice that `customer_id` is defined as an `INTEGER`, whereas the identifier for a tape was defined as a `CHAR(8)`. A `tape_id` can contain alphabetic characters, but a `customer_id` is entirely numeric^[4].

^[4] The decision to define `customer_id` as an `INTEGER` was arbitrary. I simply wanted to show a few more data types here.

I've used two other data types here that you may not have seen before: `DATE` and `NUMERIC`. A `DATE` column can hold date values (century, year, month, and day). PostgreSQL offers other date/time data types that can store different date/time components. For example, a `TIME` column can store time values (hours, minutes, seconds, and microseconds). A `TIMESTAMP` column gives you both date and time components—centuries through microseconds.

A `NUMERIC` column, obviously, holds numeric values. When you create a `NUMERIC` column, you have to tell PostgreSQL the total number of digits that you want to store and the number of fractional digits (that is, the number of digits to the right of the decimal point). The `balance` column contains a total of seven digits, with two digits to the right of the decimal point.

Now, let's create the `rentals` table:

```
CREATE TABLE rentals (  
    tape_id      CHARACTER(8),  
    customer_id  INTEGER,  
    rental_date  DATE  
);
```

When a customer comes in to rent a tape, you will add a row to the `rentals` table to record the transaction. There are three pieces of information that you need to record for each rental: the `tape_id`, the `customer_id`, and the date that the rental occurred. Notice that each row in the `rentals` table refers to a customer (`customer_id`) and a tape (`tape_id`). In most cases, when one row refers to another row, you want to use the same data type for both columns.

What Makes a Relational Database Relational?

Notice that each row in the `rentals` table refers to a row in the `customer` table (and a row in the `tapes` table). In other words, there is a relationship between `rentals` and `customers` and a relationship between `rentals` and `tapes`. The relationship between two rows is established by including an identifier from one row within the other row. Each row in the `rentals` table refers to a customer by including the `customer_id`. That's the heart of the relational database model—the relationship between two entities is established by including the unique identifier of one entity within the other.

Viewing Table Descriptions

At this point, you've defined three tables in the `movies` database: `tapes`, `customers`, and `rentals`. If you want to view the table definitions, you can use the `\d` meta-command in `psql` (remember that a meta-command is not really a SQL command, but a command understood by the `psql` client). The `\d` meta-command comes in two flavors: If you include a table name (`\d customers`), you will see the definition of that table; if you don't include a table name, `\d` will show you a list of all the tables defined in your database.

Code View: [Scroll](#) / Show All

```
$ psql -d movies
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

movies=# \d
          List of relations
  Name      | Type   | Owner
-----+-----+-----
 customers | table  | bruce
 rentals   | table  | bruce
 tapes     | table  | bruce
(3 rows)

movies=# \d tapes
          Table "tapes"
  Column      |      Type      | Modifiers
-----+-----+-----
 tape_id     | character(8)    |
 title       | character varying(80) |
 duration    | interval        |

Indexes:
    "tapes_tape_id_key" UNIQUE, btree (tape_id)

movies=# \d customers
          Table "customers"
  Attribute   |      Type      | Modifier
-----+-----+-----
 customer_id | integer        |
 customer_name | character varying(50) |
 phone       | character(8)    |
 birth_date  | date           |
 balance     | numeric(7,2)    |
Index: customers_customer_id_key

movies=# \d rentals
          Table "rentals"
  Attribute   |      Type      | Modifier
-----+-----+-----
 tape_id     | character(8)    |
 customer_id | integer         |
 rental_date | date            |

movies=#
```

I'll point out a few things about the `\d` meta-command.

Notice that for each column in a table, the `\d` meta-command returns three pieces of information: the column name (or Attribute), the data type, and a Modifier.

The data type reported by the `\d` meta-command is spelled out; you won't see `char(n)` or `varchar(n)`, you'll see `character(n)` and `character varying(n)` instead.

The Modifier column shows additional column attributes. The most commonly encountered modifiers are `NOT NULL` and `DEFAULT ...`. The `NOT NULL` modifier appears when you create a mandatory column—mandatory means that each row in the table must have a value for that column. The `DEFAULT ...` modifier appears when you create a column with a default value. A default value is inserted into a column when you don't specify a value for a column. If you don't specify a default value, PostgreSQL inserts the special value `NULL`. I'll discuss `NULL` values and default values in more detail in [Chapter 2](#).

You might have noticed that the listing for the `tapes` and `customers` tables show that an index has been created. PostgreSQL automatically creates an index for you when you define `UNIQUE` columns. An index is a data structure that PostgreSQL can use to ensure uniqueness. Indexes are also used to increase performance. I'll cover indexes in more detail in [Chapter 3](#), "PostgreSQL SQL Syntax and Use."

Depending on which version of PostgreSQL you're using, you may see each table name listed as "Table "public.table-name". The "public" part is the name of the schema that the table is defined in.

Adding New Records to a Table

The two previous sections showed you how to create some simple tables and how to view the table definitions. Now let's see how to insert data into these tables.

Using the `INSERT` Command

The most common method to get data into a table is by using the `INSERT` command. Like most SQL commands, there are a number of different formats for the `INSERT` command. Let's look at the simplest form first:

```
INSERT INTO table VALUES ( expression [,...] );
```

A Quick Introduction to Syntax Diagrams

In many books that describe a computer language (such as SQL), you will see syntax diagrams. A syntax diagram is a precise way to describe the syntax for a command. Here is an example of a simple syntax diagram:

```
INSERT INTO table VALUES ( expression [,...] );
```

In this book, I'll use the following conventions:

- Words that are presented in uppercase must be entered literally, as shown, except for the case. When you enter these words, it doesn't matter if you enter them in uppercase, lowercase, or mixed case, but the spelling must be the same. SQL keywords are traditionally typed in uppercase to improve readability, but the case does not really matter otherwise.
- A lowercase italic word is a placeholder for user-provided text. For example, the table placeholder shows where you would enter a table name, and expression shows where you would enter an expression.
- Optional text is shown inside a pair of square brackets (`[]`). If you include optional text, don't include the square brackets.
- Finally, `,...` means that you can repeat the previous component one or more times, separating multiple occurrences with commas.

So, the following `INSERT` commands are (syntactically) correct:

```
INSERT INTO states VALUES ( 'WA', 'Washington' );
INSERT INTO states VALUES ( 'OR' );
```

This command would not be legal:

```
INSERT states VALUES ( 'WA' 'Washington' );
```

There are two problems with this command. First, I forgot to include the `INTO` keyword (following `INSERT`). Second, the two values that I provided are not separated by a comma.

When you use an `INSERT` statement, you have to provide the name of the table and the values that you want to include in the new row. The following command inserts a new row into the `customers` table:

```
INSERT INTO customers VALUES
(
  1,
  'William Rubin',
  '555-1212',
  '1970-12-31',
  0.00
);
```

This command creates a single row in the `customers` table. Notice that you did not have to tell PostgreSQL how to match up

each value with a specific column: In this form of the `INSERT` command, PostgreSQL assumes that you listed the values in column order. In other words, the first value that you provide will be placed in the first column, the second value will be stored in the second column, and so forth. (The ordering of columns within a table is defined when you create the table.)

If you don't include one (or more) of the trailing values, PostgreSQL will insert default values for those columns. The default value is typically `NULL`.

Notice that I have included single quotes around some of the data values. Numeric data should not be quoted; most other data types must be. In [Chapter 2](#), I'll cover the literal value syntax for each data type.

In the second form of the `INSERT` statement, you include a list of columns and a list of values:

```
INSERT INTO table ( column [,...] ) VALUES ( expression [,...] );
```

Using this form of `INSERT`, I can specify the order of the column values:

```
INSERT INTO customers
(
    customer_name, birth_date, phone, customer_id, balance
)
VALUES
(
    'William Rubin',
    '1970-12-31',
    '555-1212',
    1,
    0.00
);
```

As long as the column values match up with the order of the column names that you specified, everybody's happy.

The advantage to this second form is that you can omit the value for any column (at least any column that allows `NULL`s). If you use the first form (without column names), you can only omit values for trailing columns. You can't omit a value in the middle of the row because PostgreSQL can only match up column values in left to right order.

Here is an example that shows how to `INSERT` a customer who wasn't willing to give you his date of birth:

```
INSERT INTO customers
(
    customer_name, phone, customer_id, balance
)
VALUES
(
    'William Rubin',
    '555-1212',
    1,
    0.00
);
```

This is equivalent to either of the following statements:

```
INSERT INTO customers
(
    customer_name, birth_date, phone, customer_id, balance
)
VALUES
(
    'William Rubin',
    NULL,
    '555-1212',
    1,
    0.00
);
```

or

```
INSERT INTO customers VALUES
(
    1,
    'William Rubin',
    '555-1212',
    NULL,
```

```
0.00
);
```

There are two other forms for the `INSERT` command. If you want to create a row that contains only default values, you can use the following form:

```
INSERT INTO table DEFAULT VALUES;
```

Of course, if any of the columns in your table are `unique`, you can only insert a single row with default values.

The final form for the `INSERT` statement allows you to insert one or more rows based on the results of a query:

```
INSERT INTO table ( column [,...] ) SELECT query;
```

I haven't really talked extensively about the `SELECT` statement yet (that's in the next section), but I'll show you a simple example here:

```
INSERT INTO customer_backup SELECT * from customers;
```

This `INSERT` command copies every row in the `customers` table into the `customer_backup` table. It's unusual to use `INSERT...SELECT...` to make an exact copy of a table (in fact, there are easier ways to do that). In most cases, you will use the `INSERT...SELECT...` command to make an altered version of a table; you might add or remove columns or change the data using expressions.

Using the `COPY` Command

If you need to load a lot of data into a table, you might want to use the `COPY` command. The `COPY` command comes in two forms. `COPY ... TO` writes the contents of a table into an external file. `COPY ... FROM` reads data from an external file into a table.

Let's start by exporting the `customers` table:

```
COPY customers TO '/tmp/customers.txt';
```

This command copies every row in the `customers` table into a file named `'/tmp/customers.txt'`. Take a look at the `customers.txt` file:

```
1      Jones, Henry    555-1212      1970-10-10    0.00
2      Rubin, William 555-2211      1972-07-10    15.00
3      Panky, Henry   555-1221      1968-01-21    0.00
4      Wonderland, Alison 555-1122      1980-03-05    3.00
```

If you compare the file contents with the definition of the `customers` table:

```
movies=# \d customers
          Table "customers"
  Attribute |          Type          | Modifier
-----+-----+-----
 customer_id | integer                |
 customer_name | character varying(50) |
  phone      | character(8)           |
 birth_date  | date                   |
  balance     | numeric(7,2)           |
Index: customers_customer_id_key
```

You can see that the columns in the text form match (left to right) with the columns defined in the table: The leftmost column is the `customer_id`, followed by `customer_name`, `phone`, and so on. Each column is separated from the next by a tab character and each row ends with an invisible newline character. You can choose a different column separator (with the `DELIMITERS 'delimiter'` option), but you can't change the line terminator. That means that you have to be careful editing a `COPY` file using a DOS (or Windows) text editor because most of these editors terminate each line with a carriage-return/newline combination. That will confuse the `COPY ... FROM` command when you try to import the text file.

The inverse of `COPY ... TO` is `COPY ... FROM`. `COPY ... FROM` imports data from an external file into a PostgreSQL table. When you use `COPY ... FROM`, the format of the text file is very important. The easiest way to find the correct format is to export a few rows using `COPY ... TO`, and then examine the text file.

If you decide to create your own text file for use with the `COPY . . . FROM` command, you'll have to worry about a lot of details like proper quoting, column delimiters, and such. Consult the PostgreSQL reference documentation for more details.

Installing the Sample Database

If you want, you can download a sample database from this book's website: <http://www.conjectrix.com/pgbook>.

After you have downloaded the `bookdata.tar.gz` file, you can unpack it with either of the following commands:

```
$ tar -zxvf bookdata.tar.gz
```

or

```
$ gunzip -c bookdata.tar.gz | tar -xvf -
```

The `bookdata.tar.gz` file contains a number of files and will unpack into your current directory. After unpacking, you will see a subdirectory for each chapter (okay, for most chapters—not all chapters include sample code or sample data).

You can use the `chapter1/load_sample.sql` file to create and populate the three tables that I have discussed (`tapes`, `customers`, and `rentals`). To use the `load_sample.sql` file, execute the following command:

```
$ psql -d movies -f chapter1/load_sample.sql
```

This command drops the `tapes`, `customers`, and `rentals` tables (if they exist), creates them, and adds a few sample rows to each one.

Retrieving Data from the Sample Database

At this point, you should have a sample database (movies) that contains three tables (`tapes`, `customers`, and `rentals`) and a few rows in each table. You know how to get data into a table; now let's see how to view that data.

The `SELECT` statement is used to retrieve data from a database. `SELECT` is the most complex statement in the SQL language, and the most powerful. Using `SELECT`, you can retrieve entire tables, single rows, a group of rows that meet a set of constraints, combinations of multiple tables, expressions, and more. To help you understand the basics of the `SELECT` statement, I'll try to break it down into each of its forms and move from the simple to the more complex.

`SELECT` Expression

In its simplest form, you can use the `SELECT` statement to retrieve one or more values from a set of predefined functions. You've already seen how to retrieve your PostgreSQL user id:

```
movies=# select user;
      current_user
-----
      korrry
(1 row)

movies=# \q
```

Other values that you might want to see are

```
select 5;          -- returns the number 5 (whoopie)
select sqrt(2.0);  -- returns the square root of 2
select timeofday();-- returns current date/time
select now();      -- returns time of start of transaction
select version();  -- returns the version of PostgreSQL you are using

select now(), timeofday();
```

Commenting

The `--` characters introduce a comment—any text that follows is ignored.

The previous example shows how to `SELECT` more than one piece of information—just list all the values that you want, separated by commas.

The PostgreSQL User's Guide contains a list of all the functions that are distributed with PostgreSQL. In [Chapter 2](#), I'll show you how to combine columns, functions, operators, and literal values into more complex expressions.

`SELECT * FROM Table`

You probably won't use the first form of the `SELECT` statement very often—it just isn't very exciting. Moving to the next level of complexity, let's see how to retrieve data from one of the tables that you created earlier:

Code View: [Scroll](#) / [Show All](#)

```
movies=# SELECT * FROM customers;

 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
      3 | Panky, Henry | 555-1221 | 1968-01-21 | 0.00
      1 | Jones, Henry | 555-1212 | 1970-10-10 | 0.00
      4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 | 3.00
      2 | Rubin, William | 555-2211 | 1972-07-10 | 15.00(4 rows)
```

When you write a `SELECT` statement, you have to tell PostgreSQL what information you are trying to retrieve. Let's take a closer look at the components of this `SELECT` statement.

Following the `SELECT` keyword, you specify a list of the columns that you want to retrieve. I used an asterisk (*) here to tell PostgreSQL that we want to see all the columns in the `customers` table.

Next, you have to tell PostgreSQL which table you want to view; in this case, you want to see the `customers` table.

Now let's look at the results of this query. A `SELECT` statement returns a result set. A result set is a table composed of all the rows and columns (or fields) that you request. A result set may be empty.

You asked PostgreSQL to return all the columns in the `customers` table—notice that the columns are displayed (from left to right) in the order that you specified when you created the table. You may have noticed that the rows are returned in an (apparently) arbitrary order. That's an important thing to keep in mind: Unless you specifically request that PostgreSQL return rows in a particular order, you won't be able to predict which rows will come first^[5]. This is a performance feature; if you don't care about row ordering, let PostgreSQL return the rows in the fastest possible way.

^[5] Okay, some people probably could predict the order in which the rows will appear. Those people have way too much free time and consider a propeller to be fashionable headwear. They are also very good at inducing sleep.

Making the Most of the `psql` Console

You'll be spending a lot of time using the `psql` console so it's a good idea to get to know it well. `psql` can do more than just send a command to the server and display the result. You can use `bash`-style tab completion to reduce the amount of typing you have to do. To use tab completion, just type in the first few characters of a word and then press the TAB key—`psql` will try to complete the rest of the word. Tab completion is smart. If you type in the first few characters of a command and then press TAB, `psql` tries to complete the command name. If you've already entered `DROP DATABASE` and then press TAB, `psql` shows you a list of databases; type in the first few characters of a database, press TAB, and `psql` completes the name of the database. If you press TAB in a context where `psql` expects to find a username (like `DROP USER <TAB>`), you'll see a list of users. `psql` can complete column names, data type names, domain names, aggregate names, function names, index names, table names, view names, database names, encodings, languages, schemas, and users.

You can also change the format that `psql` uses to display query results. By default, `psql` uses a format named `aligned` (each column is preceded by a column header and values are aligned within a grid). You can also choose `unaligned`, `html`, or `latex` format. To change the output format, use the command `\pset format format-name`. For example, to switch to `html` format, type in the command `\pset format html`. Once you're in HTML-mode, query results will include the HTML tags required to display the results in tabular form. You probably want to send HTML output to a file (rather than seeing all of the formatting commands in your terminal window). Use the `\o filename` command to route query results to the given filename. See the `psql` manual page (`$ man psql`) for complete details. Play around with the formatting options. Play around with tab completion. Change your `psql` prompt. `psql` packs a lot of power into an easy-to-use interface.

SELECT Single-Column FROM Table

If you don't want to view all of the columns from a table, you can replace the `*` (following the `SELECT` keyword) with the name of a column:

```
movies=# SELECT title FROM tapes;
      title
-----
The Godfather
The Godfather
Casablanca
Citizen Kane
Rear Window
(5 rows)
```

Again, the rows are presented in an arbitrary order. But this time you see only a single column. You may have noticed that "The Godfather" appears twice in this list. That happens because our imaginary video store owns two copies of that movie. I'll show you how to get rid of duplicates in a moment.

SELECT Column-List FROM Table

So far, you have seen how to select all the columns in a table and how to select a single column. Of course, there is a middle ground—you can select a list of columns:

```
movies=# SELECT customer_name, birth_date FROM customers;
      customer_name      | birth_date
-----+-----
Jones, Henry             | 1970-10-10
Rubin, William           | 1972-07-10
Panky, Henry             | 1968-01-21
Wonderland, Alice N.    | 1969-03-05
(4 rows)
```

Instead of naming a single column after the `SELECT` keyword, you can provide a column-separated list of column names. Column names can appear in any order, and the results will appear in the order you specify.

SELECT Expression-List FROM Table

In addition to selecting columns, you can also select expressions. Remember, an expression is a combination of columns, functions, operators, literal values, and other expressions that will evaluate to a single value. Here is an example:

```
movies=# SELECT
movies-#      customer_name,
movies-#      birth_date,
movies-#      age( birth_date )
movies-# FROM customers;
      customer_name      | birth_date |      age
-----+-----+-----
Jones, Henry             | 1970-10-10 |      35
Rubin, William           | 1972-07-10 |      33
Panky, Henry             | 1968-01-21 |      37
Wonderland, Alice N.    | 1969-03-05 |      36
```

```

Jones, Henry           | 1970-10-10 | 31 years 4 mons 3 days 01:00
Rubin, William        | 1972-07-10 | 29 years 7 mons 3 days 01:00
Panky, Henry          | 1968-01-21 | 34 years 23 days
Wonderland, Alice N.  | 1969-03-05 | 32 years 11 mons 8 days
(4 rows)

```

In this example, I've selected two columns and an expression. The expression `age(birth_date)` is evaluated for each row in the table. The `age()` function subtracts the given date from the current date^[6].

^[6] Technically, the `age()` function subtracts the given timestamp (date+time) from the current date and time.

Selecting Specific Rows

The preceding few sections have shown you how to specify which columns you want to see in a result set. Now let's see how to choose only the rows that you want.

First, I'll show you to how to eliminate duplicate rows; then I'll introduce the `WHERE` clause.

```
SELECT [ALL | DISTINCT | DISTINCT ON]
```

In an earlier example, you selected the titles of all the videotapes owned by your video store:

```

movies=# SELECT title from tapes;
      title
-----
The Godfather
The Godfather
Casablanca
Citizen Kane
Rear Window
(5 rows)

```

Notice that "The Godfather" is listed twice (you own two copies of that video). You can use the `DISTINCT` clause to filter out duplicate rows:

```

movies=# SELECT DISTINCT title FROM tapes;
      title
-----
Casablanca
Citizen Kane
Rear Window
The Godfather
(4 rows)

```

You now have a single row with the value "The Godfather." Let's see what happens when you add the `tape_id` back into the previous query:

```

movies=# SELECT DISTINCT title, tape_id FROM tapes;
      title | tape_id
-----+-----
Casablanca | MC-68873
Citizen Kane | OW-41221
Rear Window | AH-54706
The Godfather | AB-12345
The Godfather | AB-67472
(5 rows)

```

We're back to seeing "The Godfather" twice. What happened? The `DISTINCT` clause removes duplicate rows, not duplicate column values; and when the tape IDs are added to the result, the rows containing "The Godfather" are no longer identical.

If you want to filter rows that have duplicate values in one (or more) columns, use the `DISTINCT ON()` form:

```

movies=# SELECT DISTINCT ON (title) title, tape_id FROM tapes;
      title | tape_id
-----+-----
Casablanca | MC-68873
Citizen Kane | OW-41221
Rear Window | AH-54706
The Godfather | AB-12345
(4 rows)

```

Notice that one of the "The Godfather" rows has been omitted from the result set. If you don't include an `ORDER BY` clause (I'll cover that in a moment), you can't predict which row in a set of duplicates will be included in the result set.

You can list multiple columns (or expressions) in the `DISTINCT ON()` clause.

The WHERE Clause

The next form of the `SELECT` statement includes the `WHERE` clause. Here is the syntax diagram for this form:

```
SELECT expression-list FROM table WHERE conditions
```

Using the `WHERE` clause, you can filter out rows that you don't want included in the result set. Let's see a simple example. First, here is the complete `customers` table:

```
movies=# SELECT * FROM customers;
```

customer_id	customer_name	phone	birth_date	balance
3	Panky, Henry	555-1221	1968-01-21	0.00
1	Jones, Henry	555-1212	1970-10-10	0.00
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
2	Rubin, William	555-2211	1972-07-10	15.00

(4 rows)

Now pick out only those customers who owe you some money:

```
movies=# SELECT * FROM customers WHERE balance > 0;
```

customer_id	customer_name	phone	birth_date	balance
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
2	Rubin, William	555-2211	1972-07-10	15.00

(2 rows)

In this example, I've used a single condition to restrict the rows included in the result set: `balance > 0`.

When PostgreSQL executes a `SELECT` statement, it evaluates the `WHERE` clause as it processes each row. If all the conditions specified by the `WHERE` clause are met, the row will be included in the result set (if a row meets all the conditions in the `WHERE` clause, the row satisfies the `WHERE` clause).

Here is an example that is slightly more complex:

```
movies=# SELECT customer_name, phone FROM customers
movies=# WHERE
movies=#   ( balance = 0 )
movies=#   AND
movies=#   ( AGE( birth_date ) < '35 years' )
movies=# ;
customer_name | phone
-----+-----
Jones, Henry  | 555-1212
(1 row)
```

In this query, I've specified two conditions, separated by an `AND` operator. The conditions are: `balance = 0` and `AGE(birth_date) < '34 years'`^[7]. As before, PostgreSQL reads each row in the `customers` table and evaluates the `WHERE` clause. If a given row is to be included in the result set, it must satisfy two constraints—`balance` must be equal to zero and the customer must be younger than 35 years of age. If either of these conditions is false for a given row, that row will not be included in the result set.

^[7] I'll show you how to format various date/time related values in [Chapter 2](#).

`AND` is one of the logical operators supported by PostgreSQL. A logical operator is used to combine logical expressions. A logical expression is an expression that evaluates to `TRUE`, `FALSE`, or unknown (`NULL`). The other two logical operators are `OR` and `NOT`.

Let's see how the `OR` operator works:

```
movies=# SELECT customer_id, customer_name, balance, AGE(birth_date)
movies=# FROM customers
movies=# WHERE
movies=#   ( balance = 0 )
movies=#   OR
movies=#   ( AGE( birth_date ) < '35 years' )
movies=# ;
customer_id | customer_name | balance | age
-----+-----+-----+-----
3 | Panky, Henry | 0.00 | 36 years 8 mons 29 days 23:00:00
1 | Jones, Henry | 0.00 | 34 years 10 days
2 | Rubin, William | 15.00 | 32 years 3 mons 10 days
(3 rows)
```

The `OR` operator evaluates to `TRUE` if either (or both) of the conditions is `TRUE`. The first row (`id = 1`) is included in the result set because it satisfies the first condition (`balance = 0`). It is included even if it does not satisfy the second condition. The second row (`id = 2`) is included in the result set because it satisfies the second condition, but not the first. You can see the difference between `AND` and `OR`. A row satisfies the `AND` operator if both conditions are `TRUE`. A row satisfies the `OR` operator if either condition is `TRUE` (or if both are `TRUE`).

The NOT operator is simple:

```
movies=# SELECT * FROM customers
movies=# WHERE
movies=#     NOT ( balance = 0 )
movies=# ;
```

customer_id	customer_name	phone	birth_date	balance
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
2	Rubin, William	555-2211	1972-07-10	15.00

(2 rows)

NOT evaluates to TRUE if its operand is FALSE and evaluates to FALSE if its operand is TRUE. The NOT operator inverts (or reverses) a test. Without the NOT operator, the previous example would have returned all customers where the balance column was equal to zero. With the NOT operator, you get the other rows instead.

One other point that I should mention about the WHERE clause. Just because you mention a column in the WHERE clause does not mean that you have to include the column in the result set. For example:

```
movies=# SELECT customer_id, customer_name FROM customers
movies=#     WHERE
movies=#         balance != 0
movies=# ;
```

customer_id	customer_name
4	Wonderland, Alice N.
2	Rubin, William

(2 rows)

This example also shows a more common alternative to the NOT operator. The != operator means "is not equal to." The != operator is not an exact replacement for NOT—it can only be used to check for inequality, whereas NOT is used to reverse the sense of any logical expression.

NULL Values

Sometimes when you add data to a table, you find that you don't know what value you should include for a column. For example, you may encounter a customer who does not want to provide you with his or her birthday. What value should be recorded in the birth_date column for that customer? You don't really want to make up an answer—you want a date value that means "unknown." This is what the NULL value is for. NULL usually means that you don't know what value should be entered into a column, but it can also mean that a column does not apply. A NULL value in the birth_date column certainly means that we don't know a customer's birth_date, not that birth_date does not apply^[8]. On the other hand, you might want to include a rating column in the tapes table. A NULL value in the rating column might imply that the movie was produced before ratings were introduced and therefore the rating column does not apply.

^[8] I am making the assumption that the customers for your video store have actually been born. For some of you, that may not be a valid assumption.

Some columns should not allow NULL values. In most cases, it would not make sense to add a customer to your customers table unless you know the customer's name. Therefore, the customer_name column should be mandatory (in other words, customer_name should not allow NULL values).

Let's drop and re-create the customers table so that you can tell PostgreSQL which columns should allow NULL values:

```
movies=# DROP TABLE customers;
DROP
movies=# CREATE TABLE customers (
movies=#     customer_id    INTEGER UNIQUE NOT NULL,
movies=#     customer_name  VARCHAR(50)    NOT NULL,
movies=#     phone         CHAR(8),
movies=#     birth_date    DATE,
movies=#     balance       DECIMAL(7,2)
movies=# );
CREATE
```

The NOT NULL modifier tells PostgreSQL that the customer_id and customer_name columns are mandatory. If you don't specify NOT NULL, PostgreSQL assumes that a column is optional. You can include the keyword NULL to make your choices more obvious:

```
movies=# DROP TABLE customers;
DROP
movies=# CREATE TABLE customers (
movies=#     customer_id    INTEGER UNIQUE NOT NULL,
movies=#     customer_name  VARCHAR(50)    NOT NULL,
movies=#     phone         CHAR(8)         NULL,
movies=#     birth_date    DATE            NULL,
movies=#     balance       DECIMAL(7,2)    NULL
movies=# );
CREATE
```

Notice that a column of any data type can support `NULL` values.

The `NULL` value has a unique property that is often the source of much confusion. `NULL` is not equal to any value, not even itself. `NULL` is not less than any value, and `NULL` is not greater than any value. Let's add a customer with a `NULL` balance:

```
movies=# INSERT INTO customers
movies=# VALUES
movies=# (
movies=# 5, 'Funkmaster, Freddy', '555-FUNK', NULL, NULL
movies=# )
movies=# ;
```

Now we have five customers:

```
movies=# SELECT * FROM customers;
```

customer_id	customer_name	phone	birth_date	balance
3	Panky, Henry	555-1221	1968-01-21	0.00
1	Jones, Henry	555-1212	1970-10-10	0.00
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
2	Rubin, William	555-2211	1972-07-10	15.00
5	Funkmaster, Freddy	555-FUNK		

(5 rows)

One of these customers has a `NULL` balance. Let's try a few queries:

```
movies=# SELECT * FROM customers WHERE balance > NULL;
```

customer_id	customer_name	phone	birth_date	balance
-------------	---------------	-------	------------	---------

(0 rows)

This query did not return any rows. You might think that it should have customer number 2 (Rubin, William); after all, 15.00 is surely greater than 0. But remember, `NULL` is not equal to, greater than, or less than any other value. `NULL` is not the same as zero. Rather than using relational operators ('=', '!=', '<', or '>'), you should use either the `IS` or `IS NOT` operator.

```
movies=# SELECT * FROM customers WHERE balance IS NULL;
```

customer_id	customer_name	phone	birth_date	balance
5	Funkmaster, Freddy	555-FUNK		

(1 row)

```
movies=# SELECT * FROM customers WHERE balance IS NOT NULL;
```

customer_id	customer_name	phone	birth_date	balance
3	Panky, Henry	555-1221	1968-01-21	0.00
1	Jones, Henry	555-1212	1970-10-10	0.00
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
2	Rubin, William	555-2211	1972-07-10	15.00

(4 rows)

The `NULL` value introduces another complication. If `NULL` is not greater than, equal to, or less than any other value, what would '`NULL` + 4' mean? Is `NULL` + 4 greater than `NULL`? It can't be because that would imply that `NULL` is less than `NULL` + 4 and, by definition, `NULL` can't be less than another value. What does all this mean? It means that you can't do math with a `NULL` value.

Code View: [Scroll](#) / [Show All](#)

```
movies=# SELECT customer_id, customer_name, balance, balance+4 FROM customers;
```

customer_id	customer_name	balance	?column?
3	Panky, Henry	0.00	4.00
1	Jones, Henry	0.00	4.00
4	Wonderland, Alice N.	3.00	7.00
2	Rubin, William	15.00	19.00
5	Funkmaster, Freddy		

(5 rows)

This query shows what happens when you try to perform a mathematical operation using `NULL`. When you try to add '4' to `NULL`, you end up with `NULL`.

The `NULL` value complicates logic operators as well. Most programmers are familiar with two-valued logic operators (that is, logic operators that are defined for the values `TRUE` and `FALSE`). When you add in `NULL` values, the logic operators become a bit more complex. [Tables 1.4](#), [1.5](#), and [1.6](#) show the truth tables for each logical operator.

Table 1.4. Truth Table for Three-Valued AND Operator

a	b	a AND b
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
TRUE	NULL	NULL
FALSE	FALSE	FALSE
FALSE	NULL	FALSE
NULL	NULL	NULL

Source: PostgreSQL User's Guide

Table 1.5. Truth Table for Three-Valued OR Operator

a	b	a OR b
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	NULL	TRUE
FALSE	FALSE	FALSE
FALSE	NULL	NULL
NULL	NULL	NULL

Source: PostgreSQL User's Guide

Table 1.6. Truth Table for Three-Valued NOT Operator

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Source: PostgreSQL User's Guide

I don't mean to scare you away from the `NULL` value—it's very useful and often necessary—but you do have to understand the complications that it introduces.

NULLIF() and COALESCE()

PostgreSQL offers two operators that can convert a `NULL` value to some other value or convert a specific value into `NULL`.

The `COALESCE()` operator will substitute a default value whenever it encounters a `NULL`. For example, pretend that you've added two more columns, `male_lead` and `female_lead` to the `tapes` table so that it looks like this:

```
movies=# SELECT * from tapes;
tape_id | title | male_lead | female_lead | duration
-----+-----+-----+-----+-----
AB-12345 | The Godfather | Marlon Brando | | 02:55:00
AB-67472 | The Godfather | Marlon Brando | | 02:55:00
MC-68873 | Casablanca | Humphrey Bogart | Ingrid Bergman | 01:42:00
OW-41221 | Citizen Kane | | | 01:59:00
AH-54706 | Rear Window | James Stewart | Grace Kelly | 
AH-44289 | The Birds | | Tippi Hedren | 01:59:00
(6 rows)
```

You can use the `COALESCE()` operator to transform a `NULL` `male_lead` into the word 'Unknown':

```
movies=# SELECT title, COALESCE( male_lead, 'Unknown' ) FROM tapes;
title | coalesce
-----+-----
The Godfather | Marlon Brando
The Godfather | Marlon Brando
Casablanca | Humphrey Bogart
Citizen Kane | Unknown
Rear Window | James Stewart
The Birds | Unknown
(6 rows)
```

The `COALESCE()` operator is more talented than we've shown here—it can search through a list of values, returning the first non-`NULL` value

it finds. For example, the following query prints the `male_lead`, or, if `male_lead` is `NULL`, the `female_lead`, or if both are `NULL`, 'Unknown':

```
movies=# SELECT title, COALESCE( male_lead, female_lead, 'Unknown' )
movies-#       AS "Starring"
movies-#       FROM TAPES;
   title   | Starring
-----+-----
The Godfather | Marlon Brando
The Godfather | Marlon Brando
Casablanca   | Humphrey Bogart
Citizen Kane | Unknown
Rear Window  | James Stewart
The Birds    | Tippi Hedren
(6 rows)
```

You can string together any number of expressions inside of the `COALESCE()` operator (as long as all expressions evaluate to the same type) and `COALESCE()` will evaluate to the leftmost non-`NULL` value in the list. If all of the expressions inside `COALESCE()` are `NULL`, the entire expression evaluates to `NULL`.

The `NULLIF()` operator translates a non-`NULL` value into `NULL`. `NULLIF()` is often used to do the opposite of `COALESCE()`. `COALESCE()` transforms `NULL` into a default value—`NULLIF()` translates a default value into `NULL`. In many circumstances, you want to treat a numeric value and a `NULL` value as being the same thing. For example, the `balance` column (in the `customers` table) is `NULL` until a customer actually rents a tape: A `NULL` balance implies that you haven't actually done any business with the customer yet. But a `NULL` balance also implies that the customer owes you no money. To convert a `NULL` balance to 0, use `COALESCE(balance, 0)`. To convert a zero balance to `NULL`, use `NULLIF(balance, 0)`. When PostgreSQL evaluates an `NULLIF(arg1, arg2)` expression, it compares the two arguments; if they are equal, the expression evaluates to `NULL`; if they are not equal, the expression evaluates to the value of `arg1`.

The CASE Expression

The `CASE` expression is a more generic form of `NULLIF()` and `COALESCE()`. A `CASE` expression lets you map any given value into some other value. You can write a `CASE` expression in two different forms. The first form (called the simple form) looks like this:

```
CASE expression1
  WHEN value1 THEN result1
  WHEN value2 THEN result2
  ...
  [ ELSE resultn ]
END
```

When PostgreSQL evaluates a simple `CASE` expression, it computes the value of `expression1` then compares the result to `value1`. If `expression1` equals `value1`, the `CASE` expression evaluates to `result1`. If not, PostgreSQL compares `expression1` to `value1`; if they match, the `CASE` expression evaluates to `result2`. PostgreSQL continues searching through the `WHEN` clauses until it finds a match. If none of the values match `expression1`, the expression evaluates to the value specified in the `ELSE` clause. If PostgreSQL gets all the way to the end of the list and you haven't specified an `ELSE` clause, the `CASE` expression evaluates to `NULL`. Note that `result1`, `result2`, ... `resultn` must all have the same data type.

You can see that `NULLIF(balance, 0)` is equivalent to

```
CASE balance
  WHEN 0 THEN NULL
  ELSE balance
END
```

The second, more flexible form of the `CASE` expression is called the searched form:

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ...
  [ ELSE resultn ]
END
```

When PostgreSQL evaluates a searched `CASE` expression, it first evaluates `condition1`. If `condition1` evaluates to `true`, the value of the `CASE` expression is `result1`. If `condition1` evaluates to `false`, PostgreSQL evaluates `condition2`. If that condition evaluates to `true`, the value of the `CASE` expression is `result2`. Otherwise, PostgreSQL moves on to the next condition. PostgreSQL continues to evaluate each condition until it finds one that evaluates to `true`. If none of the conditions is `true`, the `CASE` expression evaluates to `resultn`. If PostgreSQL gets all the way to the end of the list and you haven't specified an `ELSE` clause, the `CASE` expression evaluates to `NULL`.

Like the simple form, `result1`, `result2`, ... `resultn` must all have the same data type. However, in the searched form, the conditions don't have to be similar to each other. For example, if you want to classify the titles in your `tapes` collection (and you're a big Jimmy Stewart fan), you might use a `CASE` expression like this:

```
movies=# SELECT
movies-#   title, male_lead, duration,
movies-#   CASE
movies-#     WHEN duration < '1 hour 45 min' THEN 'short movie'
movies-#     WHEN male_lead = 'James Stewart' THEN 'great movie'
movies-#     WHEN duration > '2 hours' THEN 'long movie'
movies-#   END
movies-# FROM
movies-#   tapes;
   title   | male_lead | duration | case
-----+-----+-----+-----
The Godfather | Marlon Brando | 02:55:00 | long movie
The Godfather | Marlon Brando | 02:55:00 | long movie
Casablanca   | Humphrey Bogart | 01:42:00 | short movie
Citizen Kane |              | 01:59:00 |
Rear Window  | James Stewart |          | great movie
The Birds    |              | 01:59:00 |
(6 rows)
```

The ORDER BY Clause

So far, all the queries that you have seen return rows in an arbitrary order. You can add an `ORDER BY` clause to a `SELECT` command if you need to impose a predictable ordering. The general form of the `ORDER BY` clause is^[9]

^[9] PostgreSQL supports another form for the `ORDER BY` clause: `ORDER BY expression [USING operator] [, ...]`. This might seem a little confusing at first. When you specify `ASC`, PostgreSQL uses the `<` operator to determine row ordering. When you specify `DESC`, PostgreSQL uses the `>` operator. The second form of the `ORDER BY` clause allows you to specify an alternative operator.

```
ORDER BY expression [ ASC | DESC ] [, ...]
```

The `ASC` and `DESC` terms mean ascending and descending, respectively. If you don't specify `ASC` or `DESC`, PostgreSQL assumes that you want to see results in ascending order. The expression following `ORDER BY` is called a sort key.

Let's look at a simple example:

```
movies=# SELECT * FROM customers ORDER BY balance;
```

customer_id	customer_name	phone	birth_date	balance
3	Panky, Henry	555-1221	1968-01-21	0.00
1	Jones, Henry	555-1212	1970-10-10	0.00
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
2	Rubin, William	555-2211	1972-07-10	15.00
5	Funkmaster, Freddy	555-FUNK		

(5 rows)

You can see that this `SELECT` command returns the result set in ascending order of the `balance` column. Here is the same query, but in descending order:

```
movies=# SELECT * FROM customers ORDER BY balance DESC;
```

customer_id	customer_name	phone	birth_date	balance
5	Funkmaster, Freddy	555-FUNK		
2	Rubin, William	555-2211	1972-07-10	15.00
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
3	Panky, Henry	555-1221	1968-01-21	0.00
1	Jones, Henry	555-1212	1970-10-10	0.00

(5 rows)

This time, the largest balance is first, followed by successively smaller values.

You may have noticed something odd about how the `ORDER BY` clause handles the customer named Freddy Funkmaster. Recall from the previous section that `NULL` cannot be compared to other values. By its very nature, the `ORDER BY` clause must compare values. PostgreSQL resolves this issue with a simple rule: `NULL` is always considered larger than all other values when evaluating an `ORDER BY` clause.

You can include multiple sort keys in the `ORDER BY` clause. The following query sorts customers in ascending balance order, and then in descending birth_date order:

```
movies=# SELECT * FROM customers ORDER BY balance, birth_date DESC;
```

customer_id	customer_name	phone	birth_date	balance
1	Jones, Henry	555-1212	1970-10-10	0.00
3	Panky, Henry	555-1221	1968-01-21	0.00
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
2	Rubin, William	555-2211	1972-07-10	15.00
5	Funkmaster, Freddy	555-FUNK		

(5 rows)

When an `ORDER BY` clause contains multiple sort keys, you are telling PostgreSQL how to break ties. You can see that customers 1 and 3 have the same value (0.00) in the `balance` column—you have asked PostgreSQL to order rows using the `balance` column. What happens when PostgreSQL finds two rows with the same balance? When two sort key values are equal, PostgreSQL moves to the next sort key to break the tie. If two sort key values are not equal, sort keys with a lower precedence are ignored. So, when PostgreSQL finds that customers 1 and 3 have the same balance, it moves to the `birth_date` column to break the tie.

If you don't have a sort key with a lower precedence, you won't be able to predict the ordering of rows with duplicate sort key values.

You can include as many sort keys as you like.

LIMIT and OFFSET

Occasionally, you will find that you want to answer a question such as "Who are my top 10 salespeople?" In most relational databases, this is a difficult question to ask. PostgreSQL offers two extensions that make it easy to answer "Top n" or "Bottom n"-type questions. The first extension is the `LIMIT` clause. The following query shows the two customers who owe you the most money:

```
movies=# SELECT * FROM customers ORDER BY balance DESC LIMIT 2;
```

customer_id	customer_name	phone	birth_date	balance
5	Funkmaster, Freddy	555-FUNK		
2	Rubin, William	555-2211	1972-07-10	15.00

(2 rows)

You can see here that I used an `ORDER BY` clause so that the rows are sorted such that the highest balances appear first—in most cases, you won't use a `LIMIT` clause without also using an `ORDER BY` clause. Let's change this query a little—this time we want the top five

customers who have a balance over \$10:

```
movies=# SELECT * FROM customers
movies=# WHERE
movies=#     balance >= 10
movies=# ORDER BY balance DESC
movies=# LIMIT 5;
```

customer_id	customer_name	phone	birth_date	balance
2	Rubin, William	555-2211	1972-07-10	15.00

(1 row)

This example shows that the `LIMIT` clause won't always return the number of rows that were specified. Instead, `LIMIT` returns no more than the number of rows that you request. In this sample database, you have only one customer who owes you more than \$10.

The second extension is the `OFFSET n` clause. The `OFFSET n` clause tells PostgreSQL to skip the first `n` rows of the result set. For example:

```
movies=# SELECT * FROM customers ORDER BY balance DESC OFFSET 1;
```

customer_id	customer_name	phone	birth_date	balance
2	Rubin, William	555-2211	1972-07-10	15.00
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
3	Panky, Henry	555-1221	1968-01-21	0.00
1	Jones, Henry	555-1212	1970-10-10	0.00

(4 rows)

In this case, we are viewing all the customers except the customer with the greatest balance. It's common to use `LIMIT` and `OFFSET` together:

```
movies=# SELECT * FROM customers
movies=# ORDER BY balance DESC LIMIT 2 OFFSET 1;
```

customer_id	customer_name	phone	birth_date	balance
2	Rubin, William	555-2211	1972-07-10	15.00
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00

(2 rows)

Formatting Column Results

So far, you have seen how to tell PostgreSQL which rows you want to view, which columns you want to view, and the order in which the rows should be returned. Let's take a short side-trip here and learn how to change the appearance of the values that you select.

Take a look at the following query:

Code View: [Scroll](#) / [Show All](#)

```
movies=# SELECT customer_id, customer_name, balance, balance+4 FROM customers;
```

customer_id	customer_name	balance	?column?
3	Panky, Henry	0.00	4.00
1	Jones, Henry	0.00	4.00
4	Wonderland, Alice N.	3.00	7.00
2	Rubin, William	15.00	19.00
5	Funkmaster, Freddy		

(5 rows)

PostgreSQL inserts two lines of text between your query and the result set. These two lines are (obviously) column headings. You can see that the header for each of the first three columns contains the name of the column. What about the last column? When you `SELECT` an expression, PostgreSQL uses "`?column?`" for the field header^[10].

^[10] Actually, if you `SELECT` a function (such as `AGE()` or `SQRT()`), PostgreSQL will use the name of the function for the field header.

You can change field headers using the `AS` clause:

```
movies=# SELECT customer_id, customer_name,
movies=#     balance AS "Old balance",
movies=#     balance + 4 AS "New balance"
movies=# FROM customers;
```

customer_id	customer_name	Old Balance	New balance
3	Panky, Henry	0.00	4.00

```

1 | Jones, Henry | 0.00 | 4.00
4 | Wonderland, Alice N. | 3.00 | 7.00
2 | Rubin, William | 15.00 | 19.00
5 | Funkmaster, Freddy | |
(5 rows)

```

Notice that you can provide a field header for table columns as well as for expressions. If you rename a field and the query includes an `ORDER BY` clause that refers to the field, the `ORDER BY` should use the new name, not the original one:

```

movies=# SELECT id, customer_name,
movies-#         balance AS "Old balance",
movies-#         balance + 4 AS "New balance"
movies-# FROM customers
movies-# ORDER BY "Old balance";

```

customer_id	customer_name	Old Balance	New balance
3	Panky, Henry	0.00	4.00
1	Jones, Henry	0.00	4.00
4	Wonderland, Alice N.	3.00	7.00
2	Rubin, William	15.00	19.00
5	Funkmaster, Freddy		

(5 rows)

This section explained how to change the column headers for a `SELECT` command. You can also change the appearance of the data values. In the next section, I'll show you a few examples using date values for illustration.

Working with Date Values

PostgreSQL supports six basic date, time, and date/time data types, as shown in [Table 1.7](#). I'll use the term temporal to cover date, time, and date/time data types.

Table 1.7. PostgreSQL Temporal Data Types

Data Type Name	Type of Data Stored	Earliest Date/ Time	Latest Date/ Time
TIMESTAMP	Date/Time	4713 BC	1465001 AD
TIMESTAMP WITH TIME ZONE	Date/Time	1903 AD	2037 AD
INTERVAL	Interval	-178000000 years	178000000 years
DATE	Date	4713 BC	32767 AD
TIME	Time	00:00:00.00	23:59:59.99
TIME WITH TIME ZONE	Time	00:00:00.00+12	23:59:59.99-12

I'll cover the details of the date/time data types in [Chapter 2](#). You have already seen two of these temporal data types. The `customers` table contains a `DATE` column (`birth_date`):

```

movies=# \d customers
          Table "customers"

```

Column	Type	Modifiers
customer_id	integer	not null
customer_name	character varying(50)	not null
phone	character(8)	
birth_date	date	
balance	numeric(7,2)	

```

Indexes:
    "customers_customer_id_key" UNIQUE, btree (customer_id)

```

```

movies=# SELECT customer_name, birth_date FROM customers;

```

customer_name	birth_date
Jones, Henry	1970-10-10
Rubin, William	1972-07-10
Panky, Henry	1968-01-21
Wonderland, Alice N.	1969-03-05
Funkmaster, Freddy	

(5 rows)

You've also seen the `INTERVAL` data type—the `AGE()` function returns an `INTERVAL`:

```

movies=# SELECT customer_name, AGE( birth_date ) FROM customers;

```

customer_name	age
Jones, Henry	31 years 4 mons 8 days 01:00
Rubin, William	29 years 7 mons 8 days 01:00

```
Panky, Henry          | 34 years 28 days
Wonderland, Alice N. | 32 years 11 mons 13 days
Funkmaster, Freddy   |
(5 rows)
```

Date/time values are usually pretty easy to work with, but there is a complication that you need to be aware of. Let's say that I need to add a new customer:

```
movies=# INSERT INTO customers
movies=# VALUES
movies=# (
movies=# 7, 'Gull, Jonathon LC', '555-1111', '02/05/1984', NULL
movies=# );
```

This customer has a `birth_date` of `'02/05/1984'`—does that mean "February 5th 1984", or "May 2nd 1984"? How does PostgreSQL know which date I meant? The problem is that a date such as `'02/05/1984'` is ambiguous—you can't know which date this string represents without knowing something about the context in which it was entered. `'02/05/1984'` is ambiguous. `'May 02 1984'` is unambiguous.

PostgreSQL enables you to enter and display dates in a number of formats—some date formats are ambiguous and some are unambiguous. The `DATESTYLE` runtime variable tells PostgreSQL how to format dates when displaying data and how to interpret ambiguous dates that you enter.

The `DATESTYLE` variable can be a little confusing. `DATESTYLE` is composed of two parts. The first part, called the convention, tells PostgreSQL how to interpret ambiguous dates. The second part, called the display format, determines how PostgreSQL displays date values. The convention controls date input and the display format controls date output. [Table 1.8](#) shows the `DATESTYLE` display formats.

Table 1.8. `DATESTYLE` Display Formats

Display Format	US Convention	European Convention
ISO	1984-05-02	1984-05-02
GERMAN	02.05.1984	02.05.1984
POSTGRES	Wed May 02 1984	Wed 02 May 1984
SQL	05/02/1984	02/05/1984

Let's talk about the display format first. PostgreSQL supports four different display formats. Three of the display formats are unambiguous and one is ambiguous.

The default display format is named `ISO`. In `ISO` format, dates always appear in the form `'YYYY-MM-DD'`. The next display format is `GERMAN`. In `GERMAN` format, dates always appear in the form `'DD.MM.YYYY'`. The `ISO` and `GERMAN` formats are unambiguous because the format never changes. The `POSTGRES` format is also unambiguous, but the display format can vary. PostgreSQL needs a second piece of information (the convention) to decide whether the month should appear before the day (US convention) or the day should appear before the month (European convention). In `POSTGRES` format, date values display the day-of-the-week and month name in abbreviated text form; for example `'Wed May 02 1984'` (US) or `'Wed 02 May 1984'` (European).

The final display format is `SQL`. `SQL` format is ambiguous. In `SQL` format, the date `'May 02 1984'` is displayed as `'05/02/1984'` (US), or as `'02/05/1984'` (European).

As I mentioned earlier, the `ISO` and `GERMAN` display formats are unambiguous. In `ISO` format, the month always precedes the day. In `GERMAN` format, the day always precedes the month. If you choose `POSTGRES` or `SQL` format, you must also specify the order in which you want the month and day components to appear. You can specify the desired display format and month/day ordering (that is, the convention) in the `DATESTYLE` runtime variable:

```
movies=# SET DATESTYLE TO 'US,ISO';           -- 1984-05-02
movies=# SET DATESTYLE TO 'US,GERMAN';        -- 02.05.1984
movies=# SET DATESTYLE TO 'US,POSTGRES';      -- Wed May 02 1984
movies=# SET DATESTYLE TO 'US,SQL';           -- 05/02/1984

movies=# SET DATESTYLE TO 'EUROPEAN,ISO';     -- 1984-05-02
movies=# SET DATESTYLE TO 'EUROPEAN,GERMAN';  -- 02.05.1984
movies=# SET DATESTYLE TO 'EUROPEAN,POSTGRES'; -- Wed 02 May 1984
movies=# SET DATESTYLE TO 'EUROPEAN,SQL';     -- 02/05/1984
```

The convention part of the `DATESTYLE` variable determines how PostgreSQL will make sense of the date values that you enter. The convention also affects the ordering of the month and day components when displaying a `POSTGRES` or `SQL` date. Note that you are not restricted to entering date values in the format specified by `DATESTYLE`. For example, if you have chosen to display dates in `'US,SQL'` format, you can still enter date values in any of the other formats.

Recall that the `ISO` and `GERMAN` date formats are unambiguous—the ordering of the month and day components is predefined. A date entered in `POSTGRES` format is unambiguous as well—you enter the name of the month so it cannot be confused with the day. If you choose to enter a date in `SQL` format, PostgreSQL will look to the first component of `DATESTYLE` (that is, the convention) to determine whether you want the value interpreted as a US or a European date. Let's look at a few examples.

```
movies=# SET DATESTYLE TO 'US,ISO';
movies=# SELECT CAST('02/05/1984' AS DATE);
```

```
1984-02-05
```

```
movies=# SET DATESTYLE TO 'EUROPEAN, ISO';
movies=# SELECT CAST( '02/05/1984' AS DATE );
1984-05-02
```

In this example, I've asked PostgreSQL to display dates in ISO format, but I've entered a date in an ambiguous format. In the first case, you can see that PostgreSQL interpreted the ambiguous date using US conventions (the month precedes the day). In the second case, PostgreSQL uses European conventions to interpret the date.

Now let's see what happens when I enter an unambiguous date:

```
movies=# SET DATESTYLE TO 'US, ISO';
SET VARIABLE
movies=# SELECT CAST( '1984-05-02' AS DATE );
1984-05-02

movies=# SET DATESTYLE TO 'EUROPEAN, ISO';
SET VARIABLE
movies=# SELECT CAST( '1984-05-02' AS DATE );
1984-05-02
```

This time, there can be no confusion—an ISO-formatted date is always entered in 'YYYY-MM-DD' format. PostgreSQL ignores the convention.

So, you can see that I can enter date values in many formats. If I choose to enter a date in an ambiguous format, PostgreSQL uses the convention part of the current DATESTYLE to interpret the date. I can also use DATESTYLE to control the display format.

Matching Patterns

In the previous two sections, you took a short detour to learn a little about how to format results. Now let's get back to the task of producing the desired results.

The WHERE clause is used to restrict the number of rows returned by a SELECT command^[11]. Sometimes, you don't know the exact value that you are searching for. For example, you may have a customer ask you for a film, but he doesn't remember the exact name, although he knows that the film has the word "Citizen" in the title. PostgreSQL provides two features that make it possible to search for partial alphanumeric values.

^[11] Technically, the WHERE clause constrains the set of rows affected by a SELECT, UPDATE, or DELETE command. I'll show you the UPDATE and DELETE commands a little later.

LIKE and NOT LIKE

The LIKE operator provides simple pattern-matching capabilities. LIKE uses two special characters that indicate the unknown part of a pattern. The underscore (_) character matches any single character. The percent sign (%) matches any sequence of zero or more characters. Table 1.9 shows a few examples.

Table 1.9. Pattern Matching with the LIKE Operator

String	Pattern	Result
The Godfather	%Godfather%	Matches
The Godfather	%Godfather	Matches
The Godfather	%Godfathe_	Matches
The Godfather	___ Godfather	Matches
The Godfather	Godfather%	Does not match
The Godfather	_Godfather	Does not match
The Godfather: Part II	%Godfather	Does not match

Now let's see how to use the LIKE operator in a SELECT command:

```
movies=# SELECT * FROM tapes WHERE title LIKE '%Citizen%';
 tape_id | title | duration
-----+-----+-----
OW-41221 | Citizen Kane |
KJ-03335 | American Citizen, An |
(2 rows)
```

The LIKE operator is case-sensitive:

```
movies=# SELECT * FROM tapes WHERE title LIKE '%citizen%';
 tape_id | title | duration
```



```
-----+-----+-----
(0 rows)
```

If you want to perform case-insensitive pattern matching, use the `ILIKE` operator:

```
movies=# SELECT * FROM tapes WHERE title ILIKE '%citizen%';
 tape_id | title | duration
-----+-----+-----
OW-41221 | Citizen Kane |
KJ-03335 | American Citizen, An |
(2 rows)
```

You can, of course, combine `LIKE` and `ILIKE` with the `NOT` operator to return rows that do not match a pattern:

```
movies=# SELECT * FROM tapes WHERE title NOT ILIKE '%citizen%';
 tape_id | title | duration
-----+-----+-----
AB-12345 | The Godfather |
AB-67472 | The Godfather |
MC-68873 | Casablanca |
AH-54706 | Rear Window |
OW-42200 | Sly | 01:36
OW-42201 | Stone | 4 days 01:36
(6 rows)
```

Pattern Matching with Regular Expressions

The `LIKE` and `ILIKE` operators are easy to use, but they aren't very powerful. Fortunately, PostgreSQL lets you search for data using regular expressions. A regular expression is a string that specifies a pattern. The language that you use to create regular expressions is far more powerful than the `LIKE` and `ILIKE` operators. You have probably used regular expressions before; programs such as `grep`, `awk`, and the Unix (and DOS) shells use regular expressions.

The `LIKE` and `ILIKE` operators define two pattern-matching characters; the regular expression operator defines far more. First, the character `.` within a regular expression operates in the same way as the `_` character in a `LIKE` pattern: it matches any single character. The characters `"."` in a regular expression operate in the same way as the `"%"` character in a `LIKE` pattern: they match zero or more occurrences of any single character.

Notice that in a regular expression, you use two characters to match a sequence of characters, whereas you use a single character in a `LIKE` pattern. The regular expression `"."` is actually two regular expressions combined into one complex expression. As I mentioned earlier, the `"."` character matches any single character. The `"*"` character matches zero or more occurrences of the pattern that precedes it. So, `"*."` means to match any single character, zero or more times. There are three other repetition operators: The `"+"` character matches one or more occurrences of the preceding pattern, and the `"?"` character matches zero or one occurrence of the preceding pattern. If you need to get really fancy (I never have), you can use the form `"{x[,y]}"` to match at least `x` and no more than `y` occurrences of the preceding pattern.

You can also search for things other than `"."`. For example, the character `"^"` matches the beginning of a string and `"$"` matches the end. The regular expression syntax even includes support for character classes. The pattern `"[:upper:]*[:digit:]"` will match any string that includes zero or more uppercase characters followed by a single digit.

The `"|"` character gives you a way to search for a string that matches either of two patterns. For example, the regular expression `"(^God)|.*Donuts.*"` would match a string that either starts with the string `"God"` or includes the word `"Donuts"`.

Regular expressions are extremely powerful, but they can get awfully complex. If you need more information, [Chapter 4](#) of the PostgreSQL User's Manual provides an exhaustive reference to the complete regular expression syntax.

[Table 1.10](#) shows how to construct regular expressions that match the same strings matched by the `LIKE` patterns in shown in [Table 1.9](#).

Table 1.10. Pattern Matching with Regular Expressions

String	Pattern	Result
The Godfather	<code>.* Godfather</code>	Matches
The Godfather	<code>.* Godfather.*</code>	Matches
The Godfather	<code>.* Godfathe.</code>	Matches
The Godfather	<code>... Godfather</code>	Matches
The Godfather	<code>Godfather.*</code>	Does not match
The Godfather	<code>.Godfather</code>	Does not match
The Godfather: Part II	<code>.* Godfather</code>	Does not match

Aggregates

PostgreSQL offers a number of aggregate functions. An aggregate is a collection of things—you can think of an aggregate as the set of rows returned by a query. An aggregate function is a function that operates on an aggregate (nonaggregate functions operate on a single row within an aggregate). Most of the aggregate functions operate on a single value extracted from each row—this is called an aggregate expression.

`COUNT()`

`COUNT()` is probably the simplest aggregate function. `COUNT()` returns the number of objects in an aggregate. The `COUNT()` function comes in four forms:

- `COUNT(*)`
- `COUNT(expression)`
- `COUNT(ALL expression)`
- `COUNT(DISTINCT expression)`

In the first form, `COUNT(*)` returns the number of rows in an aggregate:

```
movies=# SELECT * FROM customers;
```

customer_id	customer_name	phone	birth_date	balance
3	Panky, Henry	555-1221	1968-01-21	0.00
1	Jones, Henry	555-1212	1970-10-10	0.00
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
2	Rubin, William	555-2211	1972-07-10	15.00
5	Funkmaster, Freddy	555-FUNK		
7	Gull, Jonathon LC	555-1111	1984-02-05	
8	Grumby, Jonas	555-2222	1984-02-21	

(7 rows)

```
movies=# SELECT COUNT(*) FROM customers;
```

```
count
-----
      7
(1 row)
```

```
movies=# SELECT COUNT(*) FROM customers WHERE customer_id < 5;
```

```
count
-----
      4
(1 row)
```

You can see from this example that the `COUNT(*)` function pays attention to the `WHERE` clause. In other words, `COUNT(*)` returns the number of rows that filter through the `WHERE` clause; that is, the number of rows in the aggregate.

In the second form, `COUNT(expression)` returns the number of non-NULL values in the aggregate. For example, you might want to know how many customers have a non-NULL balance:

```
movies=# SELECT COUNT( balance ) FROM customers;
```

```
count
-----
      4
(1 row)
```

```
movies=# SELECT COUNT(*) - COUNT( balance ) FROM customers;
```

```
?column?
-----
      3
(1 row)
```

The first query returns the number of non-NULL balances in the `customers` table. The second query returns the number of NULL balances.

The third form, `COUNT(ALL expression)` is equivalent to the second form. PostgreSQL includes the third form for completeness; it complements the fourth form.

`COUNT(DISTINCT expression)` returns the number of distinct non-NULL values in the aggregate.

```
movies=# SELECT DISTINCT balance FROM customers;
```

```

balance
-----
    0.00
    3.00
   15.00

(4 rows)

movies=# SELECT COUNT( DISTINCT balance ) FROM customers;
count
-----
    3
(1 row)

```

You might notice a surprising result in that last example. The first query returns the distinct balances in the `customers` table. Notice that PostgreSQL tells you that it returned four rows—there are four distinct values. The second query returns a count of the distinct balances—it says that there are only three.

Is this a bug? No, both queries returned the correct information. The first query includes the `NULL` value in the result set. `COUNT()`, and in fact all the aggregate functions (except for `COUNT(*)`), ignore `NULL` values.

SUM()

The `SUM(expression)` function returns the sum of all the values in the aggregate expression. Unlike `COUNT()`, you can't use `SUM()` on entire rows^[12]. Instead, you usually specify a single column:

^[12] Actually, you can `SUM(*)`, but it probably doesn't do what you would expect. `SUM(*)` is equivalent to `COUNT(*)`.

```

movies=# SELECT SUM( balance ) FROM customers;
sum
-----
 18.00
(1 row)

```

Notice that the `SUM()` function expects an expression. The name of a numeric column is a valid expression. You can also specify an arbitrarily complex expression as long as that expression results in a numeric value.

You can also `SUM()` an aggregate of intervals. For example, the following query tells you how long it would take to watch all the tapes in your video store:

```

movies=# SELECT SUM( duration ) FROM tapes;
sum
-----
 4 days 03:12
(1 row)

```

AVG()

The `AVG(expression)` function returns the average of an aggregate expression. Like `SUM()`, you can find the average of a numeric aggregate or an interval aggregate.

```

movies=# SELECT AVG( balance ) FROM customers;
avg
-----
 4.5000000000
(1 row)

movies=# SELECT AVG( balance ) FROM customers
movies=# WHERE balance IS NOT NULL;
avg
-----
 4.5000000000
(1 row)

```

These queries demonstrate an important point: the aggregate functions completely ignore rows where the aggregate expression evaluates to `NULL`. The aggregate produced by the second query explicitly omits any rows where the balance is `NULL`. The aggregate produced by the first query implicitly omits `NULL` balances. In other words, the following queries are equivalent:

```

SELECT AVG( balance ) FROM customers;
SELECT AVG( balance ) FROM customers WHERE balance IS NOT NULL;
SELECT SUM( balance ) / COUNT( balance ) FROM customers;

```

But these queries are not equivalent:

```
SELECT AVG( balance ) FROM customers;
SELECT SUM( balance ) / COUNT( * ) FROM customers;
```

Why not? Because `COUNT(*)` counts all rows whereas `COUNT(balance)` omits any rows where the balance is NULL.

MIN() and MAX()

The `MIN(expression)` and `MAX(expression)` functions return the minimum and maximum values, respectively, of an aggregate expression. The `MIN()` and `MAX()` functions can operate on numeric, date/time, or string aggregates:

```
movies=# SELECT MIN( balance ), MAX( balance ) FROM customers;
 min | max
-----+-----
 0.00 | 15.00
(1 row)

movies=# SELECT MIN( birth_date ), MAX( birth_date ) FROM customers;
 min      | max
-----+-----
1968-01-21 | 1984-02-21
(1 row)

movies=# SELECT MIN( customer_name ), MAX( customer_name )
movies=#      FROM customers;
 min      | max
-----+-----
Funkmaster, Freddy | Wonderland, Alice N.
(1 row)
```

Other Aggregate Functions

In addition to `COUNT()`, `SUM()`, `AVG()`, `MIN()`, and `MAX()`, PostgreSQL also supports the `STDDEV(expression)` and `VARIANCE(expression)` aggregate functions. These last two aggregate functions compute the standard deviation and variance of an aggregate, two common statistical measures of variation within a set of observations.

Grouping Results

The aggregate functions are useful for summarizing information. The result of an aggregate function is a single value. Sometimes, you really want an aggregate function to apply to each of a number of subsets of your data. For example, you may find it interesting to compute some demographic information about your customer base. Let's first look at the entire `customers` table:

```
movies=# SELECT * FROM customers;

 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
          3 | Panky, Henry | 555-1221 | 1968-01-21 |    0.00
          1 | Jones, Henry | 555-1212 | 1970-10-10 |    0.00
          4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 |    3.00
          2 | Rubin, William | 555-2211 | 1972-07-10 |   15.00
          5 | Funkmaster, Freddy | 555-FUNK |          |
          7 | Gull, Jonathon LC | 555-1111 | 1984-02-05 |
          8 | Grumby, Jonas | 555-2222 | 1984-02-21 |
(7 rows)
```

Look at the `birth_date` column—notice that you have customers born in three distinct decades (four if you count NULL as a decade):

```
movies=# SELECT DISTINCT( EXTRACT( DECADE FROM birth_date ) )
movies=#      FROM customers;
 date_part
-----
      196
      197
      198
(4 rows)
```

The `EXTRACT()` function extracts a date component from a date/time value. The `DECADE` component looks a little strange, but it makes sense to know whether the decade of the '60s refers to the 1960s or the 2060s, now that we are past Y2K.

Now that you know how many decades are represented in your customer base, you might next want to know how many customers were born in each decade. The `GROUP BY` clause helps answer this kind of question:

```

movies=# SELECT COUNT(*), EXTRACT( DECADE FROM birth_date )
movies=# FROM customers
movies=# GROUP BY EXTRACT( DECADE FROM birth_date );
 count | date_part
-----+-----
      2 |      196
      2 |      197
      2 |      198
      1 |
(4 rows)

```

The `GROUP BY` clause is used with aggregate functions. PostgreSQL sorts the result set by the `GROUP BY` expression and applies the aggregate function to each group.

There is an easier way to build this query. The problem with this query is that you had to repeat the `EXTRACT(DECADE FROM birth_date)` phrase. Instead, you can use the `AS` clause to name the decade field, and then you can refer to that field by name in the `GROUP BY` clause:

```

movies=# SELECT COUNT(*), EXTRACT( DECADE FROM birth_date ) AS decade
movies=# FROM customers
movies=# GROUP BY decade;
 count | decade
-----+-----
      2 |      196
      2 |      197
      2 |      198
      1 |
(4 rows)

```

If you don't request an explicit ordering, the `GROUP BY` clause will cause the result set to be sorted by the `GROUP BY` fields. If you want a different ordering, you can use the `ORDER BY` clause with `GROUP BY`. The following query shows how many customers you have for each decade, sorted by the count:

```

movies=# SELECT
movies=# COUNT(*) as "Customers",
movies=# EXTRACT( DECADE FROM birth_date ) as "Decade"
movies=# FROM customers
movies=# GROUP BY "Decade"
movies=# ORDER BY "Customers";
 Customers | Decade
-----+-----
          1 |
          2 |      196
          2 |      197
          2 |      198
(4 rows)

```

The `NULL` decade looks a little funny in this result set. You have one customer (Freddy Funkmaster) who was too vain to tell you when he was born. You can use the `HAVING` clause to eliminate aggregate groups:

```

movies=# SELECT COUNT(*), EXTRACT( DECADE FROM birth_date ) as decade
movies=# FROM customers
movies=# GROUP BY decade
movies=# HAVING EXTRACT( DECADE FROM birth_date ) IS NOT NULL;
 count | decade
-----+-----
      2 |      196
      2 |      197
      2 |      198
(3 rows)

```

You can see that the `HAVING` clause is similar to the `WHERE` clause. The `WHERE` clause determines which rows are included in the aggregate, whereas the `HAVING` clause determines which groups are included in the result set.

Multi-Table Joins

So far, all the queries that you've seen involve a single table. Most databases contain multiple tables and there are relationships between these tables. This sample database has an example:

```
movies=# \d rentals
```

Table "rentals"		
Attribute	Type	Modifier
tape_id	character(8)	not null
rental_date	date	not null
customer_id	integer	not null

Here's a description of the `rentals` table from earlier in this chapter:

"When a customer comes in to rent a tape, we will add a row to the `rentals` table to record the transaction. There are three pieces of information that we need to record for each rental: the `tape_id`, the `customer_id`, and the date that the rental occurred. Notice that each row in the `rentals` table refers to a customer (`customer_id`) and a tape (`tape_id`)."

You can see that each row in the `rentals` table refers to a tape (`tape_id`) and to a customer (`customer_id`). If you `SELECT` from the `rentals` table, you can see the tape ID and customer ID, but you can't see the movie title or customer name. What you need here is a join. When you need to retrieve data from multiple tables, you join those tables.

PostgreSQL (and all relational databases) supports a number of join types. The most basic join type is a cross-join (or Cartesian product). In a cross join, PostgreSQL joins each row in the first table to each row in the second table to produce a result table. If you are joining against a third table, PostgreSQL joins each row in the intermediate result with each row in the third table.

Let's look at an example. We'll cross-join the `rentals` and `customers` tables. First, I'll show you each table:

```
movies=# SELECT * FROM rentals;
 tape_id | rental_date | customer_id
-----+-----+-----
 AB-12345 | 2001-11-25 |          1
 AB-67472 | 2001-11-25 |          3
 OW-41221 | 2001-11-25 |          1
 MC-68873 | 2001-11-20 |          3
(4 rows)
```

```
movies=# SELECT * FROM customers;
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
          3 | Panky, Henry | 555-1221 | 1968-01-21 |    0.00
          1 | Jones, Henry | 555-1212 | 1970-10-10 |    0.00
          4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 |    3.00
          2 | Rubin, William | 555-2211 | 1972-07-10 |   15.00
          5 | Funkmaster, Freddy | 555-FUNK |          |
          7 | Gull, Jonathon LC | 555-1111 | 1984-02-05 |
          8 | Grumby, Jonas | 555-2222 | 1984-02-21 |
(7 rows)
```

Now I'll join these tables. To perform a cross-join, we simply list each table in the `FROM` clause:

Code View: [Scroll](#) / [Show All](#)

```
movies=# SELECT rentals.*, customers.customer_id, customers.customer_name
movies=# FROM rentals, customers;
 tape_id | rental_date | customer_id | customer_id | customer_name
-----+-----+-----+-----+-----
 AB-12345 | 2001-11-25 |          1 |          3 | Panky, Henry
 AB-12345 | 2001-11-25 |          1 |          1 | Jones, Henry
 AB-12345 | 2001-11-25 |          1 |          4 | Wonderland, Alice N.
 AB-12345 | 2001-11-25 |          1 |          2 | Rubin, William
 AB-12345 | 2001-11-25 |          1 |          5 | Funkmaster, Freddy
 AB-12345 | 2001-11-25 |          1 |          7 | Gull, Jonathon LC
 AB-12345 | 2001-11-25 |          1 |          8 | Grumby, Jonas
 AB-67472 | 2001-11-25 |          3 |          3 | Panky, Henry
 AB-67472 | 2001-11-25 |          3 |          1 | Jones, Henry
 AB-67472 | 2001-11-25 |          3 |          4 | Wonderland, Alice N.
 AB-67472 | 2001-11-25 |          3 |          2 | Rubin, William
 AB-67472 | 2001-11-25 |          3 |          5 | Funkmaster, Freddy
 AB-67472 | 2001-11-25 |          3 |          7 | Gull, Jonathon LC
 AB-67472 | 2001-11-25 |          3 |          8 | Grumby, Jonas
 OW-41221 | 2001-11-25 |          1 |          3 | Panky, Henry
 OW-41221 | 2001-11-25 |          1 |          1 | Jones, Henry
 OW-41221 | 2001-11-25 |          1 |          4 | Wonderland, Alice N.
 OW-41221 | 2001-11-25 |          1 |          2 | Rubin, William
 OW-41221 | 2001-11-25 |          1 |          5 | Funkmaster, Freddy
 OW-41221 | 2001-11-25 |          1 |          7 | Gull, Jonathon LC
 OW-41221 | 2001-11-25 |          1 |          8 | Grumby, Jonas
```

MC-68873		2001-11-20		3		3		Panky, Henry
MC-68873		2001-11-20		3		1		Jones, Henry
MC-68873		2001-11-20		3		4		Wonderland, Alice N.
MC-68873		2001-11-20		3		2		Rubin, William
MC-68873		2001-11-20		3		5		Funkmaster, Freddy
MC-68873		2001-11-20		3		7		Gull, Jonathon LC
MC-68873		2001-11-20		3		8		Grumby, Jonas

(28 rows)

You can see that PostgreSQL has joined each row in the `rentals` table to each row in the `customers` table. The `rentals` table contains four rows; the `customers` table contains seven rows. The result set contains 4 × 7 or 28 rows.

Cross-joins are rarely useful—they usually don't represent real-world relationships.

The second type of join, the inner-join, is very useful. An inner-join starts with a cross-join, and then throws out the rows that you don't want. Take a close look at the results of the previous query. Here are the first seven rows again:

tape_id		rental_date		customer_id		customer_id		customer_name
AB-12345		2001-11-25		1		3		Panky, Henry
AB-12345		2001-11-25		1		1		Jones, Henry
AB-12345		2001-11-25		1		4		Wonderland, Alice N.
AB-12345		2001-11-25		1		2		Rubin, William
AB-12345		2001-11-25		1		5		Funkmaster, Freddy
AB-12345		2001-11-25		1		7		Gull, Jonathon LC
AB-12345		2001-11-25		1		8		Grumby, Jonas
.	
.	
.	

These seven rows were produced by joining the first row in the `rentals` table:

tape_id		rental_date		customer_id
AB-12345		2001-11-25		1

with each row in the `customers` table. What is the real-world relationship between a `rentals` row and a `customers` row? Each row in the `rentals` table contains a customer ID. Each row in the `customers` table is uniquely identified by a customer ID. So, given a `rentals` row, we can find the corresponding `customers` row by searching for a customer where the customer ID is equal to `rentals.customer_id`. Looking back at the previous query, you can see that the meaningful rows are those `WHERE customers.customer_id = rentals.customer_id`.

Qualifying Column Names

Notice that this `WHERE` clause mentions two columns with the same names (`customer_id`). You may find it helpful to qualify each column name by prefixing it with the name of the corresponding table, followed by a period. So, `customers.customer_id` refers to the `customer_id` column in the `customers` table and `rentals.customer_id` refers to the `customer_id` column in the `rentals` table. Adding the table qualifier is sometimes required if a command involves two columns with identical names, but is useful in other cases.

Now you can construct a query that will show us all of the rentals and the names of the corresponding customers:

```
movies=# SELECT rentals.*, customers.customer_id, customers.customer_name
movies=# FROM rentals, customers
movies=# WHERE customers.customer_id = rentals.customer_id;
```

tape_id		rental_date		customer_id		customer_id		customer_name
AB-12345		2001-11-25		1		1		Jones, Henry
AB-67472		2001-11-25		3		3		Panky, Henry
OW-41221		2001-11-25		1		1		Jones, Henry
MC-68873		2001-11-20		3		3		Panky, Henry

(4 rows)

To execute this query, PostgreSQL could start by creating the cross-join between all the tables involved, producing an intermediate result table. Next, PostgreSQL could throw out all the rows that fail to satisfy the `WHERE` clause. In practice, this would be a poor strategy: Cross-joins can get very large quickly. Instead, the PostgreSQL query optimizer analyzes the query and plans an execution strategy to minimize execution time. I'll cover query optimization in [Chapter 4](#).

Join Types

We've seen two join types so far: cross-joins and inner-joins. Now we'll look at outer-joins. An outer-join is similar to an inner-join: a relationship between two tables is established by correlating a column from each table.

In an earlier section, you wrote a query that answered the question: "Which customers are currently renting movies?" How would you answer the question: "Who are my customers and which movies are they currently renting?" You might start by trying the following query:

```
movies=# SELECT customers.*, rentals.tape_id
movies=# FROM customers, rentals
movies=# WHERE rentals.customer_id = customers.customer_id;
```

customer_id	customer_name	phone	birth_date	balance	tape_id
1	Jones, Henry	555-1212	1970-10-10	0.00	AB-12345
3	Panky, Henry	555-1221	1968-01-21	0.00	AB-67472
1	Jones, Henry	555-1212	1970-10-10	0.00	OW-41221
3	Panky, Henry	555-1221	1968-01-21	0.00	MC-68873

rows)

Well, that didn't work. This query showed you which customers are currently renting movies (and the movies that they are renting). What we really want is a list of all customers and, if a customer is currently renting any movies, all the movies rented. This is an outer-join. An outer-join preserves all the rows in one table (or both tables) regardless of whether a matching row can be found in the second table.

The syntax for an outer-join is a little strange. Here is an example:

```
movies=# SELECT customers.customer_name, rentals.tape_id
movies=# FROM customers LEFT OUTER JOIN rentals
movies=# ON customers.customer_id = rentals.customer_id;
```

customer_name	tape_id
Jones, Henry	AB-12345
Jones, Henry	OW-41221
Rubin, William	
Panky, Henry	AB-67472
Panky, Henry	MC-68873
Wonderland, Alice N.	
Funkmaster, Freddy	
Gull, Jonathon LC	
Grumby, Jonas	

rows)

This query is a left outer-join. Why left? Because you will see each row from the left table (the table to the left of the `LEFT OUTER JOIN` phrase). An inner-join would list only two customers ("Jones, Henry" and "Panky, Henry")—the other customers have no rentals.

A `RIGHT OUTER JOIN` preserves each row from the right table. A `FULL OUTER JOIN` preserves each row from both tables.

The following query shows a list of all customers, all tapes, and any rentals:

```
movies=# SELECT customers.customer_name, rentals.tape_id, tapes.title
movies=# FROM customers FULL OUTER JOIN rentals
movies=# ON customers.customer_id = rentals.customer_id
movies=# FULL OUTER JOIN tapes
movies=# ON tapes.tape_id = rentals.tape_id;
```

customer_name	tape_id	title
Jones, Henry	AB-12345	The Godfather
Panky, Henry	AB-67472	The Godfather
		Rear Window
		American Citizen, An
Panky, Henry	MC-68873	Casablanca
Jones, Henry	OW-41221	Citizen Kane
Rubin, William		
Wonderland, Alice N.		
Funkmaster, Freddy		
Gull, Jonathon LC		
Grumby, Jonas		
		Sly
		Stone

(13 rows)

UPDATE

Now that you've seen a number of ways to view your data, let's see how to modify (and delete) existing data.

The `UPDATE` command modifies data in one or more rows. The general form of the `UPDATE` command is

```
UPDATE table SET column = expression [, ...] [WHERE condition]
```

Using the `UPDATE` command is straightforward: The `WHERE` clause (if present) determines which rows will be updated and the `SET` clause determines which columns will be updated (and the new values).

You might have noticed in earlier examples that one of the tapes had a duration of '4 days, 01:36'—that's obviously a mistake. You can correct this problem with the `UPDATE` command as follows:

```
movies=# UPDATE tapes SET duration = '4 hours 36 minutes'
movies=# WHERE tape_id = 'OW-42201';
UPDATE 1
```

```
movies=# SELECT * FROM tapes;
 tape_id | title | duration
-----+-----+-----
AB-12345 | The Godfather |
AB-67472 | The Godfather |
MC-68873 | Casablanca |
OW-41221 | Citizen Kane |
AH-54706 | Rear Window |
OW-42200 | Sly | 01:36
KJ-03335 | American Citizen, An |
OW-42201 | Stone Cold | 04:36
(8 rows)
```

Using the `UPDATE` command, you can update all the rows in the table, a single row, or a set of rows—it all depends on the `WHERE` clause. The `SET` clause in this example updates a single column in all the rows that satisfy the `WHERE` clause. If you want to update multiple columns, list each assignment, separated by commas:

```
movies=# UPDATE tapes
movies=# SET duration = '1 hour 52 minutes', title = 'Stone Cold'
movies=# WHERE tape_id = 'OW-42201';
UPDATE 1
```

```
movies=# SELECT * FROM tapes;
 tape_id | title | duration
-----+-----+-----
AB-12345 | The Godfather |
AB-67472 | The Godfather |
MC-68873 | Casablanca |
OW-41221 | Citizen Kane |
AH-54706 | Rear Window |
OW-42200 | Sly | 01:36
KJ-03335 | American Citizen, An |
OW-42201 | Stone Cold | 01:52
(8 rows)
```

The `UPDATE` statement displays the number of rows that were modified. The following `UPDATE` will modify three of the seven rows in the `customers` table:

Code View: [Scroll](#) / [Show All](#)

```
movies=# SELECT * FROM customers;
```

```
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
3 | Panky, Henry | 555-1221 | 1968-01-21 | 0.00
1 | Jones, Henry | 555-1212 | 1970-10-10 | 0.00
4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 | 3.00
2 | Rubin, William | 555-2211 | 1972-07-10 | 15.00
5 | Funkmaster, Freddy | 555-FUNK |
7 | Gull, Jonathon LC | 555-1111 | 1984-02-05 |
8 | Grumby, Jonas | 555-2222 | 1984-02-21 |
(7 rows)
```

```
movies=# UPDATE customers
movies=# SET balance = 0
movies=# WHERE balance IS NULL;
UPDATE 3
movies=# SELECT * FROM customers;
```

customer_id	customer_name	phone	birth_date	balance
3	Panky, Henry	555-1221	1968-01-21	0.00
1	Jones, Henry	555-1212	1970-10-10	0.00
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
2	Rubin, William	555-2211	1972-07-10	15.00
5	Funkmaster, Freddy	555-FUNK		0.00
7	Gull, Jonathon LC	555-1111	1984-02-05	0.00
8	Grumby, Jonas	555-2222	1984-02-21	0.00

(7 rows)

DELETE

Like UPDATE, the DELETE command is simple. The general format of the DELETE command is

```
DELETE FROM table [ WHERE condition ]
```

The DELETE command removes all rows that satisfy the (optional) WHERE clause. Here is an example:

Code View: [Scroll](#) / Show All

```
movies=# SELECT * FROM tapes;
tape_id | title | duration
-----+-----+-----
AB-12345 | The Godfather |
AB-67472 | The Godfather |
MC-68873 | Casablanca |
OW-41221 | Citizen Kane |
AH-54706 | Rear Window |
OW-42200 | Sly | 01:36
KJ-03335 | American Citizen, An |
OW-42201 | Stone Cold | 01:52
(8 rows)

movies=# BEGIN WORK;
BEGIN
movies=# DELETE FROM tapes WHERE duration IS NULL;
DELETE 6
movies=# SELECT * FROM tapes;
tape_id | title | duration
-----+-----+-----
OW-42200 | Sly | 01:36
OW-42201 | Stone Cold | 01:52
(2 rows)

movies=# ROLLBACK;
ROLLBACK
```

Before we executed the DELETE command, there were eight rows in the tapes table, and six of these tapes had a NULL duration.

You can see that the DELETE statement returns the number of rows deleted ("DELETE 6"). After the DELETE statement, only two tapes remain.

If you omit the WHERE clause in a DELETE command, PostgreSQL will delete all rows. Similarly, forgetting the WHERE clause for an UPDATE command updates all rows. Be careful!

A (Very) Short Introduction to Transaction Processing

You might have noticed two new commands in this example. The `BEGIN WORK` and `ROLLBACK` commands are used for transaction processing. A transaction is a group of commands. Usually, a transaction includes one or more table modifications (`INSERTs`, `DELETES`, and `UPDATES`).

`BEGIN WORK` marks the beginning of a transaction. Inside of a transaction, any changes that you make to the database are temporary changes. There are two ways to mark the end of a transaction: `COMMIT` and `ROLLBACK`. If you `COMMIT` a transaction, you are telling PostgreSQL to write all the changes made within the transaction into the database—in other words, when you `COMMIT` a transaction, the changes become permanent. When you `ROLLBACK` a transaction, all changes made within the transaction are discarded.

You can see that transactions are handy in that you can discard your changes if you change your mind. But transactions are important for another reason. PostgreSQL guarantees that all the modifications in a transaction will complete, or none of them will complete. The classic example of the importance of this property is to pretend that you are transferring money from one bank account to another. This transaction might be written in two steps. The first step is to subtract an amount from the first account. The second step is to add the amount to the second account. Now consider what would happen if your system crashed after completing the first step, but before the second step. Somehow, you've lost money! If you wrap these steps in a transaction, PostgreSQL promises that the first step will be rolled back if the second step fails (actually, the transaction will be rolled back unless you perform a `COMMIT`).

I'll cover the transaction processing features of PostgreSQL in great detail in [Chapter 3](#).

Creating New Tables Using CREATE TABLE...AS

Let's turn our attention to something completely different. Earlier in this chapter, you learned how to use the `INSERT` statement to store data in a table. Sometimes, you want to create a new table based on the results of a `SELECT` command. That's exactly what the `CREATE TABLE...AS` command is designed to do.

The format of `CREATE TABLE...AS` is

```
CREATE [ TEMPORARY | TEMP ] TABLE table [ (column [, ...] ) ]
AS select_clause
```

When you execute a `CREATE TABLE...AS` command, PostgreSQL automatically creates a new table. Each column in the new table corresponds to a column returned by the `SELECT` clause. If you include the `TEMPORARY` (or `TEMP`) keyword, PostgreSQL will create a temporary table. This table is invisible to other users and is destroyed when you end your PostgreSQL session. A temporary table is useful because you don't have to remember to remove the table later—PostgreSQL takes care of that detail for you.

Let's look at an example. A few pages earlier in the chapter, you created a complex join between the `customers`, `rentals`, and `tapes` tables. Let's create a new table based on that query so you don't have to keep entering the same complex query^[13]:

[13] Some readers are probably thinking, "Hey, you should use a view to do that!" You're right, you'll soon see that I just needed a bad example.

Code View: [Scroll](#) / Show All

```
movies=# CREATE TABLE info AS
movies=# SELECT customers.customer_name, rentals.tape_id, tapes.title
movies=# FROM customers FULL OUTER JOIN rentals
movies=# ON customers.customer_id = rentals.customer_id
movies=# FULL OUTER JOIN tapes
movies=# ON tapes.tape_id = rentals.tape_id;
SELECT
movies=# SELECT * FROM info;
 customer_name | tape_id | title
-----+-----+-----
 Jones, Henry  | AB-12345 | The Godfather
Panky, Henry   | AB-67472 | The Godfather
               |         | Rear Window
               |         | American Citizen, An
Panky, Henry   | MC-68873 | Casablanca
Jones, Henry   | OW-41221 | Citizen Kane
Rubin, William |         |
Wonderland, Alice N. |         |
Funkmaster, Freddy |         |
Gull, Jonathon LC |         |
Grumby, Jonas  |         |
               |         | Sly
               |         | Stone Cold
(13 rows)
```

This is the same complex query that you saw earlier. I'll point out a few things about this example. First, notice that the `SELECT` command selected three columns (`customer_name`, `tape_id`, `title`)—the result table has three columns. Next, you can create a table using an arbitrarily complex `SELECT` command. Finally, notice that the `TEMPORARY` keyword is not included; therefore, `info` is a permanent table and is visible to other users.

What happens if you try to create the `info` table again?

```
movies=# CREATE TABLE info AS
movies=# SELECT customers.customer_name, rentals.tape_id, tapes.title
movies=# FROM customers FULL OUTER JOIN rentals
movies=# ON customers.customer_id = rentals.customer_id
movies=# FULL OUTER JOIN tapes
movies=# ON tapes.tape_id = rentals.tape_id;
ERROR: Relation 'info' already exists
```

As you might expect, you receive an error message because the `info` table already exists. `CREATE TABLE...AS` will not automatically drop an existing table. Now let's see what happens if you include the `TEMPORARY` keyword:

```
movies=# CREATE TEMPORARY TABLE info AS
movies=# SELECT * FROM tapes;
SELECT
movies=# SELECT * FROM info;
 tape_id | title | duration
```

```

-----+-----+-----
AB-12345 | The Godfather      |
AB-67472 | The Godfather      |
MC-68873 | Casablanca         |
OW-41221 | Citizen Kane       |
AH-54706 | Rear Window        |
OW-42200 | Sly                 | 01:36
KJ-03335 | American Citizen, An |
OW-42201 | Stone Cold         | 01:52
(8 rows)

```

This time, the `CREATE TABLE...AS` command succeeded. When I `SELECT` from `info`, I see a copy of the `tapes` table. Doesn't this violate the rule that I mentioned earlier ("`CREATE TABLE...AS` will not automatically drop an existing table")? Not really. When you create a temporary table, you are hiding any permanent table of the same name—the original (permanent) table still exists. Other users will still see the permanent table. If you `DROP` the temporary table, the permanent table will reappear:

Code View: [Scroll](#) / Show All

```

movies=# SELECT * FROM info;
 tape_id |      title      | duration
-----+-----+-----
AB-12345 | The Godfather   |
AB-67472 | The Godfather   |
MC-68873 | Casablanca      |
OW-41221 | Citizen Kane    |
AH-54706 | Rear Window     |
OW-42200 | Sly             | 01:36
KJ-03335 | American Citizen, An |
OW-42201 | Stone Cold      | 01:52
(8 rows)

movies=# DROP TABLE info;
DROP
movies=# SELECT * FROM info;
 customer_name | tape_id |      title
-----+-----+-----
Jones, Henry   | AB-12345 | The Godfather
Panky, Henry   | AB-67472 | The Godfather
               |         | Rear Window
               |         | American Citizen, An
Panky, Henry   | MC-68873 | Casablanca
Jones, Henry   | OW-41221 | Citizen Kane
Rubin, William |         |
Wonderland, Alice N. |         |
Funkmaster, Freddy |         |
Gull, Jonathon LC |         |
Grumby, Jonas  |         |
               |         | Sly
               |         | Stone Cold
(13 rows)

```

Using view

In the previous section, I used the `CREATE TABLE...AS` command to create the `info` table so that you didn't have to type in the same complex query over and over again. The problem with that approach is that the `info` table is a snapshot of the underlying tables at the time that the `CREATE TABLE...AS` command was executed. If any of the underlying tables change (and they probably will), the `info` table will be out of synch.

Fortunately, PostgreSQL provides a much better solution to this problem—the view. A view is a named query. The syntax you use to create a view is nearly identical to the `CREATE TABLE...AS` command:

```
CREATE VIEW view AS select_clause;
```

Let's get rid of the `info` table and replace it with a view:

```
movies=# DROP TABLE info;
DROP
movies=# CREATE VIEW info AS
movies-#   SELECT customers.customer_name, rentals.tape_id,tapes.title
movies-#   FROM customers FULL OUTER JOIN rentals
movies-#   ON customers.customer_id = rentals.customer_id
movies-#   FULL OUTER JOIN tapes
movies-#   ON tapes.tape_id = rentals.tape_id;
CREATE
```

While using `psql`, you can see a list of the views in your database using the `\dv` meta-command:

```
movies=# \dv
      List of relations
  Name | Type | Owner
-----+-----+-----
 info | view | bruce
(1 row)
```

You can see the definition of a view using the `\d view-name` meta-command:

```
movies=# \d info
View "info"
Attribute | Type | Modifier
-----+-----+-----
customer_name | character varying(50) |
tape_id | character(8) |
title | character varying(80) |
View definition: SELECT customers.customer_name,
                  rentals.tape_id, tapes.title
                  FROM (( customers FULL JOIN rentals
                  ON ((customers.customer_id = rentals.customer_id)))
                  FULL JOIN tapes
                  ON ((tapes.tape_id = rentals.tape_id)));
```

You can `SELECT` from a view in exactly the same way that you can `SELECT` from a table:

```
movies=# SELECT * FROM info WHERE tape_id IS NOT NULL;
 customer_name | tape_id | title
-----+-----+-----
 Jones, Henry | AB-12345 | The Godfather
Panky, Henry | AB-67472 | The Godfather
Panky, Henry | MC-68873 | Casablanca
 Jones, Henry | OW-41221 | Citizen Kane
(4 rows)
```

The great thing about a view is that it is always in synch with the underlying tables. Let's add a new `rentals` row:

```
movies=# INSERT INTO rentals VALUES( 'KJ-03335', '2001-11-26', 8 );
INSERT 38488 1
```

and then repeat the previous query:

```
movies=# SELECT * FROM info WHERE tape_id IS NOT NULL;
 customer_name | tape_id | title
-----+-----+-----
 Jones, Henry | AB-12345 | The Godfather
```

```
Panky, Henry | AB-67472 | The Godfather
Grumby, Jonas | KJ-03335 | American Citizen, An
Panky, Henry | MC-68873 | Casablanca
Jones, Henry | OW-41221 | Citizen Kane
(5 rows)
```

To help you understand how a view works, you might imagine that the following sequence of events occurs each time you `SELECT` from a view:

1. PostgreSQL creates a temporary table by executing the `SELECT` command used to define the view.
2. PostgreSQL executes the `SELECT` command that you entered, substituting the name of temporary table everywhere that you used the name of the view.
3. PostgreSQL destroys the temporary table.

This is not what actually occurs under the covers, but it's the easiest way to think about views.

Unlike other relational databases, PostgreSQL treats all views as read-only—you can't `INSERT`, `DELETE`, or `UPDATE` a view.

To destroy a view, you use the `DROP VIEW` command:

```
movies=# DROP VIEW info;
DROP
```


Summary

This chapter has given you a gentle introduction to PostgreSQL. You have seen how to install PostgreSQL on your system and how to configure it for use. You've also created a sample database that you'll use throughout the rest of this book.

In the next chapter, I'll discuss the many PostgreSQL data types in more depth, and I'll give you some guidelines for choosing between them.

Chapter 2. Working with Data in PostgreSQL

When you create a table in PostgreSQL, you specify the type of data that you will store in each column. For example, if you are storing a customer name, you will want to store alphabetic characters. If you are storing a customer's birth date, you will want to store values that can be interpreted as dates. An account balance would be stored in a numeric column.

Every value in a PostgreSQL database is defined within a data type. Each data type has a name (`NUMERIC`, `TIMESTAMP`, `CHARACTER`, and so on) and a range of valid values. When you enter a value in PostgreSQL, the data that you supply must conform to the syntax required by the type. PostgreSQL defines a set of functions that can operate on each data type; you can also define your own functions. Every data type has a set of operators that can be used with values of that type. An operator is a symbol used to build up complex expressions from simple expressions. You're already familiar with arithmetic operators such as `+` (addition) and `-` (subtraction). An operator represents some sort of computation applied to one or more operands. For example, in the expression `5 + 3`, `+` is the operator and `5` and `3` are the operands. Most operators require two operands, some require a single operand, and others can function in either context. An operator that works with two operands is called a binary operator. An operator that works with one operand is called a unary operator.

You can convert most values from one data type to another. I'll describe type conversion at the end of this chapter.

This chapter explores each of the data types built into a standard PostgreSQL distribution (yes, you can also define your own custom data types). For each type, I'll show you the range of valid values, the syntax required to enter a value of that type, and a list of operators that you can use with that type.

Each section includes a table showing which operators you can use with a specific data type. For example, in the discussion of character data types, you will see that the string concatenation operator (`||`) can be used to append one string value to the end of another string value. The operator table in that section shows that you use the string concatenation operator to join two `CHARACTER` values, two `VARCHAR` values, or two `TEXT` values. What the table does not show is that you can use the string concatenation operator to append an `INTEGER` value to the end of a `VARCHAR`. PostgreSQL automatically converts the `INTEGER` value into a string value and then applies the `||` operator. It's important to keep this point in mind as you read through this chapter—the operator tables don't show all possible combinations, only the combinations that don't require type conversion.

Later in this chapter, I'll give a brief description of the process that PostgreSQL uses to decide whether an operator (or function) is applicable, and if so, which values require automatic type conversion. For a detailed explanation of the process, see Chapter 5 of the PostgreSQL User's Guide that came with your copy of PostgreSQL.

Besides the operators listed in this section, PostgreSQL offers a huge selection of functions that you can call from within expressions. For a complete, up-to-date list of functions, see the PostgreSQL User's Guide.

NULL Values

`NULL` values represent missing, unknown, or not-applicable values. For example, let's say that you want to add a `membership_expiration_date` to the `customers` table. Some customers might be permanent members—their memberships will never expire. For those customers, the `membership_expiration_date` is not applicable and should be set to `NULL`. You may also find some customers who don't want to provide you with their birth dates. The `birth_date` column for these customers should be `NULL`.

In one case, `NULL` means not applicable. In the other case, `NULL` means don't know. A `NULL membership_expiration_date` does not mean that you don't know the expiration date, it means that the expiration date does not apply. A `NULL birth_date` does not mean that the customer was never born(!); it means that the date of birth is unknown.

Of course, when you create a table, you can specify that a given column cannot hold `NULL` values (`NOT NULL`). When you do so, you aren't affecting the data type of the column; you're just saying that `NULL` is not a legal value for that particular column. A column that prohibits `NULL` values is mandatory; a column that allows `NULL` values is optional.

You may be wondering how a data type could hold all values legal for that type, plus one more value. The answer is that PostgreSQL knows whether a given column is `NULL` not by looking at the column itself, but by first examining a `NULL` indicator (a single bit) stored separately from the column. If the `NULL` indicator for a given row/column is set to `TRUE`, the data stored in the row/column is meaningless. This means that a data row is composed of values for each column plus an array of indicator bits—one bit for each optional column.

Character Values

There are three character (or, as they are more commonly known, string) data types offered by PostgreSQL. A string value is just that—a string of zero or more characters. The three string data types are `CHARACTER(n)`, `CHARACTER VARYING(n)`, and `TEXT`.

A value of type `CHARACTER(n)` can hold a fixed-length string of `n` characters. If you store a value that is shorter than `n`, the value is padded with spaces to increase the length to exactly `n` characters. You can abbreviate `CHARACTER(n)` to `CHAR(n)`. If you omit the "(n)" when you create a `CHARACTER` column, the length is assumed to be 1.

The `CHARACTER VARYING(n)` type defines a variable-length string of at most `n` characters. `VARCHAR(n)` is a synonym for `CHARACTER VARYING(n)`. If you omit the "(n)" when creating a `CHARACTER VARYING` column, you can store strings of any length in that column.

The last string type is `TEXT`. A `TEXT` column is equivalent to a `VARCHAR` column without a specified length—a `TEXT` column can store strings of any length.

Syntax for Literal Values

A string value is a sequence of characters surrounded by a pair of delimiters. Prior to PostgreSQL version 8.0, you had to use a pair of single quote characters to delimit a string value. Starting with version 8.0, you can also define your own delimiters for each string value using a form known as dollar quoting. Each of the following is a valid string value:

```
'I am a string'
'3.14159265'
''
```

You can also write these same string values using dollar quoting as follows:

```
$$I am a string$$
$$3.14159265$$
$$$$
```

The first example is obviously a string value. `'3.14159265'` is also a string value—at first glance it may look like a numeric value but the fact it is surrounded by single quotes tells you that it is really a string. The third example (`''`) is also a valid string: It is the string composed of zero characters (that is, it has a length of zero). It is important to understand that an empty string is not the same as a `NULL` value. An empty string means that you have a known value that just happens to be empty, whereas `NULL` implies that the value is unknown. Consider, for example, that you are storing an employee name in your database. You might create three columns to hold the complete name: `first_name`, `middle_name`, and `last_name`. If you find an employee whose `middle_name` is `NULL`, that should imply that the employee might have a middle name, but you don't know what it is. On the other hand, if you find an employee who has no middle name, you should store that `middle_name` as an empty string. Again, `NULL` implies that you don't have a piece of information; an empty string means that you do have the information, but it just happens to be empty.

If a string is delimited with single quotes, how do you represent a string that happens to include a single quote? There are four choices. First, you can embed a single quote within a string by entering two adjacent quotes. For example, the string "Where's my car?" could be entered as:

```
'Where's my car?'
```

Two other alternatives involve an escape character. An escape is a special character that tells PostgreSQL that the character (or characters) following the escape is to be interpreted as a directive instead of as a literal value. In PostgreSQL, the escape character is the backslash (`\`). When PostgreSQL sees a backslash in a string literal, it discards the backslash and interprets the following characters according to the following rules:

Code View: [Scroll](#) / [Show All](#)

```
\b is the backspace character
\f is the form feed character
\r is the carriage-return character
\n is the newline character
\t is the tab character

\xxx (where xxx is an octal number) means the character whose ASCII value is xxx.
```

If any character, other than those mentioned, follows the backslash, it is treated as its literal value. So, if you want to include a single quote in a string, you can escape the quote by preceding it with a backslash:

```
'Where\'s my car?'
```

Or you can embed a single quote (or any character) within a string by escaping its ASCII value (in octal), as in

```
'Where\047s my car?'
```

Finally, you can use dollar quoting. To write the string "Where's my car?" in dollar-quoted form, use this format:

```
$$Where's my car?$$
```

Notice that in this form, the embedded single quote doesn't cause any problems. When you write a string in dollar-quoted form, the single quote character has no special meaning—it's just another character. You may be thinking that dollar quoting just trades one special delimiter (a single quote) for another (two dollar signs). After all, what happens if you want to embed two consecutive dollar signs in a string value? OK, that's not very likely, but PostgreSQL doesn't just ignore the problem; it lets you define your own delimiters.

In its most simple form, a dollar-quote delimiter is just a pair of dollar signs. To define your own delimiter, simply include a tag between the two dollar signs at the beginning of the string and include the same tag between the two dollar signs at the end of the string. You get to choose the tag but be aware that tags are case sensitive. Here's a string written using a custom delimiter:

```
$MyTag$That restaurant's rated 3 $$$; it must be expensive$MyTag$
```

When you define your own delimiter, embedded single quotes lose their special meaning and so do consecutive dollar signs. You can define a custom delimiter for each string value that you write, but remember that you don't have to define a custom delimiter unless your string contains consecutive dollar signs.

To summarize, here are the four ways that you can embed a single quote within a string:

```
'It's right where you left it'
'It\'s right where you left it'
'It\047s right where you left it'
$It's right where you left it$
```

Supported Operators

PostgreSQL offers a large number of string operators. One of the most basic operations is string concatenation. The concatenation operator (||) is used to combine two string values into a single TEXT value. For example, the expression

```
'This is ' || 'one string'
```

will evaluate to the value: 'This is one string'. And the expression

```
'The current time is ' || now()
```

will evaluate to a TEXT value such as, 'The current time is 2002-01-01 19:45:17-04'.

PostgreSQL also gives you a variety of ways to compare string values. All comparison operators return a BOOLEAN value; the result will be TRUE, FALSE, or NULL. A comparison operator will evaluate to NULL if either of the operands are NULL.

The equality (=) and inequality (<>) operators behave the way you would expect—two strings are equal if they contain the same characters (in the same positions); otherwise, they are not equal. You can also determine whether one string is greater than or less than another (and of course, greater than or equal to and less than or equal to).

Table 2.1^[1] shows a few sample string comparisons.

^[1] You might find the format of this table a bit confusing at first. In the first column, I use the 'θ' character to represent any one of the operators listed in the remaining columns. So, the first row of the table tells you that 'string' < 'string' evaluates to FALSE, 'string' <= 'string' evaluates to TRUE, 'string' = 'string' evaluates to TRUE, and so forth. I'll use the 'θ' character throughout this chapter to indicate an operator.

Table 2.1. Sample String Comparisons

Operator (θ)						
Expression	<	<=	=	<>	>=	>
'string' θ 'string'	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE
'string1' θ 'string'	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE
'String1' θ 'string'	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE

You can also use pattern-matching operators with string values. PostgreSQL defines eight pattern-matching operators, but the names are a bit contrived and not particularly intuitive.

Table 2.2 contains a summary of the string operators.

Table 2.2. String Operators

Operator	Meaning	Case Sensitive?
----------	---------	-----------------

<code> </code>	Concatenation	Not applicable
<code>~</code>	Matches regular expression	Yes
<code>~~</code>	Matches <code>LIKE</code> expression	Yes
<code>~*</code>	Matches regular expression	No
<code>~~*</code>	Matches <code>LIKE</code> expression	No
<code>!~</code>	Does not match regular expression	Yes
<code>!~~</code>	Does not match <code>LIKE</code> expression	Yes
<code>!~*</code>	Does not match regular expression	No
<code>!~~*</code>	Does not match <code>LIKE</code> expression	No

The first set of pattern-matching operators is related to the `LIKE` keyword. `~~` is equivalent to `LIKE`. The `~~*` operator is equivalent to `ILIKE` — it is a case-insensitive version of `LIKE`. `!~~` and `!~~*` are equivalent to `NOT LIKE` and `NOT ILIKE`, respectively.

The second set of pattern-matching operators is used to match a string value against a regular expression (regular expressions are described in more detail in [Chapter 1](#), "Introduction to PostgreSQL and SQL"). The naming convention for the regular expression operators is similar to that for the `LIKE` operators—regular expression operators are indicated with a single tilde and `LIKE` operators use two tildes. The `~` operator compares a string against a regular expression (returning `True` if the string satisfies the regular expression). `~*` compares a string against a regular expression, ignoring differences in case. The `!~` operator returns `False` if the string value matches the regular expression (and returns `True` if the string satisfies the regular expression). The `!~*` operator returns `False` if the string value matches the regular expression, ignoring differences in case, and returns `True` otherwise.

Type Conversion Operators

There are two important operators that you should know about before we go much further—actually it's one operator, but you can write it two different ways.

The `CAST()` operator is used to convert a value from one data type to another. There are two ways to write the `CAST()` operator:

```
CAST(expression AS type)
expression::type
```

No matter which way you write it, the `expression` is converted into the specified type. Of course, not every value can be converted into every type. For example, the expression `CAST('abc' AS INTEGER)` results in an error (specifically, `pg_atoi: error in "abc": can't parse "abc"`) because `'abc'` obviously can't be converted into an integer.

Most often, your casting requirements will come in either of two forms: you will need to `CAST()` a string value into some other type, or you will need to convert between related types (for example, `INTEGER` into `NUMERIC`). When you `CAST()` a string value into another data type, the string must be in the form required by the literal syntax for the target data type. Each of the following sections describes the literal syntax required by each type. When you convert between related data types, you may gain or lose precision. For example, when you convert from a fractional numeric type into an integer type, the value is rounded:

```
movies=# SELECT CAST( CAST( 12345.67 AS FLOAT8 ) AS INTEGER );
?column?
-----
 12346
```

Numeric Values

PostgreSQL provides a variety of numeric data types. Of the six numeric types, four are exact (`SMALLINT`, `INTEGER`, `BIGINT`, `NUMERIC(p, s)`) and two are approximate (`REAL`, `DOUBLE PRECISION`).

Three of the four exact numeric types (`SMALLINT`, `INTEGER`, and `BIGINT`) can store only integer values. The fourth (`NUMERIC(p, s)`) can accurately store any value that fits within the specified number (`p`) of digits.

The approximate numeric types, on the other hand, cannot store all values exactly. Instead, an approximate data type stores an approximation of a real number. The `DOUBLE PRECISION` type, for example, can store a total of 15 significant digits, but when you perform calculations using a `DOUBLE PRECISION` value, you can run into rounding errors. It's easy to see this problem:

```
movies=# select 2000.3 - 2000.0;
?column?
-----
0.2999999999999955
(1 row)
```

Size, Precision, and Range-of-Values

The four exact data types can accurately store any value within a type-specific range. The exact numeric types are described in [Table 2.3](#).

Table 2.3. Exact Numeric Data Types

Type Name	Size in Bytes	Minimum Value	Maximum Value
<code>SMALLINT</code>	2	-32768	+ 32767
<code>INTEGER</code>	4	-2147483648	+ 2147483647
<code>BIGINT</code>	8	-9223372036854775808	+ 9223372036854775807
<code>NUMERIC(p, s)</code>	11+ (p/2)	No limit	No limit

The `NUMERIC(p, s)` data type can accurately store any number that fits within the specified number of digits. When you create a column of type `NUMERIC(p, s)`, you can specify the total number of decimal digits (`p`) and the number of fractional digits (`s`). The total number of decimal digits is called the precision, and the number of fractional digits is called the scale.

[Table 2.3](#) shows that there is no limit to the values that you can store in a `NUMERIC(p, s)` column. In fact, there is a limit (normally 1,000 digits), but you can adjust the limit by changing a symbol and rebuilding your PostgreSQL server from source code.

The two approximate numeric types are named `REAL` and `DOUBLE PRECISION`. [Table 2.4](#) shows the size and range for each of these data types, while [Table 2.5](#) shows alternative names for the data types.

Table 2.4. Approximate Numeric Data Types

Type Name	Size in Bytes	Range
<code>REAL</code>	4	6 decimal digits
<code>DOUBLE PRECISION</code>	8	15 decimal digits

Table 2.5. Alternate Names for Numeric Data Types

Common Name	Synonyms
<code>SMALLINT</code>	<code>INT2</code>
<code>INTEGER</code>	<code>INT</code> , <code>INT4</code>
<code>BIGINT</code>	<code>INT8</code>
<code>NUMERIC(p, s)</code>	<code>DECIMAL(p, s)</code>
<code>REAL</code>	<code>FLOAT</code> , <code>FLOAT4</code>

SERIAL, BIGSERIAL, and SEQUENCES

Besides the numeric data types already described, PostgreSQL supports two "advanced" numeric types: `SERIAL` and `BIGSERIAL`. A `SERIAL` column is really an unsigned `INTEGER` whose value automatically increases (or decreases) by a defined increment as you add new rows. Likewise, a `BIGSERIAL` is a `BIGINT` that increases in value. When you create a `BIGSERIAL` or `SERIAL` column, PostgreSQL will automatically create a `SEQUENCE` for you. A `SEQUENCE` is an object that generates sequence numbers for you. I'll talk more about `SEQUENCES` later in this chapter.

Syntax for Literal Values

When you need to enter a numeric literal, you must follow the formatting rules defined by PostgreSQL. There are two distinct styles for numeric literals: integer and fractional (the PostgreSQL documentation refers to fractional literals as floating-point literals).

Let's start by examining the format for fractional literals. Fractional literals can be entered in any of the following forms^[2]:

^[2] Syntax diagrams are described in detail in [Chapter 1](#).

```
[-]digits.[digits][E[+|-]digits]
[-][digits].[digits][E[+|-]digits]
[-]digits[+|-]digits
```

Here are some examples of valid fractional literals:

```
3.14159
2.0e+15
0.2e-15
4e10
```

A numeric literal that contains only digits is considered to be an integer literal:

```
[-]digits
```

Here are some examples of valid integer literals:

```
-100
55590332
9223372036854775807
-9223372036854775808
```

A fractional literal is always considered to be of type `DOUBLE PRECISION`. An integer literal is considered to be of type `INTEGER`, unless the value is too large to fit into an integer—in which case, it will be promoted first to type `BIGINT`, then to `NUMERIC` or `REAL` if necessary.

Supported Operators

PostgreSQL supports a variety of arithmetic, comparison, and bit-wise operators for the numeric data types. [Tables 2.6](#) and [2.7](#) give some examples of the arithmetic operators.

Table 2.6. Arithmetic Operators for Integers

Data Types	Valid Operators (θ)
INT2 θ INT2	+ - * / %
INT2 θ INT4	+ - * / %
INT4 θ INT2	+ - * / %
INT4 θ INT4	+ - * / %

INT4 θ INT8	+ - * /
INT8 θ INT4	+ - * /
INT8 θ INT8	+ - * / %

Table 2.7. Arithmetic Operators for Floats

Data Types	Valid Operators (θ)
FLOAT4 θ FLOAT4	* + - /
FLOAT4 θ FLOAT8	* + - /
FLOAT8 θ FLOAT4	* + - /
FLOAT8 θ FLOAT8	* + - / ^

You use the comparison operators to determine the relationship between two numeric values. PostgreSQL supports the usual operators: $<$, $<=$, $>$, $>=$, $=$, $<>$ (not equal), $=$, $>$, and $>=$. You can use the comparison operators with all possible combinations of the numeric data types (some combinations will require type conversion).

PostgreSQL also provides a set of bit-wise operators that you can use with the integer data types. Bit-wise operators work on the individual bits that make up the two operands.

The easiest way to understand the bit-wise operators is to first convert your operands into binary notation—for example:

```
decimal 12 = binary 00001100
decimal 7  = binary 00000111
decimal 21 = binary 00010101
```

Next, let's look at each operator in turn.

The **AND** ($\&$) operator compares corresponding bits in each operand and produces a 1 if both bits are 1 and a 0 otherwise. For example:

```
00001100 & 00000111 &
00010101   00010101
-----
00000100   00000101
```

The **OR** ($|$) operator compares corresponding bits in each operand and produces a 1 if either (or both) bit is 1 and a 0 otherwise. For example:

```
00001100 | 00000111 |
00010101   00010101
-----
00011101   00010111
```

The **XOR** ($\#$) operator is similar to **OR**. **XOR** compares corresponding bits in each operand, and produces a 1 if either bit, but not both bits, is 1, and produces a 0 otherwise.

```
00001100 # 00000111 #
00010101   00010101
-----
00011001   00010010
```

PostgreSQL also provides two bit-shift operators.

The left-shift operator ($<<$) shifts the bits in the first operand n bits to the left, where n is the second operand. The leftmost n bits are discarded, and the rightmost n bits are set to 0. A left-shift by n bits is equivalent to multiplying the first operand by 2^n —for example:

```
00001100 << 2(decimal) = 00110000
00010101 << 3(decimal) = 10101000
```


The right-shift operator (`>>`) `>` shifts the bits in the first operand `n` bits to the right, where `n` is the second operand. The rightmost `n` bits are discarded, and the leftmost `n` bits are set to 0. A right-shift by `n` bits is equivalent to dividing the first operand by 2^n :

```
00001100 >> 2(decimal) = 00000011
00010101 >> 3(decimal) = 00000010
```

The final bit-wise operator is the binary NOT (`~`). Unlike the other bit-wise operators, NOT is a unary operator—it takes a single operand. When you apply the NOT operator to a value, each bit in the original value is toggled: ones become zeroes and zeroes become ones. For example:

```
~00001100 = 11110011
~00010101 = 11101010
```

Table 2.8 shows the data types that you can use with the bit-wise operators.

Table 2.8. Bit-Wise Operators for Integers	
Data Types	Valid Operators (θ)
INT2 θ INT2	# & << >>
INT4 θ INT4	# & << >>
INT8 θ INT4	<< >>
INT8 θ INT8	# &

Date/ Time Values

PostgreSQL supports four basic temporal data types plus a couple of extensions that deal with time zone issues.

The `DATE` type is used to store dates. A `DATE` value stores a century, year, month, and day.

The `TIME` data type is used to store a time-of-day value. A `TIME` value stores hours, minutes, seconds, and microseconds. It is important to note that a `TIME` value does not contain a time zone—if you want to include a time zone, you should use the type `TIME WITH TIME ZONE`. `TIMETZ` is a synonym for `TIME WITH TIME ZONE`.

The `TIMESTAMP` data type combines a `DATE` and a `TIME`, storing a century, year, month, day, hour, minutes, seconds, and microseconds. Unlike the `TIME` data type, a `TIMESTAMP` does include a time zone. If, for some reason, you want a date/time value that does not include a time zone, you can use the type `TIMESTAMP WITHOUT TIME ZONE`.

The last temporal data type is the `INTERVAL`. An `INTERVAL` represents a span of time. I find that the easiest way to think about `INTERVAL` values is to remember that an `INTERVAL` stores some (possibly large) number of seconds, but you can group the seconds into larger units for convenience. For example, the `CAST('1 week' AS INTERVAL)` is equal to `CAST('604800 seconds' AS INTERVAL)`, which is equal to `CAST('7 days' AS INTERVAL)`—you can use whichever format you find easiest to work with.

Table 2.9 lists the size and range for each of the temporal data types.

Table 2.9. Temporal Data Type Sizes and Ranges

Data Type	Size (in bytes)	Range
<code>DATE</code>	4	-01-MAR-4801 BC 31-DEC-32767
<code>TIME [WITHOUT TIME ZONE]</code>	4	-00:00:00.00 23:59:59.99
<code>TIME WITH TIME ZONE</code>	12	-00:00:00.00+ 12 23:59:59.00-12
<code>TIMESTAMP [WITH TIME ZONE]</code>	8	-24-NOV-4714 BC 31-DEC- 5874897
<code>TIMESTAMP WITHOUT TIME ZONE</code>	8	-24-NOV-4714 BC 31-DEC- 5874897
<code>INTERVAL</code>	12	-- 178000000 YEARS + 178000000 YEARS

The data types that contain a time value (`TIME`, `TIME WITH TIME ZONE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `INTERVAL`) have microsecond precision. The `DATE` data type has a precision of one day.

Syntax for Literal Values

I covered date literal syntax pretty thoroughly in [Chapter 1](#); see the section titled "[Working with Date Values](#)."

You may recall from [Chapter 1](#) that date values can be entered in many formats, and you have to tell PostgreSQL how to interpret ambiguous values. Fortunately, the syntax for `TIME`, `TIMESTAMP`, and `INTERVAL` values is much more straightforward.

A `TIME` value stores hours, minutes, seconds, and microseconds. The syntax for a `TIME` literal is

```
hh:mm[:ss[.μ]] [AM|PM]μ
```

where `hh` specifies the hour, `mm` specifies the number of minutes past the hour, `ss` specifies the number of seconds, and `μ` specifies the number of microseconds. If you include an `AM` or `PM` indicator, the `hh` component must be less than or equal to 12; otherwise, the hour can range from 0 to 24.

Entering a `TIME WITH TIME ZONE` value is a bit more complex. A `TIME WITH TIME ZONE` value is a `TIME` value, plus a time zone. The time zone component can be specified in two ways. First, you can include an offset (in minutes and hours) from UTC:

```
hh:mm[:ss[.μ]] [AM|PM] [{+|-}HH[:MM]]
```

where `HH` is the number of hours and `MM` is the number of minutes distant from UTC. Negative values are considered to be

west of the prime meridian, and positive values are east of the prime meridian.

You can also use a standard time zone abbreviation (such as UTC, PDT, or EST) to specify the time zone:

```
hh:mm[:ss[.p ]][AM|PM][ZZZ]
```

Table 2.10 shows all the time zone abbreviations accepted by PostgreSQL version 8.0.

Table 2.10. PostgreSQL Time Zone Names

Names	Offset	Description
FJST	-13:00	Fiji Summer Time
FJT	-12:00	Fiji Time
IDLW	-12:00	International Date Line, West
BST	-11:00	Bering Summer Time
NT	-11:00	Nome Time
NUT	-11:00	Niue Time
AHST	-10:00	Alaska-Hawaii Std Time
CAT	-10:00	Central Alaska Time
HST	-10:00	Hawaii Std Time
THAT	-10:00	Tahiti Time
TKT	-10:00	Tokelau Time
MART	-09:30	Marquesas Time
AKST	-09:00	Alaska Standard Time
GAMT	-09:00	Gambier Time
HDT	-09:00	Hawaii/Alaska Daylight Time
YST	-09:00	Yukon Standard Time
AKDT	-08:00	Alaska Daylight Time
PST	-08:00	Pacific Standard Time
YDT	-08:00	Yukon Daylight Time
MST	-07:00	Mountain Standard Time
PDT	-07:00	Pacific Daylight Time
CST	-06:00	Central Standard Time
EAST	-06:00	Easter Island Time
GALT	-06:00	Galapagos Time
MDT	-06:00	Mountain Daylight Time
ZP6	-06:00	UTC + 6 hours
ACT	-05:00	Atlantic/Porto Acre Time
CDT	-05:00	Central Daylight Time
COT	-05:00	Columbia Time
EASST	-05:00	Easter Island Summer Time
ECT	-05:00	Ecuador Time
EST	-05:00	Eastern Standard Time
PET	-05:00	Peru Time
ZP5	-05:00	UTC + 5 hours
ACST	-04:00	Atlantic/Porto Acre Summer Time
AMT	-04:00	Amazon Time (Porto Velho)
AST	-04:00	Atlantic Std Time (Canada)

BOT	-04:00	Bolivia Time
CLT	-04:00	Chile Time
ECT	-04:00	Eastern Caribbean Time
EDT	-04:00	Eastern Daylight Time
GYT	-04:00	Guyana Time
PYT	-04:00	Paraguay Time
VET	-04:00	Venezuela Time
ZP4	-04:00	UTC + 4 hours
NFT	-03:30	Newfoundland Standard Time
NST	-03:30	Newfoundland Standard Time
ADT	-03:00	Atlantic Daylight Time
AMST	-03:00	Amazon Summer Time (Porto Velho)
ART	-03:00	Argentina Time
AWT	-03:00	Brazil Time
BRT	-03:00	Brasilia Time
BST	-03:00	Brazil Standard Time
CLST	-03:00	Chile Summer Time
FKST	-03:00	Falkland Islands Summer Time
GFT	-03:00	French Guiana Time
PYST	-03:00	Paraguay Summer Time
UYT	-03:00	Uruguay Time
WGT	-03:00	West Greenland Time
NDT	-02:30	Newfoundland Daylight Time
BRST	-02:00	Brasilia Summer Time
FKT	-02:00	Falkland Islands Time
FNT	-02:00	Fernando de Noronha Time
PMDT	-02:00	Pierre & Miquelon Daylight Time
UYST	-02:00	Uruguay Summer Time
WGST	-02:00	West Greenland Summer Time
AZOT	-01:00	Azores Time
EGT	-01:00	East Greenland Time
FNST	-01:00	Fernando de Noronha Summer Time
SET	-01:00	Seychelles Time
WAT	-01:00	West Africa Time
AZOST	+ 00:00	Azores Summer Time
EGST	+ 00:00	East Greenland Summer Time
GMT	+ 00:00	Greenwich Mean Time
UTC	+ 00:00	Universal Coordinated Time
UT	+ 00:00	Universal Time
WET	+ 00:00	Western Europe
ZULU	+ 00:00	Universal Time
Z	+ 00:00	ISO-8601 Universal Time
BST	+ 01:00	British Summer Time
CET	+ 01:00	Central European Time
DNT	+ 01:00	Dansk Normal Time

FST	+ 01:00	French Summer Time
MET	+ 01:00	Middle Europe Time
MEWT	+ 01:00	Middle Europe Winter Time
MEZ	+ 01:00	Middle Europe Zone
NOR	+ 01:00	Norway Standard Time
SWT	+ 01:00	Swedish Winter Time
WEST	+ 01:00	Western Europe Summer Time
WETDST	+ 01:00	Western Europe Daylight Savings Time
BDST	+ 02:00	British Double Summer Time
CEST	+ 02:00	Central European Dayl.Time
CETDST	+ 02:00	Central European Dayl.Time
EET	+ 02:00	Eastern Europe, USSR Zone 1
FWT	+ 02:00	French Winter Time
IST	+ 02:00	Israel Time
MEST	+ 02:00	Middle Europe Summer Time
METDST	+ 02:00	Middle Europe Daylight Time
SST	+ 02:00	Swedish Summer Time
BT	+ 03:00	Baghdad Time
EAT	+ 03:00	East Africa Time
EAT	+ 03:00	Indian Antananarivo Time
EEST	+ 03:00	Eastern Europe Summer Time
EETDST	+ 03:00	Eastern Europe Daylight Time
HMT	+ 03:00	Hellas Mediterranean Time
MSK	+ 03:00	Moscow Time
IRT	+ 03:30	Iran Time
IT	+ 03:30	Iran Time
AMT	+ 04:00	Armenia Time (Yerevan)
AZT	+ 04:00	Azerbaijan Time
EAST	+ 04:00	Indian Antananarivo Savings Time
GET	+ 04:00	Georgia Time
MSD	+ 04:00	Moscow Summer Time
MUT	+ 04:00	Mauritius Island Time
RET	+ 04:00	Reunion Island Time
SCT	+ 04:00	Mahe Island Time
AFT	+ 04:30	Kabul Time
AMST	+ 05:00	Armenia Summer Time (Yerevan)
AZST	+ 05:00	Azerbaijan Summer Time
GEST	+ 05:00	Georgia Summer Time
IOT	+ 05:00	Indian Chagos Time
KGT	+ 05:00	Kyrgyzstan Time
MVT	+ 05:00	Maldives Island Time
PKT	+ 05:00	Pakistan Time
TFT	+ 05:00	Kerguelen Time
TJT	+ 05:00	Tajikistan Time
TMT	+ 05:00	Turkmenistan Time

UZT	+ 05:00	Uzbekistan Time
YEKT	+ 05:00	Yekaterinburg Time
NPT	+ 05:45	Nepal Standard Time
ALMT	+ 06:00	Almaty Time
BDT	+ 06:00	Dacca Time
BTT	+ 06:00	Bhutan Time
DUSST	+ 06:00	Dushanbe Summer Time
KGST	+ 06:00	Kyrgyzstan Summer Time
LKT	+ 06:00	Lanka Time
MAWT	+ 06:00	Mawson, Antarctica
NOVT	+ 06:00	Novosibirsk Standard Time
OMST	+ 06:00	Omsk Time
UZST	+ 06:00	Uzbekistan Summer Time
YEKST	+ 06:00	Yekaterinburg Summer Time
CCT	+ 06:30	Indian Cocos (Island) Time
MMT	+ 06:30	Myanmar Time
ALMST	+ 07:00	Almaty Savings Time
CVT	+ 07:00	Christmas Island Time (Indian Ocean)
CXT	+ 07:00	Christmas Island Time (Indian Ocean)
DAVT	+ 07:00	Davis Time (Antarctica)
ICT	+ 07:00	Indochina Time
JAVT	+ 07:00	Java Time
KRAST	+ 07:00	Krasnoyarsk Summer Time
NOVST	+ 07:00	Novosibirsk Summer Time
OMSST	+ 07:00	Omsk Summer Time
WAST	+ 07:00	West Australian Std Time
JT	+ 07:30	Java Time
AWST	+ 08:00	Western Australia
BNT	+ 08:00	Brunei Darussalam Time
BORT	+ 08:00	Borneo Time (Indonesia)
CCT	+ 08:00	China Coast Time
HKT	+ 08:00	Hong Kong Time
IRKT	+ 08:00	Irkutsk Time
KRAT	+ 08:00	Krasnoyarsk Standard Time
MYT	+ 08:00	Malaysia Time
PHT	+ 08:00	Phillipine Time
ULAT	+ 08:00	Ulan Bator Time
WADT	+ 08:00	West Australian DST
WST	+ 08:00	West Australian Standard Time
MT	+ 08:30	Moluccas Time
AWSST	+ 09:00	Western Australia Time
IRKST	+ 09:00	Irkutsk Summer Time
JAYT	+ 09:00	Jayapura Time (Indonesia)
JST	+ 09:00	Japan Std Time, USSR Zone 8
KST	+ 09:00	Korea Standard Time

PWT	+ 09:00	Palau Time
ULAST	+ 09:00	Ulan Bator Summer Time
WDT	+ 09:00	West Australian DST
YAKT	+ 09:00	Yakutsk Time
ACST	+ 09:30	Central Australia
CAST	+ 09:30	Central Australian ST
SAST	+ 09:30	South Australian Std Time
SAT	+ 09:30	South Australian Std Time
AEST	+ 10:00	Australia Eastern Std Time
DDUT	+ 10:00	Dumont-d'Urville Time (Antarctica)
EAST	+ 10:00	East Australian Std Time
EST	+ 10:00	Australia Eastern Std Time
GST	+ 10:00	Guam Std Time, USSR Zone 9
KDT	+ 10:00	Korea Daylight Time
LIGT	+ 10:00	From Melbourne, Australia
MPT	+ 10:00	North Mariana Islands Time
PGT	+ 10:00	Papua New Guinea Time
TRUK	+ 10:00	Truk Time
VLAT	+ 10:00	Vladivostok Time
YAKST	+ 10:00	Yakutsk Summer Time
YAPT	+ 10:00	Yap Time (Micronesia)
ACSST	+ 10:30	Central Australia Time
CADT	+ 10:30	Central Australian DST
CST	+ 10:30	Australia Central Std Time
LHST	+ 10:30	Lord Howe Standard Time, Australia
SADT	+ 10:30	South Australian Daylight Time
AESST	+ 11:00	Eastern Australia
LHDT	+ 11:00	Lord Howe Daylight Time, Australia
MAGT	+ 11:00	Magadan Time
NCT	+ 11:00	New Caledonia Time
PONT	+ 11:00	Ponape Time (Micronesia)
VLAST	+ 11:00	Vladivostok Summer Time
VUT	+ 11:00	Vanuata Time
ANAT	+ 12:00	Anadyr Time (Russia)
CKT	+ 12:00	Cook Islands Time
GILT	+ 12:00	Gilbert Islands Time
IDLE	+ 12:00	International Date Line, East
KOST	+ 12:00	Kosrae Time
MAGST	+ 12:00	Magadan Summer Time
MHT	+ 12:00	Kwajalein Time
NZST	+ 12:00	New Zealand Standard Time
NZT	+ 12:00	New Zealand Time
PETT	+ 12:00	Petropavlovsk-Kamchatski Time
TVT	+ 12:00	Tuvalu Time
WAKT	+ 12:00	Wake Time

WFT	+ 12:00	Wallis and Futuna Time
CHAST	+ 12:45	Chatham Island Time
ANAST	+ 13:00	Anadyr Summer Time (Russia)
NZDT	+ 13:00	New Zealand Daylight Time
PETST	+ 13:00	Petropavlovsk-Kamchatski Summer Time
PHOT	+ 13:00	Phoenix Islands (Kiribati) Time
TOT	+ 13:00	Tonga Time
CHADT	+ 13:45	Chatham Island Daylight Time
LINT	+ 14:00	Line Islands Time (Kiribati)

I mentioned earlier in this section that an `INTERVAL` value represents a time span. I also mentioned that an `INTERVAL` stores some number of seconds. The syntax for an `INTERVAL` literal allows you to specify the number of seconds in a variety of units.

The format of an `INTERVAL` value is

```
quantity unit [quantity unit ...][AGO]
```

The `unit` component specifies a number of seconds, as shown in [Table 2.11](#). The `quantity` component acts as a multiplier (and may be fractional). If you have multiple `quantity unit` groups, they are all added together. The optional phrase `AGO` will cause the `INTERVAL` to be negative.

Table 2.11. INTERVAL Units

Description	Seconds	Unit Names
Microsecond ^[3]	.000001	us, usec, usecs, useconds, microsecon
Millisecond ^[3]	.001	-ms, msec, msecs, mseconds, millisecon
Second	1	s, sec, secs, second, seconds
Minute	60	m, min, mins, minute, minutes
Hour	3600	h, hr, hrs, hours
Day	86400	d, day, days
Week	604800	w, week, weeks
Month (30 days)	2592000	mon, mons, month, months
Year	31557600	y, yr, yrs, year, years
Decade	315576000	dec, decs, decade, decades
Century	3155760000	c, cent, century, centuries
Millennium	31557600000	mil, mils, millennia, millennium

^[3] Millisecond and microsecond can be used only in combination with another date/time component. For example, `CAST('1 SECOND 5000 MSEC' AS INTERVAL)` results in an interval of six seconds.

You can use the `EXTRACT(EPOCH FROM interval)` function to convert an `INTERVAL` into a number of seconds. A few sample `INTERVAL` values are shown in [Table 2.12](#). The `Display` column shows how PostgreSQL would format the Input Value for display. The `EPOCH` column shows the value that would be returned by extracting the `EPOCH` from the Input Value.

Table 2.12. Sample INTERVAL Values

Input Value	Display	EPOCH
-------------	---------	-------

.5 minutes	00:00:30	30
22 seconds 1 msec	00:00:22.00	22.001
22.001 seconds	00:00:22.00	22.001
10 centuries 2 decades	1020 years	32188752000
1 week 2 days 3.5 msec	9 days 00:00:00.00	777600.0035

Supported Operators

There are two types of operators that you can use with temporal values: arithmetic operators (addition and subtraction) and comparison operators.

You can add an `INT4`, a `TIME`, or a `TIMETZ` to a `DATE`. When you add an `INT4`, you are adding a number of days. Adding a `TIME` or `TIMETZ` to a `DATE` results in a `TIMESTAMP`. [Table 2.13](#) lists the valid data type and operator combinations for temporal data types. The last column in [Table 2.14](#) shows the data type of the resulting value.

Table 2.13. Arithmetic Date/Time Operators

Data Types	Valid Operators (θ)	Result Type
DATE θ DATE	–	INTEGER
DATE θ TIME	+	TIMESTAMP
DATE θ TIMETZ	+	TIMESTAMP WITH TIMEZONE
DATE θ INT4	+ –	DATE
TIME θ DATE	+	TIMESTAMP
TIME θ INTERVAL	+ –	TIME
TIMETZ θ DATE	+	TIMESTAMP WITH TIMEZONE
TIMETZ θ INTERVAL	+ –	TIMETZ
TIMESTAMP θ TIMESTAMP	–	INTERVAL
TIMESTAMP θ INTERVAL	+ –	TIMESTAMP WITH TIMEZONE
INTERVAL θ TIME	+	TIME WITHOUT TIMEZONE

Table 2.14. Arithmetic Date/Time Operator Examples

Example	Result
'23-JAN-2003'::DATE - '23-JAN-2002'::DATE	365
'23-JAN-2003'::DATE + '2:35 PM'::TIME	2003-01-23 14:35:00
'23-JAN-2003'::DATE + '2:35 PM GMT'::TIMETZ	2003-01-23 09:35:00-05
'23-JAN-2003'::DATE + 2::INT4	2003-01-25
'2:35 PM'::TIME + '23-JAN-2003'::DATE	2003-01-23 14:35:00
'2:35 PM'::TIME + '2 hours 5 minutes'::INTERVAL	16:40:00
'2:35 PM EST'::TIMETZ + '23-JAN-2003'::DATE	2003-01-23 14:35:00-05
'2:35 PM EST'::TIMETZ + '2 hours 5 minutes'::INTERVAL	16:40:00-05
'23-JAN-2003 2:35 PM EST'::TIMESTAMP - '23-JAN-2002 1:00 PM EST'::TIMESTAMP	365 days 01:35
'23-JAN-2003 2:35 PM EST'::TIMESTAMP + '3 days 2 hours 5 minutes'::INTERVAL	2003-01-26 16:40:00-05
'2 hours 5 minutes'::INTERVAL + '2:34 PM'::TIME	16:39:00

Table 2.14 shows how each of the arithmetic operators behave when applied to date/time values.

Using the temporal comparison operators, you can determine the relationship between two date/time values. For purposes of comparison, an earlier date/time value is considered to be less than a later date/time value.

Table 2.15 shows how you can combine the various temporal types with comparison operators.

Table 2.15. Date/Time Comparison Operators	
Data Types	Valid Operators (θ)
date θ date	< <= <> = >= >
time θ time	< <= <> = >= >
timetz θ timetz	< <= <> = >= >
timestamp θ timestamp	< <= <> = >= >

Boolean (Logical) Values

PostgreSQL supports a single Boolean (or logical) data type: `BOOLEAN` (`BOOLEAN` can be abbreviated as `BOOL`).

Size and Valid Values

A `BOOLEAN` can hold the values `TRUE`, `FALSE`, or `NULL`, and consumes a single byte of storage.

Syntax for Literal Values

Table 2.16 shows the alternate spellings for `BOOLEAN` literals.

Table 2.16. <code>BOOLEAN</code> Literal Syntax	
Common Name	Synonyms
<code>TRUE</code>	<code>true</code> , <code>'t'</code> , <code>'y'</code> , <code>'yes'</code> , <code>1</code>
<code>FALSE</code>	<code>false</code> , <code>'f'</code> , <code>'n'</code> , <code>'no'</code> , <code>0</code>

Supported Operators

The only operators supported for the `BOOLEAN` data type are the logical operators shown in Table 2.17:

Table 2.17. Logical Operators for <code>BOOLEAN</code>	
Data Types	Valid Operators (θ)
<code>BOOLEAN</code> θ <code>BOOLEAN</code>	<code>AND</code> <code>OR</code> <code>NOT</code>

I covered the `AND`, `OR`, and `NOT` operators in Chapter 1. For a complete definition of these operators, see Tables 1.3, 1.4, and 1.5.

Geometric Data Types

PostgreSQL supports six data types that represent two-dimensional geometric objects. The most basic geometric data type is the `POINT`—as you might expect, a `POINT` represents a point within a two-dimensional plane.

A `POINT` is composed of an x-coordinate and a y-coordinate—each coordinate is a `DOUBLE PRECISION` number.

The `LSEG` data type represents a two-dimensional line segment. When you create a `LSEG` value, you specify two points—the starting `POINT` and the ending `POINT`.

A `BOX` value is used to define a rectangle—the two points that define a box specify opposite corners.

A `PATH` is a collection of an arbitrary number of `POINT`s that are connected. A `PATH` can specify either a closed path or an open path. In a closed path, the beginning and ending points are considered to be connected, and in an open path, the first and last points are not connected. PostgreSQL provides two functions to force a `PATH` to be either open or closed: `POPEN()` and `PCLOSE()`. You can also specify whether a `PATH` is open or closed using special literal syntax (described later).

A `POLYGON` is similar to a closed `PATH`. The difference between the two types is in the supporting functions.

A center `POINT` and a (`DOUBLE PRECISION`) floating-point radius represent a `CIRCLE`.

Table 2.18 summarizes the geometric data types.

Table 2.18. Geometric Data Types

Type	Meaning	Defined By
<code>POINT</code>	2D point on a plane	x- and y-coordinates
<code>LSEG</code>	Line segment	Two points
<code>BOX</code>	Rectangle	Two points
<code>PATH</code>	Open or closed path	n points
<code>POLYGON</code>	Polygon	n points
<code>CIRCLE</code>	Circle	Center point and radius

Syntax for Literal Values

When you enter a value for geometric data type, keep in mind that you are working with a list of two-dimensional points (except in the case of a `CIRCLE`, where you are working with a `POINT` and a radius).

A single `POINT` can be entered in either of the following two forms:

```
'( x, y )'  
' x, y '
```

The `LSEG` and `BOX` types are constructed from a pair of `POINT`s. You can enter a pair of `POINT`s in any of the following formats:

```
'(( x1, y1 ), ( x2, y2 ))'  
'( x1, y1 ), ( x2, y2 )'  
'x1, y1, x2, y2'
```

The `PATH` and `POLYGON` types are constructed from a list of one or more `POINT`s. Any of the following forms is acceptable for a `PATH` or `POLYGON` literal:

```
'(( x1, y1 ), ..., ( xn, yn ))'  
'( x1, y1 ), ..., ( xn, yn )'  
'( x1, y1, ..., xn, yn )'  
'x1, y1, ..., xn, yn'
```

You can also use the syntax `'[(x1, y1), ..., (xn, yn)]'` to enter a `PATH` literal. A `PATH` entered in this form is considered to be an open `PATH`.

A `CIRCLE` is described by a central point and a floating point radius. You can enter a `CIRCLE` in any of the following forms:

```
'< ( x, y ), r >'  
'(( x, y ), r )'
```

```
'( x, y ), r'
'x, y, r'
```

Notice that the surrounding single quotes are required around all geometric literals—in other words, geometric literals are entered as string literals. If you want to create a geometric value from individual components, you will have to use a geometric conversion function. For example, if you want to create a `POINT` value from the results of some computation, you would use:

```
POINT( 4, 3*height )
```

The `POINT(DOUBLE PRECISION x, DOUBLE PRECISION y)` function creates a `POINT` value from two `DOUBLE PRECISION` values. There are similar functions that you can use to create any geometric type starting from individual components. [Table 2.19](#) lists the conversion functions for geometric types.

Table 2.19. Type Conversion Operators for the Geometric Data Types

Result Type	Meaning
<code>POINT</code>	<code>POINT(DOUBLE PRECISION x, DOUBLE PRECISION y)</code>
<code>LSEG</code>	<code>LSEG(POINT p1, POINT p2)</code>
<code>BOX</code>	<code>BOX(POINT p1, POINT p2)</code>
<code>PATH</code>	<code>PATH(POLYGON poly)</code>
<code>POLYGON</code>	<code>POLYGON(PATH path)</code>
	<code>POLYGON(BOX b)</code> yields a 12-point polygon
	<code>POLYGON(CIRCLE c)</code> yields a 12-point polygon
	<code>POLYGON(INTEGER n, CIRCLE c)</code> yields an n point polygon
<code>CIRCLE</code>	<code>CIRCLE(BOX b)</code> <code>CIRCLE(POINT radius, DOUBLE PRECISION point)</code>

Sizes and Valid Values

[Table 2.20](#) lists the size of each geometric data type.

Table 2.20. Geographic Data Type Storage Requirements

Type	Size (in bytes)
<code>POINT</code>	16 (2 * sizeof <code>DOUBLE PRECISION</code>)
<code>LSEG</code>	32 (2 * sizeof <code>POINT</code>)
<code>BOX</code>	32 (2 * sizeof <code>POINT</code>)
<code>PATH</code>	4+ (32* number of points) ^[4]
<code>POLYGON</code>	4+ (32* number of points) ^[4]
<code>CIRCLE</code>	24 (sizeof <code>POINT</code> + sizeof <code>DOUBLE PRECISION</code>)

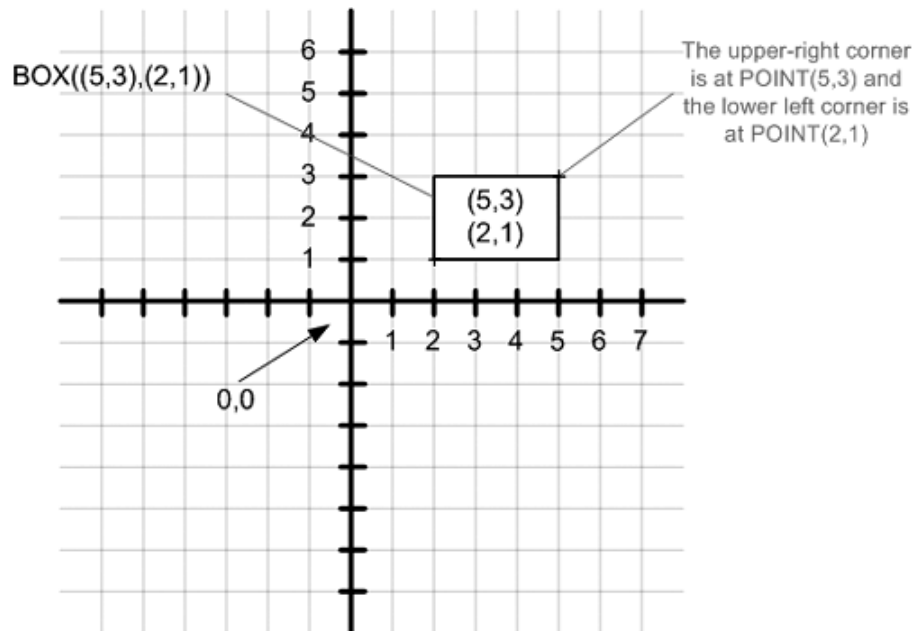
^[4] The size of a `PATH` or `POLYGON` is equal to 4 + (size of `LSEG` * number of segments).

Supported Operators

PostgreSQL features a large collection of operators that work with the geometric data types. I've divided the geometric operators into two broad categories (transformation and proximity) to make it a little easier to talk about them.

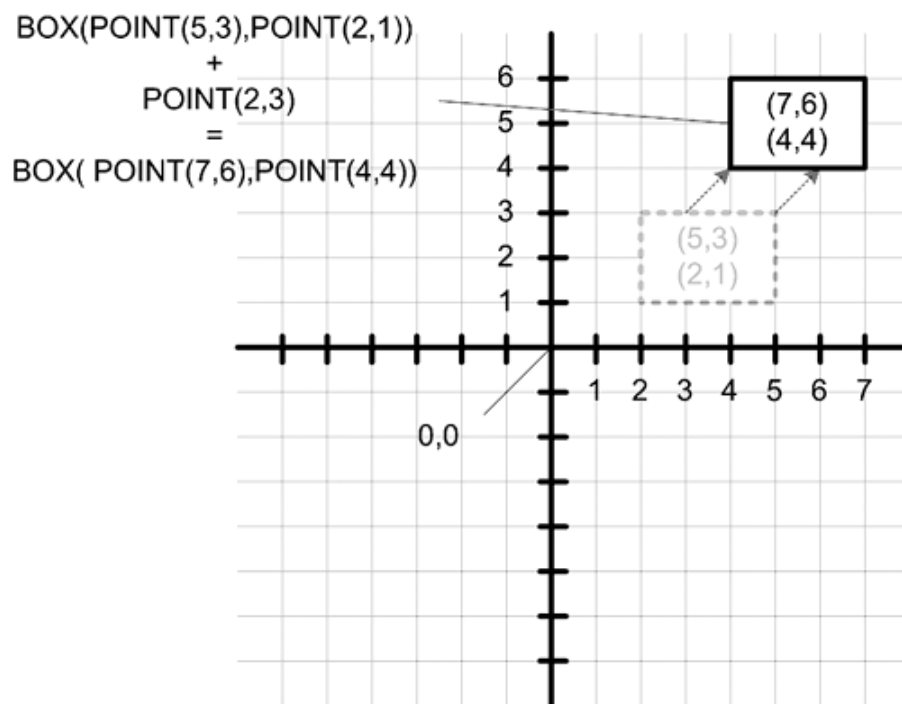
Using the transformation operators, you can translate, rotate, and scale geometric objects. The `+` and `-` operators translate a geometric object to a new location. Consider [Figure 2.1](#), which shows a `BOX` defined as `BOX(POINT(3,5), POINT(1,2))`.

Figure 2.1. `BOX(POINT(3,5), POINT(1,2))`.



If you use the + operator to add the POINT(2,1) to this BOX, you end up with the object shown in [Figure 2.2](#).

Figure 2.2. Geometric translation.



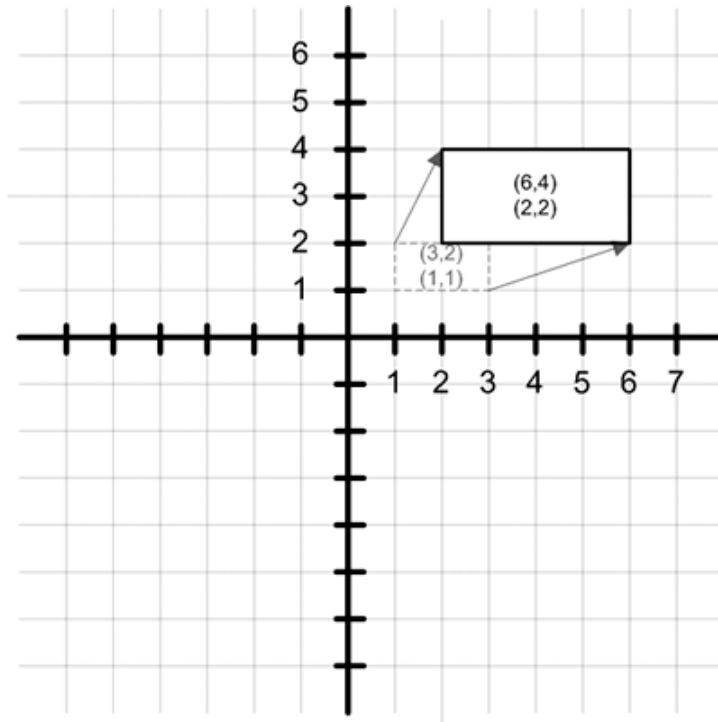
You can see that the x-coordinate of the POINT is added to each of the x-coordinates in the BOX, and the y-coordinate of the POINT is added to the y-coordinates in the BOX. The - operator works in a similar fashion: the x-coordinate of the POINT is subtracted from the x-coordinates of the BOX, and the y-coordinate of the POINT is subtracted from each y-coordinate in the BOX.

Using the + and - operators, you can move a POINT, BOX, PATH, or CIRCLE to a new location. In each case, the x-coordinate in the second operand (a POINT), is added or subtracted from each x-coordinate in the first operand, and the y-coordinate in the second operand is added or subtracted from each y-coordinate in the first operand.

The multiplication and division operators (* and /) are used to scale and rotate. The multiplication and division operators treat the operands as points in the complex plane. Let's look at some examples.

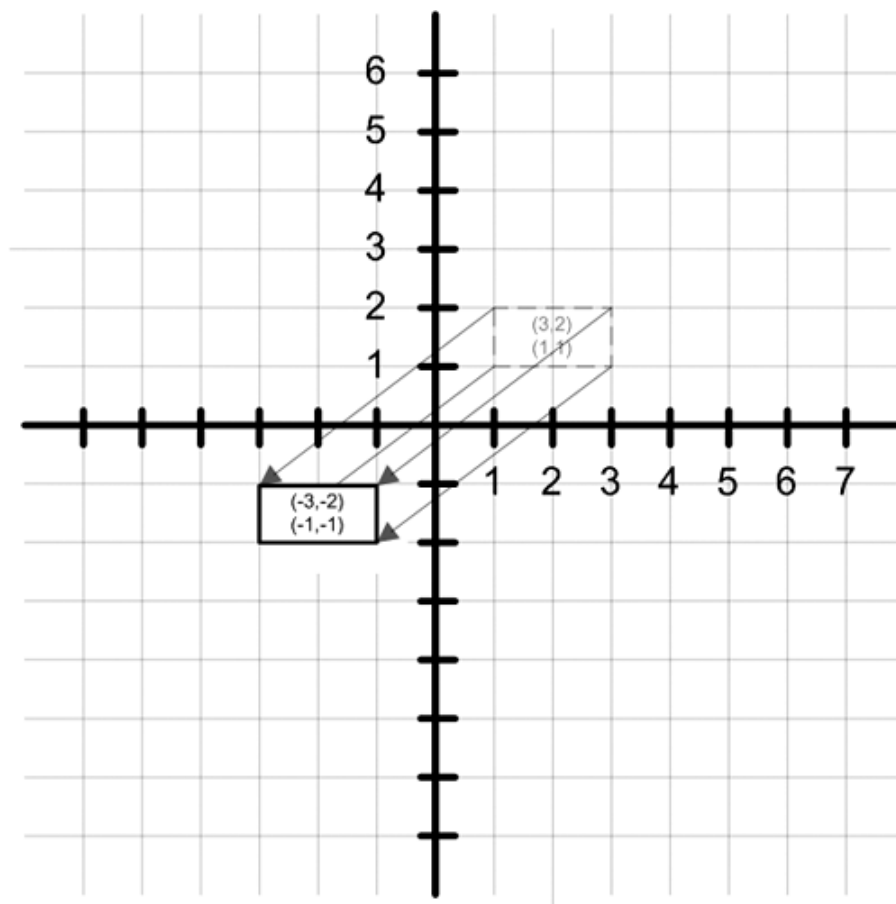
[Figure 2.3](#) shows the result of multiplying $\text{BOX}(\text{POINT}(3,2), \text{POINT}(1,1))$ by $\text{POINT}(2,0)$.

Figure 2.3. Point multiplication-scaling by a positive value.



You can see that each coordinate in the original box is multiplied by the x-coordinate of the point, resulting in `BOX (POINT (6,4), POINT (2,2))`. If you had multiplied the box by `POINT (0.5,0)`, you would have ended up with `BOX (POINT (1.5,1), POINT (0.5,0.5))`. So the effect of multiplying an object by `POINT (x,0)` is that each coordinate in the object moves away from the origin by a factor x . If x is negative, the coordinates move to the other side of the origin, as shown in [Figure 2.4](#).

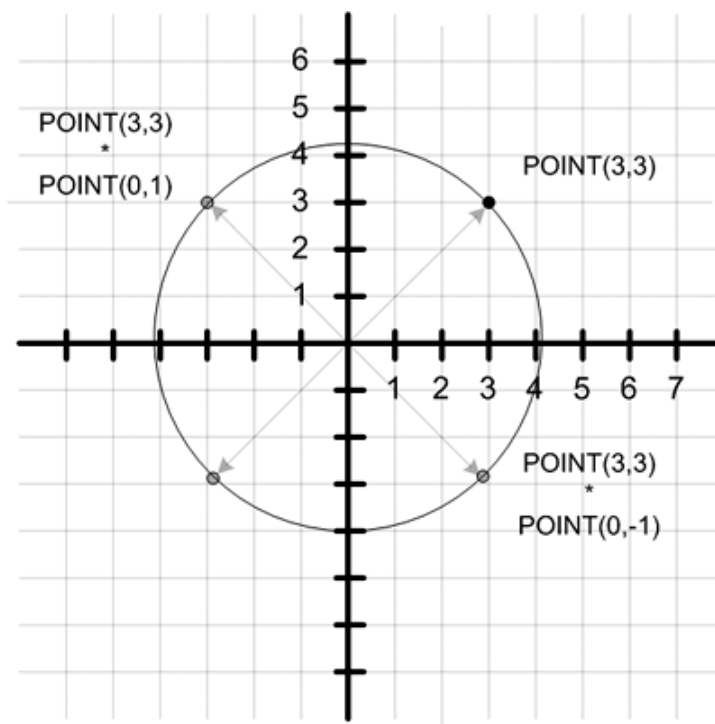
Figure 2.4. Point multiplication-scaling by a negative value.



The x-coordinate controls scaling. The y-coordinate controls rotation. When you multiply any given geometric object by `POINT (0,y)`, each point in the object is rotated around the origin. When y is equal to 1, each point is rotated counterclockwise by 90° about the origin. When y is equal to -1 , each point is rotated -90° about the origin (or 270°). When you rotate a point without

scaling, the length of the line segment drawn between the point and origin remains constant, as shown in Figure 2.5.

Figure 2.5. Point multiplication-rotation.



You can combine rotation and scaling into the same operation by specifying non-zero values for both the x- and y-coordinates. For more information on using complex numbers to represent geometric points, see <http://www.clarku.edu/~djoyce/complex>.

Table 2.21 shows the valid combinations for geometric types and geometric operators.

Table 2.21. Transformation Operators for the Geometric Types

Data Types	Valid Operators (θ)
POINT θ POINT	* + - /
BOX θ POINT	* + - /
PATH θ POINT	* + - /
CIRCLE θ POINT	* + - /

The proximity operators allow you to determine the spatial relationships between two geometric objects.

First, let's look at the three containment operators. The \sim operator evaluates to **TRUE** if the left operand contains the right operand. The $\@$ operator evaluates to **TRUE** if the left operand is contained within the right operand. The $\sim=$ returns **TRUE** if the left operand is the same as the right operand—two geographic objects are considered identical if the points that define the objects are identical (two circles are considered identical if the radii and center points are the same).

The next two operators are used to determine the distance between two geometric objects.

The $\#\#$ operator returns the closest point between two objects. You can use the $\#\#$ operator with the following operand types shown in Table 2.22.

Table 2.22. Closest-Point Operators

Operator	Description
LSEG a $\#\#$ BOX b	Returns the point in BOX b that is closest to LSEG a
LSEG a $\#\#$ LSEG b	Returns the point in LSEG b that is closest to LSEG a
POINT a $\#\#$ BOX b	Returns the point in BOX b that is closest to

	POINT <i>a</i>
POINT <i>a</i> ## LSEG <i>b</i>	Returns the point in LSEG <i>b</i> that is closest to POINT <i>a</i>

The distance (<->) operator returns (as a DOUBLE PRECISION number) the distance between two geometric objects. You can use the distance operator with the operand types in [Table 2.23](#).

Table 2.23. Distance Operators

Operator	Description (or Formula)
BOX <i>a</i> <-> BOX <i>b</i>	((@ BOX <i>a</i>) <-> (@@ BOX <i>b</i>))
CIRCLE <i>a</i> <-> CIRCLE <i>b</i>	((@ CIRCLE <i>a</i>) <-> (@@ CIRCLE <i>b</i>)) - (radius <i>a</i> + radius <i>b</i>)
CIRCLE <i>a</i> <-> POLYGON <i>b</i>	-0 if any point in POLYGON <i>b</i> is inside CIRCLE <i>a</i> otherwise, distance between center of CIRCLE <i>a</i> and closest point in POLYGON <i>b</i>
LSEG <i>a</i> <-> BOX <i>b</i>	(LSEG ## BOX) <-> (LSEG ## (LSEG ## BOX))
LSEG <i>a</i> <-> LSEG <i>b</i>	Distance between closest points (0 if LSEG <i>a</i> intersects LSEG <i>b</i>)
PATH <i>a</i> <-> PATH <i>b</i>	Distance between closest points
POINT <i>a</i> <-> BOX <i>b</i>	POINT <i>a</i> <-> (POINT <i>a</i> ## BOX <i>b</i>)
POINT <i>a</i> <-> CIRCLE <i>b</i>	POINT <i>a</i> <-> ((@ CIRCLE <i>b</i>) - CIRCLE <i>b</i> radius)
POINT <i>a</i> <-> LSEG <i>b</i>	POINT <i>a</i> <-> (POINT <i>a</i> ## LSEG <i>b</i>)
POINT <i>a</i> <-> PATH <i>b</i>	Distance between POINT <i>a</i> and closest points
POINT <i>a</i> <-> POINT <i>b</i>	$-\text{SQRT}((\text{POINT } a.x - \text{POINT } b.x)^2 + (\text{POINT } a.y - \text{POINT } b.y)^2)$

Next, you can determine the spatial relationships between two objects using the left-of (<<), right-of (>>), below (<^), and above (>^) operators.

There are three overlap operators. && evaluates to TRUE if the left operand overlaps the right operand. The &> operator evaluates to TRUE if the leftmost point in the first operand is left of the rightmost point in the second operand. The &< evaluates to TRUE if the rightmost point in the first operand is right of the leftmost point in the second operand.

The intersection operator (#) returns the intersecting points of two objects. You can find the intersection of two BOX es, or the intersection of two LSEG s. The intersection of two BOX es evaluates to a BOX. The intersection of two LSEG s evaluates to a single POINT.

Finally, the θ# operator evaluates to TRUE if the first operand intersects with or overlaps the second operand.

The final set of geometric operators determines the relationship between a line segment and an axis, or the relationship between two line segments.

The θ- operator evaluates to TRUE if the given line segment is horizontal (that is, parallel to the x-axis). The θ| operator evaluates to TRUE if the given line segment is vertical (that is, parallel to the y-axis). When you use the θ- and θ| operators with a line segment, they function as prefix unary operators. You can also use the θ- and θ| operators as infix binary operators (meaning that the operator appears between two values), in which case they operate as if you specified two points on a line segment.

The θ-| operator evaluates to TRUE if the two operands are perpendicular. The θ|| operator evaluates to TRUE if the two operands are parallel. The perpendicular and parallel operators can be used only with values of type LSEG.

The final geometric operator (@@) returns the center point of an LSEG, PATH, BOX, POLYGON, or CIRCLE.

[Tables 2.24](#) summarizes the proximity operators.

Table 2.24. Proximity Operators for the Geometric Types

Data Types	Valid Operators (θ)
POINT θ POINT	<-> << <^ >> >^ θ- θ @
POINT θ LSEG	## <-> @
POINT θ BOX	## <-> @
POINT θ PATH	<-> @
POINT θ POLYGON	@
POINT θ CIRCLE	<-> @
LSEG θ LSEG	# ## < <-> <= <> = > >= θ# θ- θ
LSEG θ BOX	## <-> θ# @
BOX θ POINT	* + - /
BOX θ BOX	# && &< &> < <-> << <= <^ = > >= >> >^ θ# @ ~ ~=
PATH θ POINT	* + - / ~
PATH θ PATH	+ < <-> <= = > >= θ#
POLYGON θ POINT	~
POLYGON θ POLYGON	&& &< &> <-> >> << @ ~ ~=
CIRCLE θ POINT	* + - / ~
CIRCLE θ POLYGON	<->
CIRCLE θ CIRCLE	&& &< &> > <-> << <= <> <^ = > >= >> >^ @ ~ ~=

Table 2.25 summarizes the names of the proximity operators for geometric types.

Table 2.25. Geometric Proximity Operator Names

Data Types	Valid Operators (θ)
#	Intersection or point count(for polygons)
##	Point of closest proximity
<->	Distance between
<<	Left of θ
>>	Right of θ
<^	Below θ
>^	Above θ
&&	Overlaps
&>	Overlaps to left
&<	Overlaps to right
θ#	Intersects or overlaps
@	Contained in
~	Contains
~=	Same as
θ-	Horizontal
θ	Vertical
θ-	Perpendicular
θ	Parallel
@@	Center

Object IDs (OID)

An `OID` is a 32-bit, positive whole number. Every row^[5] in a PostgreSQL database contains a unique identifier^[6]—the object ID (or `OID`). Normally, the `OID` column is hidden. You can see the `OID` for a row by including the `OID` column in the target list of a `SELECT` statement:

^[5] By default, all tables are created such that every row contains an `OID`. You can omit the object IDs using the `WITHOUT OIDS` clause of the `CREATE TABLE` command.

^[6] The PostgreSQL documentation warns that object IDs are currently unique within a database cluster; but in a future release, an `OID` may be unique only within a single table.

Code View: [Scroll](#) / Show All

```
movies=# SELECT OID, * FROM customers;
```

oid	customer_id	customer_name	phone	birth_date	balance
33876	3	Panky, Henry	555-1221	1968-01-21	0.00
33877	1	Jones, Henry	555-1212	1970-10-10	0.00
33878	4	Wonderland, Alice N.	555-1122	1969-03-05	3.00
33879	2	Rubin, William	555-2211	1972-07-10	15.00
33889	5	Funkmaster, Freddy	555-FUNK		0.00
33890	7	Gull, Jonathon LC	555-1111	1984-02-05	0.00
33891	8	Grumby, Jonas	555-2222	1984-02-21	0.00

You can create a column of type `OID` if you want to explicitly refer to another object (usually a row in another table). Think back to the `rentals` table that you developed in [Chapter 1](#). Each row in the `rentals` table contains a `tape_id`, a `customer_id`, and a rental date. The `rentals` table currently looks like this:

```
movies=# \d rentals
Table "public.rentals"
Attribute | Type      | Modifier
-----+-----+-----
tape_id   | character(8) | not null
rental_date | date         | not null
customer_id | integer      | not null

movies=# SELECT * FROM rentals;
tape_id | rental_date | customer_id
-----+-----+-----
AB-12345 | 2001-11-25 | 1
AB-67472 | 2001-11-25 | 3
OW-41221 | 2001-11-25 | 1
MC-68873 | 2001-11-20 | 3
KJ-03335 | 2001-11-26 | 8
(5 rows)
```

Each value in the `tape_id` column refers to a row in the `tapes` table. Each value in the `customer_id` column refers to a row in the `customers` table. Rather than storing the `tape_id` and `customer_id` in the `rentals` table, you could store `OID`s for the corresponding rows. The following `CREATE TABLE ... AS` command creates a new table, `rentals2`, that is equivalent to the original `rentals` table:

```
movies=# CREATE TABLE rentals2 AS
movies=# SELECT
movies=#   t.oid AS tape_oid, c.oid AS customer_oid, r.rental_date
movies=# FROM
movies=#   tapes t, customers c, rentals r
movies=# WHERE
movies=#   t.tape_id = r.tape_id
movies=#   AND
movies=#   c.id = r.customer_id;
```

This statement (conceptually) works as follows. First, you retrieve a row from the `rentals` table. Next, you use the `rentals.customer_id` column to retrieve the matching `customers` row and the `rentals.tape_id` column to retrieve the matching `tapes` row. Finally, you store the `OID` of the `customers` row and the `OID` of the `tapes` row (and the `rental_date`) in a new `rentals2` row.

Now, when you `SELECT` from the `rentals2` table, you will see the object IDs for the `customers` row and the `tapes` row:

```
movies=# SELECT * FROM rentals2;
tape_oid | customer_oid | rental_date
-----+-----+-----
38337 | 38333 | 2001-11-25
38338 | 38335 | 2001-11-25
38394 | 38393 | 2001-11-26
38339 | 38335 | 2001-11-20
38340 | 38333 | 2001-11-25
```

You can re-create the data in the original table by joining the corresponding `customers` and `tapes` records, based on their respective `OID` s:

```
movies=# SELECT t.tape_id, r.rental_date, c.id
movies-# FROM
movies-#   tapes t, rentals2 r, customers c
movies-# WHERE
movies-#   t.oid = r.tape_oid AND
movies-#   c.oid = r.customer_oid
movies-# ORDER BY t.tape_id;

tape_id | rental_date | id
-----+-----+---
AB-12345 | 2001-11-25 | 1
AB-67472 | 2001-11-25 | 3
KJ-03335 | 2001-11-26 | 8
MC-68873 | 2001-11-20 | 3
OW-41221 | 2001-11-25 | 1
(5 rows)
```

Here are a couple of warnings about using `OID` s in your own tables.

The first concern has to do with backups. The standard tool for performing a backup of a PostgreSQL database is `pg_dump`. By default, `pg_dump` will not archive `OID` s. This means that if you back up a table that contains an `OID` column (referring to another object) and then restore that table from the archive, the relationships between objects will be lost, unless you remembered to tell `pg_dump` to archive `OID` s. This happens because when you restore a row from the archive, it might be assigned a different `OID`.

The second thing you should consider when using `OID` s is that they offer no real performance advantages. If you are coming from an Oracle or Sybase environment, you might be thinking that an `OID` sounds an awful lot like a `ROWID`. It's true that an `OID` and a `ROWID` provide a unique identifier for a row, but that is where the similarity ends. In an Oracle environment, you can use a `ROWID` as the fastest possible way to get to a specific row. A `ROWID` encodes the location (on disk) of the row that it belongs to—when you retrieve a row by `ROWID`, you can bypass any index⁷ searches and go straight to the data. An `OID` is just a 32-bit number—you can create an index on the `OID` column, but you could also create an index on any other (unique) column to achieve the same results. In fact, the only time that it might make sense to use an `OID` to identify a row is when the primary key^[7] for a table is very long.

[7] Don't be too concerned if you aren't familiar with the concept of indexes or primary keys. I'll cover each of those topics a bit later.

Finally, I should point out that `OID` s can wrap. In an active database cluster, it's certainly possible that 4 billion objects can be created. That doesn't mean that all 4 billion objects have to exist at the same time, just that 4 billion `OID` s have been created since the cluster was created. When the `OID` generator wraps, you end up with duplicate values. This may sound a little far-fetched, but it does happen and it is not easy to recover from. There really is no good reason to use an `OID` as a primary key—use `SERIAL` (or `BIGSERIAL`) instead.

Syntax for Literal Values

The format in which you enter literal `OID` values is the same that you would use for unsigned `INTEGER` values. An `OID` literal is simply a sequence of decimal digits.

Size and Valid Values

As I mentioned earlier, an `OID` is an unsigned 32-bit (4-byte) integer. An `OID` column can hold values between 0 and 4294967295. The value 0 represents an invalid `OID`.

Supported Operators

You can compare two `OID` values, and you can compare an `OID` value against an `INTEGER` value. Table 2.26 shows which operators you can use with the `OID` data type.

Table 2.26. <code>OID</code> Operators	
Data Types	Valid Operators
<code>OID</code> 0 <code>OID</code>	<code><</code> <code><=</code> <code><></code> <code>=</code> <code>>=</code> <code>></code>
<code>OID</code> 0 <code>INT4</code>	<code><</code> <code><=</code> <code><></code> <code>=</code> <code>>=</code> <code>></code>
<code>INT4</code> 0 <code>OID</code>	<code><</code> <code><=</code> <code><></code> <code>=</code> <code>>=</code> <code>></code>

BLOBs

Most database systems provide a data type that can store raw data, and PostgreSQL is no exception. I use the term *raw data* to mean that the database doesn't understand the structure or meaning of a value. In contrast, PostgreSQL does understand the structure and meaning of other data types. For example, when you define an `INTEGER` column, PostgreSQL knows that the bytes of data that you place into that column are supposed to represent an integer value. PostgreSQL knows what an integer is—it can add integers, multiply them, convert them to and from string form, and so on. Raw data, on the other hand, is just a collection of bits—PostgreSQL can't infer any meaning in the data.

PostgreSQL offers the type `BYTEA` for storing raw data. A `BYTEA` column can theoretically hold values of any length, but it appears that the maximum length is 1GB.

The size of a `BYTEA` value is 4 bytes plus the actual number of bytes in the value.

Syntax for Literal Values

Entering a `BYTEA` value can be a little tricky. A `BYTEA` literal is entered as a string literal: It is just a string of characters enclosed within single quotes. Given that, how do you enter a `BYTEA` value that includes a single quote? If you look back to the discussion of string literal values (earlier in this chapter), you'll see that you can include special characters in a string value by escaping them. In particular, a single quote can be escaped in one of three ways:

- Double up the single quotes (`'This is a single quote'''`)
- Precede the single quote with a backslash (`'This is a single quote \'`)
- Include the octal value of the character instead (`'This is a single quote \047'`)

There are two other characters that you must escape when entering `BYTEA` literals. A byte whose value is zero (not the character `0`, but the null byte) must be escaped, and the backslash character must be escaped. You can escape any character using the `"\ddd"` form (where `ddd` is an octal number). You can escape any printable character using the `"\c"` form. So, if you want to store a `BYTEA` value that includes a zero byte, you could enter it like this:

```
'This is a zero byte \\000'
```

If you want to store a `BYTEA` value that includes a backslash, you can enter it in either of the following forms:

```
'This is a backslash \\\\'
'This is also a backslash \\134'
```

If you compare these rules to the rules for quoting string literals, you'll notice that `BYTEA` literals require twice as many backslash characters. This is a quirk of the design of the PostgreSQL parser. `BYTEA` literals are processed by two different parsers. The main PostgreSQL parser sees a `BYTEA` literal as a string literal (gobbling up the first set of backslash characters). Then, the `BYTEA` parser processes the result, gobbling up the second set of backslash characters.

So, if you have a `BYTEA` value such as `This is a backslash \`, you quote it as `'This is a backslash \\\\'`. After the string parser processes this string, it has been turned into `'This is a backslash \\'`. The `BYTEA` parser finally transforms this into `This is a backslash \`.

Supported Operators

PostgreSQL offers a single `BYTEA` operator: concatenation. You can append one `BYTEA` value to another `BYTEA` value using the concatenation (`||`) operator.

Note that you can't compare two `BYTEA` values, even for equality/inequality. You can, of course, convert a `BYTEA` value into another value using the `CAST()` operator, and that opens up other operators.

Large-Objects

The `BYTEA` data type is currently limited to storing values no larger than 1GB. If you need to store values larger than will fit into a `BYTEA` column, you can use large-objects. A large-object is a value stored outside of a table. For example, if you want to store a photograph with each row in your `tapes` table, you would add an `OID` column to hold a reference to the corresponding large-object:

```
movies=# ALTER TABLE tapes ADD COLUMN photo_id OID;
ALTER
```

Each value in the `photo_id` column refers to an entry in the `pg_largeobject` system table. PostgreSQL provides a function that will load an external file (such as a JPEG file) into the `pg_largeobject` table:

```
movies=# INSERT INTO tapes VALUES
movies=# (
movies(# 'AA-55892',
movies(# 'Casablanca',
movies(# '102 min',
movies(# lo_import('/tmp/casablanca.jpg' )
movies(# );
```

The `lo_import()` function loads the named file into `pg_largeobject` and returns an `OID` value that refers to the large-object. Now when you `SELECT` this row, you see the `OID`, not the actual bits that make up the photo:

```
movies=# SELECT * FROM tapes WHERE tape_id = 'AA-55892';
```

tape_id	title	duration	photo_id
AA-55892	Casablanca	01:42:00	510699

If you want to write the photo back into a file, you can use the `lo_export()` function:

```
movies=# SELECT lo_export( 510699, '/tmp/Casablanca.jpg' );
lo_export
-----
1
(1 row)
```

To see all large-objects in the current database, use `psql`'s `\lo_list` metacommand:

```
movies=# \lo_list
Large objects
ID | Description
-----+-----
510699 |
(1 row)
```

You can remove large-objects from your database using the `lo_unlink()` function:

```
movies=# SELECT lo_unlink( 510699 );
lo_unlink
-----
1
(1 row)
```

```
movies=# \lo_list
Large objects
ID | Description
----+-----
(0 rows)
```

How do you get to the actual bits behind the reference `OID`? You can't—at least not with `psql`. Large-object support must be built into the client application that you are using. `psql` is a text-oriented tool and has no way to display a photograph, so the best that you can do is to look at the raw data in the `pg_largeobject` table. A few client applications, such as the Conjectrix Workstation, do support large-objects and can interpret the raw data properly, in most cases.

Network Address Data Types

PostgreSQL supports three data types that are designed to hold network addresses, both IP^[8] (logical) and MAC^[9] (physical) addresses. I don't think there are many applications that require the storage of an IP or MAC address, so I won't spend too much time describing them. The PostgreSQL User's Guide contains all the details that you might need to know regarding network data types.

^[8] IP stands for Internet Protocol, the substrate of the Internet.

^[9] The acronym MAC stands for one or more of the following: Machine Address Code, Media Access Control, or Macaroni And Cheese.

MACADDR

The `MACADDR` type is designed to hold a MAC address. A MAC address is a hardware address, usually the address of an ethernet interface.

CIDR

The `CIDR` data type is designed to hold an IP network address. A `CIDR` value contains an IP network address and an optional netmask (the netmask determines the number of meaningful bits in the network address).

INET

An `INET` value can hold the IP address of a network or of a network host. An `INET` value contains a network address and an optional netmask. If the netmask is omitted, it is assumed that the address identifies a single host (in other words, there is no discernible network component in the address).

Note that an `INET` value can represent a network or a host, but a `CIDR` is designed to represent the address of a network.

Syntax for Literal Values

The syntax required for literal network values is shown in [Table 2.27](#).

Table 2.27. Literal Syntax for Network Types

Type	Syntax	Examples
INET	a.b.c.d[/e]	192.168.0.1_192.168.150.0/26_130.155.16.1/20
CIDR	a[.b[.c[.d]]]/[e]	192.168.0.0/16_192.168/16
MACADDR	xxxxxx:xxxxxx	0004E2:3695C0
	xxxxxx-xxxxxx	0004E2-3695C0
	xxxx.xxxx.xxxx	0004.E236.95C0
	xx-xx-xx-xx-xx-xx	00-04-E2-36-95-C0
	xx:xx:xx:xx:xx:xx	00:04:E2:36:95:C0

Starting with version 7.4, you can also store IPv6 (colon-separated) addresses in an `INET` or `CIDR` value.

An `INET` or `CIDR` value consumes either 12 bytes or 24 bytes of storage (depending on the number of bits in the address). A `MACADDR` value consumes 6 bytes of storage.

Supported Operators

PostgreSQL provides comparison operators that you can use to compare two `INET` values, two `CIDR` values, or two `MACADDR` values. The comparison operators work by first checking the common bits in the network components of the two addresses; then, if those are equal, the address with the greatest number of netmask bits is considered the largest value. If the number of bits in the netmask is equal (and the network components of the addresses are equal), then the entire address is compared. The net effect (pun intended) is that `192.168.0.22/24` is considered greater than `192.168.0.22/20`.

When you are working with two `INET` (or `CIDR`) values, you can also check for containership. [Table 2.28](#) describes the network address operators.

Table 2.28. Network Address Operators

Operator	Meaning
$\text{INET}_1 < \text{INET}_2$	True if operand ₁ is less than operand ₂
$\text{CIDR}_1 < \text{CIDR}_2$	
$\text{MACADDR}_1 < \text{MACADDR}_2$	
$\text{INET}_1 \leq \text{INET}_2$	True if operand ₁ is less than or equal to operand ₂
$\text{CIDR}_1 \leq \text{CIDR}_2$	
$\text{MACADDR}_1 \leq \text{MACADDR}_2$	
$\text{INET}_1 <> \text{INET}_2$	True if operand ₁ is not equal to operand ₂
$\text{CIDR}_1 <> \text{CIDR}_2$	
$\text{MACADDR}_1 <> \text{MACADDR}_2$	
$\text{INET}_1 = \text{INET}_2$	True if operand ₁ is equal to operand ₂
$\text{CIDR}_1 = \text{CIDR}_2$	
$\text{MACADDR}_1 = \text{MACADDR}_2$	
$\text{INET}_1 \geq \text{INET}_2$	True if operand ₁ is greater than or equal to operand ₂
$\text{CIDR}_1 \geq \text{CIDR}_2$	
$\text{MACADDR}_1 \geq \text{MACADDR}_2$	
$\text{INET}_1 > \text{INET}_2$	True if operand ₁ is greater than operand ₂
$\text{CIDR}_1 > \text{CIDR}_2$	
$\text{MACADDR}_1 > \text{MACADDR}_2$	
$\text{INET}_1 \ll \text{INET}_2 \text{ CIDR}_1 \ll \text{CIDR}_2$	True if operand ₁ is contained within operand ₂
$\text{INET}_1 \ll= \text{INET}_2 \text{ CIDR}_1 \ll= \text{CIDR}_2$	True if operand ₁ is contained within operand ₂ or if operand ₁ is equal to operand ₂
$\text{INET}_1 \gg \text{INET}_2 \text{ CIDR}_1 \gg \text{CIDR}_2$	True if operand ₁ contains operand ₂
$\text{INET}_1 \gg= \text{INET}_2 \text{ CIDR}_1 \gg= \text{CIDR}_2$	True if operand ₁ contains operand ₂ or if operand ₁ is equal to operand ₂

Sequences

One problem that you will most likely encounter in your database life is the need to generate unique identifiers. We've already seen one example of this in the `customers` table—the `customer_id` column is nothing more than a unique identifier. Sometimes, an entity that you want to store in your database will have a naturally unique identifier. For example, if you are designing a database to track employee information (in the United States), a Social Security number might make a good identifier. Of course, if you employ people who are not U.S. citizens, the Social Security number scheme will fail. If you are tracking information about automobiles, you might be tempted to use the license plate number as a unique identifier. That would work fine until you needed to track autos in more than one state. The VIN (or Vehicle Identification Number) is a naturally unique identifier.

Quite often, you will need to store information about an entity that has no naturally unique ID. In those cases, you are likely to simply assign a unique number to each entity. After you have decided to create a uniquifier^[10], the next problem is coming up with a sequence of unique numbers.

[10] I'm not sure that "uniquifier" is a real word, but I've used it for quite some time and it sure is a lot easier to say than "disambiguator."

PostgreSQL offers help in the form of a `SEQUENCE`. A `SEQUENCE` is an object that automatically generates sequence numbers. You can create as many `SEQUENCE` objects as you like: Each `SEQUENCE` has a unique name.

Let's create a new `SEQUENCE` that you can use to generate unique identifiers for rows in your `customers` table. You already have a few customers, so start the sequence numbers at 10:

```
movies=# CREATE SEQUENCE customer_id_seq START 10;
CREATE
```

The `\ds` command (in `psql`) shows you a list of the `SEQUENCE` objects in your database:

```
movies=# \ds
          List of relations
   Name          |  Type   | Owner
-----+-----+-----
 customer_id_seq | sequence | korry
(1 row)
```

Now, let's try using this `SEQUENCE`. PostgreSQL provides a number of functions that you can call to make use of a `SEQUENCE`. The one that you are most interested in at the moment is the `nextval()` function. When you call the `nextval()` function, you provide (in the form of a string) the name of the `SEQUENCE` as the only argument.

For example, when you `INSERT` a new row in the `customers` table, you want PostgreSQL to automatically assign a unique `customer_id`:

```
movies=# INSERT INTO
movies-#   customers( customer_id, customer_name )
movies-#   VALUES
movies-#   (
movies-#       nextval( 'customer_id_seq' ), 'John Gomez'
movies-#   );

movies=# SELECT * FROM customers WHERE customer_name = 'John Gomez';
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
          10 | John Gomez    |      |            |
(1 row)
```

You can see that the `SEQUENCE` (`customer_id_seq`) generated a new `customer_id`, starting with the value that you requested. You can use the `currval()` function to find the value that was just generated by your server process:

```
movies=# SELECT currval( 'customer_id_seq' );
 currval
-----
        10
```

The complete syntax for the `CREATE SEQUENCE` command is

```
CREATE SEQUENCE name
  [ INCREMENT increment ]
  [ MINVALUE min ]
  [ MAXVALUE max ]
  [ START start_value ]
  [ CACHE cache_count ]
```

[CYCLE]

Notice that the only required item is the name.

The `INCREMENT` attribute determines the amount added to generate a new sequence number. This value can be positive or negative, but not zero. Positive values cause the sequence numbers to increase in value as they are generated (that is, 0, 1, 2, and so on). Negative values cause the sequence numbers to decrease in value (that is, 3, 2, 1, 0, and so on).

The `MINVALUE` and `MAXVALUE` attributes control the minimum and maximum values (respectively) for the `SEQUENCE`.

What happens when a `SEQUENCE` has reached the end of its valid range? You get to decide: If you include the `CYCLE` attribute, the `SEQUENCE` will wrap around. For example, if you create a cyclical `SEQUENCE` with `MINVALUE 0` and `MAXVALUE 3`, you will retrieve the following sequence numbers: 0, 1, 2, 3, 0, 1, 2, 3, If you don't include the `CYCLE` attribute, you will see: 0, 1, 2, 3, error: reached `MAXVALUE`.

The `START` attribute determines the first sequence number generated by a `SEQUENCE`. The value for the `START` attribute must be within the `MINVALUE` and `MAXVALUE` range.

The default values for most of the `SEQUENCE` attributes depend on whether the `INCREMENT` is positive or negative. The default value for the `INCREMENT` attribute is 1. If you specify a negative `INCREMENT`, the `MINVALUE` defaults to -2147483647, and `MAXVALUE` defaults to -1. If you specify a positive `INCREMENT`, `MINVALUE` defaults to 1, and `MAXVALUE` defaults to 2147483647. The default value for the `START` attribute is also dependent on the sign of the `INCREMENT`. A positive `INCREMENT` defaults the `START` value to the `MINVALUE` attribute. A negative `INCREMENT` defaults the `START` value to the `MAXVALUE` attribute.

Remember, these are the defaults—you can choose any meaningful combination of values that you like (within the valid range of a `BIGINT`).

The default `SEQUENCE` attributes are summarized in Table 2.29.

Table 2.29. Sequence Attributes

Attribute Name	Default Value
<code>INCREMENT</code>	1
<code>MINVALUE</code>	<code>-INCREMENT > 0 ? 1 - INCREMENT < 0 ? -2147483647</code>
<code>MAXVALUE</code>	<code>INCREMENT > 0 ? 2147483647 INCREMENT < 0 ? -1</code>
<code>START</code>	<code>INCREMENT > 0 ? MINVALUE INCREMENT < 0 ? MAXVALUE</code>
<code>CACHE</code>	1
<code>CYCLE</code>	False

The `CACHE` attribute is a performance-tuning parameter; it determines how many sequence numbers are generated and held in memory. In most cases, you can simply use the default value (1). If you suspect that sequence number generation is a bottleneck in your application, you might consider increasing the `CACHE` attribute, but be sure to read the warning in the PostgreSQL documentation (see the `CREATE SEQUENCE` section).

You can view the attributes of a `SEQUENCE` by treating it as a table and selecting from it^[11]:

^[11] There are four other columns in a `SEQUENCE`, but they hold bookkeeping information required to properly maintain the `SEQUENCE`.

```
movies=# SELECT
movies=#   increment_by, max_value, min_value, cache_value, is_cycled
movies=# FROM
movies=#   customer_id_seq;
```

```
increment_by | max_value | min_value | cache_value | is_cycled
-----+-----+-----+-----+-----
1 | 3 | 0 | 1 | f
```

PostgreSQL provides three functions that work with `SEQUENCE` s. I described the `nextval()` and `currval()` functions earlier; `nextval()` generates (and returns) a new value from a `SEQUENCE`, and `currval()` retrieves the most recently generated value. You can reset a `SEQUENCE` to any value between `MINVALUE` and `MAXVALUE` by calling the `setval()` function. For example:

```
movies=# SELECT nextval( 'customer_id_seq' );
ERROR:  customer_id_seq.nextval: reached MAXVALUE (3)
```

```
movies=# SELECT setval( 'customer_id_seq', 0 );
setval
```

```

-----
      0
(1 row)

movies=# SELECT nextval( 'customer_id_seq' );
      nextval
-----
      1

```

Now that you know how `SEQUENCE`s work in PostgreSQL, let's revisit the `SERIAL` data type. I mentioned earlier in this chapter that a `SERIAL` is really implemented as a `SEQUENCE` (see the "[SERIAL, BIGSERIAL, and SEQUENCES](#)" sidebar). Remember that a `SERIAL` provides an automatically increasing (or decreasing) unique identifier. That sounds just like a `SEQUENCE`, so what's the difference? A `SEQUENCE` is a standalone object, whereas `SERIAL` is a data type that you can assign to a column.

Let's create a new table that contains a `SERIAL` column:

```

movies=# CREATE TABLE serial_test ( pkey SERIAL, payload INTEGER );

NOTICE: CREATE TABLE will create implicit
sequence 'serial_test_pkey_seq' for
SERIAL column 'serial_test.pkey'
NOTICE: CREATE TABLE/UNIQUE will create implicit
index 'serial_test_pkey_key' for table 'serial_test'
CREATE

```

The `CREATE TABLE` command is normally silent. When you create a table with a `SERIAL` column, PostgreSQL does a little extra work on your behalf. First, PostgreSQL creates a `SEQUENCE` for you. The name of the `SEQUENCE` is based on the name of the table and the name of the column. In this case, the `SEQUENCE` is named `serial_test_pkey_seq`. Next, PostgreSQL creates a unique index. We haven't really talked about indexes yet: for now, know that a unique index on the `pkey` column ensures that you have no duplicate values in that column. PostgreSQL performs one more nicety for you when you create a `SERIAL` column. The `\d` command (in `psql`) shows you this last step:

```

movies=# \d serial_test
                                Table "public.serial_test"
Attribute | Type      | Modifier
-----+-----+-----
pkey      | integer   | not null default nextval('serial_test_pkey_seq')
payload   | integer   |
Index: serial_test_pkey_key

```

PostgreSQL has created a default value for the `pkey` column. A column's default value is used whenever you insert a row but omit a value for that column. For example, if you execute the command `INSERT INTO serial_test(payload) VALUES(24307);`, you have not provided an explicit value for the `pkey` column. In this case, PostgreSQL evaluates the default value for `pkey` and inserts the resulting value. Because the default value for `pkey` is a call to the `nextval()` function, each new row is assigned a new (unique) sequence number.

Arrays

One of the unique features of PostgreSQL is the fact that you can define a column to be an array. Most commercial database systems require that a single column within a given row can hold no more than one value. With PostgreSQL, you aren't bound by that rule—you can create columns that store multiple values (of the same data type).

The `customers` table defined in [Chapter 1](#) contained a single `balance` column. What change would you have to make to the database if you wanted to store a month-by-month balance for each customer, going back at most 12 months? One alternative would be to create a separate table to store monthly balances. The primary key of the `cust_balance` might be composed of the `customer_id` and the month number (either 0-11 or 1-12, whichever you found more convenient)^[12]. This would certainly work, but in PostgreSQL, it's not the only choice.

[12] The relationship between the `customers` table and the `cust_balance` is called a parent/child relationship. In this case, the `customers` table is the parent and `cust_balance` is the child. The primary key of a child table is composed of the parent key plus a unifier (that is, a value, such as the month number, that provides a unique identifier within a group of related children).

You know that there are never more than 12 months in a year and that there are never fewer than 12 months in a year. Parent/child relationships are perfect when the parent has a variable number of children, but they aren't always the most convenient choice when the number of child records is fixed.

Instead, you could store all 12 monthly balance values inside the `customers` table. Here is how you might create the `customers` table using an array to store the monthly balances:

```
CREATE TABLE customers (
    customer_id    INTEGER UNIQUE,
    customer_name  VARCHAR(50),
    phone          CHAR(8),
    birth_date     DATE,
    balance        DECIMAL(7,2),
    monthly_balances DECIMAL(7,2) [12]
);
```

Notice that I have added a new column named `monthly_balances`—this is an array of 12 `DECIMAL` values. I'll show you how to put values into an array in a moment.

You can define an array of any data type: the built-in types, user-defined types, even other arrays. When you create an array of arrays, you are actually creating a multidimensional array. For example, if we wanted to store month-by-month balances for the three previous years, I could have created the `monthly_balances` field as

```
monthly_balances DECIMAL(7,2) [3][12]
```

This would give you three arrays of 12-element arrays.

There is no limit to the number of members in an array. There is also no limit to the number of dimensions in a multidimensional array.

Now, let's talk about inserting and updating array values. When you want to insert a new row into the `customers` table, you provide values for each member in the `monthly_balances` array as follows:

```
INSERT INTO customers
(
    customer_id, customer_name, phone, birth_date, balance, monthly_balances
)
VALUES
(
    8,
    'Wink Wankel',
    '555-1000',
    '1988-12-25',
    0.00,
    '{1,2,3,4,5,6,7,8,9,10,11,12}'
);
```

To `INSERT` values into an array, you enclose all the array elements in single quotes and braces (`{}`) and separate multiple elements with a comma. Starting with PostgreSQL version 7.4, you can use an alternate form to write an array value. To express the same array value in constructor syntax form, you would write:

```
ARRAY[1,2,3,4,5,6,7,8,9,10,11,12]
```

There are two advantages to the new array constructor syntax. First, the meaning is a bit more obvious when you're looking at a piece of unfamiliar SQL code. Second, if you write an array value expressed '`{ element1, element2, ... }`' form, you have to

double up the quotes if the array contains string values. In array constructor form, you don't have to do that. The following array values are equivalent:

```
{ 'Panky, Henry', 'Rubin, William', 'Grumby, Jonas' }'  
ARRAY[ 'Panky, Henry', 'Rubin, William', 'Grumby, Jonas' ]
```

Inserting values into a multidimensional array is treated as if you were inserting an array of arrays. For example, if you had a table defined as

```
CREATE TABLE arr  
(  
    pkey serial,  
    val int[2][3]  
);
```

you would INSERT a row as

```
INSERT INTO arr( val ) VALUES( '{ {1,2,3}, {4,5,6} }' );
```

Or, to write the same array value in constructor form:

```
INSERT INTO arr( val ) VALUES( ARRAY[ [1,2,3], [4,5,6] ] );
```

Looking back at the `customers` table now; if you `SELECT` the row that you `INSERT` ed, you'll see:

```
movies=# \x  
Expanded display is on  
  
movies=# SELECT  
movies-#     customer_name, monthly_balances  
movies-# FROM customers  
movies-# WHERE customer_id = 8;  
-[ RECORD 1 ]-----  
  
customer_name      | Wink Wankel  
  
monthly_balances   | {1.00,2,3,4,5,6,7,8,9,10,11,12.00}
```

To make this example a little more readable in book form, I have used `psql`'s `\x` command to rearrange the display format here. I have also edited out some of the trailing zeroes in the `monthly_balances` column.

You can retrieve specific elements within an array:

```
movies=# SELECT  
movies-#     customer_name, monthly_balances[3]  
movies-# FROM customers  
movies-# WHERE customer_id = 8;  
customer_name | monthly_balances  
-----  
Wink Wankel   | 3.00  
(1 row)
```

Or you can ask for a range^[13] of array elements:

^[13] The PostgreSQL documentation refers to a contiguous range of array elements as a slice.

```
movies=# SELECT  
movies-#     customer_name, monthly_balances[1:3]  
movies-# FROM customers  
movies-# WHERE customer_id = 8;  
customer_name | monthly_balances  
-----  
Wink Wankel   | {"1.00","2.00","3.00"}  
(1 row)
```

The index for an array starts at 1 by default. I'll show you how to change the range of an index in a moment.

You can use an array element in any situation where you can use a value of the same data type. For example, you can use an array element in a `WHERE` clause:

```
movies=# SELECT
```

```

movies=#      customer_name, monthly_balances[1:3]
movies=# FROM customers
movies=# WHERE monthly_balances[1] > 0;
  customer_name |      monthly_balances
-----+-----
Wink Wankel    | {"1.00","2.00","3.00"}
(1 row)

```

There are three ways to UPDATE an array. If you want to UPDATE all elements in an array, simply SET the array to a new value:

```

movies=# UPDATE customers SET
movies=#   monthly_balances = '{12,11,10,9,8,7,6,5,4,3,1}'
movies=# WHERE customer_id = 8;

```

If you want to UPDATE a single array element, simply identify the element:

```

movies=# UPDATE customers SET monthly_balances[1] = 22;

```

Finally, you can UPDATE a contiguous range of elements:

```

movies=# UPDATE customers SET monthly_balances[1:3] = '{11,22,33}';

```

Now, there are a few odd things you should know about arrays in PostgreSQL.

First, the array bounds that you specify when you create a column are optional. I don't just mean that you can omit an array bound when you create a column (although you can), I mean that PostgreSQL won't enforce any limits that you try to impose. For example, you created the `monthly_balances` column as a 12-element array. PostgreSQL happily lets you put a value into element 13, 14, or 268. The `array_dims()` function tells the upper and lower bounds of an array value:

```

movies=# SELECT array_dims( monthly_balances ) FROM customers
movies=#   WHERE
movies=#     customer_id = 8;

array_dims
-----
[1:12]

```

You can increase the size of an array by updating values adjacent to those that already exist^[14]. For example, the `monthly_balances` column for customer 8 (Wink Wankel) contains 12 elements, numbered 1 through 12. You can add new elements at either end of the range (array subscripts can be negative):

^[14] The PostgreSQL documentation warns that you can't expand a multidimensional array.

Code View: [Scroll](#) / Show All

```

movies=# UPDATE customers SET
movies=#   monthly_balances[13] = 13
movies=#   WHERE
movies=#     customer_id = 8;
UPDATE 1

movies=# SELECT array_dims( monthly_balances ) FROM customers
movies=#   WHERE
movies=#     customer_id = 8;
  array_dims
-----
[1:13]

movies=# UPDATE customers SET
movies=#   monthly_balances[-1:0] = '{ -1, 0 }'
movies=#   WHERE
movies=#     customer_id = 8;
UPDATE 1

movies=# SELECT array_dims( monthly_balances ) FROM customers
movies=#   WHERE
movies=#     customer_id = 8;
  array_dims
-----
[-1:13]

```

Note that you can expand an array only by updating elements that are directly adjacent to the existing elements. For example, customer number 8 now contains elements -1:13. We can't add an element 15 without first adding element 14:

```
movies=# UPDATE customers SET
movies=#   monthly_balances[15] = 15
movies=# WHERE
movies=#   customer_id = 8;
ERROR:  Invalid array subscripts
```

Next, the syntax for inserting or updating array values is a bit misleading. Let's say that you want to insert a new row in your customers table, but you only want to provide a balance for month number 3:

```
movies=# INSERT INTO customers
movies=# ( customer_id, customer_name, monthly_balances[3] )
movies=# VALUES
movies=# ( 9, 'Samuel Boney', '{300}' );
```

This appears to work, but there is danger lurking here. Let's go back and retrieve the data that you just inserted:

```
movies=# SELECT customer_name, monthly_balances[3]
movies=#   FROM customers
movies=#   WHERE
movies=#     customer_id = 9;
 customer_name | monthly_balances
-----+-----
 Samuel Boney |
```

Where'd the data go? If you SELECT all array elements, the data is still there:

```
movies=# SELECT customer_name, monthly_balances
movies=#   FROM customers
movies=#   WHERE
movies=#     customer_id = 9;
 customer_name | monthly_balances
-----+-----
 Samuel Boney | {"300"}
```

The array_dims() function gives you a pretty good hint:

```
movies=# SELECT array_dims( monthly_balances ) FROM customers
movies=#   WHERE
movies=#     customer_id = 9;

array_dims
-----
 [1:1]
```

According to array_dims(), the high and low subscript values are both 1. You explicitly INSERT ed the value 300 into array element 3, but PostgreSQL (silently) decided to place it into element 1 anyway. This seems a bit mysterious to me, but that's how it works.

The final oddity concerns how PostgreSQL handles NULL values and arrays. An array can be NULL, but an individual element cannot—you can't have an array in which some elements are NULL and others are not. Furthermore, PostgreSQL silently ignores an attempt to UPDATE an array member to NULL :

```
movies=# SELECT customer_name, monthly_balances
movies=#   FROM
movies=#     customers
movies=#   WHERE
movies=#     customer_id = 8;
-[ RECORD 1 ]-----+-----
id              | 8
customer_name   | Wink Wankel
phone           | 555-1000
birth_date      | 1988-12-25
monthly_balances | {1.00,2,3,4,5,6,7,8,9,10,11,12.00}

movies=# UPDATE customers SET
movies=#   monthly_balances[1] = NULL
movies=# WHERE
movies=#   customer_id = 8;
UPDATE 1
```

You won't get any error messages when you try to change an array element to NULL, but a SELECT statement will show that the UPDATE had no effect:

```

movies=# SELECT customer_name, monthly_balances
movies=# FROM
movies=# customers
movies=# WHERE
movies=# customer_id = 8;
-[ RECORD 1 ]-----+-----
id              | 8
customer_name   | Wink Wankel
phone           | 555-1000
birth_date      | 1988-12-25
monthly_balances | {1.00,2,3,4,5,6,7,8,9,10,11,12.00}

```

If you keep these three oddities in mind, arrays can be very useful. Remember, though, that an array is not a substitute for a child table. You should use an array only when the number of elements is fixed by some real-world constraint (12 months per year, 7 days per week, and so on).

Column Constraints

When you create a PostgreSQL table, you can define column constraints^[15]. A column constraint is a rule that must be satisfied whenever you insert or update a value in that column.

^[15] You can also define table constraints. A table constraint applies to the table as a whole, not just a single column. We'll discuss table constraints in [Chapter 3](#), "PostgreSQL SQL Syntax and Use."

It's very important to understand that when you define a column constraint, PostgreSQL won't ever let your table get into a state in which the constraints are not met. If you try to `INSERT` a value that violates a constraint, the insertion will fail. If you try to `UPDATE` a value in such a way that it would violate a constraint, the modification will be rejected.

You can also define constraints that establish relationships between two tables. For example, each row in the `rentals` table contains a `tape_id` (corresponding to a row in the `tapes` table). You could define a constraint to tell PostgreSQL that the `rentals.tape_id` column `REFERENCES` the `tapes.tape_id` column. I'll discuss the implications of a `REFERENCES` constraint in a moment.

Needless to say, column constraints are a very powerful feature.

NULL/NOT NULL

Let's start with the most basic column constraints: `NULL` and `NOT NULL`. You've already seen some examples of the `NOT NULL` constraint (in [Chapter 1](#)):

```
CREATE TABLE customers (  
    customer_id    INTEGER UNIQUE NOT NULL,  
    name           VARCHAR(50)     NOT NULL,  
    phone          CHAR(8),  
    birth_date     DATE,  
    balance        DECIMAL(7,2)  
);
```

I have specified that the `customer_id` and `name` columns are `NOT NULL`. The meaning of a `NOT NULL` constraint is pretty clear: The column is not allowed to contain a `NULL` value^[16]. If you try to `INSERT` a `NULL` value into the `customer_id` or `name` columns, you will receive an error:

^[16] A column that has been defined to be `NOT NULL` is also known as a mandatory column. A column that can accept `NULL` values is said to be optional.

```
INSERT INTO customers VALUES  
(  
    11,  
    NULL,  
    '555-1984',  
    '10-MAY-1980',  
    0  
);  
  
ERROR:  ExecAppend: Fail to add null value in not null  
         attribute customer_name
```

You'll also get an error if you try to `UPDATE` either column in such a way that the result would be `NULL`:

```
UPDATE customers SET customer_name = NULL WHERE customer_id = 1;  
  
ERROR:  ExecReplace: Fail to add null value in not null  
         attribute customer_name
```

The opposite of `NOT NULL` is `NULL`. You can explicitly define a `NULL` constraint, but it really doesn't function as a constraint. A `NULL` constraint does not force a column to contain only `NULL` values (that would be pretty pointless). Instead, a `NULL` constraint simply tells PostgreSQL that `NULL` values are allowed in a particular column. If you don't specify that a column is mandatory, it is considered optional.

UNIQUE

The `UNIQUE` constraint ensures that a column will contain unique values; that is, there will be no duplicate values in the column. If you look back to the previous section, you'll see that you specified that the `customer_id` column should be `UNIQUE`. If you try to `INSERT` a duplicate value into a `UNIQUE` column, you will receive an error message:

```
movies=# SELECT * FROM customers;
```

customer_id	customer_name	phone	birth_date	balance
1	Jones, Henry	555-1212	1970-10-10	0.00
2	Rubin, William	555-2211	1972-07-10	15.00
3	Panky, Henry	555-1221	1968-01-21	0.00
4	Wonderland, Alice N.	555-1122	1969-03-05	3.00

```
movies=# INSERT INTO customers VALUES
```

```
movies=# (
movies=# 1,
movies=# 'John Gomez',
movies=# '555-4272',
movies=# '1982-06-02',
movies=# 0.00
movies=# );
```

```
ERROR: Cannot insert a duplicate key into unique
        index customers_customer_id_key
```

When you create a `UNIQUE` column, PostgreSQL will ensure that an index exists for that column. If you don't create one yourself, PostgreSQL will create one for you. We'll talk more about indexes in [Chapter 3](#).

PRIMARY KEY

Almost every table that you create will have one column (or possibly a set of columns) that uniquely identifies each row. For example, each tape in the `tapes` table is uniquely identified by its `tape_id`. Each customer in your `customers` table is identified by a `UNIQUE` `customer_id`. In relational database lingo, the set of columns that act to identify a row is called the primary key.

Quite often, you will find that a table has more than one unique column. For example, a table holding employee information might have an `employee_id` column and a `social_security_number` (SSN) column. You could argue that either of these would be a reasonable primary key. The `employee_id` would probably be the better choice for at least three reasons. First, you are likely to refer to an employee record in other tables (for example, `withholdings` and `earnings`)—an `employee_id` is (most likely) shorter than an SSN. Second, an SSN is considered private information, and you don't want to expose an employee's SSN to everyone who has access to one of the related files. Third, it is entirely possible that some of your employees may not have Social Security numbers (they may not be U.S. citizens)—you can't define a column as the `PRIMARY KEY` if that column allows `NULL` values.

PostgreSQL provides a constraint, `PRIMARY KEY`, that you can use to define the primary key for a table. Practically speaking, identifying a column (or a set of columns) as a `PRIMARY KEY` is the same as defining the column to be `NOT NULL` and `UNIQUE`. But the `PRIMARY KEY` constraint does offer one advantage over `NOT NULL` and `UNIQUE`: documentation. When you create a `PRIMARY KEY`, you are stating that the columns that comprise the key should be used when you need to refer to a row in that table. Each row in the `rentals` table, for example, contains a reference (`rentals.tape_id`) to a tape and a reference (`rentals.customer_id`) to a customer. You should define the `customers.customer_id` column as the `PRIMARY KEY` of the `customers` table:

```
CREATE TABLE customers (
    customer_id  INTEGER PRIMARY KEY,
    name         VARCHAR(50) NOT NULL,
    phone        CHAR(8),
    birth_date    DATE,
    balance       DECIMAL(7,2)
);
```

You should also define the `tapes.tape_id` column as the primary key of the `tapes` table:

```
CREATE TABLE tapes (
    tape_id      CHARACTER(8) PRIMARY KEY,
    title        CHARACTER VARYING(80)
);
```

Now, let's look at the other half of the equation: the `REFERENCES` constraint.

REFERENCES

A foreign key is a column (or group of columns) in one table that refers to a row in another table. Usually, but not always, a foreign key refers to the primary key of another table.

The `REFERENCES` constraint tells PostgreSQL that one table refers to another table (or more precisely, a foreign key in one table refers to the primary key of another). Let's look at an example:

```
CREATE TABLE rentals (
```

```

        tape_id      CHARACTER(8) REFERENCES tapes,
        customer_id  INTEGER      REFERENCES customers,
        rental_date  DATE
    );

```

I've now defined `rentals.tape_id` and `rentals.customer_id` to be foreign keys. In this example, the `rentals.tape_id` column is also called a reference and the `tapes.tape_id` column is called the referent.

There are a few implications to the `REFERENCES` constraint that you will need to consider. First, the `REFERENCES` constraint is a constraint: PostgreSQL does not allow you to change the database in such a way that the constraint would be violated. You cannot add a `rentals` row that refers to a nonexistent tape (or to a nonexistent customer):

```

movies=# SELECT * FROM tapes;
 tape_id | title
-----+-----
AB-12345 | The Godfather
AB-67472 | The Godfather
MC-68873 | Casablanca
OW-41221 | Citizen Kane
AH-54706 | Rear Window

movies=# INSERT INTO rentals VALUES
movies=# (
movies(#      'OW-00000',
movies(#      1,
movies(#      '2002-02-21'
movies(# );
ERROR:  <unnamed> referential integrity violation -
        key referenced from rentals not found in tapes

```

The next thing to consider is that you cannot (normally) `DELETE` a referent—doing so would violate the `REFERENCES` constraint:

```

movies=# SELECT * FROM rentals;
 tape_id | customer_id | rental_date
-----+-----+-----
AB-12345 |          1 | 2001-11-25
AB-67472 |          3 | 2001-11-25
OW-41221 |          1 | 2001-11-25
MC-68873 |          3 | 2001-11-20
(4 rows)

movies=# DELETE FROM tapes WHERE tape_id = 'AB-12345';
ERROR:  <unnamed> referential integrity violation -
        key in tapes still referenced from rentals

```

Sometimes, it's not appropriate for a `REFERENCES` constraint to block the deletion of a referent. You can specify the action that PostgreSQL should take when the referent is deleted. The default action (also known as `NO ACTION` and `RESTRICT`) is to prevent the deletion of a referent if there are still any references to it. The next alternative, `CASCADE`, deletes all rows that refer to a value when the referent is deleted. The final two choices break the link between the reference and the referent: `SET NULL` updates any references to `NULL` whenever a referent is deleted, whereas `SET DEFAULT` updates any references to their default values when a referent is deleted.

If you want to specify one of the alternatives, you would use the following syntax when you create the `REFERENCES` constraint:

```

REFERENCES table [ (column) ] ON DELETE
    NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT

```

By default, a `REFERENCES` constraint also prevents you from changing data in such a way that the constraint would be violated. You can use the `ON UPDATE` clause to relax the constraint a little, much the same as the `ON DELETE` clause.

The syntax required for `ON UPDATE` is

```

REFERENCES table [ (column) ] ON UPDATE
    NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT

```

There is a subtle difference between the `ON UPDATE` clause and `ON DELETE` clause. When you `DELETE` a referent, the entire row disappears, so the behavior of the `ON DELETE` clause is obvious. When you `UPDATE` a referent row, you may change values other than the referent column(s). If you `UPDATE` a referent row, but you don't update the referent column, you can't introduce a constraint violation, so the `ON UPDATE` action doesn't come into play. If you do change the referent column, the `ON UPDATE` action is triggered.

The `NO ACTION` and `RESTRICT` actions simply prevent a constraint violation—this is identical to the `ON DELETE` clause. The

CASCADE action causes all references to be updated whenever a referent changes. **SET NULL** and **SET DEFAULT** actions work the same for **ON UPDATE** as for **ON DELETE**.

CHECK()

By defining a **CHECK()** constraint on a column, you can tell PostgreSQL that any values inserted into that column must satisfy an arbitrary Boolean expression. The syntax for a **CHECK()** constraint is

```
[CONSTRAINT constraint-name] CHECK( boolean-expression )
```

For example, if you want to ensure that the `customer_balance` column is a positive value, but less than \$10,000.00, you might use the following:

```
CREATE TABLE customers
(
    customer_id    INTEGER UNIQUE,
    customer_name  VARCHAR(50),
    phone          CHAR(8),
    birth_date     DATE,
    balance        DECIMAL(7,2)
    CONSTRAINT invalid_balance
        CHECK( balance > 0 AND balance < 10000 )
);
```

Now, if you try to **INSERT** an invalid value into the `customer_balance` table, you'll cause an error:

```
INSERT INTO customers VALUES
(
    10,
    'John Smallberries',
    '555-8426',
    '1970-JAN-02',
    20000
);
```

```
ERROR:  ExecAppend: rejected due to CHECK constraint invalid_balance
```

Expression Evaluation and Type Conversion

Now that you have seen all the standard PostgreSQL data types, it's time to talk about how you can combine values of different types into complex expressions.

First, you should understand that an expression represents a value. In a well-designed language, you can use an expression anywhere you can use a value. An expression can be as simple as a single value: `3.14159` is an expression. A complex expression is created by combining two simple expressions with an operator. An operator is a symbol that represents some sort of operation to be applied to one or two operands. For example, the expression `"customer_balance * 1.10"` uses the multiplication operator (`*`) to multiply `customer_balance` by `1.10`. In this example, `customer_balance` is the left operand, `*` is the operator, and `1.10` is the right operand. This expression combines two different kinds of values: `customer_balance` is (presumably) a column in one of your tables; whereas `1.10` is a literal value (informally called a constant). You can combine column values, literal values, function results, and other expressions to build complex expressions.

Most operators (such as `*`, `+`, and `<`) require two operands: these are called binary operators. Other operators (such as `!!`, the factorial operator) work with a single value: these are called unary operators^[17]. Some operators (such as `-`) can function as either.

^[17] You may also see the terms dyadic (meaning two-valued) and monadic (meaning single-valued). These terms have the distinct advantage that you will never have to worry about accidentally saying "urinary operator" in polite company.

For some expressions, particularly those expressions that mix data types, PostgreSQL must perform implicit type conversions^[18]. For example, there is no predefined operator that allows you to add an `INT2` to a `FLOAT8`. PostgreSQL can convert the `INT2` into a `FLOAT8` before performing the addition, and there is an operator that can add two `FLOAT8` values. Every computer language defines a set of rules^[19] that govern automatic type conversion; PostgreSQL is no exception.

^[18] A type conversion that is automatically provided by PostgreSQL is called a coercion. A type conversion caused explicitly by the programmer (using the `CAST()` or `::` operator) is called a cast.

^[19] A given language might simply prohibit automatic type conversion, but most languages try to help out the programmer a bit.

PostgreSQL is rather unique in its depth of support for user-defined data types. In most RDBMSs, you can define new data types, but you are really just providing a different name for an existing data type (although you might be able to constrain the set of legal values in the new type). With PostgreSQL, you can add new data types that are not necessarily related to the existing data types. When you add a new data type to PostgreSQL, you can also define a set of operators that can operate on the new type. Each operator is implemented as an operator function; usually, but not necessarily, written in C. When you use an operator in an expression, PostgreSQL must find an operator function that it can use to evaluate the expression. The point of this short digression is that although most languages can define a static set of rules governing type conversion, the presence of user-defined data types requires a more dynamic approach. To accommodate user-defined data types, PostgreSQL consults a table named `pg_operator`. Each row in the `pg_operator` contains an operator name (such as `+` or `#`), the operand data types, and the data type of the result. For example, (in PostgreSQL version 7.1.2) there are 31 rows in `pg_operator` that describe the `+` operator: One row describes the `+` operator when applied to two `POINT` values, another row describes the `+` operator when applied to two `INTERVAL` values, and a third row describes the `+` operator when applied to an `INT2` and an `INT4`.

You can see the complete list of operators using the `"\do"` command in the `psql` query tool.

When searching for an operator function, PostgreSQL first searches the `pg_operator` table for an operator that exactly matches data types involved in the expression. For example, given the expression:

```
CAST( 1.2 AS DECIMAL ) + CAST( 5 AS INTEGER )
```

PostgreSQL searches for a function named `+` that takes a `DECIMAL` value as the left operand and an `INTEGER` value as right operand. If it can't find a function that meets those criteria, the next step is to determine whether it can coerce one (or both) of the values into a different data type. In our example, PostgreSQL could choose to convert either value: The `DECIMAL` value could be converted into an `INTEGER`, or the `INTEGER` value could be converted into a `DECIMAL`. Now we have two operator functions to choose from: One function can add two `DECIMAL` values and the other can add two `INTEGER` values. If PostgreSQL chooses the `INTEGER + INTEGER` operator function, it will have to convert the `DECIMAL` value into an `INTEGER` — this will result in loss of precision (the fractional portion of the `DECIMAL` value will be rounded to the nearest whole number). Instead, PostgreSQL will choose the `DECIMAL + DECIMAL` operator, coercing the `INTEGER` value into a `DECIMAL`.

So to summarize, PostgreSQL first looks for an operator function in which the operand types exactly match the expression being evaluated. If it can't find one, PostgreSQL looks through the list of operator functions that could be applied by coercing one (or both) operands into a different type. If type coercion would result in more than one alternative, PostgreSQL tries to find the operator function that will maintain the greatest precision.

The process of selecting an operator function can get complex and is described more fully in Chapter 5 of the PostgreSQL User's Guide.

Table 2.30 lists the type conversion functions supplied with a standard PostgreSQL distribution.

Table 2.30. Explicit Type Conversion Functions

Result Type	Source Type
BOX	CIRCLE, POLYGON
DATE	TIMESTAMPTZ, DATE, TEXT
INTERVAL	INTERVAL, TEXT, TIME
LSEG	BOX
MACADDR	TEXT
NUMERIC	BIGINT, SMALLINT, INTEGER, REAL, DOUBLE PRECISION
OID	TEXT
PATH	POLYGON
POINT	PATH, LSEG, BOX, POLYGON, CIRCLE
POLYGON	PATH, CIRCLE, BOX
TEXT	INET, DOUBLE PRECISION, NAME, OID, SMALLINT, INTEGER, INTERVAL, TIMESTAMP WITH TIME ZONE, TIME WITH TIME ZONE, TIME, BIGINT, DATE, MACADDR, CHAR, REAL
TIME	TEXT, TIME, TIMESTAMP WITH TIME ZONE, INTERVAL

Creating Your Own Data Types

PostgreSQL allows you to create your own data types. This is not unique among relational database systems, but PostgreSQL's depth of support is unique. In other RDBMSs, you can define one data type in terms of another (predefined) data type. For example, you might create a new numeric data type to hold an employee's age, with valid values between 18 and 100. This is still a numeric data type—you must define the new type as a subset of an existing type. PostgreSQL calls such a "refined" data type a domain. Starting with PostgreSQL version 8.0, you can also create composite data types. A composite type is a single data type made up of multiple fields. For example, you might define a composite type named `address` that contains a street number, city, state/province, and postal code. When you define a composite type, each component has a separate name and data type.

With PostgreSQL, you can create entirely new types that have no relationship to existing types. When you define a custom data type (in PostgreSQL), you determine the syntax required for literal values, the format for internal data storage, the set of operators supported for the new type, and the set of (predefined) functions that can operate on values of that type.

There are a number of contributed packages that add new data types to the standard PostgreSQL distribution. For example, the PostGIS project (<http://postgis.refractory.net>) adds geographic data types based on specifications produced by the Open GIS Consortium. The `/contrib` directory of a standard PostgreSQL distribution contains a cube data type as well as an implementation of ISBN/ISSN (International Standard Book Number/International Standard Serial Number) data types.

Creating a new data type is too advanced for this chapter. If you are interested in defining a new data type, see [Chapter 6](#), "Extending PostgreSQL." In the next two sections, we'll show you how to create and work with domains and composite data types.

Refining Data Types with `CREATE DOMAIN`

A domain is a user-defined data type that refines an existing data type. You typically create a domain when you need to store the same kind of data in many tables (or many times within the same table). For example, if you create a `phone_number` domain, you can store a phone number in a customer table (in fact, you may store many phone numbers per customer), in a salesman table, a vendor table, and so on. At first glance, you may think that you could simply add a `CHARACTER(13)` column to each table, but a phone number isn't simply a collection of 13 characters—it has a specific format. Here in the U.S., a phone number is often written as

(800) 555-1212

The three-digit area code is surrounded by parentheses, then you see the prefix, a dash, and the last four digits of the phone number. To create a `phone_number` domain that enforces these constraints, you could execute the command

```
CREATE DOMAIN phone_number AS CHAR(13)
CHECK( VALUE ~ '\([[:digit:]]{3}\)[[:digit:]]{3}-[[:digit:]]{4}' );
```

The first part of this command is straightforward—you're creating a domain named `phone_number` as a constrained version of a 13-character `CHAR` field. The second part of the command (the `CHECK()` clause) is a constraint. In this case, you're telling PostgreSQL that the `VALUE` stored in a `phone_number` field must match the given regular expression (if you're not accustomed to reading complex regular expressions, this one specifies that `VALUE` must be an open parenthesis followed by three digits, followed by a close parenthesis, followed by three digits, a dash, and then four digits).

Once you've created a domain, you can define columns of that type. For example, to add a home phone number to the `customers` table, use the command:

```
ALTER TABLE customers ADD COLUMN home_phone phone_number;
```

Now here's the payoff. Once you've defined a domain (and all of the constraints you want to apply to the domain), you can use the domain in multiple tables, or many times in the same table. To add two more phone numbers to the `customers` table, use this command:

Code View: [Scroll](#) / [Show All](#)

```
ALTER TABLE customers ADD COLUMN cell_phone phone_number, work_phone phone_number;
```

You've defined the constraints once, but you've created three columns that enforce those constraints. If you don't define a `phone_number` domain, you'll have to specify the constraints every time you add a phone number to a table.

When should you define a domain? Any time you store the same kind of object in multiple tables (or many times in the same table). You should also define a domain for any column that participates in a `PRIMARY/FOREIGNKEY` relationship. For example, the `rentals` table contains two foreign keys: The `rentals.customer_id` column refers to `customer.customer_id` and `rentals.tape_id` refers to `tapes.tape_id` (for the sake of simplicity, we haven't actually defined `PRIMARY/FOREIGN KEY` constraints in the sample data for this book). Given these relationships, it's clear that the data type of `rentals.customer_id` must be identical to the data type of `customer.customer_id` (and that `rentals.tape_id` and `tapes.tape_id` must have the same type). The safest way to ensure that the data types match is to create a `customer_id` domain and a `tape_id` domain and define the key columns using those types, as shown in the following sequence of commands:

```
movies=# CREATE DOMAIN tape_id AS CHARACTER(8);
CREATE DOMAIN
movies=# CREATE DOMAIN customer_id AS INTEGER;
CREATE DOMAIN
movies=# ALTER TABLE customers ALTER COLUMN customer_id TYPE customer_id;
ALTER TABLE
movies=# ALTER TABLE rentals ALTER COLUMN customer_id TYPE customer_id;
ALTER TABLE
```

```

movies=# ALTER TABLE tapes ALTER COLUMN tape_id TYPE tape_id;
ALTER TABLE
movies=# ALTER TABLE rentals ALTER COLUMN tape_id TYPE tape_id;
ALTER TABLE

```

Notice that the `tape_id` and `customer_id` domains are unconstrained. You don't have to attach constraints to a domain—an unconstrained domain is still useful because it defines a logical data type. In fact, you can attach new constraints to a domain later (or change existing constraints) using the `ALTER DOMAIN` command and PostgreSQL will ensure that existing data conforms to the new constraints (you'll be rewarded with an error message if you have any data that fails to satisfy the new constraints).

The complete syntax for `CREATE DOMAIN` is shown here:

```

CREATE DOMAIN name [AS] data_type
    [ DEFAULT expression ]
    [ CONSTRAINT constraint_name ]
    [ CONSTRAINT constraint_name ] NOT NULL
    [ CONSTRAINT constraint_name ] CHECK( expression )

```

where constraint is one or more of the following:

```

[ CONSTRAINT constraint_name ] NULL
[ CONSTRAINT constraint_name ] NOT NULL
[ CONSTRAINT constraint_name ] CHECK( expression )

```

If you include a `DEFAULT expression` clause, the given value becomes the default for any columns of type `name`. In other words, if you omit a column of type `name` in an `INSERT` command, PostgreSQL inserts `expression` instead of the usual `NULL` value. The default value should satisfy any constraints that you attach to the domain.

Once you've created a column whose type is defined by a domain, you can treat that column in the same way you would treat any other column of the base data type. For example, if you define a domain whose base type is `CHARACTER`, you can insert string values using the same syntax you would use for values of type `CHARACTER`. You can also create indexes that include domain values. A domain is a refinement of some other data type.

Creating and Using Composite Types

One of the new and powerful features introduced in PostgreSQL version 8.0 is the composite data type. A composite type is a data type composed of one or more named fields. For example, you might want to create a composite type named `address` composed of a street number, city, state/province, and postal code. The following command will do just that:

```

movies=# CREATE TYPE address AS
movies-# (
movies(#   street_number  VARCHAR,
movies(#   city           VARCHAR,
movies(#   state          CHAR(2),
movies(#   postal_code    VARCHAR
movies(# );

```

Once you've defined a composite type, you can create columns based on the new type. When you add a column of composite type, you're adding a single field that happens to be composed of multiple fields. For example, to add an `address` to the `customers` table, execute the following command:

```

movies=# ALTER TABLE customers ADD COLUMN home_address address;
ALTER TABLE

```

Now take a look at the definition of the `customers` table:

```

movies=# \d customers;
          Table "public.customers"
   Column   |      Type      | Modifiers
-----+-----+-----
customer_id | integer        | not null
customer_name | character varying(50) | not null
phone       | character(8)   |
birth_date  | date           |
balance     | numeric(7,2)   |
home_address | address        |
Indexes:
    "customers_customer_id_key" UNIQUE, btree (customer_id)

```

The `ALTER TABLE` command added a single field named `home_address`. So what happened to the `street_number`, `city`, `state`, and `postal_code` fields? They're inside the `home_address` column. Let's fill in the `home_address` for one of your customers:

Code View: [Scroll](#) / [Show All](#)

```

movies=# UPDATE customers
movies-#   SET home_address = ( '200 Main Street', 'Springfield', 'CA', '90210' )
movies-#   WHERE customer_id = 3;

```

Notice that the composite value that you're inserting is enclosed in parentheses. There are three ways that you can write a composite literal. The easiest method is the one you've just seen—simply enclose the component values in a set of parentheses. If you only have a single component to

INSERT (that is, you're inserting default values for the other components), you must use the row constructor form instead:

```
ROW( '200 Main Street' )
```

You can also write the composite value as a string:

```
'(200 Main Street, Springfield, CA, 90210)'
```

The row constructor form is typically the easiest way to build a composite literal because you don't have to worry about doubling-up any embedded quotes.

Now take a look at what PostgreSQL stored in the row you just added:

```
movies=# SELECT customer_name, home_address FROM customers;
 customer_name | home_address
-----+-----
 Jones, Henry  |
 Wonderland, Alice N. |
 Rubin, William |
 Panky, Henry   | ("200 Main Street", Springfield, CA, 90210)
(4 rows)
```

You see a single column (`home_address`) with four values in it. How do you get to the individual components in the `home_address` column? Simply refer to `(columnName).fieldname`, like this:

```
movies=# SELECT customer_name, (home_address).postal_code FROM customers;
 customer_name | postal_code
-----+-----
 Jones, Henry  |
 Wonderland, Alice N. |
 Rubin, William |
 Panky, Henry   | 90210
(4 rows)
```

That command retrieved the `postal_code` component of the `home_address` column. The parentheses are required because the PostgreSQL parser can't tell if `home_address.postal_code` refers to a column (`postal_code`) within a table (`home_address`) or a field (`postal_code`) within a composite column (`home_address`).

Of course you can UPDATE a single component in a composite column as well:

```
movies=# UPDATE customers
movies-#   SET home_address.postal_code = '94404'
movies-#   WHERE customer_id = 3;
```

Notice that you can't include the parentheses around `home_address` in this case. Why? Because the parser would never expect to see a table name following the word `SET` and therefore can't mistake `home_address` as the name of a table.

You can't easily^[20] create an index on a composite column, but you can create an index on an individual component (or on multiple components) even though the syntax is a bit mysterious. To create an index on `home_address.city` plus `home_address.state`, use the following command:

^[20] You can create an index on a composite column, but you'll have to define an index operator class for each composite type—it's much easier to create an index on each component instead.

```
movies=# CREATE INDEX customer_location ON customers
movies-# (
movies-#   (( home_address ).city ),
movies-#   (( home_address ).state )
movies-# );
```

Take careful note of the parentheses—when you create an index on a field within a composite column, you must use the syntax `'((columnName). fieldname)'`.

If you ask `psql` to display the layout of a table that contains a composite column, you won't see the component fields listed:

Code View: [Scroll](#) / [Show All](#)

```
movies=# \d customers;
          Table "public.customers"
   Column |          Type          | Modifiers
-----+-----+-----
customer_id | integer                | not null
customer_name | character varying(50) | not null
phone       | character(8)           |
birth_date  | date                   |
balance     | numeric(7,2)           |
home_address | address                 |
Indexes:
    "customers_customer_id_key" UNIQUE, btree (customer_id)
    "customer_location" btree (((home_address).city), ((home_address).state))
```

To see the definition of a composite type, use the command `\d typename:`

```
movies=# \d address
Composite type "public.address"
  Column      |      Type
-----+-----
street_number | character varying
city          | character varying
state         | character(2)
postal_code   | character varying
```

There are a few restrictions on composite types in PostgreSQL version 8.0. You can't attach constraints to a composite type. That's really not a problem because you can attach constraints to a domain and define a composite type that uses the domain. You can't create a domain whose base type is a composite type, but you can create a composite type that includes a domain. You can create a composite type that contains fields of composite type (meaning that you can nest one composite type within another). Nesting composite types can cause some confusion when you need to create a literal value of the outermost type (you need a lot of parentheses).

Overall, composite types take you one step closer to modeling complex real-world objects inside of a PostgreSQL database. When you combine composite types and domains, you have a powerful mechanism for enforcing constraints on complex objects.

Summary

As you can see, PostgreSQL offers a data type to fit almost every need. In this chapter, I've described each data type included in a standard PostgreSQL distribution. The syntax for literal values may seem a bit contrived for some of the data types, but the fact that PostgreSQL allows you to define new data types requires a few concessions (fortunately, very few).

I've listed all the standard operators in this chapter because they are a bit under-documented in the PostgreSQL User's Guide. Functions, on the other hand, are well documented (as well as constantly changing)—refer to Chapter 4 of the PostgreSQL User's Guide for an up-to-date list of functions.

In [Chapter 3](#), we'll explore a variety of topics that should round out your knowledge of PostgreSQL from the perspective of a user. Later chapters will cover PostgreSQL programming and PostgreSQL administration.

Chapter 3. PostgreSQL SQL Syntax and Use

The first two chapters explored the basics of the SQL language and looked at the data types supported by PostgreSQL. This chapter covers a variety of topics that should round out your knowledge of PostgreSQL.

We'll start by looking at the rules that you have to follow when choosing names for tables, columns, indexes, and such. Next, you'll see how to create, destroy, and view PostgreSQL databases. In [Chapter 1](#), "Introduction to PostgreSQL and SQL," you created a few simple tables; in this chapter, you'll learn all the details of the `CREATE TABLE` command. I'll also talk about indexes. I'll finish up by talking about transaction processing and locking. If you are familiar with Sybase, DB2, or Microsoft SQL Server, I think you'll find that the locking model used by PostgreSQL is a refreshing change.

PostgreSQL Naming Rules

When you create an object in PostgreSQL, you give that object a name. Every table has a name, every column has a name, and so on. PostgreSQL uses a single data type to define all object names: the `name` type.

A value of type `name` is a string of 63 or fewer characters^[1]. A `name` must start with a letter or an underscore; the rest of the string can contain letters, digits, and underscores.

^[1] You can increase the length of the name data type by changing the value of the `NAMEDATALEN` symbol before compiling PostgreSQL.

If you examine the entry corresponding to `name` in the `pg_type` table, you will find that a `name` is really 64 characters long. Because the `name` type is used internally by the PostgreSQL engine, it is a null-terminated string. So, the maximum length of a `name` value is 63 characters. You can enter more than 63 characters for an object name, but PostgreSQL stores only the first 63 characters.

Both SQL and PostgreSQL reserve certain words and normally, you cannot use those words to name objects. Examples of reserved words are

```
ANALYZE
BETWEEN
CHARACTER
INTEGER
CREATE
```

You cannot create a table named `INTEGER` or a column named `BETWEEN`. A complete list of reserved words can be found in Appendix B of the PostgreSQL User's Guide.

If you find that you need to create an object that does not meet these rules, you can enclose the name in double quotes. Wrapping a name in quotes creates a quoted identifier. For example, you could create a table whose name is `"3.14159"`—the double quotes are required, but are not actually a part of the name (that is, they are not stored and do not count against the 63-character limit). When you create an object whose name must be quoted, you have to include the quotes not only when you create the object, but every time you refer to that object. For example, to select from the table mentioned previously, you would have to write

```
SELECT filling, topping, crust FROM "3.14159";
```

Here are a few examples of both valid and invalid names:

```
my_table      -- valid
my_2nd_table  -- valid
échéanciers   -- valid: accented and non-Latin letters are allowed
"2nd_table"   -- valid: quoted identifier
"create table" -- valid: quoted identifier
"1040Forms"   -- valid: quoted identifier
2nd_table     -- invalid: does not start with a letter or an underscore
```

Quoted names are case-sensitive. `"1040Forms"` and `"1040FORMS"` are two distinct names. Unquoted names are converted to lowercase, as shown here:

```
movies=# CREATE TABLE FOO( BAR INTEGER );
CREATE
movies=# CREATE TABLE foo( BAR INTEGER );
ERROR: Relation 'foo' already exists
movies=# \d
```

```
          List of relations
-----+-----+-----
 Name      | Type | Owner
-----+-----+-----
 1040FORMS | table | bruce
 1040Forms | table | sheila
 customers | table | bruce
   foo     | table | bruce
  rentals  | table | bruce
   tapes   | table | bruce
(6 rows)
```

The names of all objects must be unique within some scope. Every database must have a unique name; the name of a schema must be unique within the scope of a single database, the name of a table must be unique within the scope of a single schema, and column names must be unique within a table. The name of an index must be unique within a database.

The Importance of the `COMMENT` Command

If you've been a programmer (or database developer) for more than, say, two days, you understand the importance of commenting your code. A comment helps new developers understand how your program (or database) is structured. It also helps you remember what you were thinking when you come back to work after a long weekend. If you're writing procedural code (in C, Java, PL/pgSQL, or whatever language you prefer), you can intersperse comments directly into your code. If you're creating objects in a PostgreSQL database, where do you store the comments? In the database, of course. The `COMMENT` command lets you associate a comment with just about any object that you can define in a PostgreSQL database. The syntax for the `COMMENT` command is very simple:

```
COMMENT ON object-type object-name IS comment-text;
```

where `object-type` and `object-name` are taken from the following:

```
DATABASE database-name
SCHEMA schema-name
TABLE table-name
COLUMN table-name.column-name
INDEX index-name
DOMAIN domain-name
TYPE data-type-name
VIEW view-name
CONSTRAINT constraint-name ON table-name
SEQUENCE sequence-name
TRIGGER trigger-name ON table-name
```

You can also define comments for other object types (functions, operators, rules, even languages), but the object types that we've shown here are the most common (see the PostgreSQL reference documentation for a complete list).

To add a comment to a table, for example, you would execute a command such as

```
COMMENT ON TABLE customers IS 'List of active customers';
```

You can only store one comment per object—if you `COMMENT ON` an object twice, the second comment replaces the first. To drop a comment, execute a `COMMENT` command, but specify `NULL` in place of the `comment-text` string, like this:

```
COMMENT ON TABLE customers IS NULL;
```

Once you have added a comment to an object, you can view the comment (in `psql`) using the command `\dd object-name-pattern`, like this:

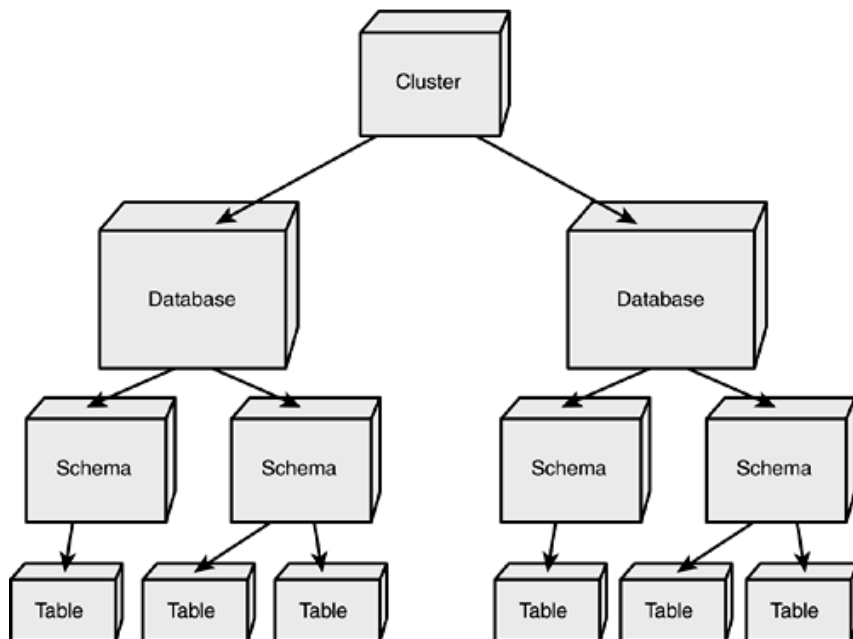
```
movies=# \dd customers
              Object descriptions
 Schema | Name      | Object | Description
-----+-----+-----+-----
 public | customers | table  | List of active customers
(1 row)
```

The `\dd` command will show you any commented object whose name matches the `object-name-pattern`. The `\dd` command will not show comments that you've assigned to a column within a table. To see column-related comments, use the command `\d+ [table-name]`. To see the comment assigned to each database, use the command `\l+`.

Creating, Destroying, and Viewing Databases

Before you can do anything else with a PostgreSQL database, you must first create the database. Before you get too much further, it might be a good idea to see where a data base fits into the overall scheme of PostgreSQL. Figure 3.1 shows the relationships between clusters, databases, schemas, and tables.

Figure 3.1. Clusters, databases, schemas, and tables.



At the highest level of the PostgreSQL storage hierarchy is the cluster. A cluster is a collection of databases. Each cluster exists within a single directory tree, and the entire cluster is serviced by a single `postmaster`. A cluster is not named—there is no way to refer to a cluster within PostgreSQL, other than by contacting the `postmaster` servicing that cluster. The `$PGDATA` environment variable should point to the root of the cluster's directory tree. A cluster is serviced by a single `postmaster` process. The `postmaster` listens for connection requests coming from client applications. When a connection request is received (and the user's credentials are authenticated), the `postmaster` starts a new server process and connects the client to the server. A single client connection can only interact with a single database at any given time (but a client application can certainly open multiple connections if it needs to interact with several databases simultaneously). A `postmaster` process can connect a client application to any of the databases in the cluster serviced by that `postmaster`.

Four system tables are shared between all databases in a cluster: `pg_group` (the list of user groups), `pg_database` (the list of databases within the cluster), `pg_shadow` (the list of valid users), and `pg_tablespace` (the list of tablespaces).

Each cluster contains one or more databases. Every database has a name that must follow the naming rules described in the previous section. Database names must be unique within a cluster. A database is a collection of schemas.

A schema is a named collection of tables (as well as functions, data types, and operators). The schema name must be unique within a database. Table names, function names, index names, type names, and operators must be unique within the schema. A schema exists primarily to provide a naming context. You can refer to an object in any schema within a single database by prefixing the object name with `schema-name`. For example, if you have a schema named `bruce`, you can create a table within that schema as

```
CREATE TABLE bruce.ratings ( ... );
SELECT * FROM bruce.ratings;
```

Each connection has a schema search path. If the object that you are referring to is found on the search path, you can omit the schema name. However, because table names are not required to be unique within a database, you may find that there are two tables with the same name within your search path (or a table may not be in your search path at all). In those circumstances, you can include the schema name to remove any ambiguity.

To view the schema search path, use the command `SHOW SEARCH_PATH`:

```
movies=# SHOW SEARCH_PATH;
search_path
-----
 $user,public
(1 row)
```

The default search path, shown here, is `$user,public`. The `$user` part equates to your PostgreSQL user name. For example, if I connect to `psql` as user `bruce`, my search path is `bruce,public`. If a schema named `bruce` does not exist, PostgreSQL will just ignore that part of the search path and move on to the schema named `public`. To change the search path, use `SET SEARCH_PATH TO`:

```
movies=# SET SEARCH_PATH TO 'bruce','sheila','public';
```

SET

You create a new schema with the `CREATE SCHEMA` command and destroy a schema with the `DROP SCHEMA` command:

```
movies=# CREATE SCHEMA bruce;
CREATE SCHEMA

movies=# CREATE TABLE bruces_table( pkey INTEGER );
CREATE TABLE

movies=# \d
      List of relations
  Name          | Schema | Type  | Owner
-----+-----+-----+-----
 bruces_table   | bruce  | table | bruce
 tapes          | public | table | bruce
(2 rows)

movies=# DROP SCHEMA bruce;
ERROR:  Cannot drop schema bruce because other objects depend on it
        Use DROP ... CASCADE to drop the dependent objects too

movies=# DROP SCHEMA bruce CASCADE;
NOTICE: Drop cascades to table bruces_table
DROP SCHEMA
```

Notice that you won't be able to drop a schema that is not empty unless you include the `CASCADE` clause. Schemas are a relatively new feature that first appeared in PostgreSQL version 7.3. Schemas are very useful. At many sites, you may need to keep a "development" system and a "production" system. You might consider keeping both systems in the same database, but in separate schemas. Another (particularly clever) use of schemas is to separate financial data by year. For example, you might want to keep one year's worth of data per schema. The table names (*invoices*, *sales*, and so on) remain the same across all schemas, but the schema name reflects the year to which the data applies. You could then refer to data for 2001 as *FY2001.invoices*, *FY2001.sales*, and so on. The data for 2002 would be stored in *FY2002.invoices*, *FY2002.sales*, and so on. This is a difficult problem to solve without schemas because PostgreSQL does not support cross-database access. In other words, if you are connected to database *movies*, you can't access tables stored in another database. Starting with PostgreSQL 7.3, you can keep all your data in a single database and use schemas to partition the data.

When you create a schema, you can specify an optional tablespace—by default, tables created within the schema will be stored in the schema's tablespace. We discuss tablespaces in more detail in the next with the `CREATE SCHEMA` section.

Tablespaces

Starting with PostgreSQL version 8.0, you can store database objects (tables and indexes) in alternate locations using a new feature called a tablespace. A tablespace is a name that you give to some directory within your computer's filesystem. Once you create a tablespace (we'll show you how in a moment), you can create schemas, tables, and indexes within that tablespace. A tablespace is defined within a single cluster—all databases within a cluster can refer to the same tablespace.

To create a new tablespace, use the `CREATE TABLESPACE` command:

```
CREATE TABLESPACE tablespacename
[ OWNER username ]
LOCATION 'directory'
```

The `tablespacename` parameter must satisfy the normal rules for all identifiers; it must be 63 characters or shorter and must start with a letter (or the name must be quoted). In addition, you can't create a tablespace whose name begins with the characters `'pg_'` since those names are reserved for the PostgreSQL development team. If you omit the `OWNER username` clause, the new tablespace is owned by the user executing the `CREATE TABLESPACE` command. By default, you can't create an object in a tablespace unless you are the owner of that tablespace (or you are a PostgreSQL superuser). You can grant `CREATE` privileges to other users with the `GRANT` command (see [Chapter 23](#), "Security" for more information on the `GRANT` command).

The interesting part of a `CREATE TABLESPACE` command is the `LOCATION 'directory'` clause. The `LOCATION` clause includes a directory—objects created within the tablespace are stored in that directory. There are a few rules that you must follow before you can create a tablespace:

- You must be a PostgreSQL superuser
- PostgreSQL must be running on a system that supports symbolic links (that means you can't create tablespaces on a Windows host)
- The `directory` must already exist (PostgreSQL won't create the directory for you)
- The `directory` must be empty
- The `directory` name must be shorter than 991 characters
- The `directory` must be owned by the owner of the postmaster process (typically a user named `postgres`)

If all of those conditions are satisfied, PostgreSQL creates the new tablespace.

When you create a tablespace, the PostgreSQL server performs a number of actions behind the scenes. First, the permissions on the

directory are changed to 700 (read, write, and execute permissions for the directory owner, all other permissions denied). Next, PostgreSQL creates a single file named `PG_VERSION` in the given directory (the `PG_VERSION` file stores the version number of the PostgreSQL server that created the tablespace—if the PostgreSQL developers change the structure of a tablespace in a future version, `PG_VERSION` will help any conversion tools understand the structure of an existing tablespace). If the permission change succeeds, PostgreSQL adds a new row to the `pg_tablespace` table (a cluster-wide table) and assigns a new OID (object-id) to that row. Next, the server uses the OID to create a symbolic link between your cluster and the given directory. For example, consider the following scenario:

```
movies# CREATE TABLESPACE mytablespace LOCATION '/fastDrive/pg';
CREATE TABLESPACE

movies# SELECT oid, spcname, spclocation
movies-# FROM
movies-#     pg_tablespace
movies-# WHERE
movies-#     spcname = 'mytablespace';
  oid | spcname | spclocation
-----+-----+-----
 34281 | mytablespace | /fastDrive/pg
```

In this case, PostgreSQL assigned the new tablespace (`mytablespace`) an OID of 34281. PostgreSQL creates a symbolic link that points from `$PGDATA/pg_tblspc/34281` to `/fastDrive/pg`. When you create an object (a table or index) inside of this tablespace, the object is not created directly inside of the `/fastDrive/pg` directory. Instead, PostgreSQL creates a subdirectory in the tablespace and then creates the object within that subdirectory. The name of the subdirectory corresponds to the OID of the database (that is, the object-id of the database's entry in the `pg_database` table) that holds the new object. If you create a new table within the `mytablespace` tablespace, like this:

```
movies# CREATE TABLE foo ( data VARCHAR ) TABLESPACE mytablespace;
CREATE TABLE
```

Then find the OID of the new table and the OID of the database (`movies`):

```
movies# SELECT oid FROM pg_class WHERE relname = 'foo';
  oid
-----
 34282
(1 row)

movies# SELECT oid FROM pg_database WHERE datname = 'movies';
  oid
-----
 17228
(1 row)
```

You can see the relationships between the tablespace, the database subdirectory, and the new table:

Code View: [Scroll](#) / [Show All](#)

```
$ ls -l $PGDATA/pg_tblspc
total 0
lrwxrwxrwx    1 postgres postgres    12 Nov  9 19:31 34281 -> /fastDrive/pg

$ ls -l /fastDrive/pg
total 8
drwx-----   2 postgres postgres   4096 Nov  9 19:50 17228
-rw-----   1 postgres postgres     4 Nov  9 19:31 PG_VERSION

$ ls -l /fastDrive/pg/17228
total 0
-rw-----   1 postgres postgres     0 Nov  9 19:50 34282
```

Notice that `$PGDATA/pg_tblspc/34281` is a symbolic link that points to `/fastDrive/pg` (34281 is the OID of `mytablespace`'s entry in the `pg_tablespace` table), PostgreSQL has created a subdirectory (17228) for the `movies` database, and the table named `foo` was created in that subdirectory (in a file whose name, 34282, corresponds to the table's OID). By creating a subdirectory for each database, PostgreSQL ensures that you can safely store objects from multiple databases within the same tablespace without worrying about OID collisions.

When you create a cluster (which is done for you automatically when you install PostgreSQL), PostgreSQL silently creates two tablespaces for you: `pg_default` and `pg_global`. PostgreSQL creates objects in the `pg_default` tablespace when it can't find a more appropriate tablespace. The `pg_default` tablespace is always located in the `$PGDATA/base` directory. The `pg_global` tablespace stores cluster-wide tables like `pg_database`, `pg_group`, and `pg_tablespace`—you can't create objects in the `pg_global` tablespace.

The name of the `pg_default` tablespace can be a bit misleading. You may think that PostgreSQL always creates an object in `pg_default` if you omit the `TABLESPACE tablespacename` clause, but that's not the case. Instead, PostgreSQL follows an inheritance hierarchy to find the appropriate tablespace. If you specify a `TABLESPACE tablespacename` clause when you execute a `CREATE TABLE` or `CREATE INDEX` command, the server creates the object in the given `tablespacename`. If you don't specify a tablespace and you're creating an index, the index is created in the tablespace of the parent table (that is, the table that you are indexing). If you don't specify a tablespace and you're creating a table, the table is created in the tablespace of the parent schema. If you are creating a schema and you don't specify a tablespace, the schema is created in the tablespace of the parent database. If you are creating a database and you don't specify a tablespace, the database is created in the tablespace of the template database (typically, `template1`). So, an index inherits its tablespace

from the parent table, a table inherits its tablespace from the parent schema, a schema inherits its tablespace from the parent database, and a database inherits its database from the template database.

To view the databases defined in a cluster, use the `\db` (or `\db+`) command in `psql`:

```
movies=# \db+
          List of tablespaces
  Name      | Owner   | Location      | Access privileges
-----+-----+-----+-----
mytablespace | postgres | /fastDrive/pg |
pg_default  | postgres |                |
pg_global   | postgres |                | {pg=C/pg}
(4 rows)
```

To see a list of objects defined with a given tablespace, use the following query:

```
SELECT relname FROM pg_class
WHERE reltablespace =
(
    SELECT oid FROM pg_tablespace WHERE spcname = 'tablespacename'
);
```

Don't confuse schemas and tablespaces—they both provide organization for the tables and indexes in your cluster, but they are definitely not the same thing. A tablespace affects the physical organization of data within a cluster (that is, it a tablespace defines where your data is stored). A schema affects the logical organization of data within a database—a schema affects name resolution; a tablespace does not. A schema acts as a part of a name; once you've created an object, you can ignore its physical location (its tablespace).

Creating New Databases

Now let's see how to create a new database and how to remove an existing one.

The syntax for the `CREATE DATABASE` command is

```
CREATE DATABASE database-name
[ WITH [ OWNER [=] {username|DEFAULT}}
      [ TEMPLATE [=] {template-name|DEFAULT}}
      [ ENCODING [=] {encoding|DEFAULT}}
      [ TABLESPACE [=] tablespace ]]
```

As I mentioned earlier, the `database-name` must follow the PostgreSQL naming rules described earlier and must be unique within the cluster.

If you don't include the `OWNER=username` clause or you specify `OWNER=DEFAULT`, you become the owner of the database. If you are a PostgreSQL superuser, you can create a database that will be owned by another user using the `OWNER=username` clause. If you are not a PostgreSQL superuser, you can still create a database if you have the `CREATEDB` privilege, but you cannot assign ownership to another user. [Chapter 21, "PostgreSQL Administration,"](#) describes the process of defining user privileges.

The `TEMPLATE=template-name` clause is used to specify a template database. A template defines a starting point for a database. If you don't include a `TEMPLATE=template-name` or you specify `TEMPLATE=DEFAULT`, the database named `template1` is copied to the new database. All tables, views, data types, functions, and operators defined in the template database are duplicated into the new database. If you add objects (usually functions, operators, and data types) to the `template1` database, those objects will be propagated to any new databases that you create based on `template1`. You can also trim down a template database if you want to reduce the size of new databases. For example, you might decide to remove the geometric data types (and the functions and operators that support that type) if you know that you won't need them. Or, if you have a set of functions that are required by your application, you can define the functions in the `template1` database and all new databases will automatically include those functions. If you want to create an as-distributed database, you can use `template0` as your template database. The `template0` database is the starting point for `template1` and contains only the standard objects included in a PostgreSQL distribution. You should not make changes to the `template0` database, but you can use the `template1` database to provide a site-specific set of default objects.

You can use the `ENCODING=character-set` clause to choose an encoding for the string values in the new database. An encoding determines how the bytes that make up a string are interpreted as characters. For example, specifying `ENCODING=SQL_ASCII` tells PostgreSQL that characters are stored in ASCII format, whereas `ENCODING=ISO-8859-8` requests ECMA-121 Latin/Hebrew encoding. When you create a database, all characters stored in that database are encoded in a single format. When a client retrieves data, the client/server protocol automatically converts between the database encoding and the encoding being used by the client. [Chapter 22, "Internationalization and Localization,"](#) discusses encoding schemes in more detail.

The `TABLESPACE=tablespace-name` clause tells PostgreSQL that you want to create the database in an alternate location (that is, the database should not be created in the usual `$PGDATA/base` directory). You must create a tablespace before you can use it. If you don't include a `TABLESPACE` clause in the `CREATE DATABASE` command, the new database is created in the same tablespace as the template database.

If you're using an older version of PostgreSQL (older than 8.0), you can't use tablespaces to create a database in a non-standard location. Instead, you must use a feature known as a location. In versions of PostgreSQL older than 8.0, the last option for the `CREATE DATABASE` command is the `LOCATION=path` clause. In most cases, you will never have to use the `LOCATION` option, which is good because it's a little strange.

If you do have need to use an alternate location, you will probably want to specify the location by using an environment variable. The environment variable must be known to the `postmaster` processor at the time the `postmaster` is started and it should contain an absolute

pathname.

The `LOCATION=path` clause can be confusing. The `path` might be specified in three forms:

- The path contains a `/`, but does not begin with a `/`—this specifies a relative path
- The path begins with a `/`—this specifies an absolute path
- The path does not include a `/`

Relative locations are not allowed by PostgreSQL, so the first form is invalid.

Absolute paths are allowed only if you defined the C/C++ preprocessor symbol `"ALLOW_ABSOLUTE_DBPATHS"` at the time you compiled your copy of PostgreSQL. If you are using a prebuilt version of PostgreSQL, the chances are pretty high that this symbol was not defined and therefore absolute paths are not allowed.

So, the only form that you can rely on in a standard distribution is the last—a path that does not include any `/` characters. At first glance, this may look like a relative path that is only one level deep, but that's not how PostgreSQL sees it. In the third form, the path must be the name of an environment variable. As I mentioned earlier, the environment variable must be known to the `postmaster` processor at the time the `postmaster` is started, and it should contain an absolute pathname. Let's look at an example:

```
$ export PG_ALTERNATE=/bigdrive/pgdata
$ initlocation PG_ALTERNATE
$ pg_ctl restart -l /tmp/pg.log -D $PGDATA
...
$ psql -q -d movies
movies=# CREATE DATABASE bigdb WITH LOCATION=PG_ALTERNATE;
...
```

First, I've defined (and exported) an environment variable named `PG_ALTERNATE`. I've defined `PG_ALTERNATE` to have a value of `/bigdrive/pgdata`—that's where I want my new database to reside. After the environment variable has been defined, I need to initialize the directory structure—the `initlocation` script will take care of that for me. Now I have to restart the `postmaster` so that it can see the `PG_ALTERNATE` variable. Finally, I can start `psql` (or some other client) and execute the `CREATE DATABASE` command specifying the `PG_ALTERNATE` environment variable.

This all sounds a bit convoluted, and it is. The PostgreSQL developers consider it a security risk to allow users to create databases in arbitrary locations. Because the `postmaster` must be started by a PostgreSQL administrator, only an administrator can choose where databases can be created. So, to summarize the process:

1. Create a new environment variable and set it to the path where you want new databases to reside.
2. Initialize the new directory using the `initlocation` application.
3. Stop and restart the `postmaster`.
4. Now, you can use the environment variable with the `LOCATION=path` clause.

createdb

The `CREATE DATABASE` command creates a new database from within a PostgreSQL client application (such as `psql`). You can also create a new database from the operating system command line. The `createdb` command is a shell script that invokes `psql` for you and executes the `CREATE DATABASE` command for you. For more information about `createdb`, see the PostgreSQL Reference Manual or invoke `createdb` with the `--help` flag:

```
$ createdb --help
createdb creates a PostgreSQL database.
```

```
Usage:
  createdb [OPTION]... [DBNAME] [DESCRIPTION]
```

Options:

<code>-D, --tablespace=TABLESPACE</code>	default tablespace for the database
<code>-E, --encoding=ENCODING</code>	encoding for the database
<code>-O, --owner=OWNER</code>	database user to own the new database
<code>-T, --template=TEMPLATE</code>	template database to copy
<code>-e, --echo</code>	show the commands being sent to the server
<code>-q, --quiet</code>	don't write any messages
<code>--help</code>	show this help, then exit
<code>--version</code>	output version information, then exit

Connection options:

<code>-h, --host=HOSTNAME</code>	database server host or socket directory
<code>-p, --port=PORT</code>	database server port
<code>-U, --username=USERNAME</code>	user name to connect as
<code>-W, --password</code>	prompt for password

By default, a database with the same name as the current user is created.

Report bugs to <pgsql-bugs@postgresql.org>.

Dropping a Database

Getting rid of an old database is easy. The `DROP DATABASE` command will delete all of the data in a database and remove the database from the cluster.

For example:

```
movies=# CREATE DATABASE redshirt;
CREATE DATABASE
movies=# DROP DATABASE redshirt;
DROP DATABASE
```

There are no options to the `DROP DATABASE` command; you simply include the name of the database that you want to remove. There are a few restrictions. First, you must own the database that you are trying to drop, or you must be a PostgreSQL superuser. Next, you cannot drop a database from within a transaction block—you cannot roll back a `DROP DATABASE` command. Finally, the database must not be in use, even by you. This means that before you can drop a database, you must connect to a different database (`template1` is a good candidate). An alternative to the `DROP DATABASE` command is the `dropdb` shell script. `dropdb` is simply a wrapper around the `DROP DATABASE` command; see the PostgreSQL Reference Manual for more information about `dropdb`.

Viewing Databases

Using `psql`, there are two ways to view the list of databases. First, you can ask `psql` to simply display the list of databases and then exit. The `-l` option does this for you:

```
$ psql -l
      List of databases
  Name  | Owner  | Encoding
-----+-----+-----
template0 | postgres | UNICODE
template1 | postgres | UNICODE
movies   | bruce   | UNICODE
(3 rows)
$
```

From within `psql`, you can use the `\l` or `\l+` meta-commands to display the databases within a cluster:

```
movies=# \l+
      List of databases
  Name  | Owner  | Encoding | Description
-----+-----+-----+-----
template0 | postgres | UNICODE | 
template1 | postgres | UNICODE | Default template database
movies   | bruce   | UNICODE | Virtual Video database
(3 rows)
```

Creating New Tables

The previous section described how to create and drop databases. Now let's move down one level in the PostgreSQL storage hierarchy and talk about creating and dropping tables.

You've created some simple tables in the first two chapters; it's time to talk about some of the more advanced features of the `CREATE TABLE` command. Here is the command that you used to create the `customers` table:

```
CREATE TABLE customers (
    customer_id    INTEGER UNIQUE,
    customer_name  VARCHAR(50),
    phone          CHAR(8),
    birth_date     DATE,
    balance        DECIMAL(7,2)
);
```

This command creates a permanent table named `customers`. A table name must meet the naming criteria described earlier in this chapter. When you create a table, PostgreSQL automatically creates a new data type^[2] with the same name as the table. This means that you can't create a table whose name is the same as an existing data type.

^[2] This seems to be a holdover from earlier days. You can't actually do anything with this data type.

When you execute this command, the `customers` table is created in the database that you are connected to. If you are using PostgreSQL 7.3 or later, the `customers` table is created in the first schema in your search path. (If you are using a version older than 7.3, your copy of PostgreSQL does not support schemas). If you want the table to be created in some other schema, you can prefix the table name with the schema qualifier, for example:

```
CREATE TABLE joes_video.customers( ... );
```

The new table is owned by you. You can't give ownership to another user at the time you create the table, but you can change it later using the `ALTER TABLE...OWNER TO` command (described later).

When you create a table (or an index), you can tell PostgreSQL to store the object in a specific tablespace by including a `TABLESPACE tablespace_name` clause, like this:

```
CREATE TABLE joes_video.customers( ... ) TABLESPACE mytablespace;
```

If you don't specify a tablespace, PostgreSQL creates the table in the tablespace assigned to the schema (if you're creating an index without specifying a tablespace, the index is created in the tablespace of the parent table).

Temporary Tables

I mentioned earlier that the `customers` table is a permanent table. You can also create temporary tables. A permanent table persists after you terminate your PostgreSQL session; a temporary table is automatically destroyed when your PostgreSQL session ends. Temporary tables are also local to your session, meaning that other PostgreSQL sessions can't see [temporary tables](#) that you create. Because temporary tables are local to each session, you don't have to worry about colliding with the name of a table created by another session.

If you create a temporary table with the same name as a permanent table, you are effectively hiding the permanent table. For example, let's create a temporary table that hides the permanent `customers` table:

```
CREATE TEMPORARY TABLE customers (
    customer_id    INTEGER UNIQUE,
    customer_name  VARCHAR(50),
    phone          CHAR(8),
    birth_date     DATE,
    balance        DECIMAL(7,2)
);
```

Notice that the only difference between this command and the command that you used to create the permanent `customers` table is the `TEMPORARY` keyword^[3]. Now you have two tables, each named `customers`. If you now `SELECT` from or `INSERT` into the `customers` table, you will be working with the temporary table. Prior to version 7.3, there was no way to get back to the permanent table except by dropping the temporary table:

^[3] You can abbreviate `TEMPORARY` to `TEMP`.

Code View: [Scroll](#) / [Show All](#)

```
movies=# SELECT * FROM customers;
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
      1 | Jones, Henry | 555-1212 | 1970-10-10 | 0.00
      2 | Rubin, William | 555-2211 | 1972-07-10 | 15.00
      3 | Panky, Henry | 555-1221 | 1968-01-21 | 0.00
      4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 | 3.00
      8 | Wink Wankel | 555-1000 | 1988-12-25 | 0.00
(5 rows)

movies=# CREATE TEMPORARY TABLE customers
movies=# (
movies(# customer_id    INTEGER UNIQUE,
movies(# customer_name  VARCHAR(50),
movies(# phone          CHAR(8),
movies(# birth_date     DATE,
movies(# balance        DECIMAL(7,2)
movies(# );
CREATE
```

```

movies=# SELECT * FROM customers;
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
(0 rows)

movies=# DROP TABLE customers;
DROP

movies=# SELECT * FROM customers;
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
          1 | Jones, Henry | 555-1212 | 1970-10-10 | 0.00
          2 | Rubin, William | 555-2211 | 1972-07-10 | 15.00
          3 | Panky, Henry | 555-1221 | 1968-01-21 | 0.00
          4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 | 3.00
          8 | Wink Wankel | 555-1000 | 1988-12-25 | 0.00
(5 rows)

```

Starting with release 7.3, you can access the permanent table by including the name of the schema where the permanent table resides.

A temporary table is like a scratch pad. You can use a temporary table to accumulate intermediate results. Quite often, you will find that a complex query can be formulated more easily by first extracting the data that interests you into a temporary table. If you find that you are creating a given temporary table over and over again, you might want to convert that table into a view. See the section titled "Using Views" in [Chapter 1](#) for more information about views.

Table Constraints

In [Chapter 2](#), "Working with Data in PostgreSQL," we explored the various constraints that you can apply to a column: NOT NULL, UNIQUE, PRIMARY KEY, REFERENCES, and CHECK(). You can also apply constraints to a table as a whole or to groups of columns within a table.

First, let's look at the CHECK() constraint. The syntax for a CHECK() constraint is

```
[CONSTRAINT constraint-name] CHECK( boolean-expression )
```

When you define a CHECK() constraint for a table, you are telling PostgreSQL that any insertions or updates made to the table must satisfy the boolean-expression given within the constraint. The difference between a column constraint and a table constraint is that a column constraint should refer only to the column to which it relates. A table constraint can refer to any column in the table.

For example, suppose that you had an `orders` table to track customer orders:

```

CREATE TABLE orders
(
    customer_number    INTEGER,
    part_number        CHAR(8),
    quantity_ordered   INTEGER,
    price_per_part     DECIMAL(7,2)
);

```

You could create a table-related CHECK() constraint to ensure that the extended price (that is, `quantity_ordered` times `price_per_part`) of any given order is at least \$5.00:

```

CREATE TABLE orders
(
    customer_number    INTEGER,
    part_number        CHAR(8),
    quantity_ordered   INTEGER,
    price_per_part     DECIMAL(7,2),

    CONSTRAINT verify_minimum_order
        CHECK (( price_per_part * quantity_ordered) >= 5.00::DECIMAL )
);

```

Each time a row is inserted into the `orders` table (or the `quantity_ordered` or `price_per_part` columns are updated), the `verify_minimum_order` constraint is evaluated. If the expression evaluates to FALSE, the modification is rejected. If the expression evaluates to TRUE or NULL, the modification is allowed.

You may have noticed that a table constraint looks very much like a column constraint. PostgreSQL can tell the difference between the two types by their placement within the CREATE TABLE statement. A column constraint is placed within a column definition—after the column's data type and before the comma. A table constraint is listed outside of a column definition. The only tricky spot is a table constraint that follows the last column definition; you normally would not include a comma after the last column. If you want a constraint to be treated as a table constraint, be sure to include a comma following the last column definition. At the moment, PostgreSQL does not treat table constraints and column constraints differently, but in a future release it may.

Each of the table constraint varieties is related to a type of column constraint.

The UNIQUE table constraint is identical to the UNIQUE column constraint, except that you can specify that a group of columns must be unique. For example, here is the `rentals` table as currently defined:

```

CREATE TABLE rentals
(
    tape_id            CHARACTER(8),
    customer_id        INTEGER,
    rental_date        DATE
);

```

Let's modify this table to reflect the business rule that any given tape cannot be rented twice on the same day:

```
CREATE TABLE rentals
(
    tape_id      CHARACTER(8),
    customer_id  INTEGER,
    rental_date  DATE,

    UNIQUE( rental_date, tape_id )
);
```

Now when you insert a row into the `rentals` table, PostgreSQL will ensure that there are no other rows with the same combination of `rental_date` and `tape_id`. Notice that I did not provide a constraint name in this example; constraint names are optional.

The `PRIMARY KEY` table constraint is identical to the `PRIMARY KEY` column constraint, except that you can specify that the key is composed of a group of columns rather than a single column.

The `REFERENCES` table constraint is similar to the `REFERENCES` column constraint. When you create a `REFERENCES` column constraint, you are telling PostgreSQL that a column value in one table refers to a row in another table. More specifically, a `REFERENCES` column constraint specifies a relationship between two columns. When you create a `REFERENCES` table constraint, you can relate a group of columns in one table to a group of columns in another table. Quite often, you will find that the unique identifier for a table (that is, the `PRIMARY KEY`) is composed of multiple columns. Let's say that the Virtual Video Store is having great success and you decide to open a second store. You might want to consolidate the data for each store into a single database. Start by creating a new table:

```
CREATE TABLE stores
(
    store_id     INTEGER PRIMARY KEY,
    location     VARCHAR
);
```

Now, change the definition of the `customers` table to include a `store_id` for each customer:

```
CREATE TABLE customers (
    store_id     INTEGER REFERENCES stores( store_id ),
    customer_id  INTEGER UNIQUE,
    customer_name VARCHAR(50),
    phone        CHAR(8),
    birth_date   DATE,
    balance      DECIMAL(7,2),

    PRIMARY KEY( store_id, customer_id )
);
```

The `store_id` column in the `customers` table refers to the `store_id` column in the `stores` table. Because `store_id` is the primary key to the `stores` table, you could have written the `REFERENCES` constraint in either of two ways:

```
store_id INTEGER REFERENCES stores( store_id )
```

or

```
store_id INTEGER REFERENCES stores
```

Also, notice that the primary key for this table is composed of two columns: `store_id` and `customer_id`. I can have two customers with the same `customer_id` as long as they have different `store_ids`.

Now you have to change the `rentals` table as well:

```
CREATE TABLE rentals
(
    store_id     INTEGER,
    tape_id      CHARACTER(8),
    customer_id  INTEGER,
    rental_date  DATE,

    UNIQUE( rental_date, tape_id )
    FOREIGN KEY( store_id, customer_id ) REFERENCES customers
);
```

The `customers` table has a two-part primary key. Each row in the `rentals` table refers to a row in the `customers` table, so the `FOREIGN KEY` constraint must specify a two-part foreign key. Again, because foreign key refers to the primary key of the `customers` table, I can write this constraint in either of two forms:

```
FOREIGN KEY( store_id, customer_id )
    REFERENCES customers( store_id, customer_id )
```

or

```
FOREIGN KEY( store_id, customer_id )
    REFERENCES customers
```

Now that I have the referential integrity constraints defined, they will behave as described in the "[Column Constraints](#)" section of [Chapter 2](#). Remember, a

table constraint functions the same as a column constraint, except that table constraints can refer to more than one column.

Dropping Tables

Dropping a table is much easier than creating a table. The syntax for the `DROP TABLE` command is

```
DROP TABLE table-name [, ...];
```

If you are using PostgreSQL 7.3 or later, you can qualify the table name with a schema. For example, here is the command to destroy the `rentals` table:

```
DROP TABLE rentals;
```

If the `rentals` table existed in some schema other than your current schema, you would qualify the table name:

```
DROP TABLE sheila.rentals;
```

You can destroy a table only if you are the table's owner or if you are a PostgreSQL superuser. Notice that I used the word destroy here rather than drop. It's important to realize that when you execute a `DROP TABLE` command, you are destroying all the data in that table.

PostgreSQL has a nice feature that I have not seen in other databases: You can roll back a `DROP TABLE` command. Try the following experiment. First, let's view the contents of the `tapes` table:

```
movies=# SELECT * FROM tapes;
```

```
tape_id |      title      | duration
-----+-----+-----
AB-12345 | The Godfather |
AB-67472 | The Godfather |
MC-68873 | Casablanca   |
OW-41221 | Citizen Kane  |
AH-54706 | Rear Window   |
(5 rows)
```

Now, start a multistatement transaction and destroy the `tapes` table:

```
movies=# BEGIN WORK;
BEGIN
```

```
movies=# DROP TABLE tapes;
```

```
NOTICE: DROP TABLE implicitly drops referential integrity trigger
        from table "rentals"
DROP
```

If you try to `SELECT` from the `tapes` table, you'll find that it has been destroyed:

```
movies=# SELECT * FROM tapes;
ERROR:  Relation "tapes" does not exist
```

If you `COMMIT` this transaction, the table will permanently disappear; let's `ROLLBACK` the transaction instead:

```
movies=# ROLLBACK;
ROLLBACK
```

The `ROLLBACK` threw out all changes made since the beginning of the transaction, including the `DROP TABLE` command. You should be able to `SELECT` from the `tapes` table again and see the same data that was there before:

```
movies=# SELECT * FROM tapes;
tape_id |      title      | duration
-----+-----+-----
AB-12345 | The Godfather |
AB-67472 | The Godfather |
MC-68873 | Casablanca   |
OW-41221 | Citizen Kane  |
AH-54706 | Rear Window   |
(5 rows)
```

This is a very nice feature. You can roll back `CREATE TABLE`, `DROP TABLE`, `CREATE VIEW`, `DROP VIEW`, `CREATE INDEX`, `DROP INDEX`, and so on. I'll discuss transactions a bit later in this chapter. For now, I'd like to point out a few details that I glossed over in the previous example. You may have noticed that the `DROP TABLE` command produced a `NOTICE`.

```
movies=# DROP TABLE tapes;
NOTICE: DROP TABLE implicitly drops referential integrity trigger
        from table "rentals"
DROP
```

When you drop a table, PostgreSQL will automatically `DROP` any indexes defined for that table as well as any triggers or rules. If other tables refer to the table that you dropped (by means of a `REFERENCE` constraint), PostgreSQL will automatically drop the constraints in the other tables. However, any views that refer to the dropped table will not be removed—a view can refer to many tables and PostgreSQL would not know how to remove a single table from a multitable `SELECT`.

Inheritance

Another PostgreSQL feature that is uncommon in relational database systems is inheritance. Inheritance is one of the foundations of the object-oriented programming paradigm. Using inheritance, you can define a hierarchy of related data types (in PostgreSQL, you define a hierarchy of related tables). Each layer in the inheritance hierarchy represents a specialization of the layer above it^[4].

^[4] We'll view an inheritance hierarchy with the most general types at the top and the most specialized types at the bottom.

Let's look at an example. The Virtual Video database defines a table that stores information about the tapes that you have in stock:

```
movies=# \d tapes
      Column |          Type          | Modifiers
-----+-----+-----
 tape_id    | character(8)           | not null
 title      | character varying(80) | not null
 duration   | interval               |
```

For each tape, you store the `tape_id`, `title`, and `duration`. Let's say that you decide to jump into the twenty-first century and rent DVDs as well as videotapes. You could store DVD records in the `tapes` table, but a tape and a DVD are not really the same thing. Let's create a new table that defines the characteristics common to both DVDs and videotapes:

```
CREATE TABLE video
(
    video_id      CHARACTER(8) PRIMARY KEY,
    title         VARCHAR(80),
    duration      INTERVAL
);
```

Now, create a table to hold the DVDs. For each DVD you have in stock, you want to store everything in the `video` table plus a `region_id` and an array of `audio_tracks`. Here is the new table definition:

```
movies=# CREATE TABLE dvds
movies=# (
movies(#   region_id    INTEGER,
movies(#   audio_tracks VARCHAR[]
movies(# ) INHERITS ( video );
```

Notice the last line in this command: You are telling PostgreSQL that the `dvds` table inherits from the `video` table. Now let's `INSERT` a new DVD:

```
movies=# INSERT INTO dvds VALUES
movies=# (
movies(#   'ASIN-750',           -- video_id
movies(#   'Star Wars',         -- title
movies(#   '121 minutes',       -- duration
movies(#   1,                   -- region_id
movies(#   '{English,Spanish}'  -- audio_tracks
movies(# );
```

Now, if you `SELECT` from the `dvds` table, you'll see the information that you just inserted:

```
 video_id | title | duration | region_id | audio_tracks
-----+-----+-----+-----+-----
 ASIN-750 | Star Wars | 02:01:00 |          1 | {English,Spanish}
```

At this point, you might be thinking that the `INHERITS` clause did nothing more than create a row template that PostgreSQL copied when you created the `dvds` table. That's not the case—if you simply want to create a table that has the same structure as another table, use the `LIKE` table-name clause instead of the `INHERITS` table-name clause. When we say that `dvds` inherits from `video`, we are not simply saying that a DVD is like a video, we are saying that a DVD is a video. Let's `SELECT` from the `video` table now; remember, you haven't explicitly inserted any data into the `video` table, so you might expect the result set to be empty:

```
movies=# SELECT * FROM video;
 video_id | title | duration
-----+-----+-----
 ASIN-750 | Star Wars | 02:01:00
```

A DVD is a video. When you `SELECT` from the `video` table, you see only the columns that comprise a `video`. When you `SELECT` from the `dvds` table, you see all the columns that comprise a DVD. In this relationship, you say that the `dvds` table specializes^[5] the more general `video` table.

^[5] Object-oriented terminology defines many different phrases for this inheritance relationship: `specialize/generalize`, `subclass/superclass`, and so on. Choose the phrase that you like.

If you are using a version of PostgreSQL older than 7.2, you must code this query as `SELECT * FROM video*` to see the DVD entries. Starting with release 7.2, `SELECT` will include descendent tables and you have to say `SELECT * FROM ONLY video` to suppress descendents.

You now have a new table to track your DVD inventory; let's go back and redefine the `tapes` table to fit into the inheritance hierarchy. For each tape, we want to store a `video_id`, a `title`, and a `duration`. This is where we started: the `video` table already stores all this information. You should still create a new table to track videotapes—at some point in the future, you may find information that relates to a videotape, but not to a DVD:

```
movies=# CREATE TABLE tapes ( ) INHERITS( video );
CREATE
```

This `CREATE TABLE` command creates a new table identical in structure to the `video` table. Each row in the `tapes` table will contain a `video_id`, a `title`,

and a duration. Insert a row into the tapes table:

```
movies=# INSERT INTO tapes VALUES
movies=# (
movies(# 'ASIN-8YD',
movies(# 'Flight To Mars(1951)',
movies(# '72 min'
movies(# );
INSERT
```

When you SELECT from the tapes table, you should see this new row:

```
movies=# SELECT * FROM tapes;
 tape_id | title | duration
-----+-----+-----
 ASIN-8YD | Flight To Mars(1951) | 01:12:00
(1 row)
```

And because a tape is a video, you would also expect to see this row in the video table:

```
movies=# SELECT * FROM video;
 video_id | title | duration
-----+-----+-----
 ASIN-750 | Star Wars | 02:01:00
 ASIN-8YD | Flight To Mars(1951) | 01:12:00
(2 rows)
```

Now here's the interesting part. A DVD is a video—any row that you add to the dvds table shows up in the video table. A tape is a video—any row that you add to the tapes table shows up in the video table. But a DVD is not a tape (and a tape is not a DVD). Any row that you add to the dvds table will not show up in the tapes table (and vice versa).

If you want a list of all the tapes you have in stock, you can SELECT from the tapes table. If you want a list of all the DVDs in stock, SELECT from the dvds table. If you want a list of all videos in stock, SELECT from the videos table.

In this example, the inheritance hierarchy is only two levels deep. PostgreSQL imposes no limit to the number of levels that you can define in an inheritance hierarchy. You can also create a table that inherits from multiple tables—the new table will have all the columns defined in the more general tables.

I should caution you about two problems with the current implementation of inheritance in PostgreSQL. First, indexes are not shared between parent and child tables. On one hand, that's good because it gives you good performance. On the other hand, that's bad because PostgreSQL uses an index to guarantee uniqueness. That means that you could have a videotape and a DVD with the same video_id. Of course, you can work around this problem by encoding the type of video in the video_id (for example, use a T for tapes and a D for DVDs). But PostgreSQL won't give you any help in fixing this problem. The other potential problem with inheritance is that triggers are not shared between parent and child tables. If you define a trigger for the topmost table in your inheritance hierarchy, you will have to remember to define the same trigger for each descendant.

We have redefined some of the example tables many times in the past two chapters. In a real-world environment, you probably won't want to throw out all your data each time you need to make a change to the definition of an existing table. Let's explore a better way to alter a table.

ALTER TABLE

Now that you have a video table, a dvds table, and a tapes table, let's add a new column to all three tables that you can use to record the rating of the video (PG, G, R, and so on).

You could add the rating column to the tapes table and to the dvds table, but you really want the rating column to be a part of every video. The ALTER TABLE ... ADD COLUMN command adds a new column for you, leaving all the original data in place:

```
movies=# ALTER TABLE video ADD COLUMN rating VARCHAR;
ALTER
```

Now, if you look at the definition of the video table, you will see the new column:

```
movies=# \d video
          Table "video"
  Column | Type | Modifiers
-----+-----+-----
 video_id | character(8) | not null
 title | character varying(80) |
 duration | interval |
 rating | character varying |
Indexes:
    "video_pkey" PRIMARY KEY, btree (video_id)
```

After the ALTER TABLE command completes, each row in the video table has a new column; the value of every rating column will be NULL. Because you have changed the definition of a video, and a DVD is a video, you might expect that the dvds table will also contain a rating column:

```
movies=# \d dvds
          Table "dvds"
  Column | Type | Modifiers
-----+-----+-----
 video_id | character(8) | not null
 title | character varying(80) |
 duration | interval |
 region_id | integer |
 audio_tracks | character varying[] |
 rating | character varying |
Inherits: video
```

Similarly, the `tapes` table will also inherit the new `rating` column:

```
movies=# \d tapes
          Table "tapes"
   Column |          Type          | Modifiers
-----+-----+-----
 video_id | character(8)           | not null
  title   | character varying(80) |
 duration | interval               |
  rating  | character varying      |
Inherits: video
```

Starting with PostgreSQL version 8.0, you can change the data type of an existing column using `ALTER TABLE`. For example, to change the data type of the `customers.customer_id` column from `INTEGER` to `NUMERIC(7, 2)`, you could execute the command:

```
ALTER TABLE customers ALTER COLUMN customer_id TYPE NUMERIC( 7,2 )
```

As long as PostgreSQL knows how to convert a value from the old data type to the new data type, you can freely change data types. If PostgreSQL doesn't know how to convert between the old and new types, you can include a `USING` expression clause to tell PostgreSQL how to perform the conversion. The expression following the `USING` keyword typically refers to the original column value. For example, if you want to change the data type of `customers.customer_id` and multiply each `customer_id` by 100 at the same time, use the following command:

Code View: [Scroll](#) / [Show All](#)

```
ALTER TABLE customers ALTER COLUMN customer_id TYPE NUMERIC( 7,2 ) USING customer_id * 100
```

You can also refer to other columns in the `USING` expression. For example, say that you are currently storing each customer name in two columns, `last_name` and `first_name`, and you've decided to combine them into a single column named `customer_name`. You can do that with the following commands:

```
movies=# ALTER TABLE customers
movies=#     ALTER COLUMN last_name
movies=#         TYPE VARCHAR USING ( last_name || ',' || first_name ),
movies=#     DROP COLUMN first_name;
ALTER TABLE

movies=# ALTER TABLE customers
movies=#     RENAME COLUMN last_name TO customer_name;
ALTER TABLE
```

The first `ALTER TABLE` command performs two alterations. First, for each row in the table, it evaluates the expression `last_name || ',' || first_name` and assigns that value to the `last_name` column (converting the result into type `VARCHAR` along the way). Next, the (first) `ALTER TABLE` command removes the `first_name` column from each row. You're left with a single column called `last_name` that contains the concatenation of the original `last_name` and `first_name` columns (with a comma in between). The second `ALTER TABLE` command renames the `last_name` column to `customer_name`.

Keep in mind that some `ALTER TABLE` commands will take longer to execute than others. It takes very little time to change the name of a column. It can take quite a while to change the data type of a column (because PostgreSQL has to traverse every row in the table and write out a new version). If you use `ALTER TABLE ... SET TABLESPACE` to move a table from one tablespace to another, the server must physically copy each block in the table. In most cases, it's faster to execute a series of `ALTER TABLE` commands than it is to read the old data into a client application, change each row, and then write the result back to the server. When you use an `ALTER TABLE` command, the entire transformation occurs within the server; if you modify the structure of a table using a custom-written client application, you have to send every row to the client, perform the transformation, and then send every row back to the server.

The `ALTER TABLE` command is useful when you are in the development stages of a project. Using `ALTER TABLE`, you can add new columns to a table, define default values, rename columns (and tables), add and drop constraints, change the data type of a column, and transfer ownership. The capabilities of the `ALTER TABLE` command seem to grow with each new release—see the PostgreSQL Reference Manual for more details.

Adding Indexes to a Table

Most of the tables that you have created so far have no indexes. An index serves two purposes. First, an index can be used to guarantee uniqueness. Second, an index provides quick access to data (in certain circumstances).

Here is the definition of the `customers` table that you created in [Chapter 1](#):

```
CREATE TABLE customers (  
    customer_id    INTEGER UNIQUE,  
    customer_name  VARCHAR(50),  
    phone          CHAR(8),  
    birth_date     DATE,  
    balance        DECIMAL(7,2)  
);
```

When you create this table, PostgreSQL will display a rather terse message:

```
NOTICE: CREATE TABLE / UNIQUE will create implicit  
index 'customers_customer_id_key' for table 'customers'
```

What PostgreSQL is trying to tell you here is that even though you didn't explicitly ask for one, an index has been created on your behalf. The implicit index is created so that PostgreSQL has a quick way to ensure that the values that you enter into the `customer_id` column are unique.

Think about how you might design an algorithm to check for duplicate values in the following list of names:

Grumby, Jonas

Hinkley, Roy

Wentworth, Eunice

Floyd, Heywood

Bowman, David

Dutton, Charles

Poole, Frank

Morbius, Edward

Farman, Jerry

Stone, Jeremy

Dutton, Charles

Manchek, Arthur

A first attempt might simply start with the first value and look for a duplicate later in the list, comparing `Grumby, Jonas` to `Hinkley, Roy`, then `Wentworth, Eunice`, and so on. Next, you would move to the second name in the list and compare `Hinkley, Roy` to `Wentworth, Eunice`, then `Floyd, Heywood`, and so on. This algorithm would certainly work, but it would turn out to be slow as the list grew longer. Each time you add a new name to the list, you have to compare it to every other name already in the list.

A better solution would be to first sort the list:

Bowman, David

Dutton, Charles

Dutton, Charles

Farman, Jerry

Floyd, Heywood

Grumby, Jonas

Hinkley, Roy
Manchek, Arthur
Morbius, Edward
Poole, Frank
Stone, Jeremy
Wentworth, Eunice

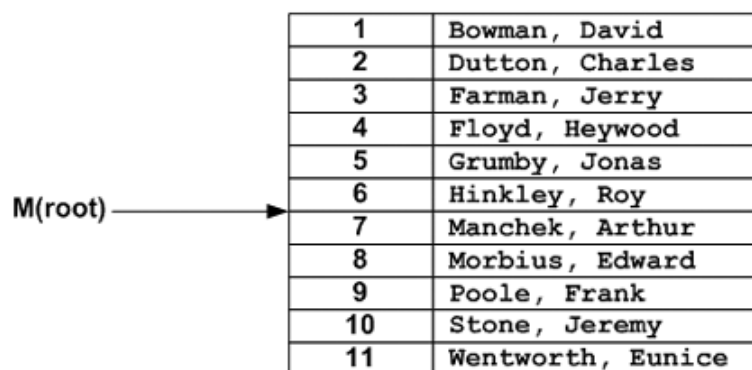
After the list is sorted, it's easy to check for duplicates—any duplicate values appear next to each other. To check the sorted list, you start with the first name, *Bowman, David* and compare it to the second name, *Dutton, Charles*. If the second name is not a duplicate of the first, you know that you won't find any duplicates later in the list. Now when you move to the second name on the list, you compare it to the third name—now you can see that there is a duplicate. Duplicate values appear next to each other after the list is sorted. Now when you add a new name to the list, you can stop searching for duplicate values as soon as you encounter a value that sorts after the name you are adding.

An index is similar in concept to a sorted list, but it's even better. An index provides a quick way for PostgreSQL to find data within a range of values. Let's see how an index can help narrow a search. First, let's assign a number to each of the names in the sorted list, just for easy reference (I've removed the duplicate value):

1. Bowman, David
2. Dutton, Charles
3. Farman, Jerry
4. Floyd, Heywood
5. Grumby, Jonas
6. Hinkley, Roy
7. Manchek, Arthur
8. Morbius, Edward
9. Poole, Frank
10. Stone, Jeremy
11. Wentworth, Eunice

Now let's build a (simplistic) index (see [Figure 3.2](#)). The English alphabet contains 26 letters—split this roughly in half and choose to keep track of where the "Ms" start in the list. In this list, names beginning with an *M* start at entry number 7. Keep track of this pair (*M*,7) and call it the root of your index.

Figure 3.2. One-level index.

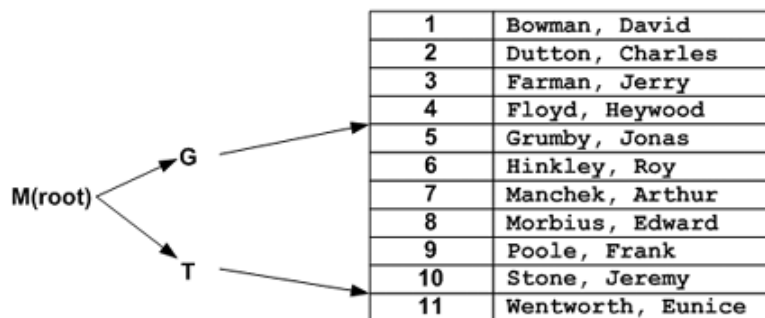


1	Bowman, David
2	Dutton, Charles
3	Farman, Jerry
4	Floyd, Heywood
5	Grumby, Jonas
6	Hinkley, Roy
7	Manchek, Arthur
8	Morbius, Edward
9	Poole, Frank
10	Stone, Jeremy
11	Wentworth, Eunice

Now when you insert a new name, *Tyrell, Eldon*, you start by comparing it to the root. The root of the index tells you that names starting with the letter *M* are found starting at entry number 7. Because the list is sorted, and you know that *Tyrell* will sort after *M*, you can start searching for the insertion point at entry 7, skipping entries 1 through 6. Also, you can stop searching as soon as you encounter a name that sorts later than *Tyrell*.

As your list of names grows, it would be advantageous to add more levels to the index (see [Figure 3.3](#)). The letter **M** splits the alphabet (roughly) in half. Add a second level to the index by splitting the range between **A** and **M** (giving you **G**), and splitting the range between **M** and **Z** (giving you **T**).

Figure 3.3. Two-level index.



Now when you want to add **Tyrell, Eldon** to the list, you compare **Tyrell** against the root and find that **Tyrell** sorts later than **M**. Moving to the next layer of the index, you find that **Tyrell** sorts later than **T**, so you can jump straight to slot number 11 and insert the new value.

You can see that you can add as many index levels as you need. Each level divides the parent's range in half, and each level reduces the number of names that you have to search to find an insertion point^[6].

^[6] Technically speaking, the index diagrams discussed here depict a clustered index. In a clustered index, the leaf nodes in the index tree are the data rows themselves. In a non-clustered index, the leaf nodes are actually row pointers—the rows are not kept in sorted order. PostgreSQL does not support clustered indexes. I've diagrammed the index trees in clustered form for clarity. A clustered index provides fast, sequential access along one index path, but it is very expensive to maintain.

Using an index is similar in concept to the way you look up words in a dictionary. If you have a dictionary handy, pull it off the shelf and take a close look at it. If it's like my dictionary, it has those little thumb-tab indentations, one for each letter of the alphabet. If I want to find the definition of the word "polyglot," I'll find the thumb-tab labeled "P" and start searching about halfway through that section. I know, because the dictionary is sorted, that "polyglot" won't appear in any section prior to "P" and it won't appear in any section following "P." That little thumb-tab saves a lot of searching.

You also can use an index as a quick way to check for uniqueness. If you are inserting a new name into the index structure shown earlier, you simply search for the new name in the index. If you find it in the index, it is obviously a duplicate.

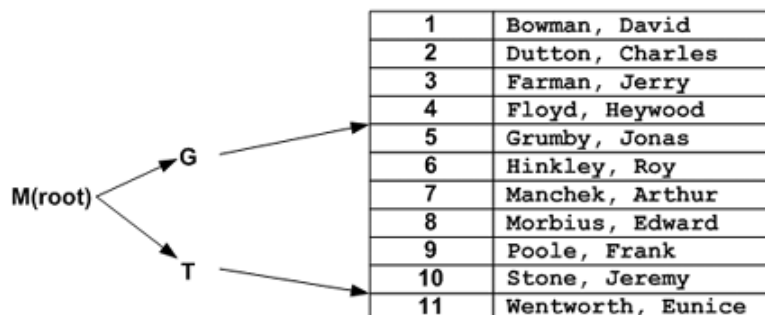
I mentioned earlier that PostgreSQL uses an index for two purposes. You've seen that an index can be used to search for unique values. But how does PostgreSQL use an index to provide faster data access?

Let's look at a simple query:

```
SELECT * FROM characters WHERE name >= 'Grumby' AND name < 'Moon';
```

Now assume that the list of names that you worked with before is actually a table named `characters` and you have an index defined for the `name` column, as in [Figure 3.4](#).

Figure 3.4. Two-level index (again).



When PostgreSQL parses through the `SELECT` statement, it notices that you are constraining the result set to a range of names and that you have an index on the `name` column. That's a convenient combination. To satisfy this statement, PostgreSQL can use the index to start searching at entry number 5. Because the rows are already sorted, PostgreSQL can stop searching as soon as it finds the first entry greater than "Moon" (that is, the search ends as soon as you hit entry number 8). This kind of operation is called a partial index scan.

Think of how PostgreSQL would process this query if the rows were not indexed. It would have to start at the beginning of the table and compare each row against the constraints; PostgreSQL can't terminate the search without processing every row in the table. This kind of operation is called a full table scan, or table scan.

Because this kind of index can access data in sorted order, PostgreSQL can use such an index to avoid a sort that would otherwise be required to satisfy an `ORDER BY` clause.

In these examples, we are working with small tables, so the performance difference between a full table scan and an indexed range read is negligible. As tables become larger, the performance difference can be huge. [Chapter 4](#), "Performance," discusses how the PostgreSQL query optimizer chooses when it is appropriate to use an index.

PostgreSQL actually supports several kinds of indexes. The previous examples show how a B-Tree index works^[7]. Another type of index is the Hash index. A Hash index uses a technique called hashing to evenly distribute keys among a number of hash buckets. Each key value added to a hash index is run through a hashing function. The result of a hashing function is a bucket number. A simplistic hashing function for string values might sum the ASCII value of each character in the string and then compute the sum modulo the number of buckets to get the result. In C, you might write this function as

[7] The "B" in B-Tree stands for "Balanced." A balanced tree is a type of data structure that retains its performance characteristics even in the face of numerous insertions and deletions. The most important feature of a B-Tree is that it takes about the same amount of time to find any given record.

```
int hash_string( char * key, int bucket_count )
{
    int hash = 0;
    int i;

    for( i = 0; i < strlen( key ); i++ )
        hash = hash + key[i];

    return( hash % bucket_count );
}
```

Let's run each of the names in the `characters` table through this function to see what kind of numbers you get back (I've used a `bucket_count` of 5):

<code>hash_string()</code> Value	Name
1	Grumby, Jonas
2	Hinkley, Roy
3	Wentworth, Eunice
4	Floyd, Heywood
4	Bowman, David
3	Dutton, Charles
3	Poole, Frank
0	Morbius, Edward
0	Farman, Jerry
0	Stone, Jeremy
4	Manchek, Arthur

The numbers returned don't really have any intrinsic meaning, they simply serve to distribute a set of keys amongst a set of buckets.

Now let's reformat this table so that the contents are grouped by bucket number:

Bucket Number	Bucket Contents
0	Morbius, Edward
	Farman, Jerry
	Stone, Jeremy
1	Grumby, Jonas
2	Hinkley, Roy
3	Wentworth, Eunice
Bucket Number	Bucket Contents

Dutton, Charles

Poole, Frank

Floyd, Heywood

Bowman, David

Manchek, Arthur

You can see that the hash function (`hash_string()`) did a respectable job of distributing the names between the five hash buckets. Notice that we did not have to assign a unique hash value to each key—hash keys are seldom unique. The important feature of a good hash function is that it distributes a set of keys fairly evenly. Now that you have a Hash index, how can you use it? First, let's try to insert a new name: `Lowell, Freeman`. The first thing you do is run this name through your `hash_string()` function, giving you a hash value of 4. Now you know that if `Lowell, Freeman` is already in the index, it will be in bucket number 4; all you have to do is search that one bucket for the name you are trying to insert.

There are a couple of important points to note about Hash indexes.

First, you may have noticed that each bucket can hold many keys. Another way to say this is that each key does not have a unique hash value. If you have too many collisions (that is, too many keys hashing to the same bucket), performance will suffer. A good hash function distributes keys evenly between all hash buckets.

Second, notice that a hash table is not sorted. The name `Floyd, Heywood` hashes to bucket 4, but `Farman, Jerry` hashes to bucket 0. Consider the `SELECT` statement that we looked at earlier:

```
SELECT * FROM characters WHERE name >= 'Grumby' AND name < 'Moon';
```

To satisfy this query using a Hash index, you have to read the entire contents of each bucket. Bucket 0 contains one row that meets the constraints (`Farman, Jerry`), bucket 2 contains one row, and bucket 4 contains one row. A Hash index offers no advantage to a range read. A Hash index is good for searches based on equality. For example, the `SELECT` statement

```
SELECT * FROM characters WHERE name = 'Grumby, Jonas';
```

can be satisfied simply by hashing the string that you are searching for. A Hash index is also useful when you are joining two tables where the join constraint is of the form `table1-column = table2-column`^[8]. A Hash read cannot be used to avoid a sort required to satisfy an `ORDER BY` clause.

^[8] This type of join is known as an equi-join.

PostgreSQL supports two other types of index structures: the R-Tree index and the GiST index. An R-Tree index is best suited for indexing spatial (that is, geometric or geographic) data. A GiST index is a B-Tree index that can be extended by defining new query predicates^[9]. More information about GiST indexes can be found at <http://gist.cs.berkeley.edu/>.

^[9] A predicate is a test. A simple predicate is the less-than operator (`<`). An expression such as `a < 5` tests whether the value of `a` is less than 5. In this expression, `<` is the predicate and it is called the less-than predicate. Other predicates are `=`, `>`, `>=`, and so on.

Tradeoffs

The previous section showed that PostgreSQL can use an index to speed the process of searching for data within a range of values (or data with an exact value). Most queries (that is, `SELECT` commands) in PostgreSQL include a `WHERE` clause to limit the result set. If you find that you are often searching for results based on a range of values for a specific column or group of columns, you might want to consider creating an index that covers those columns.

However, you should be aware that an index represents a performance tradeoff. When you create an index, you are trading read performance for write performance. An index can significantly reduce the amount of time it takes to retrieve data, but it will also increase the amount of time it takes to `INSERT`, `DELETE`, and `UPDATE` data. Maintaining an index introduces substantial overhead when you modify the data within a table.

You should consider this tradeoff when you feel the need to add a new index to a table. Adding an index to a table that is updated frequently will certainly slow the updates. A good candidate for an index is a table that you `SELECT` from frequently but seldom update. A customer list, for example, doesn't change often (possibly several times each day), but you probably query the customer list frequently. If you find that you often query the customer list by phone number, it would be beneficial to index the phone number column. On the other hand, a table that is updated frequently, but seldom queried, such as a transaction history table, would be a poor choice for an index.

Creating an Index

Now that you have seen what an index can do, let's look at the process of adding an index to a table. The process of creating a new

index can range from simple to somewhat complex.

Let's add an index to the `rentals` table. Here is the structure of the `rentals` table for reference:

```
CREATE TABLE rentals
(
    tape_id      CHARACTER(8) REFERENCES tapes,
    customer_id  INTEGER REFERENCES customers,
    rental_date  DATE
);
```

The syntax for a simple `CREATE INDEX` command is

```
CREATE [UNIQUE] INDEX index-name ON table-name( column [,...] );
```

You want to index the `rental_date` column in the `rentals` table:

```
CREATE INDEX rentals_rental_date ON rentals ( rental_date );
```

You haven't specified any optional information in this command (I'll get to the options in a moment), so PostgreSQL creates a B-Tree index named `rentals_rental_date`. PostgreSQL considers using this whenever it finds a `WHERE` clause that refers to the `rental_date` column using the `<`, `<=`, `=`, `>=`, or `>` operator. This index also can be used when you specify an `ORDER BY` clause that sorts on the `rental_date` column.

Multicolumn Indexes

A B-Tree index (or a GiST index) can cover more than one column. Multicolumn indexes are usually created when you have many values on the second column for each value in the first column. For example, you might want to create an index that covers the `rental_date` and `tape_id` columns—you have many different tapes rented on any given date. PostgreSQL can use multicolumn indexes for selection or for ordering. When you create a multicolumn index, the order in which you name the columns is important. PostgreSQL can use a multicolumn index when you are selecting (or ordering by) a prefix of the key. In this context, a prefix may be the entire key or a leading portion of the key. For example, the command `SELECT * FROM rentals ORDER BY rental_date` could not use an index that covers `tape_id` plus `rental_date`, but it could use an index that covers `rental_date` plus `tape_id`.

The `index-name` must be unique within the database: You can't have two indexes with the same name, even if they are defined on different tables. New rows are indexed as they are added, and deleted rows are removed. If you change the `rental_date` for a given row, the index will be updated automatically. If you have any data in the `rentals` table, each row will be included in the index.

Indexes and NULL Values

Earlier, I mentioned that an index includes a pointer for every row in a table. That statement isn't 100% accurate. PostgreSQL will not index `NULL` values in R-Tree, Hash, and GiST indexes. Because such an index will never include `NULL` values, it cannot be used to satisfy the `ORDER BY` clause of a query that returns all rows in a table. For example, if you define a GiST index covering the `phone` column in the `customers` table, that index would not include rows where `phone` was `NULL`. If you executed the command `SELECT * FROM customers ORDER BY phone`, PostgreSQL would have to perform a full table scan and then sort the results. If PostgreSQL tried to use the `phone` index, it would not find all rows. If the `phone` column were defined as `NOT NULL`, then PostgreSQL could use the index to avoid a sort. Or, if the `SELECT` command included the clause `WHERE phone IS NOT NULL`, PostgreSQL could use the index to satisfy the `ORDER BY` clause. An R-Tree, Hash, or GiST index that covers an optional (that is, `NULLS-allowed`) column will not be used to speed table joins, either.

A B-Tree index (the default index type) does include `NULL` values.

If you don't specify an index type when creating an index, you'll get a B-Tree index. Let's change the `rentals_rental_date` index into a Hash index. First, drop the original index:

```
DROP INDEX rentals_rental_date;
```

Then you can create a new index:

```
CREATE INDEX rentals_rental_date ON rentals USING HASH ( rental_date );
```


The only difference between this `CREATE INDEX` command and the previous one is that I have included a `USING` clause. You can specify `USING BTREE` (which is the default), `USING HASH`, `USING RTREE`, or `USING GIST`.

This index cannot be used to satisfy an `ORDER BY` clause. In fact, this index can be used only when `rental_date` is compared using the `=` operator.

I dropped the B-Tree index before creating the Hash index, but that is not strictly necessary. It is perfectly valid (but unusual) to have two or more indexes that cover the same column, as long as the indexes are uniquely named. If we had both a B-Tree index and a Hash index covering the `rental_date` column, PostgreSQL could use the Hash index for `=` comparisons and the B-Tree index for other comparisons.

Functional Indexes and Partial Indexes

Now let's look at two variations on the basic index types: functional indexes and partial indexes.

A column-based index catalogs the values found in a column (or a set of columns). A functional index (or more precisely a function-valued index) catalogs the values returned by a given function. This might be easiest to understand by looking at an example. Each row in the `customers` table contains a phone number. You can use the exchange^[10] portion of the phone number to determine whether a given customer is located close to your store. For example, you may know that the 555, 556, and 794 exchanges are within five miles of your virtual video store. Let's create a function that extracts the exchange from a phone number:

[10] In the U.S., a phone number is composed of an optional three-digit area code, a three-digit exchange, and a four-digit number.

```
-- exchange_index.sql
--
CREATE OR REPLACE FUNCTION get_exchange( CHARACTER )
    RETURNS CHARACTER AS '

DECLARE
    result CHARACTER(3);
BEGIN

    result := SUBSTR( $1, 1, 3 );

    return( result );
END;
' LANGUAGE 'plpgsql' WITH ( ISCACHEABLE );
```

Don't be too concerned if this looks a bit confusing; I'll cover the PL/pgSQL language in more detail in [Chapter 7, "PL/pgSQL."](#) This function (`get_exchange()`) accepts a single argument, presumably a phone number, and extracts the first three characters. You can call this function directly from `psql`:

```
movies=# SELECT customer_name, phone, get_exchange( phone )
movies=# FROM customers;
```

customer_name	phone	get_exchange
Jones, Henry	555-1212	555
Rubin, William	555-2211	555
Panky, Henry	555-1221	555
Wonderland, Alice N.	555-1122	555
Wink Wankel	555-1000	555

You can see that given a phone number, `get_exchange()` returns the first three digits. Now let's create a function-valued index that uses this function:

```
CREATE INDEX customer_exchange ON customers ( get_exchange( phone ));
```

When you insert a new row into a column-based index, PostgreSQL will index the values in the columns covered by that index. When you insert a new row into a function-valued index, PostgreSQL will call the function that you specified and then index the return value.

After the `customer_exchange` index exists, PostgreSQL can use it to speed up queries such as

```
SELECT * FROM customers WHERE get_exchange( phone ) = '555';
SELECT * FROM customers ORDER BY get_exchange( phone );
```

Now you have an index that you can use to search the customer list for all customers that are geographically close. Let's pretend that you occasionally want to send advertising flyers to those customers closest to you: you might never use the `customer_exchange` index for any other purpose. If you need the `customer_exchange` index for only a small set of customers, why bother maintaining that index for customers outside of your vicinity? This is where a partial index comes in handy. When you create an index, you can include a `WHERE` clause in the `CREATE INDEX` command. Each time you insert (or update) a row, the `WHERE` clause

is evaluated. If a row satisfies the constraints of the `WHERE` clause, that row is included in the index; otherwise, the row is not included in the index. Let's `DROP` the `customer_exchange` index and replace it with a partial, function-valued index:

```
movies=# DROP INDEX customer_exchange;
DROP
movies=# CREATE INDEX customer_exchange
movies-#   ON customers ( get_exchange( phone ))
movies-#   WHERE
movies-#       get_exchange( phone ) = '555'
movies-#       OR
movies-#       get_exchange( phone ) = '556'
movies-#       OR
movies-#       get_exchange( phone ) = '794';
CREATE
```

Now the `customer_exchange` partial index contains entries only for customers in the 555, 556, or 794 exchange.

There are three performance advantages to a partial index:

- A partial index requires less disk space than a full index.
- Because fewer rows are cataloged in a partial index, the cost of maintaining the index is lower.
- When a partial index is used in a query, PostgreSQL will have fewer index entries to search.

Partial indexes and function-valued indexes are variations on the four basic index types. You can create a function-valued Hash index, B-Tree index, R-tree index, or GiST index. You can also create a partial variant of any index type. And, as you have seen, you can create partial function-valued indexes (of any type). A function-valued index doesn't change the organization of an index—just the values that are actually included in the index. The same is true for a partial index.

Creating Indexes on Array Values

Most indexes cover scalar-valued columns (columns that store a single value). PostgreSQL also allows you to define indexes that cover index values. In fact, you can create an index that covers the entire array or (starting with PostgreSQL version 7.4) an index that covers individual elements within an array. In [Chapter 2](#) we showed you a modified version of the `customers` table that included an array column (`monthly_balances`). You can add this column to your working copy of the `customers` table with the following command:

```
movies=# ALTER TABLE customers
movies-#   ADD COLUMN
movies-#       monthly_balances DECIMAL( 7, 2 )[ 12 ];
ALTER TABLE
```

To create an index that covers a single element of `monthly_balances` array (say, the element corresponding to the month of February), you could execute the following command:

```
movies=# CREATE INDEX customers_feb
movies-#   ON customers (( monthly_balances[2] ));
CREATE INDEX
```

Notice that you need an extra set of parentheses around `monthly_balances[2]`. Once you've created the `customers_feb` index, PostgreSQL can use it to satisfy queries such as

```
movies=# SELECT * FROM customers WHERE monthly_balances[2] = 10;
movies=# SELECT * FROM customers ORDER BY monthly_balances[2];
```

To create an index that covers the entire `monthly_balances` array, execute the command

```
movies=# CREATE INDEX customers_by_monthly_balance
movies-#   ON customers( monthly_balances );
CREATE INDEX
```

When you create an index that covers an array column, the syntax is the same as you would use to cover a scalar (single-valued) column. The PostgreSQL optimizer can use the `customers_by_monthly_balance` index to satisfy an `ORDER BY` clause such as

```
movies=# SELECT * FROM customers ORDER BY monthly_balances;
```

However, you may be surprised to find that the optimizer will not use `customers_by_monthly_balance` to satisfy a `WHERE` clause such as

```
movies=# SELECT * FROM customers WHERE monthly_balances[1] = 10;
```

The PostgreSQL optimizer will use the `customers_by_monthly_balance` index to satisfy a `WHERE_CLAUSE` that compares the entire `monthly_balances` array against another array, like this:

```
movies=# SELECT * FROM customers WHERE monthly_balances = '{10}';
```

But be aware that these queries are not equivalent. The first `WHERE` clause (`monthly_balances[1] = 10`) selects any row where `monthly_balances[1]` is equal to 10, regardless of the other `monthly_balances` in that row. The second `WHERE` clause (`monthly_balances = '{10}'`) selects only those rows where `monthly_balances[1] = 10` and all other `monthly_balances` values are `NULL`.

Indexes and Tablespaces

When you create an index, you can tell PostgreSQL to store the index in a specific tablespace by including a `TABLESPACE tablespace_name` clause, like this:

```
CREATE INDEX rentals_rental_date
ON rentals ( rental_date ) TABLESPACE mytablespace;
```

If you don't specify a tablespace, PostgreSQL creates the index in the tablespace assigned to the table that you are indexing. You can move an existing index to a different tablespace using the `ALTER INDEX` command. For example, to move the `rentals_rental_date` index to `mytablespace`, you would execute the command

```
ALTER INDEX rentals_rental_date SET TABLESPACE mytablespace;
```

You may want to store a table and its indexes in different tablespaces in order to spread the workload among multiple physical disk drives.

Getting Information About Databases and Tables

When you create a table, PostgreSQL stores the definition of that table in the system catalog. The system catalog is a collection of PostgreSQL tables. You can issue `SELECT` statements against the system catalog tables just like any other table, but there are easier ways to view table and index definitions.

When you are using the `psql` client application, you can view the list of tables defined in your database using the `\d` meta-command:

```
movies=# \d
               List of relations
   Name      | Type   | Owner
-----+-----+-----
 customers   | table  | bruce
 rentals     | table  | bruce
 tapes       | table  | bruce
```

To see the detailed definition of a particular table, use the `\d table-name` meta-command:

```
movies=# \d tapes
               Table "tapes"
  Column      |      Type      | Modifiers
-----+-----+-----
 tape_id      | character(8)    | not null
 title        | character varying(80) | not null
 duration     | interval        |
```

You can also view a list of all indexes defined in your database. The `\di` meta-command displays indexes:

```
movies=# \di
               List of relations
 Schema |      Name      | Type | Owner | Table
-----+-----+-----+-----+-----
 public | customers_customer_id_key | index | korry | customers
```

You can see the full definition for any given index using the `\d index-name` meta-command:

```
movies=# \d customers_customer_id_key
Index "public.customers_customer_id_key"
   Column      | Type
-----+-----
 customer_id   | integer
UNIQUE, btree, for table "public.customers"
```

Table 3.1 shows a complete list of the system catalog-related meta-commands in `psql`:

Table 3.1. System Catalog Meta-Commands

Command	Result
<code>\dd object-name</code>	Display comments for object-name
<code>\db</code>	List all tablespaces
<code>\dn</code>	List all schemas
<code>\d_\dt</code>	List all tables
<code>\di</code>	List all indexes
<code>\ds</code>	List all sequences
<code>\dv</code>	List all views
<code>\dS</code>	List all PostgreSQL-defined tables
<code>\d table-name</code>	Show table definition
<code>\d index-name</code>	Show index definition
<code>\d view-name</code>	Show view definition
<code>\d sequence-name</code>	Show sequence definition
<code>\dp</code>	List all privileges
<code>\dl</code>	List all large objects
<code>\da</code>	List all aggregates
<code>\df</code>	List all functions

<code>\dc</code>	List all conversions
<code>\dC</code>	List all casts
<code>\df function-name</code>	List all functions with given name
<code>\do</code>	List all operators
<code>\do operator-name</code>	List all operators with given name
<code>\dT</code>	List all types
<code>\dD</code>	List all domains
<code>\dg</code>	List all groups
<code>\du</code>	List all users
<code>\l</code>	List all databases in this cluster

Alternative Views (Oracle-Style Dictionary Views)

One of the nice things about an open-source product is that code contributions come from many different places. One such project exists to add Oracle-style dictionary views to PostgreSQL. If you are an experienced Oracle user, you will appreciate this feature. The `orapgsqlviews` project contributes Oracle-style views such as `all_views`, `all_tables`, `user_tables`, and so on. For more information, see <http://gborg.postgresql.org>.

PostgreSQL version 8.0 introduced a set of views known as the `INFORMATION_SCHEMA`. The views defined in the `INFORMATION_SCHEMA` give you access to the information stored in the PostgreSQL system tables. The `INFORMATION_SCHEMA` is defined as part of the SQL standard and you'll find an `INFORMATION_SCHEMA` in most commercial (and a few open-source) database systems. If you become familiar with the views defined in the `INFORMATION_SCHEMA`, you'll find it much easier to move from one RDBMS system to another—every `INFORMATION_SCHEMA` contains the same set of views, each containing the same set of columns. For example, to see a list of the tables defined in your current database, you could execute the command:

```
SELECT table_schema, table_name, table_type FROM information_schema.tables;
```

You can execute that same query in DB2, MS SQL Server, or Informix (sadly, Oracle doesn't support the `INFORMATION_SCHEMA` standard at the time we are writing this). So what can you find in the `INFORMATION_SCHEMA`?

- `schemata`— Lists the schemas (in the current database) that are owned by you
- `tables`— Lists all tables in the current database (actually, you only see those tables that you have the right to access in some way)
- `columns`— Lists all columns in all tables that you have the right to access
- `views`— Lists all of the views you have access to in the current database
- `table_privileges`— Shows the privileges you hold (or that you granted) for each accessible object in the current database
- `domains`— Lists all of the domains defined in the current database
- `check_constraints`— Lists all of the `CHECK` constraints defined for the accessible tables (or domains) in the current database

There are more views in the `INFORMATION_SCHEMA` than we've described here (in fact, there are a total of 39 `INFORMATION_SCHEMA` views in PostgreSQL 8.0). See Chapter 30, "The Information Schema," of the PostgreSQL user guide for a complete list.

Why would you want to use the `INFORMATION_SCHEMA` instead of `psql`'s `\d` commands? We can think of three reasons. First, you can use the `INFORMATION_SCHEMA` inside of your own client applications—you can't do that with the `\d` commands because they are part of the `psql` console application (itself a PostgreSQL client) instead of the PostgreSQL server. Second, by using the views defined in the `INFORMATION_SCHEMA`, you can read the PostgreSQL system tables using the same queries that you would use to read the DB2 system tables (or Sybase or SQL Server). That makes your client applications a bit more portable. Finally, you can write custom queries against the views defined in the `INFORMATION_SCHEMA`—you can't customize the `\d` commands. For example, if you need to find all of the date columns in your database, just look inside of `INFORMATION_SCHEMA.columns`, like this:

```
SELECT DISTINCT table_name
FROM information_schema.columns WHERE data_type = 'date';
```

Need to know which columns can hold a `NUMERIC` value of at least seven digits? Use this query:

```
SELECT table_name, column_name, numeric_precision
FROM information_schema.columns
WHERE data_type = 'numeric' AND numeric_precision >= 7;
```

Of course, you can find all the information exposed by the `INFORMATION_SCHEMA` in the PostgreSQL system tables (`pg_class`, `pg_index`, and so on), but the `INFORMATION_SCHEMA` is often much easier to work with. The `INFORMATION_SCHEMA` views usually contain human-readable names for things like data type names, table names, and so on—the PostgreSQL system tables typically contain OIDs that you have to `JOIN` to another table in order to come up with a human-readable name.

Transaction Processing

Now let's move on to an important feature in any database system: transaction processing.

A transaction is a group of one or more SQL commands treated as a unit. PostgreSQL promises that all commands within a transaction will complete or that none of them will complete. If any command within a transaction does not complete, PostgreSQL will roll back all changes made within the transaction.

PostgreSQL makes use of transactions to ensure database consistency. Transactions are needed to coordinate updates made by two or more concurrent users. Changes made by a transaction are not visible to other users until the transaction is committed. When you commit a transaction, you are telling PostgreSQL that all the changes made within the transaction are logically complete, the changes should be made permanent, and the changes should be exposed to other users. When you roll back a transaction, you are telling PostgreSQL that the changes made within the transaction should be discarded and not made visible to other users.

To start a new transaction, execute a `BEGIN` command. To complete the transaction and have PostgreSQL make your changes permanent, execute the `COMMIT` command. If you want PostgreSQL to revert all changes made within the current transaction, execute the `ROLLBACK` command^[11].

[11] `BEGIN` can also be written as `BEGIN WORK` or `BEGIN TRANSACTION`. `COMMIT` can also be written as `COMMIT WORK` or `COMMIT TRANSACTION`. `ROLLBACK` can also be written as `ROLLBACK WORK` or `ROLLBACK TRANSACTION`.

It's important to realize that all SQL commands execute within a transaction. If you don't explicitly `BEGIN` a transaction, PostgreSQL will automatically execute each command within its own transaction.

Persistence

I used to think that single-command transactions were pretty useless: I was wrong. Single-command transactions are important because a single command can access multiple rows. Consider the following: Let's add a new constraint to the `customers` table.

```
movies=# ALTER TABLE customers ADD CONSTRAINT
movies=#   balance_exceeded CHECK( balance <= 50 );
```

This constraint ensures that no customer is allowed to have a balance exceeding \$50.00. Just to prove that it works, let's try setting a customer's balance to some value greater than \$50.00:

```
movies=# UPDATE CUSTOMERS SET balance = 100 where customer_id = 1;
ERROR:  ExecReplace: rejected due to CHECK constraint balance_exceeded
```

You can see that the `UPDATE` is rejected. What happens if you try to update more than one row? First, let's look at the data already in the `customers` table:

```
movies=# SELECT * FROM customers;
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
      1 | Jones, Henry | 555-1212 | 1970-10-10 | 0.00
      2 | Rubin, William | 555-2211 | 1972-07-10 | 15.00
      3 | Panky, Henry | 555-1221 | 1968-01-21 | 0.00
      4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 | 3.00
      8 | Wink Wankel | 555-1000 | 1988-12-25 | 0.00
(5 rows)
```

Now, try to `UPDATE` every row in this table:

```
movies=# UPDATE customers SET balance = balance + 40;
ERROR:  ExecReplace: rejected due to CHECK constraint balance_exceeded
```

This `UPDATE` command is rejected because adding \$40.00 to the balance for Rubin, William violates the `balance_exceeded` constraint. The question is, were any of the `customers` updated before the error occurred? The answer is: probably. You don't really know for sure because any changes made before the error occurred are rolled back. The net effect is that no changes were made to the database:

```
movies=# SELECT * FROM customers;
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
      1 | Jones, Henry | 555-1212 | 1970-10-10 | 0.00
      2 | Rubin, William | 555-2211 | 1972-07-10 | 15.00
      3 | Panky, Henry | 555-1221 | 1968-01-21 | 0.00
      4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 | 3.00
      8 | Wink Wankel | 555-1000 | 1988-12-25 | 0.00
```

(5 rows)

If some of the changes persisted while others did not, you would have to somehow find the persistent changes yourself and revert them. You can see that single-command transactions are far from useless. It took me awhile to learn that lesson

What about multicommand transactions? PostgreSQL treats a multicommand transaction in much the same way that it treats a single-command transaction. A transaction is atomic, meaning that all the commands within the transaction are treated as a single unit. If any of the commands fail to complete, PostgreSQL reverts the changes made by other commands within the transaction.

Transaction Isolation

I mentioned earlier in this section that the changes made within a transaction are not visible to other users until the transaction is committed. To be a bit more precise, uncommitted changes made in one transaction are not visible to other transactions^[12].

[12] This distinction is important when using (or developing) a client that opens two or more connections to the same database. Transactions are not shared between multiple connections. If you make an uncommitted change using one connection, those changes will not be visible to the other connection (until committed).

Transaction isolation helps to ensure consistent data within a database. Let's look at a few of the problems solved by transaction isolation.

Consider the following transactions:

User: bruce	Time	User:sheila
BEGIN TRANSACTION	T1	BEGIN TRANSACTION
UPDATE customers	T2	
SET balance = balance - 3		
WHERE customer_id = 2;		
	T3	SELECT SUM(balance) FROM customers;
	T4	COMMIT TRANSACTION;
ROLLBACK TRANSACTION;	T5	

At time T1, bruce and sheila each begin a new transaction. bruce updates the balance for customer 3 at time T1. At time T3, sheila computes the SUM() of the balances for all customers, completing her transaction at time T4. At time T5, bruce rolls back his transaction, discarding all changes within his transaction. If these transactions were not isolated from each other, sheila would have an incorrect answer: Her answer was calculated using data that was rolled back.

This problem is known as the dirty read problem: without transaction isolation, sheila would read uncommitted data. The solution to this problem is known as READ COMMITTED. READ COMMITTED is one of the two transaction isolation levels supported by PostgreSQL. A transaction running at the READ COMMITTED isolation level is not allowed to read uncommitted data. I'll show you how to change transaction levels in a moment.

There are other data consistency problems that are avoided by isolating transactions from each other. In the following scenario, sheila will receive two different answers within the same transaction:

User: bruce	Time	User: sheila
BEGIN TRANSACTION;	T1	BEGIN TRANSACTION;
	T2	SELECT balance
FROM customers		
WHERE customer_id = 2;		
UPDATE customers		
SET balance = 20		

WHERE	T3	
customer_id =		
2;		
COMMIT TRANSACTION;	T4	
	T5	SELECT balance
		FROM customers
		WHERE customer_id = 2;
	T6	COMMIT TRANSACTION;

Again, *bruce* and *sheila* each start a transaction at time T1. At T2, *sheila* finds that customer 2 has a balance of \$15.00. *bruce* changes the balance for customer 2 from \$15.00 to \$20.00 at time T3 and commits his change at time T4. At time T5, *sheila* executes the same query that she executed earlier in the transaction, but this time she finds that the balance is \$20.00. In some applications, this isn't a problem; in others, this interference between the two transactions is unacceptable. This problem is known as the non-repeatable read.

Here is another type of problem:

User: bruce	Time	User: sheila
BEGIN TRANSACTION;	T1	BEGIN TRANSACTION;
	T2	SELECT * FROM customers;
INSERT INTO customers VALUES	T3	
(
6,		
'Neville, Robert',		
'555-9999',		
'1971-03-20',		
0.00		
);		
COMMIT TRANSACTION;	T4	
	T5	SELECT * FROM customers;
	T6	COMMIT TRANSACTION;

In this example, *sheila* again executes the same query twice within a single transaction. This time, *bruce* has inserted a new row in between the *sheila*'s queries. Notice that this is not a case of a dirty read—*bruce* has committed his change before *sheila* executes her second query. At time T5, *sheila* finds a new row. This is similar to the non-repeatable read, but this problem is known as the phantom read problem.

The answer to both the non-repeatable read and the phantom read is the `SERIALIZABLE` transaction isolation level. A transaction running at the `SERIALIZABLE` isolation level is only allowed to see data committed before the transaction began.

In PostgreSQL, transactions usually run at the `READ COMMITTED` isolation level. If you need to avoid the problems present in `READ COMMITTED`, you can change isolation levels using the `SET TRANSACTION` command. The syntax for the `SET TRANSACTION` command is

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE };
```

The `SET TRANSACTION` command affects only the current transaction (and it must be executed before the first DML^[13] command within the transaction). If you want to change the isolation level for your session (that is, change the isolation level for future transactions), you can use the `SET SESSION` command:

^[13] A DML (data manipulation language) command is any command that can update or read the data within a table. `SELECT`, `INSERT`, `UPDATE`, `FETCH`, and `COPY` are DML commands.

```
SET SESSION CHARACTERISTICS AS
TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
```

PostgreSQL version 8.0 introduces a new transaction processing feature called a `SAVEPOINT`. A `SAVEPOINT` is a named marker that you define within the stream of commands that make up a transaction. Once you've defined a `SAVEPOINT`, you can `ROLLBACK` any changes that you've made since that point without discarding changes made prior to the `SAVEPOINT`—in other words, you can `ROLLBACK` part of a transaction (the trailing part) without rolling back the entire transaction. To create a `SAVEPOINT`, execute a

SAVEPOINT command within a transaction. The syntax for a SAVEPOINT command is very simple:

```
SAVEPOINT savepoint-name
```

The savepoint-name must follow the normal rules for an identifier; it must be unique within the first 64 characters and must start with a letter or underscore (or it must be a quoted identifier). A SAVEPOINT gives a name to a point in time; in particular, a point between two SQL commands. Consider the following sequence:

```
movies=# SELECT customer_id, customer_name FROM customers;
 customer_id | customer_name
-----+-----
          3 | Panky, Henry
          1 | Jones, Henry
          4 | Wonderland, Alice N.
          2 | Rubin, William
(4 rows)

movies=# START TRANSACTION;
START TRANSACTION

movies=# INSERT INTO customers VALUES( 5, 'Kemp, Hans' );
INSERT 44272 1

movies=# SELECT * FROM customers;
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
          3 | Panky, Henry | 555-1221 | 1968-01-21 | 0.00
          1 | Jones, Henry | 555-1212 | 1970-10-10 | 0.00
          4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 | 3.00
          2 | Rubin, William | 555-2211 | 1972-07-10 | 15.00
          5 | Kemp, Hans | | | |
(5 rows)
```

At this point, you've started a new transaction and inserted a new row, but you haven't committed your changes yet. Now define a SAVEPOINT named p1 and insert a second row:

```
movies=# SAVEPOINT P1;
SAVEPOINT

movies=# INSERT INTO customers VALUES( 6, 'Falkstein, Gerhard' );
INSERT 44273 1
```

The SAVEPOINT command inserted a marker into the transaction stream. If you execute a ROLLBACK command at this point, both of the newly inserted rows will be discarded (in other words, all of the changes you've made in this transaction will be rolled back):

```
movies=# ROLLBACK;
ROLLBACK

movies=# SELECT * FROM customers;
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
          3 | Panky, Henry | 555-1221 | 1968-01-21 | 0.00
          1 | Jones, Henry | 555-1212 | 1970-10-10 | 0.00
          4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 | 3.00
          2 | Rubin, William | 555-2211 | 1972-07-10 | 15.00
(4 rows)
```

Now repeat the same sequence of commands, but this time around, execute a qualified ROLLBACK command, like this:

```
movies=# ROLLBACK TO SAVEPOINT P1;
ROLLBACK

movies=# SELECT * FROM customers;
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
          3 | Panky, Henry | 555-1221 | 1968-01-21 | 0.00
          1 | Jones, Henry | 555-1212 | 1970-10-10 | 0.00
          4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 | 3.00
          2 | Rubin, William | 555-2211 | 1972-07-10 | 15.00
          5 | Kemp, Hans | | | |
(5 rows)
```

When you ROLLBACK to a SAVEPOINT, changes made since the SAVEPOINT are discarded, but not changes made before the SAVEPOINT. So, you see that the customers table retains the first row that you inserted, but not the second row. When you ROLLBACK to a SAVEPOINT, you are still in the middle of a transaction—you must complete the transaction with a COMMIT or ROLLBACK command.

Here are a few important points to keep in mind when you're working with `SAVEPOINTS`:

- You can nest `SAVEPOINTS`. For example, if you create a `SAVEPOINT` named `P1`, then create a second `SAVEPOINT` named `P2`, you have created a nested `SAVEPOINT` (`P2` is nested within `P1`). If you `ROLLBACK TO SAVEPOINT P2`, PostgreSQL discards any changes made since `P2`, but preserves changes made between `P1` and `P2`. On the other hand, if you `ROLLBACK TO SAVEPOINT P1`, PostgreSQL discards all changes made since `P1`, including all changes made since `P2`. Nested `SAVEPOINTS` are handy when you are working with a multilevel table structure such as `ORDERS` and `LINEITEMS` (where you have multiple line items per order). If you define a `SAVEPOINT` prior to modifying each order, and a second, nested `SAVEPOINT` prior to modifying each line item, you can `ROLLBACK` changes made to a single line item, changes made to a single order, or an entire transaction.
- You can use the same `SAVEPOINT` name as often as you like within a single transaction—the new `SAVEPOINT` simply replaces the old `SAVEPOINT`^[14]. Again, this is useful when you are working with a multilevel table structure. If you create a `SAVEPOINT` prior to processing each line item and you give each of those `SAVEPOINTS` the same name, you can `ROLLBACK` changes made to the most recently processed line item.

^[14] PostgreSQL doesn't follow the SQL standard when you create two `SAVEPOINTS` within the same transaction. PostgreSQL simply hides the old `SAVEPOINT`—the SQL standard states that the old `SAVEPOINT` should be destroyed. If you need the SQL-prescribed behavior, you can destroy the old `SAVEPOINT` with the command `RELEASE SAVEPOINT savepoint-name`.

- If you `ROLLBACK` to a `SAVEPOINT`, the `SAVEPOINT` is not destroyed—you can make more changes in the transaction and `ROLLBACK` to the `SAVEPOINT` again. However, any `SAVEPOINTS` nested within that `SAVEPOINT` will be destroyed. To continue the `ORDERS` and `LINEITEMS` example, if you `ROLLBACK` the changes made to an `ORDERS` row, you also discard changes made to the `LINEITEMS` for that `ORDER` and you are destroying the `SAVEPOINT` that you created for the most recent line item.
- If you make a mistake (such as a typing error), PostgreSQL rolls back to the most recent `SAVEPOINT`. That's a very nice feature. If you've used PostgreSQL for any length of time, you've surely exercised your vocabulary after watching PostgreSQL throw out a long and complex transaction because you made a simple typing error. If you insert `SAVEPOINTS` in your transaction, you won't lose as much work when your fingers fumble a table name.

Multi-Versioning and Locking

Most commercial (and open-source) databases use locking to coordinate multiuser updates. If you are modifying a table, that table is locked against updates and queries made by other users. Some databases perform page-level or row-level locking to reduce contention, but the principle is the same—other users must wait to read the data you have modified until you have committed your changes.

PostgreSQL uses a different model called multi-versioning, or MVCC for short (locks are still used, but much less frequently than you might expect). In a multi-versioning system, the database creates a new copy of the rows you have modified. Other users see the original values until you commit your changes—they don't have to wait until you finish. If you roll back a transaction, other users are not affected—they did not have access to your changes in the first place. If you commit your changes, the original rows are marked as obsolete and other transactions running at the `READ COMMITTED` isolation level will see your changes. Transactions running at the `SERIALIZABLE` isolation level will continue to see the original rows. Obsolete data is not automatically removed from a PostgreSQL database. It is hidden, but not removed. You can remove obsolete rows using the `VACUUM` command. The syntax of the `VACUUM` command is

```
VACUUM [ VERBOSE ] [ ANALYZE ] [ table ]
```

I'll talk about the `VACUUM` command in more detail in the next chapter.

The MVCC transaction model provides for much higher concurrency than most other models. Even though PostgreSQL uses multiple versions to isolate transactions, it is still necessary to lock data in some circumstances.

Try this experiment. Open two `psql` sessions, each connected to the `movies` database. In one session, enter the following commands:

```
movies=# BEGIN WORK;
BEGIN
movies=# INSERT INTO customers VALUES
movies-# ( 5, 'Manyjars, John', '555-8000', '1960-04-02', 0 );
INSERT
```

In the other session, enter these commands:

```
movies=# BEGIN WORK;
BEGIN
movies=# INSERT INTO customers VALUES
movies-# ( 6, 'Smallberries, John', '555-8001', '1960-04-02', 0 );
INSERT
```

When you press the `Enter` (or `Return`) key, this `INSERT` statement completes immediately. Now, enter this command into the second session:

```
movies=# INSERT INTO customers VALUES
movies-#  ( 5, 'Gomez, John', '555-8000', '1960-04-02', 0 );
```

This time, when you press `Enter`, `psql` hangs. What is it waiting for? Notice that in the first session, you already added a customer whose `customer_id` is 5, but you have not yet committed this change. In the second session, you are also trying to insert a customer whose `customer_id` is 5. You can't have two customers with the same `customer_id` (because you have defined the `customer_id` column to be the unique `PRIMARY KEY`). If you commit the first transaction, the second session would receive a duplicate value error. If you roll back the first transaction, the second insertion will continue (because there is no longer a constraint violation). PostgreSQL won't know which result to give you until the transaction completes in the first session.

Summary

[Chapter 1](#), "Introduction to PostgreSQL and SQL," showed you some of the basics of retrieving and modifying data using PostgreSQL. In [Chapter 2](#), "Working with Data in PostgreSQL," you learned about the many data types offered by PostgreSQL. This chapter has filled in some of the scaffolding—you've seen how to create new databases, new tables, and new indexes. You've also seen how PostgreSQL solves concurrency problems through its multi-versioning transaction model.

The next chapter, [Chapter 4](#), "Performance," should help you understand how the PostgreSQL server decides on the fastest way to execute your SQL commands.

Chapter 4. Performance

In the previous three chapters, you have seen how to create new databases and tables. You have also seen a variety of ways to retrieve data. Inevitably, you will run into a performance problem. At some point, PostgreSQL won't process data as quickly as you would like. This chapter should prepare you for that situation—after reading this chapter, you'll have a good understanding of how PostgreSQL executes a query and what you can do to make queries run faster.

How PostgreSQL Organizes Data

Before you can really dig into the details of performance tuning, you need to understand some of the basic architecture of PostgreSQL.

You already know that in PostgreSQL, data is stored in tables and tables are grouped into databases. At the highest level of organization, databases are grouped into clusters—a cluster of databases is serviced by a postmaster.

Let's see how this data hierarchy is stored on disk. You can see all databases in a cluster using the following query:

```
perf=# SELECT datname, oid FROM pg_database;
 datname | oid
-----+-----
 perf    | 16556
 template1 | 1
 template0 | 16555
```

From this list, you can see that I have three databases in this cluster. You can find the storage for these databases by looking in the `$PGDATA` directory:

```
$ cd $PGDATA
$ ls
base      pg_clog      pg_ident.conf  pg_xlog      postmaster.opts
global    pg_hba.conf  PG_VERSION     postgresql.conf  postmaster.pid
```

The `$PGDATA` directory has a subdirectory named `base`. The `base` subdirectory is where your databases reside:

```
$ cd ./base
$ ls -l
total 12
drwx----- 2 postgres pgadmin    4096 Jan  1 20:53 1
drwx----- 2 postgres pgadmin    4096 Jan  1 20:53 16555
drwx----- 3 postgres pgadmin    4096 Jan  1 22:38 16556
```

Notice that there are three subdirectories underneath `$PGDATA/base`. The name of each subdirectory corresponds to the `oid` of one entry in the `pg_database` table: the subdirectory named `1` contains the `template1` database, the subdirectory named `16555` contains the `template0` database, and the subdirectory named `16556` contains the `perf` database.

Let's look a little deeper:

```
$ cd ./1
$ ls
1247 16392 16408 16421 16429 16441 16449 16460 16472
1249 16394 16410 16422 16432 16442 16452 16462 16474
1255 16396 16412 16423 16435 16443 16453 16463 16475
1259 16398 16414 16424 16436 16444 16454 16465 16477
16384 16400 16416 16425 16437 16445 16455 16466 pg_internal.init
16386 16402 16418 16426 16438 16446 16456 16468 PG_VERSION
16388 16404 16419 16427 16439 16447 16457 16469
16390 16406 16420 16428 16440 16448 16458 16471
```

Again, you see a lot of files with numeric filenames. You might guess that these numbers also correspond to `oids`, and (by chance) you would often be correct. Every table (and index) in a database is catalogued in the `pg_class` system table. To find a table in `pg_class`, search for a row where the `relname` column is equal to the name of the table. For example, to find the `pg_class` entry for the `pg_group` table, execute the command

```
SELECT * FROM pg_class WHERE relname = 'pg_group';
```

The `pg_class.relfilenode` value for a table determines the name of the file that stores the table (likewise, the `pg_class.relfilenode` value for an index determines the name of the file that stores the index). In most cases, the `OID` of a table's `pg_class` entry matches the table's `relfilenode`, but that's not always the case. If you `ALTER` a table in such a way that PostgreSQL must first make a new copy of the table and then drop the original, the table's `relfilenode` will change. The `relfilenode` may also change if you `CLUSTER` the table (PostgreSQL makes a new copy of the table in the desired order and then drops the original). The `relfilenode` value for an index may change if you rebuild the index with a `REINDEX` command, or if you `ALTER` the data type of a column covered by the index.

To see the correspondence between a table's `relfilenode` and its filename, simply compare the output from the following `SELECT` command to the filesystem directory that contains the database

```
test=# SELECT relfilenode, relname FROM pg_class ORDER BY relfilenode;
 relfilenode | relname
-----+-----
          0 | pg_xactlock
        1247 | pg_type
        1249 | pg_attribute
        1255 | pg_proc
        1259 | pg_class
         ... | ...
```

Each table is stored in its own disk file and the name of the file is determined by the `oid` `relfilenode` of the table's entry in the `pg_class` table.

There are two more columns in `pg_class` that might help explain PostgreSQL's storage structure:

```
perf=# SELECT relname, oid, relpages, reltuples FROM pg_class
perf=# ORDER BY oid
 relname | oid | reltuples | relpages
-----+-----+-----+-----
 pg_type | 1247 |         2 |         2
pg_attribute | 1249 |        11 |        11
 pg_proc | 1255 |        31 |        31
 pg_class | 1259 |         2 |         2
pg_shadow | 1260 |         1 |         1
pg_group  | 1261 |         0 |         0
         ... | ... | ... | ...
```

The `reltuples` column tells you how many tuples are in each table. The `relpages` column shows how many pages are required to store the current contents of the table. How do these numbers correspond to the actual on-disk structures? If you look at the table files for a few tables, you'll see that there is a relationship between the size of the file and the number of `relpages` columns:

```
$ ls -l 1247 1249
-rw----- 1 postgres pgadmin 16384 Jan 01 20:53 1247
-rw----- 1 postgres pgadmin 90112 Jan 01 20:53 1249
```

The file named 1247 (`pg_type`) is 16,384 bytes long and consumes two pages. The file named 1249 (`pg_attribute`) is 90,112 bytes long and consumes 11 pages. A little math will show that $16,384/2 = 8,192$ and $90,112/11 = 8,192$: each page is 8,192 (8K) bytes long. In PostgreSQL, all disk I/O is performed on a page-by-page basis^[1]. When you select a single row from a table, PostgreSQL will read at least one page—it may read many pages if the row is large. When you update a single row, PostgreSQL will write the new version of the row at the end of the table and will mark the original version of the row as invalid.

[1] Actually, most disk I/O is performed on a page-by-page basis. Some configuration files and log files are accessed in other forms, but all table and index access is done in pages.

The size of a page is fixed at 8,192 bytes. You can increase or decrease the page size if you build your own copy of PostgreSQL from source, but all pages within a database will be the same size. The size of a row is not fixed—different tables will yield different row sizes. In fact, the rows within a single table may differ in size if the table contains variable-length columns. Given that the page size is fixed and the row size is variable, it's difficult to predict exactly how many rows will fit within any given page.

The perf database and the recalls Table

The sample database that you have been using so far doesn't really hold enough data to show performance relationships. Instead, I've created a new database (named `perf`) that holds some large tables. I've downloaded the `recalls` database from the U.S. National Highway Traffic Safety Administration^[2]. This database contains a single table with 39,241 rows. Here is the layout of the `recalls` table:

Code View: [Scroll](#) / [Show All](#)

```
perf=# \d recalls
Table "recalls"
Column | Type | Modifiers
-----+-----+-----
record_id | numeric(9,0) |
campno | character(9) |
maketxt | character(25) |
modeltxt | character(25) |
yeartxt | character(4) |
mfgcampno | character(10) |
compdesc | character(75) |
mgftxt | character(30) |
bgman | character(8) |
endman | character(8) |
```

```

vet          | character(1)          |
potaff       | numeric(9,0)          |
ndate        | character(8)          |
odate        | character(8)          |
influenced   | character(4)          |
mfgrname     | character(30)         |
rcdate       | character(8)          |
datea        | character(8)          |
rpno         | character(3)          |
fmvss        | character(3)          |
desc_defect  | character varying(2000) |
con_defect   | character varying(2000) |
cor_action   | character varying(2000) |
Indexes: recall_record_id

```

Notice that there is only one index and it covers the `record_id` column.

[2] This data (http://ftp.nhtsa.dot.gov/rev_recalls/) is in the form of a flat ASCII file. I had to import the data into my `perf` database.

The `recalls` table in the `perf` database contains 39,241 rows in 4,413 pages:

```

perf=# SELECT relname, reltuples, relpages, oid FROM pg_class
perf=# WHERE relname = 'recalls';
 relname | reltuples | relpages | oid
-----+-----+-----+-----
recalls |      39241 |      4413 | 96409

```

Given that a page is 8,192 bytes long, you would expect that the file holding this table (`$PGDATA/base/16556/96409`) would be 36,151,296 bytes long:

```

$ ls -l $PGDATA/base/16556/96409
-rw----- 1 postgres pgadmin 36151296 Jan 01 23:34 96409

```

Figure 4.1 shows how the `recalls` table might look on disk. (Notice that the rows are not sorted—they appear in the approximate order of insertion.)

Figure 4.1. The `recalls` table as it might look on disk.

[\[View full size image\]](#)

Page 1

record_id	campno	maketxt	...	cor_action
42009	02E009000	NXT	...	Nexl will remove the...
13621	82E018000	CATERPILLAR	...	The Dealer will inspect...
42010	02E010000	NEXL	...	Next will notify its custom...
35966	99E039000	APC	...	APS will replace these...
42011	02E010000	NEXL	...	Nexl will notify its custom...
12927	81T009000	HERCULES	...	Tires will be replaced,...
42012	02E010000	NEXL	...	Nexl will notify its custom...
35974	99E039000	APC	...	APS will replace these...

Page 2

record_id	campno	maketxt	...	cor_action
42013	02E010000	NEXT	...	Nexl will notify its custom...
13654	82T014000	GOODYEAR	...	The Dealer will replace all...
42014	02E010000	NXT	...	Nexl will notify its custom...
35133	99E018000	D	...	Dealers will inspect their...
42005	02E009000	NEXL	...	Nexl will remove the...
12924	81T008000	NANKANG	...	Defective tires will be...
41467	01X003000	BRITAX	...	Customers will receive a...
35131	99E017000	MERITOR	...	Meritor will inspect and...

...

Page
4412

record_id	campno	maketxt	...	cor_action
42054	02V074002	PONTIAC	...	Dealers will properly tight...
41863	02V044000	HYUNDAI	...	Dealers will inspect the ...
42139	02V095000	INTERNATIONAL	...	Dealers will inspect the ...
41926	02V065000	COUNTRY COACH	...	Dealers will replace the...
42138	02V095000	INTERNATIONAL	...	Dealers will inspect the...
41927	02V065000	COUNTRY COACH	...	Dealers will replace the...
42140	02V095000	INTERNATIONAL	...	Dealers will inspect the...
41930	02V065000	COUNTRY COACH	...	Dealers will replace the...

If a row is too large to fit into a single 8K block^[3], PostgreSQL will write part of the data into a TOAST^[4] table. A TOAST table acts as an extension to a normal table. It holds values too large to fit inline in the main table.

^[3] PostgreSQL tries to store at least four rows per heap page and at least four entries per index page.

^[4] The acronym TOAST stands for "the oversized attribute storage technique."

Indexes are also stored in page files. A page that holds row data is called a heap page. A page that holds index data is called an index page. You can locate the page file that stores an index by examining the index's entry in the `pg_class` table. And, just like tables, it is difficult to predict how many index entries will fit into each 8K page^[5]. If an index entry is too large, it is moved to an index TOAST table.

^[5] If you want more information about how data is stored inside a page, I recommend the `pg_filedump` utility from Red Hat.

In PostgreSQL, a page that contains row data is a heap block. A page that contains index data is an index block. You will never find heap blocks and index blocks in the same page file.

Page Caching

Two of the fundamental performance rules in any database system are:

- Memory access is fast; disk access is slow.
- Memory space is scarce; disk space is abundant.

Accordingly, PostgreSQL tries very hard to minimize disk I/O by keeping frequently used data in memory. When the first server process starts, it creates an in-memory data structure known as the buffer cache. The buffer cache is organized as a collection of 8K pages—each page in the buffer cache corresponds to a page in some page file. The buffer cache is shared between all processes servicing a given database.

When you select a row from a table, PostgreSQL will read the heap block that contains the row into the buffer cache. If there isn't enough free space in the cache, PostgreSQL will move some other block out of the cache. If a block being removed from the cache has been modified, it will be written back out to disk; otherwise, it will simply be discarded. Index blocks are buffered as well.

In the "[Gathering Performance Information](#)" section, you'll see how to measure the performance of the cache and how to change its size.

Summary

This section gave you a good overview of how PostgreSQL stores data on disk. With some of the fundamentals out of the way, you can move on to more performance issues.

Gathering Performance Information

With release 7.2, the PostgreSQL developers introduced a new collection of performance-related system views. These views return two distinct kinds of information. The `pg_stat` views characterize the frequency and type of access for each table in a database. The `pg_statio` views will tell you how much physical I/O is performed on behalf of each table.

Let's look at each set of performance-related views in more detail.

The `pg_stat_all_tables` contains one row for each table in your database. Here is the layout of `pg_stat_all_tables`:

```
perf=# \d pg_stat_all_tables
          View "pg_stat_all_tables"
   Column      |  Type   | Modifiers
-----+-----+-----
 relid         |  oid    |
 schemaname    |  name   |
 relname       |  name   |
 seq_scan      |  bigint |
 seq_tup_read  |  bigint |
 idx_scan      | numeric |
 idx_tup_fetch | numeric |
 n_tup_ins     |  bigint |
 n_tup_upd     |  bigint |
 n_tup_del     |  bigint |
```

The `seq_scan` column tells you how many sequential (that is, table) scans have been performed for a given table, and `seq_tup_read` tells you how many rows were processed through table scans. The `idx_scan` and `idx_tup_fetch` columns tell you how many index scans have been performed for a table and how many rows were processed by index scans. The `n_tup_ins`, `n_tup_upd`, and `n_tup_del` columns tell you how many rows were inserted, updated, and deleted, respectively.

Query Execution

If you're not familiar with the terms "table scan" or "index scan," don't worry—I'll cover query execution later in this chapter (see "[Understanding How PostgreSQL Executes a Query](#)").

The real value in `pg_stat_all_tables` is that you can find out which tables in your data base are most heavily used. This view does not tell you how much disk I/O is performed against each table file, nor does it tell you how much time it took to perform the operations.

The following query finds the top 10 tables in terms of number of rows read:

```
SELECT relname, idx_tup_fetch + seq_tup_read AS Total
FROM pg_stat_all_tables
WHERE idx_tup_fetch + seq_tup_read != 0
ORDER BY Total desc
LIMIT 10;
```

Here's an example that shows the result of this query in a newly created database:

```
perf=# SELECT relname, idx_tup_fetch + seq_tup_read AS Total
perf=# FROM pg_stat_all_tables
perf=# WHERE idx_tup_fetch + seq_tup_read != 0
perf=# ORDER BY Total desc
perf=# LIMIT 10;

 relname | total
-----+-----
 recalls | 78482
 pg_class | 57425
 pg_index | 20901
 pg_attribute | 5965
 pg_proc  | 1391
```

It's easy to see that the `recalls` table is heavily used—you have read 78,482 tuples from that table.

There are two variations on the `pg_stat_all_tables` view. The `pg_stat_sys_tables` view is identical to `pg_stat_all_tables`, except that it is restricted to showing system tables. Similarly, the `pg_stat_user_tables` view is restricted to showing only user-created tables.

You can also see how heavily each index is being used—the `pg_stat_all_indexes`, `pg_stat_user_indexes`, and `pg_stat_system_indexes` views expose index information.

Although the `pg_stat` view tells you how heavily each table is used, it doesn't provide any information about how much physical I/O is performed on behalf of each table. The second set of performance-related views provides that information.

The `pg_statio_all_tables` view contains one row for each table in a database. Here is the layout of `pg_statio_all_tables`:

```
perf=# \d pg_statio_all_tables
```

View "pg_statio_all_tables"		
Column	Type	Modifiers
relid	oid	
schemaname	name	
relname	name	
heap_blks_read	bigint	
heap_blks_hit	bigint	
idx_blks_read	numeric	
idx_blks_hit	numeric	
toast_blks_read	bigint	
toast_blks_hit	bigint	
tidx_blks_read	bigint	
tidx_blks_hit	bigint	

This view provides information about heap blocks (`heap_blks_read`, `heap_blks_hit`), index blocks (`idx_blks_read`, `idx_blks_hit`), toast blocks (`toast_blks_read`, `toast_blks_hit`), and index toast blocks (`tidx_blks_read`, `tidx_blks_hit`). For each of these block types, `pg_statio_all_tables` exposes two values: the number of blocks read and the number of blocks that were found in PostgreSQL's cache. For example, the `heap_blks_read` column contains the number of heap blocks read for a given table, and `heap_blks_hit` tells you how many of those pages were found in the cache.

PostgreSQL exposes I/O information for each index in the `pg_statio_all_indexes`, `pg_statio_user_indexes`, and `pg_statio_sys_indexes` views.

Let's try a few examples and see how you can use the information exposed by `pg_statio_all_tables`.

I've written a simple utility (called `timer`) that makes it a little easier to see the statistical results of a given query. This utility takes a snapshot of `pg_stat_all_tables` and `pg_statio_all_tables`, executes a given query, and finally compares the new values in `pg_stat_all_tables` and `pg_statio_all_tables`. Using this utility, you can see how much I/O was performed on behalf of the given query. Of course, the database must be idle except for the query under test.

Execute this simple query and see what kind of I/O results you get:

Code View: [Scroll](#) / [Show All](#)

```
$ timer "SELECT * FROM recalls"
```

	SEQUENTIAL I/O				INDEXED I/O			
	scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
pg_aggregate	0	0	1	0	1	1	2	0
pg_am	1	1	1	0	0	0	0	0
pg_amop	0	0	2	10	10	24	4	16
pg_amproc	0	0	1	5	6	6	2	10
pg_attribute	0	0	8	14	21	65	6	57
pg_cast	0	0	2	6	60	8	2	118
pg_class	4	740	5	32	18	17	7	34
pg_database	1	1	1	0	0	0	0	0
pg_index	2	146	3	11	8	12	4	12
pg_namespace	2	10	1	2	2	1	2	2
pg_opclass	0	0	2	11	5	73	4	6
pg_operator	0	0	4	6	10	10	4	26
pg_proc	0	0	6	8	14	14	12	31
pg_rewrite	0	0	1	1	2	2	2	2
pg_shadow	0	0	1	2	3	3	4	2
pg_statistic	0	0	3	5	33	8	2	64
pg_trigger	0	0	1	1	2	2	2	2
pg_type	0	0	2	5	7	7	2	12
recalls	1	39241	4413	0	0	0	0	0
Totals	11	40139	4458	119	202	253	61	394

The `timer` utility shows that a simple query generates a lot of buffer traffic. The PostgreSQL server must parse and plan the query and it consults a number of system tables to do so—that explains the buffer interaction incurred on behalf of all of the tables that start with `pg_`. The `recalls` table generates most of the buffer traffic.

You can invoke the `timer` utility with one argument or two. The first argument contains the text of the query that you want to measure. The second argument, if present, is the name of the table that you're interested in. If you omit the second argument, you'll see I/O measurements for every table that was hit during the query (including the PostgreSQL system tables). If you include the second argument, you'll only see the I/O measurements for that table. In most of the discussion that follows, we'll filter out the I/O performed against the system tables.

This query retrieved 39,241 rows in a single table scan. This scan read 4,413 heap blocks from disk and found none in the cache. Normally, you would hope to see a cache ratio much higher than 4,413 to 0! In this particular case, I had just started the postmaster so there were few pages in the cache and none were devoted to the `recalls` table. Now, try this experiment again to see if the cache ratio gets any better:

Code View: [Scroll](#) / [Show All](#)

```
$ timer "SELECT * FROM recalls" recalls
```

SEQUENTIAL I/O				INDEXED I/O			
scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
recalls	1	39241	4413	0	0	0	0

You get exactly the same results for the `recalls` table—no cache hits. Why not? We did not include an `ORDER BY` clause in this query so PostgreSQL returned the rows in (approximately) the order of insertion. When we execute the same query a second time, PostgreSQL starts reading at the beginning of the page file and continues until it has read the entire file. Because my cache is only 512 blocks in size, the first 512 blocks have been forced out of the cache by the time I get to the end of the table scan. The next time I execute the same query, the final 512 blocks are in the cache, but you are looking for the leading blocks. The end result is no cache hits.

Just as an experiment, try to increase the size of the cache to see if you can force some caching to take place.

The PostgreSQL cache is kept in a segment of memory shared by all backend processes. You can see this using the `ipcs -m` command^[6]:

^[6] \up7 In case you are curious, the key value uniquely identifies a shared memory segment. The key is determined by multiplying the postmaster's port number by 1,000 and then incrementing until a free segment is found. The `shmid` value is generated by the operating system (key is generated by PostgreSQL). The `nattch` column tells you how many processes are currently using the segment.

```
$ ipcs -m
----- Shared Memory Segments -----
key      shmid    owner    perms    bytes    nattch   status
0x0052e2c1 1409024  postgres 600      5021696  3
```

The shared memory segment contains more than just the buffer cache: PostgreSQL also keeps some bookkeeping information in shared memory. With 512 pages in the buffer cache and an 8K block size, you see a shared memory segment that is 5,021,696 bytes long. Let's increase the buffer cache to 513 pages and see what effect that has on the size of the shared memory segment. There are two ways that you can adjust the size of the cache. You could change PostgreSQL's configuration file (`$PGDATA/postgresql.conf`), changing the `shared_buffers` variable from 512 to 513. Or, you can override the `shared_buffers` configuration variable when you start the `postmaster`:

```
$ pg_ctl stop
waiting for postmaster to shut down.....done
postmaster successfully shut down
$ #
$ # Note: specifying -o "-B 513" is equivalent
$ #       to setting shared_buffers = 513 in
$ #       the $PGDATA/postgresql.conf file
$ #
$ pg_start -o "-B 513" -l /tmp/pg.log
postmaster successfully started
```

Now you can use the `ipcs -m` command to see the change in the size of the shared memory segment:

```
$ ipcs -m
----- Shared Memory Segments -----
key      shmid    owner    perms    bytes    nattch   status
0x0052e2c1 1409024  postgres 600      5038080  3
```

The shared memory segment increased from 5,021,696 bytes to 5,038,080 bytes. That's a difference of 16,384 bytes, which happens to be the size of two blocks. Why two? Because PostgreSQL keeps some bookkeeping information in shared memory in addition to the buffer cache—the amount of extra space required depends on the number of shared buffers. PostgreSQL won't add two blocks each time you increment `shared_buffers` by 1; it just happens that when you increase from 512 to 513, you cross a threshold that requires an extra page in the bookkeeping system.

Now, let's get back to the problem at hand. We want to find out if doubling the buffer count will result in more cache hits and therefore fewer I/O operations. Remember, a table scan on the `recalls` table resulted in 4,413 heap blocks read and 0 cache hits. Let's double the size of the shared buffer cache (from 512 to 1,024 blocks). Try the same query again and check the results:

```
$ pg_ctl stop
waiting for postmaster to shut down.....done
postmaster successfully shut down
$ pg_start -o "-B 1024" -l /tmp/pg.log
postmaster successfully started
$ ipcs -m
----- Shared Memory Segments -----
key      shmid    owner    perms    bytes    nattch   status
0x0052e2c1 1409024  postgres 600      9338880  3

$ timer "SELECT * FROM recalls" recalls

+-----+-----+
|          SEQUENTIAL I/O          |          INDEXED I/O          |
| scans | tuples | heap_blks | cached | scans | tuples | idx_blks | cached |
+-----+-----+
| recalls | 1 | 39241 | 4413 | 0 | 0 | 0 | 0 |
```

	SEQUENTIAL I/O				INDEXED I/O			
	scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
1	39241		4413	0	0	0	0	0

You have to run this query twice because you shut down and restarted the postmaster to adjust the cache size. When you shut down the postmaster, the cache is destroyed (you can use the `ipcs -m` command to verify this).

```
$ timer "SELECT * FROM recalls" recalls
```

	SEQUENTIAL I/O				INDEXED I/O			
	scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
1	39241		4413	0	0	0	0	0

Still the same results as before—PostgreSQL does not seem to buffer any of the data blocks read from the `recalls` table. Actually, each block is buffered as soon as it is read from disk; but as before, the blocks read at the beginning of the table scan are pushed out by the blocks read at the end of the scan. When you execute the same query a second time, you start at the beginning of the table and find that the blocks that you need are not in the cache.

You could increase the cache size to be large enough to hold the entire table (somewhere around $4,413 + 120$ blocks should do it), but that's a large shared memory segment, and if you don't have enough physical memory, your system will start to thrash.

Let's try a different approach. PostgreSQL has enough room for 1,024 pages in the shared buffer cache. The entire `recalls` table consumes 4,413 pages. If you use the `LIMIT` clause to select a subset of the `recalls` table, you should see some caching. I'm going to lower the cache size back to its default of 512 pages before we start:

```
$ pg_ctl stop
waiting for postmaster to shut down.....done
postmaster successfully shut down
$ pg_start -o "-B 512" -l /tmp/pg.log
postmaster successfully started
```

You know that it takes 4,413 pages to hold the 39,241 rows in `recalls`, which gives you an average of about 9 rows per page. We have 512 pages in the cache; let's assume that PostgreSQL needs about 180 of them for its own bookkeeping, leaving us 332 pages. So, you should ask for $9 * 332$ (or 2,988) rows:

Code View: [Scroll](#) / [Show All](#)

```
$ ./timer "SELECT * FROM recalls LIMIT 2988" recalls
```

	SEQUENTIAL I/O				INDEXED I/O			
	scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
recalls	1	2988	208	0	0	0	0	0

PostgreSQL read 208 heap blocks. If everything worked, those pages should still be in the cache. Let's run the query again:

Code View: [Scroll](#) / [Show All](#)

```
$ ./timer "SELECT * FROM recalls LIMIT 2988" recalls
```

	SEQUENTIAL I/O				INDEXED I/O			
	scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
recalls	1	2988	0	208	0	0	0	0

Now you're getting somewhere. PostgreSQL read 208 heap blocks and found all 208 of them in the cache.

Dead Tuples

Now let's look at another factor that affects performance. Make a simple update to the `recalls` table:

```
perf=# UPDATE recalls SET potaff = potaff + 1;
UPDATE
```

This command increments the `potaff` column of each row in the `recalls` table. (Don't read too much into this particular `UPDATE`. I chose `potaff` simply because I needed an easy way to update every row.) Now, after restarting the database, go back and `SELECT` all rows again:

Code View: [Scroll](#) / Show All

```
$ timer "SELECT * FROM recalls" recalls
```

	SEQUENTIAL I/O				INDEXED I/O			
	scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
recalls	1	39241	8803	0	0	0	0	0

That's an interesting result—you still retrieved 39,241 rows, but this time you had to read 8,803 pages to find them. What happened? Let's see if the `pg_class` table gives any clues:

```
perf=# SELECT relname, reltuples, relpages
perf=# FROM pg_class
perf=# WHERE relname = 'recalls';
 relname | reltuples | relpages
-----+-----+-----
 recalls |    39241 |     4413
```

No clues there—`pg_class` still thinks you have 4,413 heap blocks in this table. Let's try counting the individual rows:

```
perf=# SELECT count(*) FROM recalls;
 count
-----
 39241
```

At least that gives you a consistent answer. But why does a simple `update UPDATE` cause you to read twice as many heap blocks as before?

When you `UPDATE` a row, PostgreSQL performs the following operations:

1. The new row values are written to the table.
2. The old row is deleted from the table.
3. The deleted row remains in the table, but is no longer accessible.

This means that when you executed the statement "`UPDATE recalls SET potaff = potaff + 1`", PostgreSQL inserted 39,241 new rows and deleted 39,241 old rows. We now have 78,482 rows, half of which are inaccessible.

Why does PostgreSQL carry out an `UPDATE` command this way? The answer lies in PostgreSQL's MVCC (multi-version concurrency control) feature. Consider the following commands:

```
perf=# BEGIN WORK;
BEGIN
perf=# UPDATE recalls SET potaff = potaff + 1;
UPDATE
```

Notice that you have started a new transaction, but you have not yet completed it. If another user were to `SELECT` rows from the `recalls` table at this point, he must see the old values—you might roll back this transaction. In other database systems (such as DB2, Sybase, and SQL Server), the other user would have to wait until you either committed or rolled back your transaction before his query would complete. PostgreSQL, on the other hand, keeps the old rows in the table, and other users will see the original values until you commit your transaction. If you roll back your changes, PostgreSQL simply hides your modifications from all transactions (leaving you with 78,482 rows, half of which are inaccessible).

When you `DELETE` rows from a table, PostgreSQL follows a similar set of rules. The deleted rows remain in the table, but are hidden. If you roll back a `DELETE` command, PostgreSQL will simply make the rows visible again.

Now you also know the difference between a tuple and a row. A tuple is some version of a row.

When you make a change to a table, the tuples that you've changed are hidden from other users until you `COMMIT` your changes. If you `INSERT` 100,000 new rows into a table that previously contained only a few rows, another user might suddenly see a decrease in query performance even though he can't see the new rows. If you roll back your changes, other users will never see the new rows, but the dead tuples that you've created will continue to affect performance until someone `VACUUM`s the table.

You can see that these hidden tuples can dramatically affect performance—updating every row in a table doubles the number of heap blocks required to read the entire table.

There are at least three ways to remove dead tuples from a database. One way is to export all (visible) rows and then import them again using `pg_dump` and `pg_restore`. Another method is to use `CREATE TABLE ... AS` to make a new copy of the table, drop the original table, and rename the copy. The preferred way is to use the `VACUUM` command. I'll show you how to use the `VACUUM` command a little later (see the section "[Table Statistics](#)").

Index Performance

You've seen how PostgreSQL batches all disk I/O into 8K blocks, and you've seen how PostgreSQL maintains a buffer cache to reduce disk I/O. Let's find out what happens when you throw an index into the mix. After restarting the postmaster (to clear the cache), execute the following query:

```
$ timer "SELECT * FROM recalls ORDER BY record_id;" recalls
```

SEQUENTIAL I/O				INDEXED I/O			
scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
0	0	26398	12843	1	39241	146	0

You can see that PostgreSQL chose to execute this query using an index scan (remember, you have an index defined on the `record_id` column). This query read 146 index blocks and found none in the buffer cache. You also processed 26,398 heap blocks and found 12,843 in the cache. You can see that the buffer cache helped the performance a bit, but you still processed over 26,000 heap blocks, and you need only 4,413 to hold the entire `recalls` table. Why did you need to read each heap block (approximately) five times?

Think of how the `recalls` table is stored on disk (see [Figure 4.2](#)).

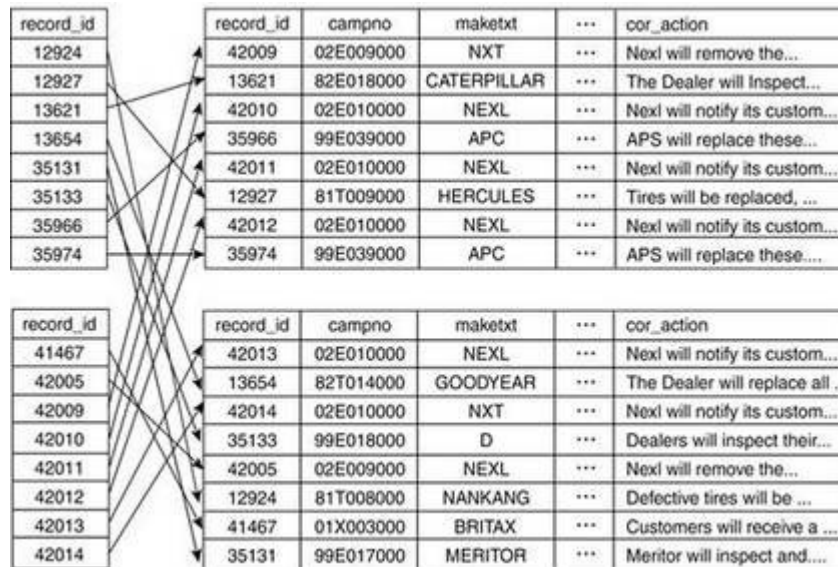
Figure 4.2. The `recalls` table on disk.

Page 1	record_id	campno	maketxt	...	cor_action
	42009	02E009000	NXT	...	Next will remove the...
	13621	82E018000	CATERPILLAR	...	The Dealer will inspect...
	42010	02E010000	NEXL	...	Next will notify its custom...
	35966	99E039000	APC	...	APS will replace these...
	42011	02E010000	NEXL	...	Next will notify its custom...
	12927	81T009000	HERCULES	...	Tires will be replaced,...
	42012	02E010000	NEXL	...	Next will notify its custom...
	35974	99E039000	APC	...	APS will replace these...
...					
Page 2	record_id	campno	maketxt	...	cor_action
	42013	02E010000	NEXT	...	Next will notify its custom...
	13654	82T014000	GOODYEAR	...	The Dealer will replace all...
	42014	02E010000	NXT	...	Next will notify its custom...
	35133	99E018000	D	...	Dealers will inspect their...
	42005	02E009000	NEXL	...	Next will remove the...
	12924	81T008000	NANKANG	...	Defective tires will be...
	41467	01X003000	BRITAX	...	Customers will receive a...
	35131	99E017000	MERITOR	...	Meritor will inspect and...
...					
Page 4412	record_id	campno	maketxt	...	cor_action
	42054	02V074002	PONTIAC	...	Dealers will properly tight...
	41863	02V044000	HYUNDAI	...	Dealers will inspect the ...
	42139	02V095000	INTERNATIONAL	...	Dealers will inspect the ...
	41926	02V065000	COUNTRY COACH	...	Dealers will replace the...
	42138	02V095000	INTERNATIONAL	...	Dealers will inspect the...
	41927	02V065000	COUNTRY COACH	...	Dealers will replace the...
	42140	02V095000	INTERNATIONAL	...	Dealers will inspect the...
	41930	02V065000	COUNTRY COACH	...	Dealers will replace the...

Notice that the rows are not stored in `record_id` order. In fact, they are stored in order of insertion. When you create an index on the `record_id` column, you end up with a structure like that shown in [Figure 4.3](#).

Figure 4.3. The `recalls` table structure after creating an index.

[\[View full size image\]](#)



Consider how PostgreSQL uses the `record_id` index to satisfy the query. After the first block of the `record_id` index is read into the buffer cache, PostgreSQL starts scanning through the index entries. The first index entry points to a `recalls` row on heap block 2, so that heap block is read into the buffer cache. Now, PostgreSQL moves on to the second index entry—this one points to a row in heap block 1. PostgreSQL reads heap block 1 into the buffer cache, throwing out some other page if there is no room in the cache. Figure 4.2 shows a partial view of the `recalls` table: remember that there are actually 4,413 heap blocks and 146 index blocks needed to satisfy this query. It's the random ordering of the rows within the `recalls` table that kills the cache hit ratio.

Let's try reordering the `recalls` table so that rows are inserted in `record_id` order. First, create a work table with the same structure as `recalls`:

```
perf=# CREATE TABLE work_recalls AS
perf=# SELECT * FROM recalls ORDER BY record_id;
SELECT
```

Then, drop the original table, rename the work table, and re-create the index:

```
perf=# DROP TABLE recalls;
DROP
perf=# ALTER TABLE work_recalls RENAME TO recalls;
ALTER
perf=# CREATE INDEX recalls_record_id ON recalls( record_id );
CREATE
```

At this point, you have the same data as before, consuming the same amount of space:

```
perf=# SELECT relname, relpages, reltuples FROM pg_class
perf=# WHERE relname IN ('recalls', 'recalls_record_id' );
 relname | relpages | reltuples
-----+-----+-----
recalls_record_id | 146 | 39241
recalls | 4422 | 39241
(2 rows)
```

After restarting the `postmaster` (again, this clears out the buffer cache so you get consistent results), let's re-execute the previous query:

```
$ timer "SELECT * FROM recalls ORDER BY record_id;" recalls
+-----+-----+-----+-----+-----+-----+-----+-----+
| SEQUENTIAL I/O | INDEXED I/O |
| scans | tuples | heap_blks | cached | scans | tuples | idx_blks | cached |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 | 0 | 4423 | 34818 | 1 | 39241 | 146 | 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

That made quite a difference. Before reordering, you read 26,398 heap blocks from disk and found 12,843 in the cache for a 40% cache hit ratio. After physically reordering the rows to match the index, you read 4,423 heap blocks from disk and found 34,818 in the cache for hit ratio of 787%. This makes a huge performance difference. Now as you read through each index page, the heap records appear next to each other; you won't be thrashing heap pages in and out of the cache. Figure 4.4 shows how the `recalls` table looks after reordering.

Figure 4.4. The `recalls` table on disk after reordering.

[\[View full size image\]](#)

record_id		record_id	campno	maketxt	...	cor_action
12924	→	12924	81T008000	NANKANG	...	Defective tires will...
12927	→	12927	81T009000	HERCULES	...	Tires will be replaced...
13621	→	13621	82E018000	CATERPILLAR	...	The Dealer will inspect...
13654	→	13654	82T014000	GOODYEAR	...	The Dealer will replace all...
35131	→	35131	99E017000	MERITOR	...	Meritor will inspect and...
35133	→	35133	99E018000	D	...	Dealers will inspect their...
35966	→	35966	99E039000	APC	...	APS will replace these...
35974	→	35974	99E039000	APC	...	APS will replace these...

record_id		record_id	campno	maketxt	...	cor_action
41467	→	41467	01X003000	BRITAX	...	Customers will receive a...
42005	→	42005	02E009000	NEXL	...	Nexl will remove the...
42009	→	42009	02E009000	NXT	...	Nexl will notify its custom...
42010	→	42010	02E010000	NEXL	...	Nexl will notify its custom...
42011	→	42011	02E010000	NEXL	...	Nexl will notify its custom...
42012	→	42012	02E010000	NEXL	...	Nexl will notify its custom...
42013	→	42013	02E010000	NEXL	...	Nexl will notify its custom...
42014	→	42014	02E010000	NXT	...	Nexl will notify its custom...

We reordered the `recalls` table by creating a copy of the table (in the desired order), dropping the original table, and then renaming the copy back to the original name. You can also use the `CLUSTER` command—it does exactly the same thing.

Understanding How PostgreSQL Executes a Query

Before going much further, you should understand the procedure that PostgreSQL follows whenever it executes a query on your behalf.

After the PostgreSQL server receives a query from the client application, the text of the query is handed to the parser. The parser scans through the query and checks it for syntax errors. If the query is syntactically correct, the parser will transform the query text into a parse tree. A parse tree is a data structure that represents the meaning of your query in a formal, unambiguous form.

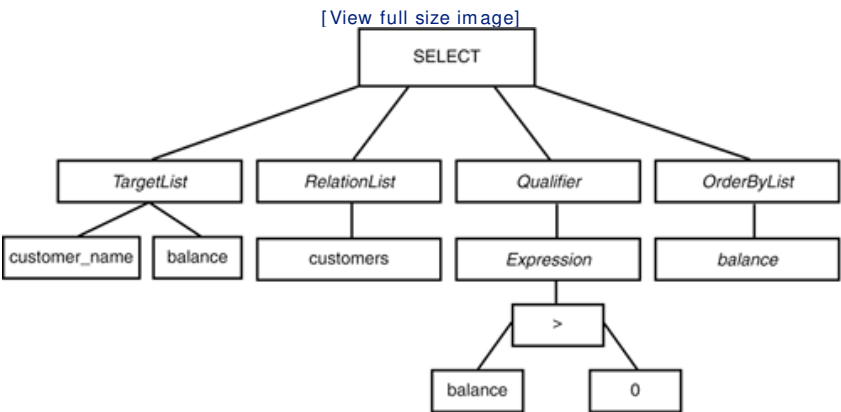
Given the query

Code View: [Scroll](#) / [Show All](#)

```
SELECT customer_name, balance FROM customers WHERE balance > 0 ORDER BY balance
```

the parser might come up with a parse tree structured as shown in [Figure 4.5](#).

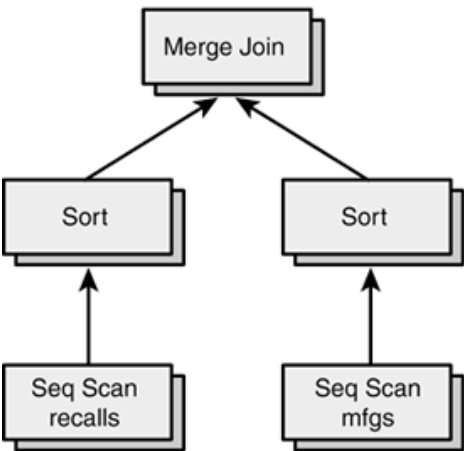
Figure 4.5. A sample parse tree.



After the parser has completed parsing the query, the parse tree is handed off to the planner/optimizer.

The planner is responsible for traversing the parse tree and finding all possible plans for executing the query. The plan might include a sequential scan through the entire table and index scans if useful indexes have been defined. If the query involves two or more tables, the planner can suggest a number of different methods for joining the tables. The execution plans are developed in terms of query operators. Each query operator transforms one or more input sets into an intermediate result set. The *Seq Scan* operator, for example, transforms an input set (the physical table) into a result set, filtering out any rows that don't meet the query constraints. The *Sort* operator produces a result set by reordering the input set according to one or more sort keys. I'll describe each of the query operators in more detail a little later. [Figure 4.6](#) shows an example of a simple execution plan (it is a new example; it is not related to the parse tree in [Figure 4.5](#)).

Figure 4.6. A simple execution plan.



You can see that complex queries are broken down into simple steps. The input set for a query operator at the bottom of the tree is usually a physical table. The input set for an upper-level operator is the result set of a lower-level operator.

When all possible execution plans have been generated, the optimizer searches for the least-expensive plan. Each plan is assigned an estimated execution cost. Cost estimates are measured in units of disk I/O. An operator that reads a single block of 8,192 bytes (8K) from the disk has a cost of one unit. CPU time is also measured in disk I/O units, but usually as a fraction. For example, the amount of CPU time

required to process a single tuple is assumed to be $1/100^{\text{th}}$ of a single disk I/O. You can adjust many of the cost estimates. Each query operator has a different cost estimate. For example, the cost of a sequential scan of an entire table is computed as the number of 8K blocks in the table, plus some CPU overhead.

After choosing the (apparently) least-expensive execution plan, the query executor starts at the beginning of the plan and asks the topmost operator to produce a result set. Each operator transforms its input set into a result set—the input set may come from another operator lower in the tree. When the topmost operator completes its transformation, the results are returned to the client application.

EXPLAIN

The `EXPLAIN` statement gives you some insight into how the PostgreSQL query planner/optimizer decides to execute a query.

First, you should know that the `EXPLAIN` statement can be used only to analyze `SELECT`, `INSERT`, `DELETE`, `UPDATE`, and `DECLARE...CURSOR` commands.

The syntax for the `EXPLAIN` command is

```
EXPLAIN [ANALYZE] [VERBOSE] query;
```

Let's start by looking at a simple example:

```
perf=# EXPLAIN ANALYZE SELECT * FROM recalls;
NOTICE: QUERY PLAN:
Seq Scan on recalls (cost=0.00..9217.41 rows=39241 width=1917)
      (actual time=69.35..3052.72 rows=39241 loops=1)
Total runtime: 3144.61 msec
```

The format of the execution plan can be a little mysterious at first. For each step in the execution plan, `EXPLAIN` prints the following information:

- The type of operation required.
- The estimated cost of execution.
- If you specified `EXPLAIN ANALYZE`, the actual cost of execution. If you omit the `ANALYZE` keyword, the query is planned but not executed, and the actual cost is not displayed.

In this example, PostgreSQL has decided to perform a sequential scan of the `recalls` table (`Seq Scan on recalls`). There are many operators that PostgreSQL can use to execute a query. I'll explain the operation type in more detail in a moment.

There are three data items in the cost estimate. The first set of numbers (`cost=0.00..9217.41`) is an estimate of how "expensive" this operation will be. "Expensive" is measured in terms of disk reads. Two numbers are given: The first number represents how quickly the first row in the result set can be returned by the operation; the second (which is usually the most important) represents how long the entire operation should take. The second data item in the cost estimate (`rows=39241`) shows how many rows PostgreSQL expects to return from this operation. The final data item (`width=1917`) is an estimate of the width, in bytes, of the average row in the result set.

If you include the `ANALYZE` keyword in the `EXPLAIN` command, PostgreSQL will execute the query and display the actual execution costs.

Cost Estimates

I will remove the cost estimates from some of the `EXPLAIN` results in this chapter to make the plan a bit easier to read. Don't be confused by this—the `EXPLAIN` command will always print cost estimates.

This was a simple example. PostgreSQL required only one step to execute this query (a sequential scan on the entire table). Many queries require multiple steps and the `EXPLAIN` command will show you each of those steps. Let's look at a more complex example:

```
perf=# EXPLAIN ANALYZE SELECT * FROM recalls ORDER BY yeartxt;
NOTICE: QUERY PLAN:

Sort (cost=145321.51..145321.51 rows=39241 width=1911)
      (actual time=13014.92..13663.86 rows=39241 loops=1)

  ->Seq Scan on recalls (cost=0.00..9217.41 rows=39241 width=1917)
        (actual time=68.99..3446.74 rows=39241 loops=1)
Total runtime: 16052.53 msec
```

This example shows a two-step query plan. In this case, the first step is actually listed at the end of the plan. When you read a query plan, it is important to remember that each step in the plan produces an intermediate result set. Each intermediate result set is fed into the next step of the plan.

Looking at this plan, PostgreSQL first produces an intermediate result set by performing a sequential scan (`Seq Scan`) on the entire `recalls` table. That step should take about 9,217 disk page reads, and the result set will have about 39,241 rows, averaging 1,917 bytes each. Notice that these estimates are identical to those produced in the first example—and in both cases, you are executing a sequential scan on the entire table.

After the sequential scan has finished building its intermediate result set, it is fed into the next step in the plan. The final step in this particular plan is a sort operation, which is required to satisfy our `ORDER BY` clause^[7]. The sort operation reorders the result set produced by the sequential scan and returns the final result set to the client application.

^[7] An `ORDER BY` clause does not require a `Sort` operation in all cases. The planner/optimizer may decide that it can use an index to order the result set.

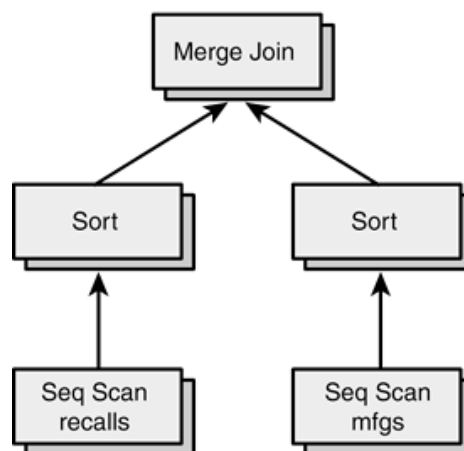
The `Sort` operation expects a single operand—a result set. The `Seq Scan` operation expects a single operand—a table. Some operations require more than one operand. Here is a join between the `recalls` table and the `mfgs` table:

```
perf=# EXPLAIN SELECT * FROM recalls, mfgs
perf=# WHERE recalls.mfgname = mfgs.mfgname;
NOTICE: QUERY PLAN:

Merge Join
-> Sort
    -> Seq Scan on recalls
-> Sort
    -> Seq Scan on mfgs
```

If you use your imagination, you will see that this query plan is actually a tree structure, as illustrated in Figure 4.7.

Figure 4.7. Execution plan viewed as a tree.



When PostgreSQL executes this query plan, it starts at the top of the tree. The `Merge Join` operation requires two result sets for input, so PostgreSQL must move down one level in the tree; let's assume that you traverse the left child first. Each `Sort` operation requires a single result set for input, so again the query executor moves down one more level. At the bottom of the tree, the `Seq Scan` operation simply reads a row from a table and returns that row to its parent. After a `Seq Scan` operation has scanned the entire table, the left-hand `Sort` operation can complete. As soon as the left-hand `Sort` operation completes, the `Merge Join` operator will evaluate its right child. In this case, the right-hand child evaluates the same way as the left-hand child. When both `Sort` operations complete, the `Merge Join` operator will execute, producing the final result set.

So far, you've seen three query execution operators in the execution plans. PostgreSQL currently has 19 query operators. Let's look at each in more detail.

Seq Scan

The `Seq Scan` operator is the most basic query operator. Any single-table query can be carried out using the `Seq Scan` operator.

`Seq Scan` works by starting at the beginning of the table and scanning to the end of the table. For each row in the table, `Seq Scan` evaluates the query constraints^[8] (that is, the `WHERE` clause); if the constraints are satisfied, the required columns are added to the result set.

^[8] The entire `WHERE` clause may not be evaluated for each row in the input set. PostgreSQL evaluates only the portions of the clause that apply to the given row (if any). For a single-table `SELECT`, the entire `WHERE` clause is evaluated. For a multi-table join, only the portion that applies to the given row is evaluated.

As you saw earlier in this chapter, a table can include dead (that is, deleted) rows and rows that may not be visible because they have not been committed. `Seq Scan` does not include dead rows in the result set, but it must read the dead rows, and that can be expensive in a heavily updated table.

The cost estimate for a `Seq Scan` operator gives you a hint about how the operator works:

```
Seq Scan on recalls (cost=0.00..9217.41 rows=39241 width=1917)
```

The startup cost is always 0.00. This implies that the first row of a `Seq Scan` operator can be returned immediately and that `Seq Scan` does not read the entire table before returning the first row. If you open a cursor against a query that uses the `Seq Scan` operator (and no other operators), the first `FETCH` will return immediately—you won't have to wait for the entire result set to be materialized before you can `FETCH`

the first row. Other operators (such as `Sort`) do read the entire input set before returning the first row.

The planner/optimizer chooses a `Seq Scan` if there are no indexes that can be used to satisfy the query. A `Seq Scan` is also used when the planner/optimizer decides that it would be less expensive (or just as expensive) to scan the entire table and then sort the result set to meet an ordering constraint (such as an `ORDER BY` clause).

Index Scan

An `Index Scan` operator works by traversing an index structure. If you specify a starting value for an indexed column (`WHERE record_id >= 1000`, for example), the `Index Scan` will begin at the appropriate value. If you specify an ending value (such as `WHERE record_id < 2000`), the `Index Scan` will complete as soon as it finds an index entry greater than the ending value.

The `Index Scan` operator has two advantages over the `Seq Scan` operator. First, a `Seq Scan` must read every row in the table—it can only remove rows from the result set by evaluating the `WHERE` clause for each row. `Index Scan` may not read every row if you provide starting and/or ending values. Second, a `Seq Scan` returns rows in table order, not in sorted order. `Index Scan` will return rows in index order.

Not all indexes are scannable. The `B-Tree`, `R-Tree`, and `GiST` index types can be scanned; a `Hash` index cannot.

The planner/optimizer uses an `Index Scan` operator when it can reduce the size of the result set by traversing a range of indexed values, or when it can avoid a sort because of the implicit ordering offered by an index.

Sort

The `Sort` operator imposes an ordering on the result set. PostgreSQL uses two different sort strategies: an in-memory sort and an on-disk sort. You can tune a PostgreSQL instance by adjusting the value of the `sort_mem` runtime parameter. If the size of the result set exceeds `sort_mem`, `Sort` will distribute the input set to a collection of sorted work files and then merge the work files back together again. If the result set will fit in `sort_mem*1024` bytes, the sort is done in memory using the `QSort` algorithm.

A `Sort` operator never reduces the size of the result set—it does not remove rows or columns.

Unlike `Seq Scan` and `Index Scan`, the `Sort` operator must process the entire input set before it can return the first row.

The `Sort` operator is used for many purposes. Obviously, a `Sort` can be used to satisfy an `ORDER BY` clause. Some query operators require their input sets to be ordered. For example, the `Unique` operator (we'll see that in a moment) eliminates rows by detecting duplicate values as it reads through a sorted input set. `Sort` will also be used for some join operations, group operations, and for some set operations (such as `INTERSECT` and `UNION`).

Unique

The `Unique` operator eliminates duplicate values from the input set. The input set must be ordered by the columns, and the columns must be unique. For example, the following command

```
SELECT DISTINCT mfgrname FROM recalls;
```

might produce this execution plan:

```
Unique
-> Sort
   -> Seq Scan on recalls
```

The `Sort` operation in this plan orders its input set by the `mfgrname` column. `Unique` works by comparing the unique column(s) from each row to the previous row. If the values are the same, the duplicate is removed from the result set.

The `Unique` operator removes only rows—it does not remove columns and it does not change the ordering of the result set.

`Unique` can return the first row in the result set before it has finished processing the input set.

The planner/optimizer uses the `Unique` operator to satisfy a `DISTINCT` clause. `Unique` is also used to eliminate duplicates in a `UNION`.

LIMIT

The `LIMIT` operator is used to limit the size of a result set. PostgreSQL uses the `LIMIT` operator for both `LIMIT` and `OFFSET` processing. The `LIMIT` operator works by discarding the first `x` rows from its input set, returning the next `y` rows, and discarding the remainder. If the query includes an `OFFSET` clause, `x` represents the offset amount; otherwise, `x` is zero. If the query includes a `LIMIT` clause, `y` represents the `LIMIT` amount; otherwise, `y` is at least as large as the number of rows in the input set.

The ordering of the input set is not important to the `LIMIT` operator, but it is usually important to the overall query plan. For example, the query plan for this query

```
perf=# EXPLAIN SELECT * FROM recalls LIMIT 5;
NOTICE: QUERY PLAN:

Limit (cost=0.00..0.10 rows=5 width=1917)
-> Seq Scan on recalls (cost=0.00..9217.41 rows=39241 width=1917)
```

shows that the `LIMIT` operator rejects all but the first five rows returned by the `Seq Scan`. On the other hand, this query

```
perf=# EXPLAIN ANALYZE SELECT * FROM recalls ORDER BY yeartxt LIMIT 5;
NOTICE: QUERY PLAN:
```

```
Limit (cost=0.00..0.10 rows=5 width=1917)
->Sort (cost=145321.51..145321.51 rows=39241 width=1911)
->Seq Scan on recalls (cost=0.00..9217.41 rows=39241 width=1917)
```

shows that the `LIMIT` operator returns the first five rows from an ordered input set.

The `LIMIT` operator never removes columns from the result set, but it obviously removes rows.

The planner/optimizer uses a `LIMIT` operator if the query includes a `LIMIT` clause, an `OFFSET` clause, or both. If the query includes only a `LIMIT` clause, the `LIMIT` operator can return the first row before it processes the entire set.

Aggregate

The planner/optimizer produces an `Aggregate` operator whenever the query includes an aggregate function. The following functions are aggregate functions: `AVG()`, `COUNT()`, `MAX()`, `MIN()`, `STDDEV()`, `SUM()`, and `VARIANCE()`.

`Aggregate` works by reading all the rows in the input set and computing the aggregate values. If the input set is not grouped, `Aggregate` produces a single result row. For example:

```
movies=# EXPLAIN SELECT COUNT(*) FROM customers;
Aggregate (cost=22.50..22.50 rows=1 width=0)
-> Seq Scan on customers (cost=0.00..20.00 rows=1000 width=0)
```

If the input set is grouped, `Aggregate` produces one result row for each group:

```
movies=# EXPLAIN
movies-# SELECT COUNT(*), EXTRACT( DECADE FROM birth_date )
movies-# FROM customers
movies-# GROUP BY EXTRACT( DECADE FROM birth_date );
NOTICE: QUERY PLAN:

Aggregate (cost=69.83..74.83 rows=100 width=4)
-> Group (cost=69.83..72.33 rows=1000 width=4)
-> Sort (cost=69.83..69.83 rows=1000 width=4)
-> Seq Scan on customers (cost=0.00..20.00 rows=1000 width=4)
```

Notice that the row estimate of an ungrouped aggregate is always 1; the row estimate of a group aggregate is $1/10^{\text{th}}$ of the size of the input set.

Append

The `Append` operator is used to implement a `UNION`. An `Append` operator will have two or more input sets. `Append` works by returning all rows from the first input set, then all rows from the second input set, and so on until all rows from all input sets have been processed.

Here is a query plan that shows the `Append` operator:

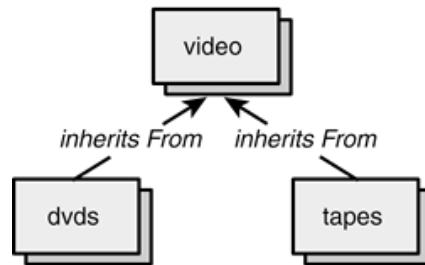
```
perf=# EXPLAIN
perf-# SELECT * FROM recalls WHERE mfgname = 'FORD'
perf-# UNION
perf-# SELECT * FROM recalls WHERE yeartxt = '1983';

Unique
->Sort
->Append
->Subquery Scan *SELECT* 1
->Seq Scan on recalls
->Subquery Scan *SELECT* 2
->Seq Scan on recalls
```

The cost estimate for an `Append` operator is simply the sum of cost estimates for all input sets. An `Append` operator can return its first row before processing all input rows.

The planner/optimizer uses an `Append` operator whenever it encounters a `UNION` clause. `Append` is also used when you select from a table involved in an inheritance hierarchy. In [Chapter 3](#), "PostgreSQL SQL Syntax and Use," I defined three tables, as shown in [Figure 4.8](#).

Figure 4.8. Inheritance hierarchy.



The `dvd` table inherits from `video`, as does the `tape` table. If you `SELECT` from `dvd` or `video`, PostgreSQL will respond with a simple query plan:

```

movies=# EXPLAIN SELECT * FROM dvd;
      Seq Scan on dvd  (cost=0.00..20.00 rows=1000 width=122)

movies=# EXPLAIN SELECT * FROM tape;
      Seq Scan on tape (cost=0.00..20.00 rows=1000 width=86)
  
```

Remember, because of the inheritance hierarchy, a `dvd` is a `video` and a `tape` is a `video`. If you `SELECT` from `video`, you would expect to see all `dvd`s, all `tapes`, and all `videos`. The query plan reflects the inheritance hierarchy:

```

movies=# EXPLAIN SELECT * FROM video;

Result (cost=0.00..60.00 rows=3000 width=86)
->Append(cost=0.00..60.00 rows=3000 width=86)
->Seq Scan on video  (cost=0.00..20.00 rows=1000 width=86)
->Seq Scan on tapes video  (cost=0.00..20.00 rows=1000 width=86)
->Seq Scan on dvd video  (cost=0.00..20.00 rows=1000 width=86)
  
```

Look closely at the `width` clause in the preceding cost estimates. If you `SELECT` from the `dvd` table, the `width` estimate is 122 bytes per row. If you `SELECT` from the `tape` table, the `width` estimate is 86 bytes per row. When you `SELECT` from `video`, all rows are expected to be 86 bytes long. Here are the commands used to create the `tapes` and `dvd` tables:

```

movies=# CREATE TABLE tapes ( ) INHERITS( video );

movies=# CREATE TABLE dvd
movies-# (
movies(#   region_id      INTEGER,
movies(#   audio_tracks  VARCHAR[]
movies(# ) INHERITS ( video );
  
```

You can see that a row from the `tapes` table is identical to a row in the `video` table—you would expect them to be the same size (86 bytes). A row in the `dvd` table contains a `video` plus a few extra columns, so you would expect a `dvd` row to be longer than a `video` row. When you `SELECT` from the `video` table, you want all `videos`. PostgreSQL discards any columns that are not inherited from the `video` table.

Result

The `Result` operator is used in three contexts.

First, a `Result` operator is used to execute a query that does not retrieve data from a table:

```

movies=# EXPLAIN SELECT timeofday();
      Result
  
```

In this form, the `Result` operator simply evaluates the given expression(s) and returns the results.

`Result` is also used to evaluate the parts of a `WHERE` clause that don't depend on data retrieved from a table. For example:

```

movies=# EXPLAIN SELECT * FROM tapes WHERE 1 <> 1;
      Result
->Seq Scan on tapes
  
```

This might seem like a silly query, but some client applications will generate a query of this form as an easy way to retrieve the metadata (that is, column definitions) for a table.

In this form, the `Result` operator first evaluates the constant part of the `WHERE` clause. If the expression evaluates to `FALSE`, no further processing is required and the `Result` operator completes. If the expression evaluates to `TRUE`, `Result` will return its input set.

The planner/optimizer also generates a `Result` operator if the top node in the query plan is an `Append` operator. This is a rather obscure rule that has no performance implications; it just happens to make the query planner and executor a bit simpler for the PostgreSQL developers to maintain.

Nested Loop

The `Nested Loop` operator is used to perform a join between two tables. A `Nested Loop` operator requires two input sets (given that a `Nested Loop` joins two tables, this makes perfect sense).

`Nested Loop` works by fetching each row from one of the input sets (called the outer table). For each row in the outer table, the other input (called the inner table) is searched for a row that meets the join qualifier.

Here is an example:

```
perf=# EXPLAIN
perf=#   SELECT * FROM customers, rentals
perf=#   WHERE customers.customer_id = rentals.customer_id;

Nested Loop
  -> Seq Scan on rentals
  -> Index Scan using customer_id on customers
```

The outer table is always listed first in the query plan (in this case, `rentals` is the outer table). To execute this plan, the `Nested Loop` operator will read each row^[9] in the `rentals` table. For each `rentals` row, `Nested Loop` reads the corresponding `customers` row using an indexed lookup on the `customer_id` index.

^[9] Actually, `Nested Loop` reads only those rows that meet the query constraints.

A `Nested Loop` operator can be used to perform inner joins, left outer joins, and unions.

Because `Nested Loop` does not process the entire inner table, it can't be used for other join types (full, right join, and so on).

Merge Join

The `Merge Join` operator also joins two tables. Like the `Nested Loop` operator, `Merge Join` requires two input sets: an outer table and an inner table. Each input set must be ordered by the join columns.

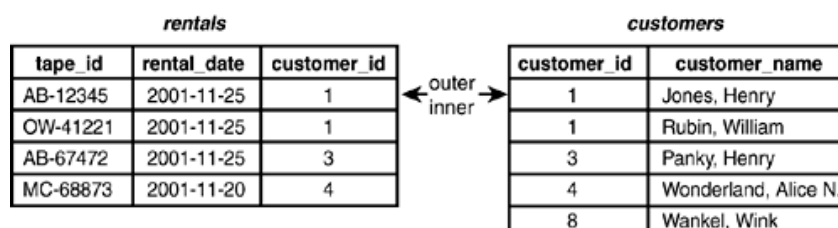
Let's look at the previous query, this time executed as a `Merge Join`:

```
perf=# EXPLAIN
perf=#   SELECT * FROM customers, rentals
perf=#   WHERE customers.customer_id = rentals.customer_id;

Merge Join
  -> Sort
  -> Seq Scan on rentals
  -> Index Scan using customer_id on customers
```

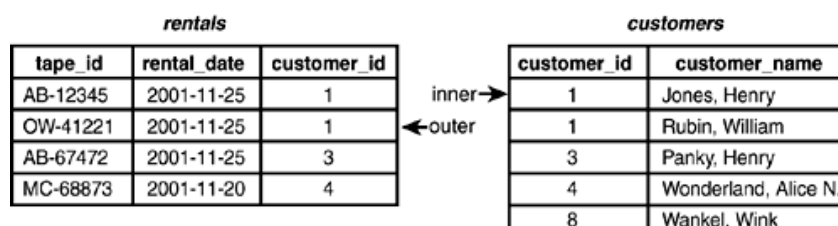
`Merge Join` starts reading the first row from each table (see [Figure 4.9](#)).

Figure 4.9. Merge Join—Step 1.



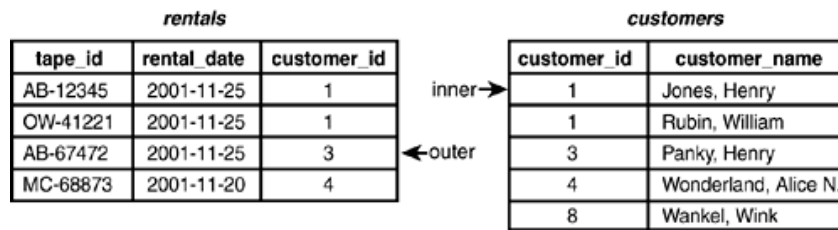
If the join columns are equal (as in this case), `Merge Join` creates a new row containing the necessary columns from each input table and returns the new row. `Merge Join` then moves to the next row in the outer table and joins it with the corresponding row in the inner table (see [Figure 4.10](#)).

Figure 4.10. Merge Join—Step 2.



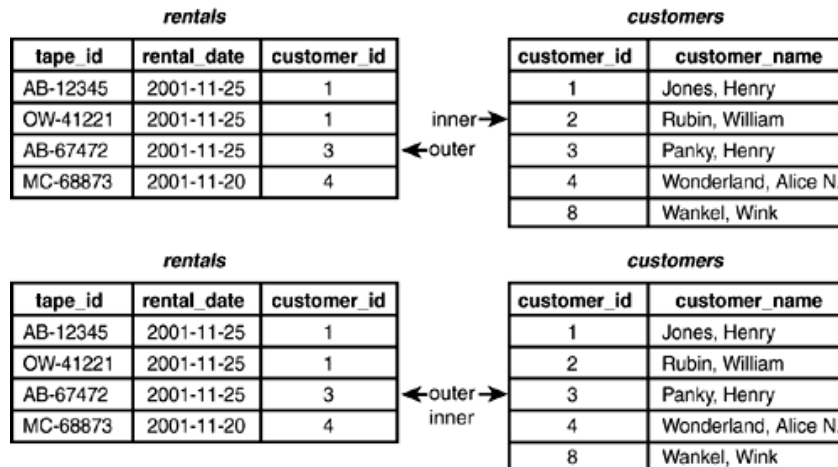
Next, `Merge Join` reads the third row in the outer table (see [Figure 4.11](#)).

Figure 4.11. Merge Join—Step 3.



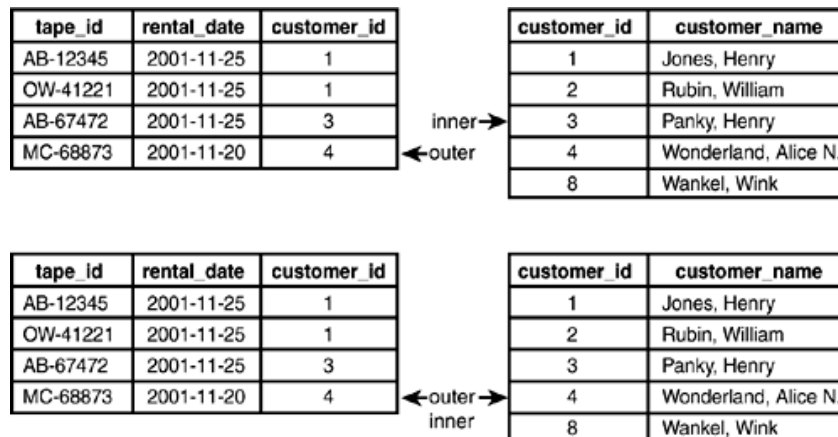
Now Merge Join must advance the inner table twice before another result row can be created (see Figure 4.12).

Figure 4.12. Merge Join—Step 4.



After producing the result row for `customer_id = 3`, Merge Join moves to the last row in the outer table and then advances the inner table to a matching row (see Figure 4.13).

Figure 4.13. Merge Join—Step 5.



Merge Join completes by producing the final result row (`customer_id = 4`).

You can see that Merge Join works by walking through two sorted tables and finding matches—the trick is in keeping the pointers synchronized.

This example shows an inner join, but the Merge Join operator can be used for other join types by walking through the sorted input sets in different ways. Merge Join can do inner joins, outer joins, and unions.

Hash and Hash Join

The Hash and Hash Join operators work together. The Hash Join operator requires two input sets, again called the outer and inner tables. Here is a query plan that uses the Hash Join operator:

```
movies=# EXPLAIN
movies=#   SELECT * FROM customers, rentals
movies=#     WHERE rentals.customer_id = customers.customer_id;
```

```

Hash Join
-> Seq Scan on customers
-> Hash
    -> Seq Scan on rentals

```

Unlike other join operators, `Hash Join` does not require either input set to be ordered by the join column. Instead, the inner table is always a hash table, and the ordering of the outer table is not important.

The `Hash Join` operator starts by creating its inner table using the `Hash` operator. The `Hash` operator creates a temporary Hash index that covers the join column in the inner table.

Once the hash table (that is, the inner table) has been created, `Hash Join` reads each row in the outer table, hashes the join column (from the outer table), and searches the temporary Hash index for a matching value.

A `Hash Join` operator can be used to perform inner joins, left outer joins, and unions.

Group

The `Group` operator is used to satisfy a `GROUP BY` clause. A single input set is required by the `Group` operator, and it must be ordered by the grouping column(s).

`Group` can work in two distinct modes. If you are computing a grouped aggregate, `Group` will return each row in its input set, following each group with a `NULL` row to indicate the end of the group (the `NULL` row is for internal bookkeeping only, and it will not show up in the final result set). For example:

```

movies=# EXPLAIN
movies-#   SELECT COUNT(*), EXTRACT( DECADE FROM birth_date )
movies-#   FROM customers
movies-#   GROUP BY EXTRACT( DECADE FROM birth_date );
NOTICE:  QUERY PLAN:

Aggregate (cost=69.83..74.83 rows=100 width=4)
-> Group (cost=69.83..72.33 rows=1000 width=4)
    -> Sort (cost=69.83..69.83 rows=1000 width=4)
        -> Seq Scan on customers (cost=0.00..20.00 rows=1000 width=4)

```

Notice that the row count in the `Group` operator's cost estimate is the same as the size of its input set.

If you are not computing a group aggregate, `Group` will return one row for each group in its input set. For example:

```

movies=# EXPLAIN
movies-#   SELECT EXTRACT( DECADE FROM birth_date ) FROM customers
movies-#   GROUP BY EXTRACT( DECADE FROM birth_date );

Group (cost=69.83..69.83 rows=100 width=4)
-> Sort (cost=69.83..69.83 rows=1000 width=4)
    -> Seq Scan on customers (cost=0.00..20.00 rows=1000 width=4)

```

In this case, the estimated row count is $1/10^{\text{th}}$ of the `Group` operator's input set.

Subquery Scan and Subplan

A `Subquery Scan` operator is used to satisfy a `UNION` clause; `Subplan` is used for subselects. These operators scan through their input sets, adding each row to the result set. Each of these operators are used for internal bookkeeping purposes and really don't affect the overall query plan—you can usually ignore them.

Just so you know when they are likely to be used, here are two sample query plans that show the `Subquery Scan` and `Subplan` operators:

```

perf=# EXPLAIN
perf-#   SELECT * FROM recalls WHERE mfgname = 'FORD'
perf-#   UNION
perf-#   SELECT * FROM recalls WHERE yeartxt = '1983';

Unique
->Sort
    ->Append
        ->Subquery Scan *SELECT* 1
            ->Seq Scan on recalls
        ->Subquery Scan *SELECT* 2
            ->Seq Scan on recalls

movies=# EXPLAIN
movies-#   SELECT * FROM customers
movies-#   WHERE customer_id IN
movies-#   (
movies-#       SELECT customer_id FROM rentals
movies-#   );
NOTICE:  QUERY PLAN:

Seq Scan on customers (cost=0.00..3.66 rows=2 width=47)

```

```
SubPlan
-> Seq Scan on rentals (cost=0.00..1.04 rows=4 width=4)
```

Tid Scan

The **Tid Scan** (tuple ID scan) operator is rarely used. A tuple is roughly equivalent to a row. Every tuple has an identifier that is unique within a table—this is called the tuple ID. When you select a row, you can ask for the row's tuple ID:

```
movies=# SELECT ctid, customer_id, customer_name FROM customers;
 ctid | customer_id | customer_name
-----+-----+-----
(0,1) |          3 | Panky, Henry
(0,2) |          1 | Jones, Henry
(0,3) |          4 | Wonderland, Alice N.
(0,4) |          2 | Rubin, William
(4 rows)
```

The "ctid" is a special column (similar to the `OID`) that is automatically a part of every row. A tuple ID is composed of a block number and a tuple number within the block. All the rows in the previous sample are stored in block 0 (the first block of the table file). The `customers` row for "Panky, Henry" is stored in tuple 3 of block 0.

After you know a row's tuple ID, you can request that row again by using its ID:

```
movies=# SELECT customer_id, customer_name FROM customers
movies=# WHERE ctid = '(0,3)';
 customer_id | customer_name
-----+-----
          4 | Wonderland, Alice N.
(1 row)
```

The tuple ID works like a bookmark. A tuple ID, however, is valid only within a single transaction. After the transaction completes, the tuple ID should not be used.

The **Tid Scan** operator is used whenever the planner/optimizer encounters a constraint of the form `ctid = expression` or `expression = ctid`.

The fastest possible way to retrieve a row is by its tuple ID. When you **SELECT** by tuple ID, the **Tid Scan** operator reads the block specified in the tuple ID and returns the requested tuple.

Materialize

The **Materialize** operator is used for some subselect operations. The planner/optimizer may decide that it is less expensive to materialize a subselect once than to repeat the work for each top-level row.

Materialize will also be used for some merge-join operations. In particular, if the inner input set of a **Merge Join** operator is not produced by a **Seq Scan**, an **Index Scan**, a **Sort**, or a **Materialize** operator, the planner/optimizer will insert a **Materialize** operator into the plan. The reasoning behind this rule is not obvious—it has more to do with the capabilities of the other operators than with the performance or the structure of your data. The **Merge Join** operator is complex; one requirement of **Merge Join** is that the input sets must be ordered by the join columns. A second requirement is that the inner input set must be repositionable; that is, **Merge Join** needs to move backward and forward through the input set. Not all ordered operators can move backward and forward. If the inner input set is produced by an operator that is not repositionable, the planner/optimizer will insert a **Materialize**.

Setop (Intersect, Intersect All, Except, Except All)

There are four **Setop** operators: **Setop Intersect**, **Setop Intersect All**, **Setop Except**, and **Setop Except All**. These operators are produced only when the planner/optimizer encounters an **INTERSECT**, **INTERSECT ALL**, **EXCEPT**, or **EXCEPT ALL** clause, respectively.

All **Setop** operators require two input sets. The **Setop** operators work by first combining the input sets into a sorted list, and then groups of identical rows are identified. For each group, the **Setop** operator counts the number of rows contributed by each input set. Finally, each **Setop** operator uses the counts to determine how many rows to add to the result set.

I think this will be easier to understand by looking at an example. Here are two queries; the first selects all `customers` born in the 1960s:

```
movies=# SELECT * FROM customers
movies=# WHERE EXTRACT( DECADE FROM birth_date ) = 196;
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
          3 | Panky, Henry | 555-1221 | 1968-01-21 | 0.00
          4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 | 3.00
```

The second selects all `customers` with a balance greater than 0:

```
movies=# SELECT * FROM customers WHERE balance > 0;
 customer_id | customer_name | phone | birth_date | balance
-----+-----+-----+-----+-----
          2 | Rubin, William | 555-2211 | 1972-07-10 | 15.00
          4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 | 3.00
```

Now, combine these two queries with an `INTERSECT` clause:

```
movies=# EXPLAIN
movies=#  SELECT * FROM customers
movies=#    WHERE EXTRACT( DECADE FROM birth_date ) = 196
movies=#  INTERSECT
movies=#    SELECT * FROM customers WHERE balance > 0;
SetOp Intersect
-> Sort
-> Append
-> Subquery Scan *SELECT* 1
-> Seq Scan on customers
-> Subquery Scan *SELECT* 2
-> Seq Scan on customers
```

The query executor starts by executing the two subqueries and then combining the results into a sorted list. An extra column is added that indicates which input set contributed each row:

customer_id	customer_name	birth_date	balance	input set
2	Rubin, William	1972-07-10	15.00	inner
3	Panky, Henry	1968-01-21	0.00	outer
4	Wonderland, Alice N.	1969-03-05	3.00	outer
4	Wonderland, Alice N.	1969-03-05	3.00	inner

The `SetOp` operator finds groups of duplicate rows (ignoring the input set pseudo-column). For each group, `SetOp` counts the number of rows contributed by each input set. The number of rows contributed by the outer set is called `count(outer)`. The number of rows contributed by the inner result set is called `count(inner)`.

Here is how the sample looks after counting each group:

customer_id	customer_name	birth_date	balance	input set
2	Rubin, William	1972-07-10	15.00	inner
				count(outer) = 0
				count(inner) = 1
3	Panky, Henry	1968-01-21	0.00	outer
				count(outer) = 1
				count(inner) = 0
4	Wonderland, Alice N.	1969-03-05	3.00	outer
4	Wonderland, Alice N.	1969-03-05	3.00	inner
				count(outer) = 1
				count(inner) = 1

The first group contains a single row, contributed by the inner input set. The second group contains a single row, contributed by the outer input set. The final group contains two rows, one contributed by each input set.

When `SetOp` reaches the end of a group of duplicate rows, it determines how many copies to write into the result set according to the following rules:

- **INTERSECT**— If `count(outer) > 0` and `count(inner) > 0`, write one copy of the row to the result set; otherwise, the row is not included in the result set.
- **INTERSECT ALL**— If `count(outer) > 0` and `count(inner) > 0`, write `n` copies of the row to the result set; where `n` is the greater `count(outer)` and `count(inner)`.
- **EXCEPT**— If `count(outer) > 0` and `count(inner) = 0`, write one copy of the row to the result set.
- **EXCEPT ALL**— If `count(inner) >= count(outer)`, write `n` copies of the row to the result set; where `n` is `count(outer) - count(inner)`.

Execution Plans Generated by the Planner

The `EXPLAIN` command only shows you the execution plan that PostgreSQL considered to be the least expensive. Unfortunately, you can't convince PostgreSQL to show the other execution plans that it considered. The most common performance question that we hear is "why didn't the database use my index?" If you could see all the alternatives, you could usually (but not always) answer that question.

When the optimizer generates a set of execution plans for a query, it starts by generating a set of plans that traverse each base table involved in the query. For a single-table query, there is only one base table and the planner generates a single set of execution plans. For a multitable query (a join), the planner starts by generating a set of traversal plans for each table.

There are only three ways that PostgreSQL can scan an individual table: a table scan (`Seq Scan`), an `Index Scan`, or a tuple-ID scan (`TID Scan`). For each table involved in the query, the planner generates one plan that makes a pass over the table using a `Seq Scan` operator. If the `WHERE` clause of the query selects one or more rows by `ctid` value, the planner generates a `TID Scan`. Next, the planner examines each index defined for the table. In theory, any single-table query can be satisfied by making a complete scan of a B-Tree index (assuming that the index is not a partial index), but the planner knows that a complete `Index Scan` is always more expensive than a complete `Seq Scan` and won't consider an `Index Scan` unless it offers some advantage.

An index is useful if it can reduce the number of tuples read from the table. If the `WHERE` clause for a query contains an expression of the form `indexCol operator constant-expression` or `constant-expression operator indexCol`, PostgreSQL may be able to use the index to read a subset of the table. For example, the `recalls` table has a B-Tree index that covers the `record_id` column. If you execute the query

```
perf=# SELECT * FROM recalls WHERE record_id > 8000;
```

the planner examines the expression `record_id > 8000` and finds that it is written in the form `indexCol operator constant-expression`—the `record_id` column is an `indexCol`. Notice that PostgreSQL looks for a `constant-expression`, not just a constant. That means that an expression such as `record_id > (800 * 10)` is acceptable as well. You can also include function calls in the `constant-expression` as long as the functions are not volatile. A volatile function (such as `random()`) can change value from row to row as PostgreSQL scans through the table. A `constant-expression` can include any operator that is not implemented by a volatile function.

Prior to version 8.0, PostgreSQL would only use an index if the data type of the indexed value exactly matched the data type of the `constant-expression`. Starting with 8.0, PostgreSQL will use an index if `constant-expression` can be coerced (that is, converted) to the same type as the indexed value.

An index is also useful if it can produce rows in a desired order. If an index produces rows in the sequence required by the `ORDER BY` clause, the planner will generate an `Index Scan` plan for the table. Some of the query operators (`MergeJoin`, `Unique`, `Group`, and `Setop`) require an ordered input set. For example, the `Unique` operator requires its input set to be ordered by the set of columns required to be unique. If an index can produce rows in the order required by one of these operators, the planner will generate an `Index Scan` plan for the table. A Hash index cannot produce rows in any particular order and therefore can't contribute to the ordering of a table.

What happens if you have two (or more) indexes that are useful to a given query? The planner generates a plan for each index and the optimizer chooses the least expensive plan among all of the alternatives.

Once the planner has generated a set of plans for each base table, it generates a set of plans to join the tables together according to the `WHERE` clause. PostgreSQL can join two tables together using any of three query operators: `Merge Join`, `Hash Join`, or `Nested Loop`. Consider a simple two-table query such as

```
movies=# SELECT * FROM rentals, customers
        WHERE rentals.customer_id = customers.customer_id;
```

Assuming that you have a B-Tree indexes that cover `rentals.customer_id` and `customers.customer_id`, the planner would generate the following plans to traverse each table individually:

```
SeqScan( rentals )
IndexScan( rentals.customer_id )
SeqScan( customers )
IndexScan( customers.customer_id )
```

To join these two tables together, the planner produces a set of execution plans.

First, the planner joins `rentals` and `customers` using the `MergeJoin` operator. Given that there are two paths through each table, the planner produces four `MergeJoin` plans for the combination of `rentals` and `customers`:

Code View: [Scroll](#) / Show All

```
MergeJoin( IndexScan( rentals.customer_id ), IndexScan( customers.customer_id ))
MergeJoin( Sort( SeqScan( rentals ) ), Sort( SeqScan( customers ) ) )
MergeJoin( IndexScan( rentals.customer_id ), Sort( SeqScan( customers ) ) )
MergeJoin( Sort( SeqScan( rentals ) ), IndexScan( customers.customer_id ) )
```

Notice that the `MergeJoin` operator requires both input set to be ordered by the join column—because a `SeqScan` operator does not produce rows in any particular order, the planner inserts a `Sort` operator where needed.

Next, the planner produces a set of four `NestedLoop` plans:

Code View: [Scroll](#) / Show All

```
NestedLoop( IndexScan( rentals.customer_id ), IndexScan( customers.customer_id ))
NestedLoop( SeqScan( rentals ), SeqScan( customers ))
NestedLoop( IndexScan( rentals.customer_id ), SeqScan( customers ))
NestedLoop( SeqScan( rentals ), IndexScan( customers.customer_id ))
```

Then, the planner considers a set of four HashJoin plans:

```
HashJoin
(
  IndexScan( rentals.customer_id ),
  Hash( IndexScan( customers.customer_id ))
)

HashJoin
(
  SeqScan( rentals ),
  Hash( SeqScan( customers ))
)

HashJoin
(
  IndexScan( rentals.customer_id ),
  Hash( SeqScan( customers ))
)

HashJoin
(
  SeqScan( rentals ),
  Hash( IndexScan( customers.customer_id ))
)
```

The HashJoin operator requires the inner input set to be a hash table so the planner inserts a Hash operator in front of each of the inner tables.

For a simple join, the planner has considered 12 plans. But it's not finished yet. The planner generates a second set of plans using customers as the outer table and rentals as the inner table (in the first set of join plans, rentals served as the outer table and customers served as the inner table):

Code View: [Scroll](#) / Show All

```
MergeJoin( IndexScan( customers.customer_id ), IndexScan( rentals.customer_id ))
MergeJoin( Sort( SeqScan( customers ) ), Sort( SeqScan( rentals ) ))
MergeJoin( IndexScan( customers.customer_id ), Sort( SeqScan( rentals ) ))
MergeJoin( Sort( SeqScan( customers ) ), IndexScan( rentals.customer_id ))
NestedLoop( IndexScan( customers.customer_id ), IndexScan( rentals.customer_id ))
NestedLoop( SeqScan( customers ), SeqScan( rentals ))
NestedLoop( IndexScan( customers.customer_id ), SeqScan( rentals ))
NestedLoop( SeqScan( customers ), IndexScan( rentals.customer_id ))
HashJoin( IndexScan( customers.customer_id ), Hash( IndexScan( rentals.customer_id ) ))
HashJoin( SeqScan( customers ), Hash( SeqScan( rentals ) ))
HashJoin( IndexScan( customers.customer_id ), Hash( SeqScan( rentals ) ))
HashJoin( SeqScan( customers ), Hash( IndexScan( rentals.customer_id ) ))
```

Once it's finished, the planner has considered 24 plans to join these two tables. In general, the planner will consider

```
joinOperatorCount x (( pathCount( table1 ) x pathCount( table2 ) ) x 2 )
```

plans to join any two tables, where pathCount(table) is the number of possible paths (SeqScans, Index Scans, and TID Scans) through a given table and joinOperatorCount is always 3 in PostgreSQL (MergeJoin, NestedLoop, and HashJoin).

As you've seen, a two-table join will result in 24 possible plans (assuming that there are two paths through each table). Add a third table and the number of possible plans skyrockets. So how does the planner generate plans for a three-table join? It first generates a set of plans to join two of the three tables into a single result set then generates a set of plans to join the intermediate result set to the remaining table. With three tables (a, b, and c), you find the following combinations (note – I've abbreviated MergeJoin and HashJoin here to better fit the printed page):

Code View: [Scroll](#) / Show All

```
Merge( a, Join( b, c ) ) NestedLoop( a, Join( b, c ) ) Hash( a, Join( b, c ) )
Merge( a, Join( c, b ) ) NestedLoop( a, Join( c, b ) ) Hash( a, Join( c, b ) )
Merge( b, Join( a, c ) ) NestedLoop( b, Join( a, c ) ) Hash( b, Join( a, c ) )
Merge( b, Join( c, a ) ) NestedLoop( b, Join( c, a ) ) Hash( b, Join( c, a ) )
Merge( c, Join( a, b ) ) NestedLoop( c, Join( a, b ) ) Hash( c, Join( a, b ) )
Merge( c, Join( c, b ) ) NestedLoop( c, Join( c, b ) ) Hash( c, Join( c, b ) )
Merge( Join( a, b ), c ) NestedLoop( Join( a, b ), c ) Hash( Join( a, b ), c )
Merge( Join( a, c ), b ) NestedLoop( Join( a, c ), b ) Hash( Join( a, c ), b )
```

```

Merge( Join( b, a ), c ) NestedLoop( Join( b, a ), c ) Hash( Join( b, a ), c )
Merge( Join( b, c ), a ) NestedLoop( Join( b, c ), a ) Hash( Join( b, c ), a )
Merge( Join( c, a ), b ) NestedLoop( Join( c, a ), b ) Hash( Join( c, a ), b )
Merge( Join( c, b ), a ) NestedLoop( Join( c, b ), a ) Hash( Join( c, b ), a )

```

And considering that any of these two-table join results in 24 possible plans, you're suddenly looking at 864 possible plans! If you add a fourth table, the planner considers the plans needed to join three of the four tables into an intermediate result, then joins the fourth table to that.

In practice, the planner won't take the time to generate every possible plan—the planner contains a number of heuristics that avoid generating plans that are known to be more expensive than plans already seen. For example, the planner knows that a complete `Index Scan` is more expensive than a complete `Seq Scan` and it won't generate a plan that includes a complete `Index Scan` unless the ordering of the result set is important.

In fact, when you reach a certain point, the plan generator switches from a near-exhaustive search to an algorithm known as the genetic query optimizer. The genetic optimizer evolves a plan by mutating and recombining possible join plans and then evaluating each generation for its "fitness." As each generation emerges, the genetic optimizer selects those mutations and recombinations that result in lower execution plans. The plan that eventually evolves is not guaranteed to be the best possible plan, but it is typically a "good" plan. By default, PostgreSQL uses the genetic query optimizer when the `FROM` clause of a query refers to 12 or more tables.

The ARC Buffer Manager

It's important to keep a few points in mind when you're trying to tune a PostgreSQL database. First, the shared buffer cache is shared. All of the examples in this chapter were built using a single-session database—if you try to reproduce these experiments, be sure you're the only one using the database or your results will vary widely. Second, the important part of a buffer management scheme isn't the part that determines what goes into the cache, it's the part that determines what gets thrown out of the cache. When PostgreSQL reads data from a table, it first checks to see if the required page is in the cache. If PostgreSQL finds it in the cache, it stops looking. If the required page isn't in the cache, PostgreSQL must read it in from disk. That means that every page in every table is read into the cache as soon as a query (or other command) refers to the page. If your buffer cache is large enough, PostgreSQL will never evict a page from the cache (although it will write modified pages to disk).

When PostgreSQL adds a page to the shared cache and finds that the cache is already full, it must evict some other page. Prior to release 8.0, PostgreSQL would always evict the least-recently-used page. Pretend that you have a very small buffer cache (say three pages). When you execute a command that causes a table scan (a scan of every page from beginning to end), the server starts by reading the first page into the cache. Next, the server processes every tuple on that page (ignoring dead and uncommitted tuples as it goes).

Once it has finished processing the first page, it unpins that page in the cache (meaning that that page is no longer in use and can be evicted if necessary). The server then moves on to the second page. Because there are still two free pages in the cache, PostgreSQL just reads the second page from disk, stores that page in the cache, processes each tuple in that page, and then unpins that page. The server repeats this sequence for the third page. When PostgreSQL comes to the fourth page, it finds that the cache is full and evicts the least-recently-used page (page one) from the cache, replacing it with page four. That leaves you with pages two, three, and four in the cache. When the server reads in page five, it evicts page two (the least-recently-used page) from the cache, leaving you with pages three, four, and five in the cache. That sequence continues until the server has finished reading the entire table—when you're finished, the cache contains the last three pages from the table. If you execute the same command again, the sequence is the same except that PostgreSQL will have to evict one of the last three pages before it can add the first page to the cache. If instead you execute a command that can be satisfied by looking at the last three pages of the table, PostgreSQL will find those pages in the cache and won't read them from disk.

Of course, if another user is running a command at the same time, he's using the same buffer cache and the eviction sequence will be completely different.

As you've seen earlier in this chapter, a table scan can evict all of the pages from an LRU cache.

Starting with version 8.0, PostgreSQL uses a new caching mechanism that constantly adapts itself to a changing workload. The ARC (adaptive replacement cache) scheme effectively uses two caches: One is a traditional LRU cache and the other is a LFU cache. LFU stands for "least-frequently-used" as opposed to "least-recently-used." PostgreSQL divides the shared memory segment into one cache that buffers recently used pages and a second cache that buffers frequently used pages. That means that if your shared buffer cache contains 1,024 pages, some pages will contain recently used pages and some will contain frequently used pages. How many pages does PostgreSQL devote to each cache? It depends on your workload: PostgreSQL adjusts the relative size of each cache as it runs. If the server sees a period of high "locality of reference" (meaning that the current workload is frequently accessing a small set of pages), it devotes more space to the LFU cache (taking pages away from the LRU cache). If the server sees a request for a page that was recently evicted from the LRU cache, it devotes more space to the LRU cache (taking pages away from the LFU cache). To see how ARC affects the shared buffer cache, we'll show you two simple queries—we'll run the queries first in PostgreSQL version 7.4.2, then again in version 8.0.

Code View: [Scroll](#) / [Show All](#)

```
$ timer \
> "SELECT * FROM recalls WHERE record_id > 8000 AND record_id < 8050" recalls
+-----+-----+-----+-----+-----+-----+-----+-----+
|          |          SEQUENTIAL I/O          |          INDEXED I/O          |          |
|          |scans|tuples|heap_blks|cached|scans|tuples|idx_blks|cached|
+-----+-----+-----+-----+-----+-----+-----+-----+
|recalls  |    0 |    0 |        4 |   45 |    1 |    49 |        3 |    0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

To satisfy this query, PostgreSQL uses the index that covers the `record_id` column. Because we just restarted the 7.4 server (and we have enough room for 512 pages in the shared buffer cache), PostgreSQL had to read all three of the index blocks that we hit from disk.

Code View: [Scroll](#) / [Show All](#)

```
$ timer \
> "SELECT * FROM recalls WHERE record_id > 8000 AND record_id < 8050" recalls
+-----+-----+-----+-----+-----+-----+-----+-----+
|          |          SEQUENTIAL I/O          |          INDEXED I/O          |          |
|          |scans|tuples|heap_blks|cached|scans|tuples|idx_blks|cached|
+-----+-----+-----+-----+-----+-----+-----+-----+
|recalls  |    0 |    0 |        0 |   49 |    1 |    49 |        0 |    3 |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

When we execute the same query again, PostgreSQL reads the same three index blocks, but this time, it finds them in the cache. Now we'll execute a query that causes a table scan:

Code View: [Scroll](#) / [Show All](#)

```
$ timer "SELECT * FROM recalls" recalls
+-----+-----+-----+-----+-----+-----+-----+-----+
|          |          SEQUENTIAL I/O          |          INDEXED I/O          |          |
```

	scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
recalls	1	39241	4400	0	0	0	0	0

This query made a complete pass through the table, shuffling all 4,400 heap blocks through a cache that can only hold 512 blocks. When the query completes, the last 512 or so heap blocks that we read are still in the cache. Now go back and execute the first query (the one that causes a partial index scan):

Code View: [Scroll](#) / Show All

```
$ timer \
> "SELECT * FROM recalls WHERE record_id > 8000 AND record_id < 8050" recalls
```

	SEQUENTIAL I/O				INDEXED I/O			
	scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
recalls	0	0	4	45	1	49	3	0

Notice that PostgreSQL had to read the same three index blocks again, but the intervening table scan has evicted them from the cache and they must be read from disk.

Now here is the same sequence running in a version 8.0 server. Again, we'll execute the same query that caused a partial index scan:

Code View: [Scroll](#) / Show All

```
$ timer \
> "SELECT * FROM recalls WHERE record_id > 8000 AND record_id < 8050" recalls
```

	SEQUENTIAL I/O				INDEXED I/O			
	scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
recalls	0	0	5	0	1	49	4	0

And we'll execute it again just to make sure that the index blocks did in fact stay in the cache:

Code View: [Scroll](#) / Show All

```
$ timer \
> "SELECT * FROM recalls WHERE record_id > 8000 AND record_id < 8050" recalls
```

	SEQUENTIAL I/O				INDEXED I/O			
	scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
recalls	0	0	0	5	1	49	0	4

Now we'll execute a query that, in version 7.4.2, threw the index blocks out of the cache:

Code View: [Scroll](#) / Show All

```
$ timer "SELECT * FROM recalls" recalls
```

	SEQUENTIAL I/O				INDEXED I/O			
	scans	tuples	heap_blks	cached	scans	tuples	idx_blks	cached
recalls	1	39241	4400	5	0	0	0	0

And finally, we'll repeat the partial index scan query:

Code View: [Scroll](#) / Show All

```
$ timer \
```

```
> "SELECT * FROM recalls WHERE record_id > 8000 AND record_id < 8050" recalls
+-----+-----+-----+-----+-----+-----+-----+-----+
|          |          SEQUENTIAL I/O          |          INDEXED I/O          |
|          |scans|tuples|heap_blks|cached|scans|tuples|idx_blks|cached|
+-----+-----+-----+-----+-----+-----+-----+-----+
|recalls  |    0 |    0 |        0 |    5 |    1 |    49 |    0 |    4 |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

This time, the 8.0 server has retained the frequently used index blocks in the LFU cache.

Table Statistics

You've seen all the operators that PostgreSQL can use to execute a query. Remember that the goal of the optimizer is to find the plan with the least overall expense. Each operator uses a different algorithm for estimating its cost of execution. The cost estimators need some basic statistical information to make educated estimates.

Table statistics are stored in two places in a PostgreSQL database: `pg_class` and `pg_statistic`.

The `pg_class` system table contains one row for each table defined in your database (it also contains information about views, indexes, and sequences). For any given table, the `pg_class.relpages` column contains an estimate of the number of 8KB pages required to hold the table. The `pg_class.reltuples` column contains an estimate of the number of tuples currently contained in each table.

Note that `pg_class` holds only estimates—when you create a new table, the `relpages` estimate is set to 10 pages and `reltuples` is set to 1,000 tuples. As you `INSERT` and `DELETE` rows, PostgreSQL does not maintain the `pg_class` estimates. You can see this here:

```
movies=# SELECT * FROM tapes;
```

tape_id	title	duration
AB-12345	The Godfather	
AB-67472	The Godfather	
MC-68873	Casablanca	
OW-41221	Citizen Kane	
AH-54706	Rear Window	

(5 rows)

```
movies=# CREATE TABLE tapes2 AS SELECT * FROM tapes;
```

```
SELECT
```

```
movies=# SELECT reltuples, relpages FROM pg_class
```

```
movies=# WHERE relname = 'tapes2';
```

reltuples	relpages
-----------	----------

1000	10
------	----

Create the `tapes2` table by duplicating the `tapes` table. You know that `tapes2` really holds five tuples (and probably requires a single disk page), but PostgreSQL has not updated the initial default estimate.

There are three commands that you can use to update the `pg_class` estimates: `VACUUM`, `ANALYZE`, and `CREATE INDEX`.

The `VACUUM` command removes any dead tuples from a table and recomputes the `pg_class` statistical information:

```
movies=# VACUUM tapes2;
```

```
VACUUM
```

```
movies=# SELECT reltuples, relpages FROM pg_class WHERE relname = 'tapes2';
```

reltuples	relpages
-----------	----------

5	1
---	---

(1 row)

The `pg_statistic` system table holds detailed information about the data in a table. Like `pg_class`, `pg_statistic` is not automatically maintained when you `INSERT` and `DELETE` data. The `pg_statistic` table is not updated by the `VACUUM` or `CREATE INDEX` command, but it is updated by the `ANALYZE` command:

```
movies=# SELECT staattnum, stawidth, stanullfrac FROM pg_statistic
```

```
movies=# WHERE starelid =
```

```
movies=# (
```

```
movies=# SELECT oid FROM pg_class WHERE relname = 'tapes2'
```

```
movies=# );
```

staattnum	stawidth	stanullfrac
-----------	----------	-------------

--	--	--

(0 rows)

```
movies=# ANALYZE tapes2;
```

```
ANALYZE
```

```
movies=# SELECT staattnum, stawidth, stanullfrac FROM pg_statistic
```

```
movies=# WHERE starelid =
```

```
movies=# (
```

```
movies=# SELECT oid FROM pg_class WHERE relname = 'tapes2'
```

```
movies=# );
```

staattnum	stawidth	stanullfrac
-----------	----------	-------------

1	12	0
2	15	0
3	4	0

(3 rows)

PostgreSQL defines a view (called `pg_stats`) that makes the `pg_statistic` table a little easier to deal with. Here is what the `pg_stats` view tells us about the `tapes2` table:

```
movies=# SELECT attname, null_frac, avg_width, n_distinct FROM pg_stats
```

```
movies=# WHERE tablename = 'tapes2';
```

```

attnname | null_frac | avg_width | n_distinct
-----+-----+-----+-----
tape_id |          0 |        12 |         -1
title   |          0 |        15 |        -0.8
(2 rows)

```

You can see that `pg_stats` (and the underlying `pg_statistics` table) contains one row for each column in the `tapes2` table (except for the `duration` column where every value happens to be `NULL`). The `null_frac` value tells you the percentage of rows where a given column contains `NULL`. In this case, there are no `NULL` values in the `tapes2` table, so `null_frac` is set to 0 for each column. `avg_width` contains the average width (in bytes) of the values in a given column. The `n_distinct` value tells you how many distinct values are present for a given column. If `n_distinct` is positive, it indicates the actual number of distinct values. If `n_distinct` is negative, it indicates the percentage of rows that contain a distinct value. A value of -1 tells you that every row in the table contains a unique value for that column.

`pg_stats` also contains information about the actual values in a table:

```

movies=# SELECT attnname, most_common_vals, most_common_freqs
movies=# FROM pg_stats
movies=# WHERE tablename = 'tapes2';
 attnname | most_common_vals | most_common_freqs
-----+-----+-----
tape_id | |
title | {"The Godfather"} | {0.4}
(2 rows)

```

The `most_common_vals` column is an array containing the most common values in a given column. The `most_common_freqs` value tells you how often each of the most common values appear. By default, `ANALYZE` stores the 10 most common values (and the frequency of those 10 values). You can increase or decrease the number of common values using the `ALTER TABLE ... SET STATISTICS` command.

Looking back at the `recalls` table, you can see that the `datea` column contains 825 distinct values:

Code View: [Scroll](#) / [Show All](#)

```

perf=# \x
Expanded display is on.
perf=# SELECT
perf=#   n_distinct, most_common_vals, most_common_freqs
perf=# FROM
perf=#   pg_stats
perf=# WHERE
perf=#   tablename = 'recalls' AND attnname = 'datea';
-[ RECORD 1 ]-----+-----
n_distinct          | 825
most_common_vals    | {19791012,19921230,20001129,19980814,19950524,19950901,...}
most_common_freqs   | {0.319667,0.005,0.005,0.00466667,0.00433333,0.00433333,...}

```

(We've turned on `psql`'s expanded display (with the `\x` command) and trimmed the results a bit to make them easier to read.) The most commonly found `datea` value is 19791012 and it occurs in approximately 32% of all rows. The second most common value is 19921230 and that value is found in .5% of all rows. Go ahead and create an index that covers the `datea` column and then `ANALYZE` the `recalls` table:

```

perf=# CREATE INDEX recalls_by_datea ON recalls( datea );
CREATE INDEX
perf=# ANALYZE recalls;
ANALYZE

```

You might expect PostgreSQL to use this index in a query that selects rows based on `datea` values, and sometimes it does:

```

perf=# EXPLAIN SELECT * FROM recalls WHERE datea = '19921230';
               QUERY PLAN
-----
Index Scan using recalls_by_datea (cost=0.00..31.44 rows=31 width=1908)
  Index Cond: (datea = '19921230'::bpchar)

```

You can see that the optimizer chose a partial index scan (using the `recalls_by_datea` index) to satisfy this query. Now try to select a different set of rows:

```

perf=# EXPLAIN SELECT * FROM recalls WHERE datea = '19791012';
               QUERY PLAN
-----
Seq Scan on recalls (cost=0.00..9015.09 rows=10690 width=1908)
  Filter: (datea = '19791012'::bpchar)

```

In this case, the optimizer thinks it would be faster to perform a complete table scan on the `recalls` table, ignoring the rows that fail to satisfy the `WHERE` clause.

The only thing that's changed between the two queries is the value that you're searching for. The `recalls` table hasn't changed; the statistics haven't changed. Why would PostgreSQL use an index to retrieve the second most-frequently-found value, but not the most-frequently-found

value? Because the optimizer knows (based on the `pg_stats`.`most_common_vals` and `pg_stats`.`most_common_freqs`) that it must process 32% of the `recalls` table in the second case and reading a table via an index is more expensive than reading it via a sequential scan. To retrieve the second most-frequently-found value, the optimizer knows that it will only read .5% of the table via the index.

Another statistic exposed by `pg_stat` is called `histogram_bounds`. The `histogram_bounds` column contains an array of values for each column in your table. These values are used to partition your data into approximately equally sized chunks. For example, here are the `histogram_bounds` values for the `recalls.potaff` column:

```
perf=# SELECT histogram_bounds FROM pg_stats
perf=# WHERE tablename = 'recalls' and attname = 'potaff';
          histogram_bounds
-----
{3,104,305,700,1503,3203,6503,15263,48003,210003,32000003}
```

Because there are 11 values shown, the `histogram_bounds` show that 10% of the `potaff` values fall between 3 and 104, 10% of the `potaff` values fall between 104 and 305, 10% fall between 305 and 700, and so on. The optimizer uses the `histograms_bounds` to decide how much of an index it will need to traverse in order to search for a specific value. For example, if you search for a `potaff` value of 32000004 (which fits into the last histogram "bucket"), PostgreSQL knows that it will only have to traverse the last 10% of the `recalls_by_potaff` index to satisfy the query. On the other hand, if you search for the value 210003, PostgreSQL must traverse the last 20% of the index. Here's how the optimizer handles a search over the last 10% of the index:

Code View: [Scroll](#) / Show All

```
perf=# EXPLAIN SELECT * FROM recalls WHERE potaff >= 14500003;
          QUERY PLAN
-----
Index Scan using recalls_by_potaff (cost=0.00..17.99 rows=4 width=1908)
  Index Cond: (potaff >= 14500003::numeric)
```

The optimizer has chosen an index scan to satisfy this query because traversing 10% of the index is less expensive than a complete table scan. Now consider a similar query, but this time, you're searching for a value known to fall within the last 20% of the index:

```
perf=# EXPLAIN SELECT * FROM recalls WHERE potaff >= 210003;
          QUERY PLAN
-----
Seq Scan on recalls (cost=0.00..9015.09 rows=3928 width=1908)
  Filter: (potaff >= 210003::numeric)
```

This time, the optimizer has chosen a table scan. The structure of this query is identical the previous query—only the search value has changed.

The last statistic stored in `pg_stats` is an indication of whether the rows in a table are stored in column order:

```
movies=# SELECT attname, correlation FROM pg_stats
movies=# WHERE tablename = 'tapes2';
   attname | correlation
-----+-----
   tape_id |          0.7
   title   |         -0.5
(2 rows)
```

A correlation of 1 means that the rows are sorted by the given column. In practice, you will see a correlation of 1 only for brand new tables (whose rows happened to be sorted before insertion) or tables that you have reordered using the `CLUSTER` command.

Performance Tips

That wraps up the discussion of performance in PostgreSQL. Here are few tips that you should keep in mind whenever you run into an apparent performance problem:

- `VACUUM` and `ANALYZE` your database after any large change in data values. This will give the query optimizer a better idea of how your data is distributed.
- Use the `CREATE TABLE AS` or `CLUSTER` commands to cluster rows with similar key values. This makes an index traversal much faster.
- If you think you have a performance problem, use the `EXPLAIN` command to find out how PostgreSQL has decided to execute your query.
- You can influence the optimizer by disabling certain query operators. For example, if you want to ensure that a query is executed as a sequential scan, you can disable the Index Scan operator by executing the following command: `"SET ENABLE_INDEX_SCAN TO OFF;"`. Disabling an operator does not guarantee that the optimizer won't use that operator—it just considers the operator to be much more expensive. The PostgreSQL User Manual contains a complete list of runtime parameters.
- You can also influence the optimizer by adjusting the relative costs for certain query operations. See the descriptions for `CPU_INDEX_TUPLE_COST`, `CPU_OPERATOR_COST`, `CPU_TUPLE_COST`, `EFFECTIVE_CACHE_SIZE`, and `RANDOM_PAGE_COST` in the PostgreSQL User Manual.
- Minimize network traffic by doing as much work as possible in the server. You will usually get better performance if you can filter data on the server rather than in the client application.
- One source of extra network traffic that might not be so obvious is metadata. If your client application retrieves 10 rows using a single `SELECT`, one set of metadata is sent to the client. On the other hand, if you create a cursor to retrieve the same set of rows, but execute 10 `FETCH` commands to grab the data, you'll also get 10 (identical) sets of metadata.
- Use server-side procedures (triggers and functions) to perform common operations. A server-side procedure is parsed, planned, and optimized the first time you use it, not every time you use it.

Part II : Programming with PostgreSQL

- 5 Introduction to PostgreSQL Programming
- 6 Extending PostgreSQL
- 7 PL/pgSQL
- 8 The PostgreSQL C API—libpq
- 9 A Simpler C API—libpqeasy
- 10 The New PostgreSQL C++ API—libpqxx
- 11 Embedding SQL Commands in C Programs—ecpg
- 12 Using PostgreSQL from an ODBC Client Application
- 13 Using PostgreSQL from a Java Client Application
- 14 Using PostgreSQL with Perl
- 15 Using PostgreSQL with PHP
- 16 Using PostgreSQL with Tcl and Tk
- 17 Using PostgreSQL with Python
- 18 npgsql: The .NET Data Provider
- 19 Other Useful Programming Tools

Chapter 5. Introduction to PostgreSQL Programming

PostgreSQL is a client/server database. When you use PostgreSQL, there are at least two processes involved—the client and the server. In a client/server environment, the server provides a service to one or more clients. The PostgreSQL server provides data storage and retrieval services. A PostgreSQL client is an application that receives data storage and retrieval services from a PostgreSQL server. Quite often, the client and the server exist on different physical machines connected by a network. The client and server can also exist on a single host. As you will see, the client and the server do not have to be written in the same computer language. The PostgreSQL server is written in C; many client applications are written in other languages.

In this chapter, I'll introduce you to some of the concepts behind client/server programming for PostgreSQL. I'll also show you options you have for server-side programming languages and for client-side programming interfaces. I also discuss the basic structure of a PostgreSQL client application, regardless of which client-side language you choose. Finally, I explore the advantages and disadvantages of server-side versus client-side code.

Server-Side Programming

The task of programming for PostgreSQL falls into two broad categories: server-side programming and client-side programming.

Server-side code (as the name implies) is code that executes within a PostgreSQL server. Server-side code executes the same way regardless of which language was used to implement any given client. If the client and server are running on different physical hosts, all server-side code executes on the server machine and within the server process. If the client and server are running on the same machine, server-side code still runs within the server process. In most cases, server-side code is written in one of the procedural languages distributed with PostgreSQL.

PostgreSQL version 7.1 ships with three procedural languages: PL/pgSQL, PL/Tcl, and PL/Perl. Release 7.2 adds PL/Python to the mix. You can also write server-side procedures in SQL. Later versions of PostgreSQL add support for PL/Java. You can even write server-side procedures in the form of `bash` shell-scripts using PL/`bash`.

You can use procedural languages to create functions that execute within the server. A function is a named sequence of statements that you can use within an SQL expression. When you write a function in a server-side language, you are extending the server. These server extensions are also known as stored procedures.

PL/ pgSQL

If you have ever used a commercial database system—Oracle, Sybase, or SQL Server, for example—you have probably used a SQL-based procedural language. Oracle's procedural language is called PL/SQL; Sybase and SQL Server use TransactSQL. PL/pgSQL is very similar to these procedural languages.

PL/pgSQL combines the declarative nature of SQL commands with structures offered by other languages. When you create a PL/pgSQL function, you can declare local variables to store intermediate results. PL/pgSQL offers a variety of loop constructs (FOR loops, WHILE loops, and cursor iteration loops). PL/pgSQL gives you the capability to conditionally execute sections of code based on the results of a test. You can pass parameters to a PL/pgSQL function, making the function reusable. You can also invoke other functions from within a PL/pgSQL function.

Chapter 7, "PL/pgSQL," provides an in-depth description of PL/pgSQL.

Other Procedural Languages Supported by PostgreSQL

One of the more unusual aspects of PostgreSQL (compared to other database systems) is that you can write procedural code in more than one language. As noted previously, the standard distribution of PostgreSQL includes PL/pgSQL, PL/Perl, PL/Tcl, and, as of release 7.2,

PL/Python.

The latter three languages each enable you to create stored procedures using a subset of the host language. PostgreSQL restricts each to a subset of the language to ensure that a stored procedure can't do nasty things to your environment.

Specifically, the PostgreSQL procedural languages are not allowed to perform I/O external to the database (in other words, you can't use a PostgreSQL procedural language to do anything outside of the context of the server). If you find that you need to affect your external environment, you can load an untrusted procedural language, but be aware that you will be introducing a security risk when you do so.

When you install PostgreSQL from a standard distribution, none of the server-side languages are installed. You can pick and choose which languages you want to install in the server. If you don't use a given language, you can choose not to install it. I'll show you how to install server-side languages in [Chapter 7](#).

You can see which languages are currently installed in your database server with the following query:

Code View: [Scroll](#) / [Show All](#)

```
movies=# select * from pg_language;
lanname | lanispl | lanpltrusted | lanplcallfoid | lanvalidator | lanacl
-----+-----+-----+-----+-----+-----
internal | f       | f           | 0             | 2246         | 
c       | f       | f           | 0             | 2247         | 
sql     | f       | t           | 0             | 2248         | {=U/pg}
(3 rows)
```

You can see that my server currently supports three languages: internal, C, and sql. The lanispl column tells us that none of these are considered to be procedural languages. You may be thinking that C should be considered a procedural language, but in this context a procedural language is one that can be installed and de-installed from the server. You can determine whether a language is trusted by examining the lanpltrusted column. A trusted language promises not to provide elevated privileges to a user. If a language is not a trusted language, only PostgreSQL superusers can create a new function in that language.

Extending PostgreSQL Using External Languages

PostgreSQL-hosted procedural languages are not the only tools available for extending the server. You can also add extensions to a PostgreSQL server by creating custom data types, new functions, and new operators written in an external language (usually C or C+).

When you create procedural-language extensions, the source code (and the object code, if any) for those functions is stored in tables within the database. When you create a function using an external language, the function is not stored in the database. Instead, it is stored in a shared library that is linked into the server when first used.

You can find many PostgreSQL extensions on the Web. For example, the PostGIS project adds a set of data types and supporting functions for dealing with geographic data. The contrib directory of a PostgreSQL distribution contains an extension for dealing with ISBNs and ISSNs.

In [Chapter 6](#), "Extending PostgreSQL," I'll show you a few simple examples of how to add custom data types and functions written in C.

Client-Side APIs

When you want to build applications that access a PostgreSQL database, you use one (or more) of the client application programming interfaces (or APIs for short). PostgreSQL has a rich variety of APIs that support a number of programming languages.

PostgreSQL supports the APIs shown in [Table 5.1](#).

Table 5.1. PostgreSQL Client APIs

Interface Name	Supported Languages	Described In
libpq	C/C++	Chapter 8
libpqeas	C/C++	Chapter 9
libpq++	C++	Chapter 10
ecpg	C/C++	Chapter 11
ODBC	C/C++	Chapter 12
JDBC	Java	Chapter 13
Perl	Perl	Chapter 14
PHP ^[1]	PHP	Chapter 15
pgtcl	TCL	Chapter 16
PyGreSQL	Python	Chapter 17
pg.el	Emacs Lisp	Not covered

^[1] The standard PostgreSQL distribution does not include the PHP or Emacs interfaces, but they are available separately on the Web.

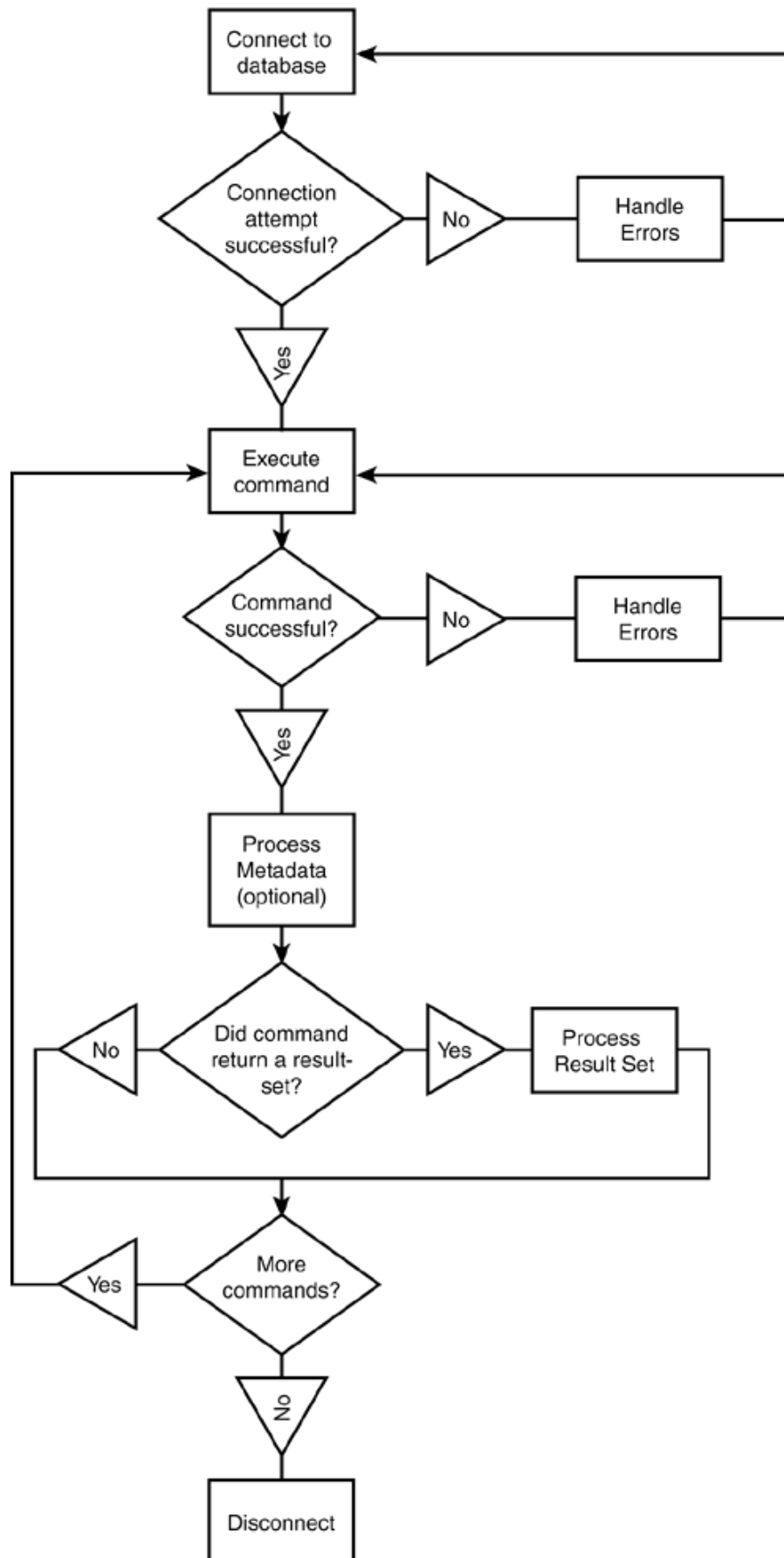
[Table 5.1](#) is not all-inclusive. You can write PostgreSQL clients using languages not mentioned in [Table 5.1](#). For example, Kylix (Borland's Pascal offering for Linux) offers a PostgreSQL interface. Also, many other languages (such as Microsoft Access and Visual Basic) provide access to PostgreSQL through the ODBC interface. In recent versions, the PostgreSQL development team has removed some interfaces from the core PostgreSQL distribution. If the language that you want to use is not directly supported in the core distribution, surf the gborg.postgresql.org website to find the interface you need.

General Structure of Client Applications

This is a good time to discuss, in general terms, how a client application interacts with a PostgreSQL database. All the client APIs have a common structure, but the details vary greatly from language to language.

Figure 5.1 illustrates the basic flow of a client's interaction with a server.

Figure 5.1. Client/ server interaction.



An application begins interacting with a PostgreSQL database by establishing a connection.

Because PostgreSQL is a client/server database, some sort of connection must exist between a client application and a database server. In the case of PostgreSQL, client/server communication takes the form of a network link. If the client and server are on different systems, the network link is a TCP/IP socket. If the client and server are on the same system, the network link is either a Unix-domain socket or a TCP/IP connection. A Unix-domain socket is a link that exists entirely within a single host—the network is

a logical network (rather than a physical network) within the OS kernel.

Connection Properties

Regardless of whether you are connecting to a local server or a remote server, the API uses a set of properties to establish the connection. Connection properties are used to identify the server (a network port number and host address), the specific database that you want to connect to, your user ID (and password if required), and various debugging and logging options. Each API allows you to explicitly specify connection properties, but you can also use default values for some (or all) of the properties. Most of the client-side APIs let you specify connection properties in the form of a string of `keyword=value` pairs. For example, to connect to a database named `accounting` on a host named `jersey`, you would use a property string such as

```
"dbname=accounting host=jersey"
```

Each `keyword=value` defines a single connection property. If you omit a connection property, PostgreSQL checks for an environment variable that corresponds to the property and, if the environment variable doesn't exist, PostgreSQL uses a hard-coded default value. See [Table 5.2](#).

Table 5.2. Keywords and Environment Variables

Keyword	Environment Variable	Description
<code>dbname</code>	<code>PGDATABASE</code>	Specifies the name of the database that you want to connect to. If not specified, the client application tries to connect to a data-base with the same name as your username.
<code>user</code>	<code>PGUSER</code>	Specifies the PostgreSQL username you want to connect as. If not specified, the client uses your operating system identity.
<code>host</code>	<code>PGHOST</code>	Specifies the name (or IP address) of the computer that hosts the database you want to connect to. If the value starts with a '/', the client assumes that you want to connect to a Unix-domain socket located in that directory. If not specified, the client connects to a Unix-domain socket in <code>/tmp</code> .
<code>hostaddr</code>	<code>PGHOSTADDR</code>	Specifies the IP address of the computer that hosts the database you want to connect to. When you specify <code>hostaddr</code> (instead of <code>host</code>), you avoid a name lookup (which can be slow on some networks). If not specified, the client uses the value of the <code>host</code> property to find the server.
<code>port</code>	<code>PGPORT</code>	Specifies the TCP/IP port number to connect to (or, if you're connecting to a Unix-domain socket, the socket filename extension). If not specified, the default <code>PGPORT</code> is 5432.
<code>connect_timeout</code>	<code>PGCONNECT_TIMEOUT</code>	Specifies the maximum amount of time (in seconds) to wait for the connection process to complete. If not specified (or if you specify a value of 0), the client will wait forever.
<code>sslmode</code>	<code>PGSSLMODE</code>	Specifies whether the client will attempt (or accept) an SSL-secured connection. Possible values are <code>disable</code> , <code>allow</code> , <code>prefer</code> , and <code>require</code> . <code>disable</code> and <code>require</code> are obvious, but <code>allow</code> and <code>prefer</code> seem a bit mysterious. If <code>sslmode</code> is <code>allow</code> , the client first attempts an insecure connection, but allows an SSL connection if an insecure connection can't be built. If <code>sslmode</code> is <code>preferred</code> , the client first attempts a secure connection, but accepts an insecure connection if a secure connection can't be built. If not specified, the default <code>sslmode</code> is <code>prefer</code> .
<code>service</code>	<code>PGSERVICE</code>	Specifies the name of a service as defined in the <code>pg_service.conf</code> file (see next section).

A more convenient way to encode connection parameters is to use the `pg_service.conf` file. When you specify a service name (with the `PGSERVICE` environment variable or the `service=service-name` connection property), the client application (actually the `libpq` library) opens a file named `$PREFIX/etc/pg_service.conf` and searches for a section that matches the `service-name` that you provided. If `libpq` locates the section that you named, it reads connection properties from that section. A typical `pg_service.conf` file might look similar to the following:

```
[accounting]
dbname=accounting
host=jersey
```

```
sslmode=required

[development]
dbname=accounting
host=guernsey
sslmode=prefer
```

Each service begins with the service name (enclosed in square brackets) and continues until the next section (or the end of the file). A service is simply a collection of connection properties in the usual `keyword=value` format. The sample above defines two services (one named `accounting` and the other named `development`).

The nice thing about using a service name is that you can consolidate all your connection properties in a single location and then give only the service name to your database users. When you connect to a database using a service name, the client application loads the service definition first then processes the connection string. That means that you can specify both a service name and a connection string (properties found in the connection string will override the properties specified in the service). Environment variables are only consulted in a last-ditch effort to find missing values.

After a server connection has been established, the API gives you a handle. A handle is nothing more than a chunk of data that you get from the API and that you give back to the API when you want to send or receive data over the connection. The exact form of a handle varies depending on the language that you are using (or more precisely, the data type of a handle varies with the API that you use). For example, in `libpq` (the C API), a handle is a void pointer—you can't do anything with a void pointer except to give it back to the API. In the case of `libpq++` and `JDBC`, a handle is embedded within a class.

After you obtain a connection handle from the API, you can use that handle to interact with the database. Typically, a client will want to execute SQL queries and process results. Each API provides a set of functions that will send a SQL command to the database. In the simplest case, you use a single function; more complex applications (and APIs) can separate command execution into two phases. The first phase sends the command to the server (for error checking and query planning) and the second phase actually carries out the command; you can repeat the execution phase as many times as you like. The advantage to a two-phase execution method is performance. You can parse and plan a command once and execute it many times, rather than parsing and planning every time you execute the command. Two-phase execution can also simplify your code by factoring the work required to generate a command into a separate function: One function can generate a command and a separate function can execute the command.

After you use an API to send a command to the server, you get back three types of results. The first result that comes back from the server is an indication of success or failure—every command that you send to the server will either fail or succeed. If your command fails, you can use the API to retrieve an error code and a translation of that code into some form of textual message.

If the server tells you that the command executed successfully, you can retrieve the next type of result: metadata. Metadata is data about data. Specifically, metadata is information about the results of the command that you just executed. If you already know the format of the result set, you can ignore the metadata.

When you execute a command such as `INSERT`, `UPDATE`, or `DELETE`, the metadata returned by the server is simply a count of the number of rows affected by the command. Some commands return no metadata. For example, when you execute a `CREATE TABLE` command, the only results that you get from the server are success or failure (and an error code if the command fails). When you execute a `SELECT` command, the metadata is more complex. Remember that a `SELECT` statement can return a set of zero or more rows, each containing one or more columns. This is called the result set. The metadata for a `SELECT` statement describes each of the columns in the result set.

Field Versus Column in Result Sets

When discussing a result set, the PostgreSQL documentation makes a distinction between a field and a column. A column comes directly from a table (or a view). A field is the result of a computation in the `SELECT` statement. For example, if you execute the command `SELECT customer_name, customer_balance * 1.05 FROM customers`, `customer_name` is a column in the result set and `customer_balance * 1.05` is a field in the result set. The difference between a field and a column is mostly irrelevant and can be ignored; just be aware that the documentation uses two different words for the same meaning.

When the server sends result set metadata, it returns the number of rows in the result set and the number of fields. For each field in the result set, the metadata includes the field name, data type information, and the size of the field (on the server).

I should mention here that most client applications don't really need to deal with all the metadata returned by the server. In general, when you write an application you already know the structure of your data. You'll often need to know how many rows were returned by a given query, but the other metadata is most useful when you are processing ad-hoc commands—commands that are not known to you at the time you are writing your application.

After you process the metadata (if you need to), your application will usually process all the rows in the result set. If you execute a `SELECT` statement, the result set will include all the rows that meet the constraints of the `WHERE` clause (if any). In some circumstances, you will find it more convenient to `DECLARE` a cursor for the `SELECT` statement and then execute multiple `FETCH` statements. When you execute the `DECLARE` statement, you won't get metadata. However, as you execute `FETCH` commands, you are constructing a new result set for each `FETCH` and the server has to send metadata describing the resulting fields—that can be expensive.

After you have finished processing the result set, you can execute more commands, or you can disconnect from the server.

LISTEN/ NOTIFY

Sometimes, you might want a client application to wait for some server-side event to occur before proceeding. For example, you might need a queuing system that writes a work order into a PostgreSQL table and then expects a client application to carry out that work order. The most obvious way to write a client of this sort is to put your client application to sleep for a few seconds (or a few minutes), then, when your application awakens, check for a new record in the work-order table. If the record exists, do your work and then repeat the whole cycle.

There are two problems with this approach. First, your client application can't be very responsive. When a new work order is added, it may take a few seconds (or a few minutes) for your client to notice (it's fast asleep after all). Second, your client application might spend a lot of time searching for work orders that don't exist.

PostgreSQL offers a solution to this problem: the `LISTEN/NOTIFY` mechanism. A PostgreSQL server can signal client applications that some event has occurred by executing a `NOTIFY eventName` command. All client applications that are listening for that event are notified that the event has occurred. You get to choose your own event names. In a work-order application, you might define an event named `workOrderReceived`. To inform the server that you are interested in that event, the client application executes a `LISTEN workOrderReceived` command (to tell the server that you are no longer interested in an event, simply `UNLISTEN workOrderReceived`). When a work order arrives at the server (via some other client application), executing the command `NOTIFY workOrderReceived` will inform all clients that a `workOrderReceived` event has occurred (actually, PostgreSQL will only notify those clients listening for that specific event).

Each client-side API offers a different `LISTEN` mechanism and you rarely execute a `LISTEN` command yourself—instead, you call an API function that executes the `LISTEN` command for you (after arranging to intercept the event in a language-specific way).

Regardless of the language that you choose, you should be aware that notifications are only sent at the end of a successful transaction. If you `ROLLBACK` a transaction, any `NOTIFY` commands executed within that transaction are ignored. That makes sense if you think about it: If your application adds a work order record, but then aborts the transaction, you don't want to wake client applications with a false alarm.

Choosing an Application Environment

When you choose an environment for your code, there are a number of issues to consider. To start with, you have to decide whether the feature that you want to build should be server-side code, client-side code, or a combination of both.

Server-Side Code

There are several advantages to adding functionality as server-side code.

The first consideration is performance. If you are creating an application that needs to access many rows of data, it will execute faster on the server. You won't have to send the data across the network to the client (network traffic is very expensive in terms of performance).

Next, you should consider code reuse. If you add a feature in the form of a server-side function, that feature can be used by any client application. You can also use server-side functions within SQL queries.

Another advantage to creating server-side functions is that you can use a server function as a trigger. A trigger function is executed whenever a particular condition occurs. For example, you can define a trigger that executes whenever a row is deleted from a particular table.

Finally, server-side code is portable. Any function that you write in a server-side procedural language runs on any platform that supports PostgreSQL. Of course, if you write a server-side function that requires specific server-side features (such as other functions or data types), those features must be installed in each server.

Client-Side Code

Client-side code is useful for building the user interface. You can't build a user interface using one of the server-side procedural languages—they execute within the context of the server and the server has no user interface.

One of the interesting things to note about the client APIs is that most of them are implemented using the libpq API (ODBC and JDBC are not). This means, for example, that if you are using libpq++ from a C++ application and you call a member function of the PgDatabase class, it will be translated into one or more calls to the libpq library.

The ODBC and JDBC interfaces are not implemented using libpq. Instead, they talk directly to the backend database using the same network protocol as libpq. If you ever decide to implement your own client API, you can choose either method: implement your API in terms of libpq (or one of the other APIs), or talk directly to the server using the same underlying network protocol.

Mixing Server-Side and Client-Side Code

A particularly powerful strategy is to create an application using a mixture of client-side code and stored-procedures. Many commercial applications are shipped with two types of code. When you use one of these packages, you install a set of stored-procedures into the database; then you install external client applications that make use of the custom procedures.

This arrangement gives you all the advantages of server-side code (performance, portability, and reusability) plus the capability to create a pleasant user interface in the client.

Chapter 6. Extending PostgreSQL

PostgreSQL is an extensible database. You can add new functions, new operators, and custom data types to the PostgreSQL server.

In this chapter, I'll show you how to add two simple functions, a new data type, and a set of operators that work with the new type. The examples build on each other, so it would be a good idea to read this chapter in sequence rather than skipping around too much. The sample code used in this chapter was developed using PostgreSQL release 8.0.

We'll start by adding a new function to the PostgreSQL server. The details are important, but the process is not difficult. After you know how to add one function to the server, it's easy to add others.

Extending the PostgreSQL Server with Custom Functions

An extension function is loaded into a running PostgreSQL server process as needed. If you don't actually use an extension, it will not be loaded. Extension functions must be created in the form of a dynamically loadable object module. In the Windows world, an extension is contained within a DLL. In the Linux/Unix environment, an extension is contained within a shared object module.

There are two phases to the process of adding an extension function to the PostgreSQL server. First, you create the extension function in the language of your choice, compiling it into a dynamic object module (`.dll` or `.so`). Next, tell the PostgreSQL server about the function. The `CREATE FUNCTION` command adds a new function to a database.

I'll show you two examples that should help clarify this process.

PostgreSQL and Portability

Some of the steps required to write a PostgreSQL extension function in C may seem rather odd at first. You may feel more comfortable with the process if you understand the problem that the PostgreSQL authors were trying to fix.

When you call a function in a typical C program, you know at the time you write your code how to call that function. You know how many arguments are required and you know the data type of each argument. If you provide an incorrect number of parameters or incorrect data types, it is highly likely that your program will crash. For example, the `fopen()` function (from the C Runtime Library) requires two parameters:

```
FILE * fopen( const char * filename, const char * mode )
```

If you omit the `mode` parameter or send a numeric data type instead of a pointer, your program will fail in some way.

Now, suppose that your program prompts the user for the name of a dynamic object module and the name of a function within that module. After you load the given module into your program, you have to call the named function. If you know which function the user will select, you can formulate your function call properly at the time you write your code. What happens if the user selects some other function that takes a completely different argument list? How can you formulate the function call if you don't know the parameter list? There is no portable way to do that, and PostgreSQL aims to be extremely portable.

So, the PostgreSQL authors decided to change the way you pass arguments to an extension function. Rather than declaring a separate formal parameter for each value passed to the function, PostgreSQL marshals all the arguments into a separate data structure and passes the address of the marshaled form to your extension. When you need to access function parameters, you get to them through the marshaled form.

This is similar in concept to the way the `main()` function of a C program behaves. You can't know, at the time you write the `main()` function, how many command-line parameters you will receive. (You might know how many parameters you should receive, but how many you will receive is not quite the same animal.) The startup routine on the C Runtime Library marshals the command-line arguments into a data structure (the `argv[]` array) and passes you the address of that structure. To find the actual values specified on the command line, you must use the data structure rather than formal parameters.

Older versions of PostgreSQL used a strategy that became less portable as operating systems advanced into the 64-bit arena. The old strategy is known as the "version-0 calling convention." The new strategy is called the "version-1 calling convention." PostgreSQL still supports both calling conventions, but you should stick to the version-1 convention for better portability.

For more information on the difference between the version-0 and version-1 conventions, see section 12 of the PostgreSQL Programmer's Guide.

There are two important consequences to the version-1 convention. First, all version-1 functions return the

same data type: a `Datum`. A `Datum` is a sort of universal data type. Any PostgreSQL data type can be accessed through a `Datum`. PostgreSQL provides a set of macros that make it easy to work with `Datums`. Second, a version-1 function makes use of a set of macros to access function arguments. Every version-1 function is declared in the same way:

```
Datum function-name(PG_FUNCTION_ARGS);
```

As you read through the examples in this chapter, keep in mind that the PostgreSQL authors had to solve the portability problem.

The first example adds a simple function, named `filesize`, to the PostgreSQL server. Given the name of a file, it returns the size of the file (in bytes). If the file does not exist, cannot be examined, or is not a regular^[1] file, this function returns `NULL`. You might find this function (and the `filelist()` function shown later) useful for performing system administration tasks from within a PostgreSQL application. After you have created the `filesize` function, you can call it like this:

^[1] In this context, a file is considered "regular" if it is not a directory, named pipe, symbolic link, device file, or socket.

```
movies=# SELECT filesize( '/bin/bash' );
 filesize
-----
    512668
```

We'll develop the `filesize` function in C (see [Listing 6.1](#)).

The `filesize` function takes a single argument—a pathname in the form of a `TEXT` value. This function returns the size of the named file as an `INTEGER` value.

Listing 6.1. `filesize.c`

Code View: [Scroll](#) / Show All

```
1 /*
2 ** Filename: filesize.c
3 */
4
5 #include "postgres.h"
6 #include "fmgr.h"
7 #include <sys/stat.h>
8
9 PG_FUNCTION_INFO_V1(filesize);
10
11 Datum filesize(PG_FUNCTION_ARGS)
12 {
13     text * fileNameText = PG_GETARG_TEXT_P(0);
14     size_t fileNameLen = VARSIZE( fileNameText ) - VARHDRSZ;
15     char * fileName = (char *)palloc( fileNameLen + 1 );
16     struct stat statBuf;
17
18     memcpy( fileName, VARDATA( fileNameText ), fileNameLen );
19     fileName[fileNameLen] = '\0';
20
21     if( stat(fileName, &statBuf) == 0 && S_ISREG(statBuf.st_mode) )
22     {
23         pfree( fileName );
24
25         PG_RETURN_INT32((int32)statBuf.st_size );
26     }
27     else
28     {
29         pfree( fileName );
30
31         PG_RETURN_NULL();
32     }
33 }
```

Lines 5 and 6 `#include` two header files supplied by PostgreSQL. These files (`postgres.h` and `fmgr.h`) provide data type

definitions, function prototypes, and macros that you can use when writing extensions. The `<sys/stat.h>` file included at line 7 defines the layout of the `struct stat` object used by the `stat()` function (described later).

Line 9 uses the `PG_FUNCTION_INFO_V1()` to tell PostgreSQL that the function (`filesize()`) uses the version-1 calling convention.

At line 11, you see the signature used for all version-1 functions. The `filesize()` function returns a `Datum` and expects a single argument. `PG_FUNCTION_ARGS` is a preprocessor symbol that expands to declare a consistently named parameter. So, your function definition expands from this:

```
Datum filesize(PG_FUNCTION_ARGS)
```

to this:

```
Datum filesize( FunctionCallInfo fcinfo )
```

This might seem a little strange at first, but the version-1 argument accessor macros are written so that the single function argument must be named `fcinfo`.

At line 13, you create a variable of type `text`. `text` is one of the data types defined in the `postgres.h` header file (or in a file included by `postgres.h`). Whenever you write an extension function, you will be working with two sets of data types. Each function parameter (and the return value) will have a SQL data type and a C data type. For example, when you call the `filesize` function from within PostgreSQL, you pass a `TEXT` parameter: `TEXT` is the SQL data type. When you implement the `filesize` function in C, you receive a `text` value: `text` is the C data type. The name for the C data type is usually similar to the name of the corresponding SQL data type. For clarity, I'll refer to the PostgreSQL data types using uppercase letters and the C data types using lowercase letters.

Notice that a macro is used to retrieve the address of the `TEXT` value. I mentioned earlier that an extension function must use macros to access parameters, and this is an example of such a macro. The `PG_GETARG_TEXT_P(n)` macro returns the `n`th parameter, which must be of type `TEXT`. The return value of `PG_GETARG_TEXT_P(n)` is of type `text`. There are many argument-accessor functions, each corresponding to a specific parameter type: `PG_GETARG_INT32(n)`, `PG_GETARG_BOOL(n)`, `PG_GETARG_OID(n)`, and so on. See the `fmgr.h` PostgreSQL header file for a complete list.

We'll be using the `stat()` function (from the C Runtime library) to find the size of a given file. `stat()` expects to find the pathname in the form of a null-terminated string. PostgreSQL has given you a `text` value, and `text` values are not null-terminated. You will need to convert `fileNameText` into a null-terminated string.

If `fileNameText` is not null-terminated, how do you know the length of the pathname? Let's take a peek at the definition of the `text` data type (from the `c.h` PostgreSQL header file):

```
struct varlena
{
    int32    vl_len;
    char     vl_data[1];
};

typedef struct varlena text;
```

You can see that a `text` value is defined by the `struct varlena` structure. The `vl_len` member tells you how many bytes are required to hold the entire structure. The characters that make up the `text` value start at the address of the `vl_data[0]` member. PostgreSQL supplies two macros that make it easy to work with variable-length data structures. The `VARHDRSZ` symbol contains the size of the fixed portion of a `struct varlena`. The `VARSIZE()` macro returns the size of the entire data structure. The `VARDATA()` macro returns a pointer to first byte of the `TEXT` value. The length of the `TEXT` value is `VARSIZE() - VARHDRSZ`. You store that length in the `fileNameLen` variable.

At line 15, you allocate enough space to hold a copy of the null-terminated string. The `palloc()` function is similar to `malloc()`: It allocates the requested number of bytes and returns a pointer to the new space. You should use `palloc()` and `pfree()` when you write extension functions rather than `malloc()` and `free()`. The `palloc()` and `pfree()` functions ensure that you can't create a memory leak in an extension function, which is something you can do if you use `malloc()` instead.

Lines 18 and 19 create a null-terminated copy of the `TEXT` value, and line 21 passes the null-terminated string to the `stat()` function. If the `stat()` function succeeds, it fills in the `statBuf` structure and returns 0.

If you succeeded in retrieving the file status information and the file is a regular file, free the null-terminated string (using `pfree()`) and return the file size. Notice that you must use a macro to translate the return value (an `int32`) into a `Datum`.

If the `stat()` function failed (or the file is not a regular file), you free the null-terminated string and return `NULL`. Again, you use a macro to produce the return value in the form of a `Datum`.

Now that you have crafted the `filesize` function, you need to compile it into a shared object module. You usually compile a C source file into a standalone executable program, but PostgreSQL expects to find the `filesize` function in a shared object module. The procedure for producing a shared object module is different for each compiler; section 31.9 of the PostgreSQL reference documentation describes the process for a number of compilers. Listing 6.2 shows the `makefile` that I've used to compile the `filesize` function using Fedora Core (Linux):

Listing 6.2. `makefile`

```
# File name: makefile
SERVER_INCLUDES += -I $(shell pg_config --includedir)
SERVER_INCLUDES += -I $(shell pg_config --includedir-server)

CFLAGS += -g $(SERVER_INCLUDES)

.SUFFIXES:      .so

.C.so:
    $(CC) $(CFLAGS) -fpic -c $<
    $(CC) $(CFLAGS) -shared -o $@ $(basename $<).o
```

To compile `filesize` using this `makefile`, you would issue the following command:

```
$ make -f makefile filesize.so
```

After the compile step is completed, you are left with a file named `filesize.so` in your current directory. The preferred location for a PostgreSQL extension can be found using the `pg_config` command:

```
$ pg_config --pkglibdir
/usr/local/pg800/lib/postgresql
```

You can copy the `filesize.so` file to this directory, but I prefer to create a symbolic link pointing back to my development directory instead. After an extension is completely debugged, I delete the symbolic link and copy the final version into the preferred location. To create a symbolic link, use the following command:

```
$ ln -s `pwd`/filesize.so `pg_config --pkglibdir`
```

At this point, you have a shared object module, but you still have to tell PostgreSQL about the function that you want to import into the server.

The `CREATE FUNCTION` command tells PostgreSQL everything it needs to know to call your function:

```
movies=# CREATE OR REPLACE FUNCTION
movies-#   filesize( TEXT ) RETURNS INTEGER AS
movies-#   'filesize.so', 'filesize' LANGUAGE 'C'
movies-#   STRICT;
CREATE
```

This command defines a function named `filesize(TEXT)`. This function returns an `INTEGER` value. The function is written in C and can be found in the file `filesize.so` in the preferred extension directory. You can specify a complete pathname to the shared object module if you want to, but in most cases it's easier to just put it where PostgreSQL expects to find it, as I've done here. You can also omit the filename extension (the `.so` part), as long as you follow the shared object module-naming rules imposed by your host operating system.

I've defined `filesize()` as a strict function. The `STRICT` attribute tells PostgreSQL that this function will always return `NULL` if any argument is `NULL`. If PostgreSQL knows that a function is `STRICT`, it can avoid calling the function with a `NULL` argument (again, a performance optimization). `STRICT` makes it easier for you to implement your extension functions; you don't have to check for `NULL` arguments if you declare your functions to be `STRICT`.

Now you can call the function from within a PostgreSQL session:

```
movies=# SELECT filesize( '/bin/bash' );
 filesize
-----
    512668
(1 row)
movies=# SELECT filesize( 'non-existent file' );
 filesize
```

(1 row)

Debugging PostgreSQL Extensions

One of the challenges you'll face in creating PostgreSQL extensions is figuring out how to debug them. Relax, it's easy. First, remember that the extension function that you create is loaded into the server (not the client). That means that when you fire up a debugger, you want to attach it to the server process. How do you find the server process? Call the `pg_backend_pid()` function once the server is up and running—`pg_backend_pid()` returns the process ID of the server that your client is connected to. Next, load the shared-object file (the file that contains your extension function) into the server with the `LOAD` command. At this point, your server is waiting for you—it's time to attach the debugger. If you're using the `gdb` debugger, you can attach to a running process with the command:

```
gdb postgres server-process-id
```

But remember, the PostgreSQL server process is owned by user `postgres`: If you try to attach without the proper privileges, `gdb` will just laugh at you. Make sure you `su postgres` before you run `gdb`. When you ask `gdb` to attach to a running process, the second argument is ignored—it has to be there, but it really doesn't matter what you string you use.

If you have the proper privileges, `gdb` should now be waiting for you to enter a command. Now you can set a breakpoint at your extension function and `gdb` will interrupt the server when that function is invoked. Notice that the server process is suspended until you tell the debugger to `continue` (if you try to execute a command from your PostgreSQL client application, the client will hang until the server wakes up again). Once you've told `gdb` to continue, you can go back to your client application and issue a command that invokes the function that you're interested in.

If you're debugging an extension function on a Windows host, the procedure is similar: Find the process ID of the server and attach a Windows debugger to that process.

To summarize:

- Start a client application (such as `psql`)
- From within the client: `SELECT pg_backend_pid();`
- From within the client: `LOAD 'extension-file.so';`
- Start another terminal session and `su postgres`
- Fire up the debugger: `$ gdb postgres server-process-id`
- Set a breakpoint: `(gdb) break my-function`
- Tell `gdb` to continue: `(gdb) cont`
- Go back to the client application and execute a command that will call your extension function

Returning Multiple Values from an Extension Function

The second extension that you will add works well with the `filesize` function. Given the name of a directory, the `filelist` function returns a list of all files (and subdirectories) contained in that directory. The `filesize` function (from the previous example) returns a single value; `filelist` will return multiple rows. An extension function that can return multiple results is called a set-returning function, or SRF.

PostgreSQL's SRF Interface

Before you read too much further, I should tell you that there's an easy way to write set-returning functions and another method that's almost as easy. I'll describe both methods, starting with the slightly more difficult approach. Starting with version 7.3, the PostgreSQL developers introduced a set of wrapper functions (and macros) that put a friendlier face on top of the original method. Under the hood, your SRF is doing the same thing whether you use the new approach or the old approach, but it's a little easier to understand the new SRF interface if you can peek under the covers.

When you are finished creating the `filelist` function, you can use it like this:

```
movies=# SELECT filelist( '/usr' );
      filelist
-----
.
..
bin
dict
etc
games
html
include
kerberos
lib
libexec
local
sbin
share
src
tmp
X11R6
(17 rows)
```

In this example, the user has invoked the `filelist` function only once, but 17 rows were returned. A SRF is actually called multiple times. In this case, the `filelist()` function is called 18 times. The first time through, `filelist()` does any preparatory work required and then returns the first result. For each subsequent call, `filelist()` returns another row until the result set is exhausted. On the 18th call, `filelist()` returns a status that tells the server that there are no more results available.

Like the `filesize` function, `filelist` takes a single argument; a directory name in the form of a `TEXT` value. This function returns a `SETOF TEXT` values. Listing 6.3 shows the first part of the `filelist.c` source file:

Listing 6.3. `filelist.c` (Part 1)

```
1 /*
2 **  Filename:  filelist.c
3 */
4
5 #include "postgres.h"
6 #include "fmgr.h"
7 #include "nodes/execnodes.h"
8
9 #include <dirent.h>
10
11 typedef struct
12 {
13     int                dir_ctx_count;
14     struct dirent **  dir_ctx_entries;
15     int                dir_ctx_current;
16 } dir_ctx;
17
18 PG_FUNCTION_INFO_V1(filelist);
19
```

`filelist.c` #includes four header files, the first three of which are supplied by PostgreSQL. `postgres.h` and `fmgr.h` provide data type definitions, function prototypes, and macros that you will need to create extensions. The `nodes/execnodes.h` header file

defines a structure (`ReturnSetInfo`) that you need because `filelist` returns a set of values. You will use the `scandir()` function to retrieve the directory contents from the operating system. The fourth header file defines a few data types that are used by `scandir()`.

Line 11 defines a structure that keeps track of your progress. In the first invocation, you will set up a context structure (`dir_ctx`) that we can use for each subsequent call. The `dir_ctx_count` member indicates the number of files and subdirectories in the given directory. The `dir_ctx_entries` member is a pointer to an array of `struct dirent` structures. Each member of this array contains a description of a file or subdirectory. `dir_ctx_current` keeps track of the current position as you traverse the `dir_ctx_entries` array.

Line 18 tells PostgreSQL that `filelist()` uses the version-1 calling convention.

Listing 6.4 shows the `filelist()` function:

Listing 6.4. `filelist.c` (Part 2)

Code View: [Scroll](#) / Show All

```
20 Datum filelist(PG_FUNCTION_ARGS)
21 {
22     FmgrInfo      * fmgr_info  = fcinfo->flinfo;
23     ReturnSetInfo * resultInfo = (ReturnSetInfo *)fcinfo->resultinfo;
24     text          * startText  = PG_GETARG_TEXT_P(0);
25     int           len          = VARSIZE( startText ) - VARHDRSZ;
26     char          * start      = (char *)palloc( len+1 );
27     dir_ctx       * ctx;
28
29     memcpy( start, startText->vl_dat, len );
30     start[len] = '\0';
31
32     if( fcinfo->resultinfo == NULL )
33         elog(ERROR, "filelist: context does not accept a set result");
34
35     if( !IsA( fcinfo->resultinfo, ReturnSetInfo ) )
36         elog(ERROR, "filelist: context does not accept a set result");
37
38     if( fmgr_info->fn_extra == NULL )
39     {
40         dir_ctx      * new_ctx;
41
42         fmgr_info->fn_extra = MemoryContextAlloc( fmgr_info->fn_mcxt,
43                                                  sizeof( dir_ctx ) );
44
45         new_ctx = (dir_ctx *)fmgr_info->fn_extra;
46
47         new_ctx->dir_ctx_count = scandir( start,
48                                         &new_ctx->dir_ctx_entries,
49                                         NULL,
50                                         alphasort );
51
52         new_ctx->dir_ctx_current = 0;
53     }
54
55     ctx = (dir_ctx *)fmgr_info->fn_extra;
56
57     if( ctx->dir_ctx_count == -1 )
58     {
59         pfree( fmgr_info->fn_extra );
60
61         fmgr_info->fn_extra = NULL;
62
63         resultInfo->isDone = ExprEndResult;
64
65         PG_RETURN_NULL();
66     }
67
68     if( ctx->dir_ctx_current < ctx->dir_ctx_count )
69     {
70         struct dirent * entry;
71         size_t        nameLen;
72         size_t        resultLen;
73         text          * result;
74
75         entry = ctx->dir_ctx_entries[ctx->dir_ctx_current];
76         nameLen = strlen( entry->d_name );
77         resultLen = nameLen + VARHDRSZ;
78
79         result = (text *)palloc( resultLen );
80
81         VARATT_SIZEP( result ) = resultLen;
82
83         memcpy( VARDATA( result ), entry->d_name, nameLen );
84
85         resultInfo->isDone = ExprMultipleResult;
```

```

85
86 /*
87 ** Advance to the next entry in our array of
88 ** filenames/subdirectories
89 */
90     ctx->dir_ctx_current++;
91
92     PG_RETURN_TEXT_P( result );
93 }
94 else
95 {
96     free( ctx->dir_ctx_entries );
97
98     pfree( fmgr_info->fn_extra );
99
100     fmgr_info->fn_extra = NULL;
101
102     resultInfo->isDone = ExprEndResult;
103
104     PG_RETURN_NULL();
105 }
106 }

```

Line 20 declares `filelist()` using the standard version-1 calling convention (remember, a version-1 function always returns a Datum and uses the `PG_FUNCTION_ARGS` preprocessor symbol as an argument list).

The C preprocessor translated line 20 into

```
Datum filesize( FunctionCallInfo fcinfo )
```

As you can see, you can access the single argument to `filesize()` through the variable `fcinfo`. All version-1 extension functions expect a `FunctionCallInfo` structure. Here is the definition of the `FunctionCallInfo` data type:

```

typedef struct FunctionCallInfoData
{
    FmgrInfo    *flinfo;        /* ptr to lookup info used for this call */
    struct Node *context;       /* pass info about context of call */
    struct Node *resultinfo;    /* pass or return extra info about result */
    bool         isnull;        /* true if result is NULL */
    short        nargs;         /* # arguments actually passed */
    Datum        arg[FUNC_MAX_ARGS]; /* Function arguments */
    bool         argnull[FUNC_MAX_ARGS]; /* T if arg[i] is NULL */
} FunctionCallInfoData;

```

There is quite a bit of information in this structure. For now, you need to know about only two of the structure members; the rest of the members are manipulated using macros, so you should pretend that you don't see them. The two members that you are interested in are `flinfo` and `resultInfo`. The `flinfo` member points to a structure of type `FmgrInfo`. The `FmgrInfo` structure looks like this:

```

typedef struct FmgrInfo
{
    PGFunction    fn_addr;      /* function or handler to be called */
    Oid           fn_oid;       /* OID of function (NOT of handler, if any) */
    short         fn_nargs;     /* 0..FUNC_MAX_ARGS, or -1 if variable arg */
    bool          fn_strict;     /* func. is "strict" (NULL in = NULL out) */
    bool          fn_retset;     /* func. returns a set (multiple calls) */
    void          *fn_extra;     /* extra space for use by handler */
    MemoryContext fn_mcxt;       /* memory context to store fn_extra in */
} FmgrInfo;

```

Look closely at the `FmgrInfo` and `FunctionCallInfo` structures. Why would you need two structures to represent a function call? The `FmgrInfo` function contains information about the definition of a function; in other words, the stuff you tell PostgreSQL in the `CREATE FUNCTION` command can be found in the `FmgrInfo` structure. The `FunctionCallInfo` structure represents a single invocation of a function. If you call the same function 20 times, you'll have 20 different `FunctionCallInfo` structures, each pointing to a single `FmgrInfo` structure. You can see the difference by comparing `FmgrInfo.fn_nargs` with `FunctionCallInfo.nargs`. `FmgrInfo.fn_nargs` tells you how many arguments were listed in the `CREATE FUNCTION` command; `FmgrInfo.fn_nargs` tells you how many arguments were passed to this particular invocation.

Line 23 declares a variable called `fmgr_info`; you'll use this to get to the `FmgrInfo` structure for this function. Line 24 declares a variable that you will use to get to the `ReturnSetInfo` structure. I'll describe the `ReturnSetInfo` structure in a moment.

Lines 24 through 30 turn the `text` argument into a null-terminated string. This is basically the same procedure you used in the `filesize()` function.

Lines 32 through 36 perform some sanity checks. It's possible to call the `filelist()` function in an inappropriate context. We know that `filelist()` returns multiple rows, so it makes sense to call that function as a target of a `SELECT` command. You could also call `filelist()` in the `WHERE` clause of a `SELECT` command, but that would be an inappropriate context (because of that multiple-row problem). When you write a function that returns a set of values, you should ensure that your function is being called in an appropriate context the way we do here.

Line 38 is where the interesting stuff starts. `fmgr_info->fn_extra` is a pointer that you can use for your own purposes; PostgreSQL doesn't do anything with this structure member except to provide for your use. The first time `filelist()` is called, the `fmgr_info->fn_extra` member is `NULL`. In each subsequent call, `fmgr_info->fn_extra` is equal to whatever you set it to in the previous call. Sounds like a great place to keep context information. Remember the `dir_ctx` structure you looked at earlier? That structure holds the information that you use to keep track of your progress as you walk through the array of file entries in a given directory.

At line 42, you know that `fmgr_info->fn_extra` is `NULL`. That implies that you have not yet started traversing a directory list. So, you allocate a `dir_ctx` structure and point `fmgr_info->fn_extra` to the new structure. The next time you are called, `fmgr_info->fn_extra` will point to the same `dir_ctx` structure (remember, there is only one `FmgrInfo` structure, regardless of how many times this function is called).

You may be thinking that I should have used `palloc()` to allocate the `dir_ctx` structure. In most extension functions, that is precisely what you should do. But in the case of an SRF, you want to allocate information related to the `FmgrInfo` structure in a different memory context^[2], the context pointed to in the `fmgr_info` structure.

^[2] You can think of a memory context as a pool of memory. Unlike `malloc()`, the `MemoryContextAlloc()` function allocates memory from a specific pool (`malloc()` allocates all memory from the same pool). A memory context has lifetime (or scope). When the scope completes, all memory allocated within that scope is automatically released. The `palloc()` function is just a wrapper around `MemoryContextAlloc()`. The memory context used by `palloc()` is destroyed at the end of a transaction (or possibly sooner).

Lines 47 through 50 do the real grunt work. You use the `scandir()` function to create an array of `struct dirent` structures. Each element in this array (`new_ctx->dir_ctx_entries`) describes a file or subdirectory. The `scandir()` function expects four parameters. The first parameter is the name of the directory that you are interested in; you pass the null-terminated string (`start`) that you crafted earlier in this function. The second parameter is a bit complex—it's a pointer to a pointer to an array of `struct dirent` structures. You know that your `dir_ctx.dir_ctx_entries` member is a pointer to an array of structures, so you pass the address of `dir_ctx_entries` and `scandir()` points `dir_ctx_entries` to the new array. The third parameter is a pointer to a structure. If you want to choose which files and subdirectories to include in the result set, you can write your own selection function and pass its address to `scandir()`. You want all files and subdirectories so you just pass in a `NULL` to tell `scandir()` not to filter the result set. The final `scandir()` parameter is a pointer to a comparison function. If you don't provide a comparison function, `scandir()` won't sort the result set. Use the `alphasort` function from the C Runtime Library—it's already written, and you aren't too concerned about performance here. For more information on `scandir()` and `alphasort()`, see the `scandir()` man page.

Finish initializing the `dir_ctx` structure by setting `dir_ctx_current` to zero. `dir_ctx_current` is incremented as you walk through the `dir_ctx_entries`.

Now that the initialization is complete, you can return your first result. But first, a quick review. You know that PostgreSQL calls this function many times and it continues to call `filelist()` until you set `resultInfo->isDone` to `ExprEndResult`. You can detect the initial call to `filelist()` by the fact that `fmgr_info->fn_extra` is `NULL`. In the initial call, you allocate a context structure and point `fmgr_info->fn_extra` to the new structure; the next time that `filelist()` is called, `fmgr_info->fn_extra` will not be `NULL`, so you know that you can skip the initialization step. Next, populate the context structure by calling the `scandir()` function: `scandir()` allocates an array of `struct dirent` structures and gives you a pointer to that array.

Line 54 retrieves the address of your context structure from `fmgr_info->fn_extra`.

Lines 56 through 65 take care of the case where the `scandir()` function fails to return any directory entries. The `scandir()` function returns the number of directory entries retrieved—it returns -1 on failure.

The details in this section of code are important. First, you must free the context structure that you allocated in the initial call (using `pfree()`). You also set `fmgr_info->fn_extra` to `NULL`; if you forget this step, the next call to `filelist()` will find a stale context structure and won't reinitialize. Remember, there is one `FunctionCallInfo` structure for each invocation, but there is never more than one `FmgrInfo` structure; you'll get the same `FmgrInfo` structure each time `filelist()` is invoked. Line 62 tells PostgreSQL that you have reached the end of the result set and line 64 returns a `NULL Datum`.

Lines 67 through 93 take care of returning a single result to the caller.

Lines 74 through 82 create a `text` value from a null-terminated directory entry (actually, ignore most of the `struct dirent` structure and just return the name portion). You first allocate a new `text` structure using `palloc()`; then set the structure size and copy the directory entry name into place. Notice that you don't copy the null-terminator: A `text` value should not be null-terminated. At line 84, you tell PostgreSQL that you are returning a result and there may be more results, so keep calling. Next, you increment the array index so that the next call to `filelist()` will return the next directory entry. Finally, you return the directory entry to the caller in the form of a `text` value.

Notice that the context structure in this section of code has not been freed. You need to preserve the `dir_ctx` structure until you have processed the last directory entry.

You reach Lines 96 through 104 once you have returned all directory entries. This section is nearly identical to the code that deals with a `scandir()` failure (lines 58-64). In fact, the only difference is that you have one more thing to clean up. When you called the `scandir()` function, it allocated an array of `struct dirent` structures using `malloc()`. You have to `free()` that array before you finish up.

That completes the C part of this function; now you have to compile it into a shared object module and tell PostgreSQL where to find it. You can use the same `makefile` that you used to compile the `filesize` function:

```
$ make -f makefile filelist.so
```

As before, you'll create a symbolic link between `filelist.so` and PostgreSQL's preferred package directory:

```
$ ln -s `pwd`/filelist.so `pg_config --pkglibdir`
```

Now the only thing remaining is to tell PostgreSQL about the new function:

```
movies=# CREATE FUNCTION filelist( TEXT )
movies=# RETURNS SETOF TEXT
movies=# AS 'filelist.so' LANGUAGE 'C';
CREATE
```

Now, let's call `filelist()` to see how it works:

```
movies=# SELECT filelist( '/usr' );
 filelist
-----
.
..
bin
dict
etc
games
html
include
kerberos
lib
libexec
local
sbin
share
src
tmp
X11R6
(17 rows)
```

Notice that the results appear in sorted order. The ordering comes because you used the `alphasort()` function when you called `scandir()`. If you don't care about the ordering, you can specify a `NULL` comparison function instead. Of course, we can ask PostgreSQL to order the data itself:

```
movies=# SELECT filelist( '/usr' ) ORDER BY filelist DESC;
 filelist
-----
X11R6
tmp
src
share
sbin
local
libexec
lib
kerberos
include
html
games
etc
dict
bin
..
.
(17 rows)
```

Now that you know how to create an SRF the hard way, I'll describe the new SRF interface that was introduced with PostgreSQL version 7.3.

The PostgreSQL SRF Interface

First off, you should know that the new SRF interface is simply a wrapper around the old method. Set-returning functions are still invoked multiple times. The first time through, an SRF initializes its own context structure and stores that structure away so that each subsequent invocation can find it. In the old approach, an SRF examined `fmgr_info->fn_extra` to determine whether it was being invoked for the first time (if `fmgr_info->fn_extra` is `NULL`, this is the first call). In the new approach, you call the `SRF_IS_FIRSTCALL()` macro instead. You can probably guess what this macro does: It returns `TRUE` if `fmgr_info->fn_extra` is `NULL` (implying that this is the first call). In fact, here's the definition of the `SRF_IS_FIRSTCALL()` macro:

```
#define SRF_IS_FIRSTCALL() ( fcinfo->flinfo->fn_extra == NULL )
```

No great surprises there.

Once you know that you're looking at the first invocation of an SRF, you typically allocate a context structure of some sort and store the address of the structure in `fmgr_info->fn_extra`. In the new approach, you call the `SRF_FIRSTCALL_INIT()` macro. This macro allocates its own context structure (a structure of type `FuncCallContext`), records the address of the structure in `fmgr_info->fn_extra`, and returns the address back to your SRF. A `FuncCallContext` structure looks like this:

```
typedef struct FuncCallContext
{
    uint32          call_cntr;
    uint32          max_calls;
    TupleTableSlot * slot;
    void            * user_fctx;
    AttInMetadata   * attinmeta;
    MemoryContext    multi_call_memory_ctx;
    TupleDesc       tuple_desc;
} FuncCallContext;
```

If the `SRF_FIRSTCALL_INIT()` macro stores its own pointer in `fmgr_info->fn_extra`, where are you supposed to store the address of your context structure? In the `user_fctx` field—that pointer is reserved for your own personal use, just like `fmgr_info->fn_extra` was reserved for your use in the old SRF mechanism. I'll explain the other members of the `FuncCallContext` structure in a moment.

Now that you have a pointer to a spanking new `FuncCallContext` (remember, `SRF_FIRSTCALL_INIT()` returns the address of the structure), you can allocate your own context structure and store its address in `user_fctx`:

```
...
FuncCallContext * srf = SRF_FIRSTCALL_INIT();
dir_ctx          * ctx;

ctx = (dir_ctx *) MemoryContextAlloc( srf->multi_call_memory_ctx,
                                     sizeof( dir_ctx ) );

srf->usr_fctx = ctx;
...
```

Notice that the `FuncCallContext` structure holds a `MemoryContext` named `multi_call_memory_ctx`. Any data that you need to save from one invocation to the next must be allocated from the `multi_call_memory_ctx` or PostgreSQL will discard that data as soon as the first invocation completes (`multi_call_memory_ctx` is equivalent to `fmgr_info->fn_mctx` in the old SRF mechanism).

Each time your SRF is invoked (even the first time), you should call the `SRF_PERCALL_SETUP()` macro. Like `SRF_FIRSTCALL_INIT()`, `SRF_PERCALL_SETUP()` returns a pointer to the `FuncCallContext` structure. The context pointer that you saved in `user_fctx` is still there. You can use that pointer to get to the context structure that you allocated (and initialized) the first time through.

The new SRF mechanism provides two more macros: `SRF_RETURN_NEXT()` and `SRF_RETURN_DONE()`. As you might expect, these macros return information to the caller. The `SRF_RETURN_NEXT()` macros returns a value (a `Datum`) to the caller and tells the server to call you again to retrieve the next value in the result set (remember, you're writing a set-returning function; the server will call your function until you indicate that you have no more results to add to the set). The `SRF_RETURN_DONE()` macros returns a `NULL` value to the caller and tells the server that you have no more results to add to the result set. `SRF_RETURN_DONE()` also deallocates the `FuncCallContext` structure so you should perform any cleanup work before you call `SRF_RETURN_DONE()`—you won't get another chance.

To show you how all of these macros fit together, [Listing 6.5](#) shows the `filelist()` function again, this time created with the new SRF mechanism:

Listing 6.5. `filelistSRF.c`

Code View: [Scroll](#) / [Show All](#)

```
1 /*
2 **  Filename:   filelistSRF.c
3 */
4
5 #include "postgres.h"
6 #include "funcapi.h"
7
```

```

 8 #include <dirent.h>
 9 #include <sys/stat.h>
10
11 typedef struct
12 {
13     struct dirent ** dir_ctx_entries;
14 } dir_ctx;
15
16 PG_FUNCTION_INFO_V1(filelist);
17
18 Datum filelist(PG_FUNCTION_ARGS)
19 {
20     text          * startText  = PG_GETARG_TEXT_P(0);
21     int           len          = VARSIZE( startText ) - VARHDRSZ;
22     char          * start      = (char *)palloc( len+1 );
23     dir_ctx       * ctx;
24     FuncCallContext * srf;
25
26     memcpy( start, startText->vl_dat, len );
27     start[len] = '\0';
28
29     if( SRF_IS_FIRSTCALL() )
30     {
31         srf = SRF_FIRSTCALL_INIT();
32
33         srf->user_fctx = MemoryContextAlloc( srf->multi_call_memory_ctx,
34                                             sizeof( dir_ctx ) );
35
36         ctx = (dir_ctx *)srf->user_fctx;
37
38         srf->max_calls = scandir(start,&ctx->dir_ctx_entries,NULL,alphasort);
39         srf->call_cntr = 0;
40     }
41
42     srf = SRF_PERCALL_SETUP();
43     ctx = (dir_ctx *)srf->user_fctx;
44
45     if( srf->max_calls == -1 )
46         SRF_RETURN_DONE( srf );
47
48     if( srf->call_cntr < srf->max_calls )
49     {
50         struct dirent * entry;
51         size_t         nameLen;
52         size_t         resultLen;
53         text          * result;
54
55         entry          = ctx->dir_ctx_entries[srf->call_cntr];
56         nameLen        = strlen( entry->d_name );
57         resultLen      = nameLen + VARHDRSZ;
58
59         result = (text *)palloc( resultLen );
60
61         VARATT_SIZEP( result ) = resultLen;
62
63         memcpy( VARDATA( result ), entry->d_name, nameLen );
64
65         SRF_RETURN_NEXT( srf, (Datum) result );
66     }
67     else
68     {
69         SRF_RETURN_DONE( srf );
70     }
71 }

```

I'll point out a few of the differences. First, notice that you don't need quite as many `#include` files when you use the new mechanism (the new `funcapi.h` header takes care of including any required headers). Next, take a look at the `dir_ctx` structure at line 11. If you compare that to the original version, you'll notice that the new version is much shorter. The `FuncCallContext` structure already contains placeholders for some of the data that used to be in `dir_ctx`. I'll explain more in a moment.

The next significant change appears at line 29. The new version of `filelist()` calls the `SRF_IS_FIRSTCALL()` macro to decide whether to initialize itself or return the next value in the result set. At line 31 you see a call to the `SRF_FIRSTCALL_INIT()` macro. That macro returns a pointer to the `FuncCallContext` structure that you're supposed to use for this invocation and for future invocations. The call to `MemoryContextAlloc()` (line 33) allocates space for a `dir_ctx` from the `srf->multi_call_memory_ctx` context. `srf->multi_call_memory_ctx` is a memory pool that survives from invocation to invocation.

Now take a look at lines 38 and 39. The call to `scandir()` returns the number of files that it finds in the given directory. In the previous version, you stored the file count in `dir_ctx->dir_ctx_count`. In the new version, you don't need an extra field (`dir_ctx_count`) to hold the file count; the `FuncCallContext` structure already has a field that serves the same purpose: `max_calls`. The SRF mechanism doesn't actually do anything with `max_calls`, it just gives you a place to store a number. At line 38, `filelist()` stores the file count in `srf->max_calls`. The `FuncCallContext` structure also has a replacement for the `dir_ctx_current` field that you saw in the original version of this function. Each time PostgreSQL calls your function, it increments the `call_cntr` field in the `FuncCallContext` structure.

`call_cntr` starts at 0 and is incremented each time you call `SRF_RETURN_NEXT()`. To summarize, the new version of `filelist()` stores the file count in `srf->max_calls` and uses `srf->call_cntr` to index into the array of filenames.

Every time the server calls this function, `filelist()` calls `SRF_PERCALL_SETUP()` to retrieve a pointer to the `FuncCallContext` structure (see line 42). If `filelist()` decides that it has no more filenames to add to the result set, it calls `SRF_RETURN_DONE()` to tell PostgreSQL that it has finished its work (lines 46 and 69). If `filelist()` does have another result, it creates a text structure and calls `SET_RETURN_NEXT()` (see the previous version of this function for a more complete explanation). `SET_RETURN_NEXT()` increments `srf->call_cntr`, returns the `Datum (result)` to the server, and tells the server that it should call this function again to retrieve the next result.

You can see that the new version of this function is very similar to the old version. The SRF macros simply hide a few of the quirks imposed by the PostgreSQL calling convention. Which approach should you use? The down-and-dirty approach or the new SRF-macro-based approach? That depends on your goals. If you need to write an extension function that will work in an older version of PostgreSQL (older than version 7.3), use the original approach (the SRF macros were added in version 7.3). If not, consider the new approach. It's possible that the SRF calling convention may change in the future and it seems safer to assume that the PostgreSQL developers will hide as many changes as possible behind the SRF macros. If you choose to use the old approach, you may find that you have to change your source code when you upgrade to a future release.

Returning Complete Rows from an Extension Function

If you've read through the first few sections in this chapter, you know how to write an extension function that returns a single scalar value (that's what the `filesize()` function does). You also know how to return a set of scalar values (that's what the `filelist()` function does). In this section, I'll show you how to return a set of rows (or, as the PostgreSQL developers prefer to call them, tuples).

To illustrate the sequence that you must follow to return multiple tuples from an extension function, I'll create a new function, `fileinfo()`, that combines `filesize()` and `filelist()`. You call `fileinfo()` with the name of a directory and it returns a `SETOF` tuples. Each tuple contains three columns: a filename, the size of the file (or `NULL` if the size is not known), and the file type (or `NULL` if the type is not known). When you've finished, you can call the `fileinfo()` function like this:

```
movies=# SELECT * FROM fileinfo( '/dev' );
 filename | filesize | filetype 
-----+-----+-----
 .         |      9380 | d
 ..        |      4096 | d
 adsp      |         0 | c
 agpggart  |         0 | c
 arpd      |         0 | c
 audio     |         0 | c
 cdrom     |         0 | b
 console   |         0 | c
 core      | 1073156096 | -
 cpu       |        360 | d
 ...
```

To start, you must define a data type that describes each row returned by the `fileinfo()` function:

Code View: [Scroll](#) / Show All

```
movies=# CREATE TYPE _fileinfo AS ( filename TEXT, filesize INTEGER, filetype CHAR(1));
CREATE TYPE
```

I'll create a few helper functions that will simplify the `fileinfo()` function. [Listing 6.6](#) shows the `getFileInfo()`, `getFileType()`, and `text2cstring()` functions:

Listing 6.6. `fileinfo.c` (Part 1)

Code View: [Scroll](#) / Show All

```
1 /*
2 ** Filename: fileinfo.c
3 */
4 #include "postgres.h"
5 #include "funcapi.h"
6
7 #include <dirent.h>
8 #include <sys/stat.h>
9
10 typedef struct
11 {
12     struct dirent ** dir_ctx_entries;
13     char            * dir_ctx_name;
14 } dir_ctx;
15
16 static bool getFileInfo(struct stat * buf, char * dirName, char * fileName)
17 {
18     char * pathName = (char *) malloc(strlen(dirName)+1+strlen(fileName)+1);
19
20     strcpy( pathName, dirName );
21     strcat( pathName, "/" );
22     strcat( pathName, fileName );
23
24     if( stat( pathName, buf ) == 0 )
25         return( true );
26     else
27         return( false );
28 }
29
30 static char getFileType( mode_t mode )
31 {
32     if( S_ISREG(mode))
33         return( '-' );
34     if( S_ISDIR(mode))
35         return( 'd' );
36     if( S_ISCHR(mode))
37         return( 'c' );
38     if( S_ISBLK(mode))
39         return( 'b' );
40     if( S_ISFIFO(mode))
41         return( 'p' );
42     if( S_ISLNK(mode))
43         return( 'l' );
44     if( S_ISSOCK(mode))
45         return( 's' );
46
47     return( '?' );
```

```

48
49 }
50
51 static char * text2cstring( text * src )
52 {
53     int     len = VARSIZE( src ) - VARHDRSZ;
54     char    * dst = (char *)palloc( len+1 );
55
56     memcpy( dst, src->vl_dat, len );
57     dst[len] = '\0';
58
59     return( dst );
60 }
61

```

The `getFileInfo()` helper function (lines 16 through 28) calls `stat()` to retrieve metadata that describes the given file. The caller provides three parameters: the address of a `struct stat` structure that `getFileInfo()` fills in, the name of the directory where the target file resides, and the name of the target file itself. If the `stat()` function succeeds, `getFileInfo()` returns `true` and the caller can find the metadata for the file in the `struct stat` structure. If the `stat()` function fails, `getFileInfo()` returns `false`.

The second helper function, `getFileType()`, translates a `mode_t` (returned by the `stat()` function) into a single character that represents a file type. `getFileType()` returns one of the following values:

- `d` (directory)
- `c` (character device)
- `b` (block device)
- `p` (named pipe)
- `l` (symbolic link)
- `s` (socket)
- `?` (unknown)
- `-` (a "regular" file—that is, not one of the above)

The last helper function is `text2cstring()` (see lines 51 through 60). This function converts a `TEXT` value into a dynamically allocated, null-terminated string. The `fileinfo()` function (which I'll describe next) calls `text2cstring()` to convert its `TEXT` argument into the form expected by `scandir()`.

The `fileinfo()` function is shown in [Listing 6.7](#):

Listing 6.7. `fileinfo.c` (Part 2)

Code View: [Scroll](#) / Show All

```

62 PG_FUNCTION_INFO_V1(fileinfo);
63
64 Datum fileinfo(PG_FUNCTION_ARGS)
65 {
66     char            * start = text2cstring( PG_GETARG_TEXT_P(0) );
67     dir_ctx         * ctx;
68     FuncCallContext * srf;
69
70     if( SRF_IS_FIRSTCALL() )
71     {
72         TupleDesc      tupdesc;
73         MemoryContext  oldContext;
74
75         srf = SRF_FIRSTCALL_INIT();
76
77         oldContext = MemoryContextSwitchTo( srf->multi_call_memory_ctx );
78
79         ctx = (dir_ctx *) palloc( sizeof( dir_ctx ) );
80
81         tupdesc = RelationNameGetTupleDesc( "_fileinfo" );
82
83         srf->user_fctx = ctx;
84         srf->max_calls = scandir( start, &ctx->dir_ctx_entries, NULL, alphasort );
85         srf->attinmeta = TupleDescGetAttInMetadata( tupdesc );
86
87         ctx->dir_ctx_name = start;
88
89         MemoryContextSwitchTo( oldContext );
90     }
91
92     srf = SRF_PERCALL_SETUP();
93     ctx = (dir_ctx *)srf->user_fctx;
94
95     if( srf->max_calls == -1 )
96         SRF_RETURN_DONE( srf );
97

```

```

98
99  if( srf->call_cntr < srf->max_calls )
100  {
101      struct dirent * entry;
102      char          * values[3];
103      struct stat    statBuf;
104      char          fileSizeStr[10+1] = {0};
105      char          fileTypeStr[1+1]  = {0};
106      HeapTuple      tuple;
107
108      entry          = ctx->dir_ctx_entries[srf->call_cntr];
109      values[0]      = entry->d_name;
110
111      if( getFileInfo( &statBuf, ctx->dir_ctx_name, entry->d_name ))
112      {
113          snprintf( fileSizeStr, sizeof( fileSizeStr ), "%d", statBuf.st_size );
114          fileTypeStr[0] = getFileType( statBuf.st_mode );
115
116          values[1] = fileSizeStr;
117          values[2] = fileTypeStr;
118      }
119      else
120      {
121          values[1] = NULL;
122          values[2] = NULL;
123      }
124
125      tuple = BuildTupleFromCStrings( srf->attinmeta, values );
126
127      SRF_RETURN_NEXT( srf, HeapTupleGetDatum( tuple ) );
128  }
129  else
130  {
131      SRF_RETURN_DONE( srf );
132  }
133 }

```

The `fileinfo()` function calls `scandir()` to generate an array that contains the names of all files in the given directory and then calls `getFileInfo()` (which in turn calls `stat()`) to retrieve the metadata for each file. The server calls `fileinfo()` until it stops returning values. Each invocation returns a single tuple (of type `_fileinfo`) that contains a filename and the size and type of that file.

`fileinfo()` starts by converting its argument from a TEXT value into a null-terminated string (the `scandir()` function that `fileinfo()` calls at line 84 requires a null-terminated string). At line 70, `fileinfo()` calls the `SRF_IS_FIRSTCALL()` macro to decide whether it should create and initialize a new context structure or use a structure created by a prior invocation.

The `fileinfo()` function has to do a little more memory-management work than the other functions you've seen in this chapter. The earlier functions allocated memory from the `srf->multi_call_memory_ctx` (or `fmgr_info>fn_mcxt`) pool. `fileinfo()` also allocates memory from that pool, but `fileinfo()` calls other PostgreSQL functions that allocate memory as well. For example, at line 81, you see a call to `RelationNameGetTupleDesc()`. That function allocates memory using the server's `palloc()` function. You must ensure that `RelationNameGetTupleDesc()` (and any function called by `RelationNameGetTupleDesc()`) allocates memory from the correct `MemoryContext`. Each `MemoryContext` has its own lifetime (or scope). If `RelationNameGetTupleDesc()` allocates memory from a `MemoryContext` with a lifetime that's too short (that is, a lifetime that ends before that last call to `fileinfo()`), you'll find that the data created by `RelationNameGetTupleDesc()` is de-allocated out from under you. On the other hand, if `RelationNameGetTupleDesc()` allocates memory from a `MemoryContext` with a lifetime that's too long, you'll create a memory leak. Take a look at line 81. Notice that you call `RelationNameGetTupleDesc()` with a single argument (the name of tuple type). Since you can't pass a `MemoryContext` to `RelationNameGetTupleDesc()`, how do you tell that function which `MemoryContext` to use? The answer is deceptively simple. Look closely at the call to `palloc()` at line 79. `palloc()` is the most commonly used memory allocation function in the PostgreSQL server. `palloc()` allocates memory from the `MemoryContext` pointed to by the `CurrentMemoryContext` global variable. If you want to talk `RelationNameGetTupleDesc()` into using a specific `MemoryContext`, you have to point `CurrentMemoryContext` to that context. That's what the code at line 77 does. Call `MemoryContextSwitchTo()` whenever you need to change the lifetime of data allocated by `palloc()`. Notice that the call at line 77 selects `srf->multi_call_memory_ctx` (which is a `MemoryContext` that survives as long as `fileinfo()` has more tuples to return). After `MemoryContextSwitchTo()` returns, `palloc()` will allocate memory from that `MemoryContext` until somebody calls `MemoryContextSwitchTo()` again. `MemoryContextSwitchTo()` switches to a new `MemoryContext` and returns the previous value. You should restore the original `MemoryContext` when you're finished with the new one (see line 89).

Once the correct `MemoryContext` is in place, `fileinfo()` allocates a new `dir_ctx` context structure (see line 79). Next, `fileinfo()` calls the `RelationNameGetTupleDesc()` function to retrieve the `TupleDesc` that defines the `_fileinfo` type (remember, the `_fileinfo` type describes the layout of the tuples returned by `fileinfo()`; you created the `_fileinfo` type earlier with a `CREATE TYPE` command). A `TupleDesc` is a structure that describes the shape of a tuple. It contains (among other things) the number of columns in the tuple and a description of each column. You don't have to peek inside of a `TupleDesc` (unless you want to) but `fileinfo()` needs the descriptor to build a return value.

After retrieving the tuple descriptor, `fileinfo()` records the address of its new context structure so it can find the structure in future invocations (line 83). Next, `fileinfo()` calls the `scandir()` function to generate an array that contains the name of each file in the given directory (`start`). `scandir()` records the address of the array in `ctx->dir_ctx_entries`.

When the `fileinfo()` function returns a value to the caller, it does so by building a tuple out of a collection of null-terminated strings; each string corresponds to one of the columns in the tuple. The `TupleDesc` that `fileinfo()` retrieved at line 81 doesn't contain quite enough information to convert C strings into a tuple. Fortunately, PostgreSQL provides a function that translates a `TupleDesc` into a new structure that contains all of the data you'll need: `TupleDescGetAttInMetaData()`. The code at line 85 calls this function and stores the address of the resulting structure in `srf->attinmeta` (which the PostgreSQL developers conveniently included for just this purpose). A little later, `fileinfo()` will use the new structure to create the return tuple.

The initialization phase completes by storing a copy of the directory name (line 87) and restoring the `MemoryContext` that was in place when `fileinfo()` was first called (line 89).

The code at lines 93 through 99 should be familiar by now—see the previous section ("The PostgreSQL SRF Interface") if you need a refresher. Every time `fileinfo()` is called, it calls the `SRF_PERCALL_SETUP()` macro to find the appropriate `FuncCallContext` structure and then extracts the address

of the `dir_ctx` structure created by the initial invocation.

I mentioned earlier that `fileinfo()` creates a return tuple out of a collection of null-terminated strings. Each tuple contains three columns: a filename, the size of the file, and the file type. Accordingly, `fileinfo()` creates three null-terminated strings (one for each column). The `values[]` array (see line 102) contains a pointer to each null-terminated string. After filling in `values[]`, `fileinfo()` will call `BuildTupleFromCStrings()` to convert the strings into a tuple.

The first null-terminated string (`values[0]`) contains the name of one file found in the `dir_ctx_entries[]` array. The assignment statement at line 109 copies the address of the file name into `values[0]`.

The other null-terminated strings (`values[1]` and `values[2]`) contain the file size and the file type (respectively). To find the size of the file, `fileinfo()` calls the `getFileInfo()` function you saw earlier. If successful, `getFileInfo()` returns true and fills in the `statBuf` structure with (among other things), the file size and type. After converting the file size into a null-terminated string (line 113) and translating the file mode into a human-readable file type (line 114), `fileinfo()` fills in the rest of the `values[]` array.

If `getFileInfo()` fails for some reason, the return tuple should contain a NULL filesize and a NULL filetype to indicate that those values are "unknown." Setting a column to NULL is easy. Just set the corresponding entry in the `values[]` array to NULL (see lines 121 and 122).

By the time it reaches line 125, `fileinfo()` has gathered all of the information it needs to create the return tuple. The `values[]` array contains three string pointers (or one string pointer and two NULL's). To convert the null-terminated strings into a tuple, `fileinfo()` calls PostgreSQL's `BuildTupleFromCStrings()`. That function uses the tuple description produced by the earlier call to `TupleDescGetAttInMetadata()` and the pointers in `values[]` to create a tuple in the form expected by the PostgreSQL server. `fileinfo()` returns the tuple to the server by invoking the `SRF_RETURN_NEXT()` macro that I described earlier (see "The PostgreSQL SRF Interface" for more information).

When `fileinfo()` has finished processing all of the file names found in `dir_ctx_entries[]`, it invokes the `SRF_RETURN_DONE()` macro instead to tell the server that it has finished building the result set.

If you want to try this function yourself, compile and install it (as described earlier) and execute the following command to tell the PostgreSQL server how to find and invoke the function:

```
CREATE OR REPLACE FUNCTION fileinfo(TEXT)
  RETURNS SETOF _fileinfo
  AS 'filelist.so','fileinfo' LANGUAGE C
  STRICT;
```

If you are rewarded with a message that states type `_fileinfo` is not yet defined, you forgot to execute the `CREATE TYPE` command that I mentioned at the beginning of this section.

You can call the `fileinfo()` function in any context where you would normally `SELECT` from a table. For example, to find the names of all files in the `/dev` directory:

```
movies=# SELECT * FROM fileinfo( '/dev' );
 filename | filesize | filetype
-----+-----+-----
 .         |      9380 | d
 ..        |      4096 | d
 adsp      |         0 | c
 agpgart   |         0 | c
 arpd      |         0 | c
 audio     |         0 | c
 cdrom     |         0 | b
 console   |         0 | c
 core      | 1073156096 | -
 cpu       |        360 | d
 ...
```

One of the cool things about PostgreSQL functions is that you can mix functions that are written in different languages. For example, you can call `fileinfo()` (which is written in C) from a function written in PL/pgSQL (one of PostgreSQL's procedural language). In fact, Listing 6.8 shows a PL/pgSQL function that returns a `SETOF _fileinfo` tuples (just like the `fileinfo()` function). This function calls `fileinfo()` to recursively descend through an entire directory tree, returning one tuple for each file (and subdirectory) that it finds.

Listing 6.8. `dirtree.sql`

Code View: [Scroll](#) / Show All

```
1 -- File: dirtree.sql
2
3 CREATE OR REPLACE FUNCTION dirtree( TEXT ) RETURNS SETOF _fileinfo AS $$
4 DECLARE
5   file _fileinfo%rowtype;
6   child _fileinfo%rowtype;
7 BEGIN
8
9   FOR file IN SELECT * FROM fileinfo( $1 ) LOOP
10     IF file.filename != '.' and file.filename != '..' THEN
11       file.filename = $1 || '/' || file.filename;
12
13       IF file.filetype = 'd' THEN
14         FOR child IN SELECT * FROM dirtree( file.filename ) LOOP
15           RETURN NEXT child;
16         END LOOP;
17       END IF;
18       RETURN NEXT file;
19     END IF;
20   END LOOP;
21
22   RETURN;
```

```
23
24 END
25 $$ LANGUAGE 'PLPGSQL';
```

Don't worry if you don't understand the `dirtree()` function yet. I'll describe the PL/pgSQL language in full detail in [Chapter 7, "PL/pgSQL."](#) The important thing to note here is that `dirtree()`, a function written in PL/pgSQL can call `fileinfo()`, a function written in C. Adding useful extension functions to PostgreSQL is not too difficult (assuming that you are comfortable working in C). Now that you understand the mechanism for creating new functions, I'd like to turn your attention to the process of creating a new data type. When you add a new data type to PostgreSQL, you must create a few supporting extension functions, so be sure you understand the material covered so far.

Extending the PostgreSQL Server with Custom Data Types

The `customers` table in this sample application contains a column named `balance`. I've made the assumption that the values in the `balance` column are expressed in local currency (that is, U.S. dollars in the U.S., British pounds in the U.K.). This assumption serves us well until our corner video store opens a web site and starts accepting orders from foreign customers.

PostgreSQL doesn't have a predefined data type that represents a foreign currency value, so let's create one. You want to store three pieces of information for each foreign currency value: the name of the currency (pounds, dollars, drachma, and so on), the number of units, and the exchange rate at the time the foreign currency value was created. Call your new data type `FCUR` (Foreign Currency). After you have fully defined the `FCUR` data type, you can create tables with `FCUR` columns, enter and display `FCUR` values, convert between `FCUR` values and other numeric types, and use a few operators (+, -, *, /) to manipulate `FCUR` values.

Internal and External Forms

Before going much further, it is important to understand the difference between the external form of a value and the internal form.

The external form of a data type defines how the user enters a value and how a value is displayed to the user. For example, if you enter a numeric value, you might enter the characters `7218942`. If you enter these characters from a client that uses an ASCII encoding, you have entered the character values 37, 32, 31, 38, 39, 34, and 32 (in hexadecimal notation). The external form of a data type is used to interact with the user.

The internal form of a data type defines how a value is represented inside the database. The preceding numeric value form might be translated from the string `7218942` into the four-byte integer value `00 6E 26 FE` (again in hexadecimal notation). The internal form of a data type is used within the database.

Why have two forms? Most programming languages can deal with numeric values implicitly (that is, without requiring the programmer to implement simple arithmetic operations). For example, the C programming language defines a built-in data type named `int`. An `int` value can store integer (that is, whole) numbers within some range determined by the compiler. The C compiler knows how to add, subtract, multiply, and divide `int` values. A C programmer is not required to perform the bit manipulations himself; the compiler emits the code required to perform the arithmetic.

Most programmers share a common understanding of what it means to add two integer values. When you add two integer values, you expect the result to be the arithmetic sum of the values. Another way to state this is to say that the `+` operator, when applied to two integer operands, should return the arithmetic sum of the operands, most likely in the form of an integer.

What would you expect the result to be if you applied the `+` operator to two string values? If each string contained only a sequence of one or more digits, such as `'1' + '34'`, you might expect the result to be the string `'35'`. What would happen if you tried adding `'1' + 'red'`? That's pretty hard to predict. Because it is difficult to come up with a good arithmetic definition of the `+` operator when applied to strings, many programming languages define `+` to mean concatenation when applied to string operands. So, the expression `'1' + 'red'` would evaluate to the string `'1red'`.

So, to summarize a bit, the external form of a numeric value is a string of numeric digits, sign characters, and a radix point. When you choose the internal form for a numeric value, you want to choose a representation that makes it easy to define and implement mathematical operations.

You've already seen the external and internal form of the `TEXT` data type. The external form of a `TEXT` value is a string of characters enclosed in single quotes (the quotes are not part of the value; they just mark the boundaries of the value). If you need to include single quotes in a `TEXT` value, the external form defines a set of rules for doing so. The internal form of a `TEXT` value is defined by the `TEXT` data type. The `TEXT` structure contains a length and an array of characters.

Defining a Simple Data Type in PostgreSQL

Now that you understand the difference between internal and external forms, it should be obvious that PostgreSQL needs to convert values between these forms. When you define a new data type, you tell PostgreSQL how to convert a value from external form to internal form and from internal form to external form.

Let's create a simple type that mimics the built-in `TEXT` data type. Data type descriptions are stored in the `pg_type` system table. We are interested in three of the columns:

```
movies=# SELECT typinput, typoutput, typplen
movies=#   FROM pg_type
movies=#   WHERE typename = 'text';
 typinput | typoutput | typplen
-----+-----+-----
 textin   | textout   |      -1
```

The `typinput` column tells you the name of the function that PostgreSQL uses to convert a `TEXT` value from external form to internal form; in this case, the function is named `textin`. The `typoutput` column contains the name of the function (`textout`) that PostgreSQL uses to convert from internal to external form. Finally, `typplen` specifies how much space is required to hold the internal form of a `TEXT` value. `TEXT` values are of variable length, so the space required to hold the internal form is also variable (-1 in this column means variable length). If `TEXT` were a fixed-length type, the `typplen` column would contain the number of bytes required to hold the internal form.

Now you have enough information to create a new data type. Here is the command that you'll use to create a type named `mytexttype`:

```
movies=# CREATE TYPE mytexttype
movies=# (
movies=#   INPUT=textin,
movies=#   OUTPUT=textout,
movies=#   INTERNALLENGTH=VARIABLE
movies=# );
```

The `INPUT=textin` clause tells PostgreSQL which function to call when it needs to convert a `mytexttype` value from external to internal form. The `OUTPUT=textout` clause tells PostgreSQL which function converts a `mytexttype` value from internal to external form. The final clause, `INTERNALLENGTH=VARIABLE`, tells PostgreSQL how much space is required to hold the internal form of a `mytexttype` value; you specify `VARIABLE` here to tell PostgreSQL that you are not defining a fixed length data type.

You have essentially cloned the `TEXT`^[3] data type. Because you are using the same input and output functions as the `TEXT` type, the internal and external form of a `mytexttype` value is identical to the internal and external form of a `TEXT` value.

[3] You have created an extremely limited clone. At this point, you can enter and display `mytexttype` values, but you can't do anything else with them. You have not defined any operators that can manipulate `mytexttype` values.

After you execute this `CREATE TYPE` command, you can use the `mytexttype` data type to create new columns:

```
movies=# CREATE TABLE myTestTable
movies=# (
movies=#   pkey INTEGER,
movies=#   value mytexttype
movies=# );
CREATE
```

You can also enter `mytexttype` values. Because you borrowed the `textin` and `textout` functions, you have to enter values according to the rules for a `TEXT` value:

```
movies=# INSERT INTO myTestTable
movies=#   VALUES ( 1, 'This is a mytexttype value in external form' );
```

Now, let's define a new data type from scratch.

Defining the Data Type in C

We'll start out by defining the internal form for an `FCUR` value. As I mentioned before, you want to store three pieces of information for each value: the name of the currency (dollars, euros, yen, and so on), the number of units, and the exchange rate at the time the value was created. Why do you need to store the exchange rate with each value? Because exchange rates vary over time, and you need to know the rate at the time the value is created.

Because you are going to use the C programming language to implement the required conversion functions, you need to define a structure^[4] containing the three components. Listing 6.9 shows the first few lines of the implementation file:

[4] This is not necessarily the most efficient (or even realistic) way to store a foreign currency value, but it works well for purposes of illustration. In a real-world implementation, you would not want to store monetary values using floating-point data types because of their inherent lack of precision. You would also want more control over the format of the currency name.

Listing 6.9. `fcurl.c` (Part 1)

```
1 /*
2 **  File name: fcurl.c
3 */
4
5 #include "postgres.h"
6 #include "fmgr.h"
7
8 typedef struct
9 {
10     char    fcur_name[4];    /* Currency name */
11     float4   fcur_units;     /* Units of currency */
12     float4   fcur_xrate;     /* Exchange rate */
13 } fcur;
14
15 static char * baseCurrencyName    = "US$";
16 static char * unknownCurrencyName = "???"
17
```

Start by `#including` the `postgres.h` and `fmgr.h` header files, just like you did for the earlier examples. The `fcur` structure defines the internal form for your `fcur` data type. Store the currency name (`fcur_name`) as a three-character, null-terminated string. The `fcur_units` member stores the number of currency units as a floating-point number. The exchange rate is stored as a floating-point number in `fcur_xrate`.

At lines 15 and 16, you define two currency names. The `baseCurrencyName` is the name of the local currency. When the `fcur_name` of a value is equal to `baseCurrencyName`, the value is said to be normalized. A normalized value will always have an exchange rate (`fcur_xrate`) of 1.0: One U.S. dollar always equals one U.S. dollar. The `unknownCurrencyName` is used when the user enters a value containing a number of units and an exchange rate, but fails to provide the currency name. We'll use each of these variables in a moment.

Defining the Input and Output Functions in C

Now you will create the input and output functions for this data type. At this point, you have to decide what your external form will look like. You know that you need to deal with three components: the number of units, an optional exchange rate, and an optional currency name. You want the typical case (units only) to be easy to enter, so you will accept input in any of the following forms:

```
units
units(exchange-rate)
units(exchange-rate/currency-name)
```

If you see a number (and nothing else), assume that you have a number of units of the base currency. If you see a number followed by an open parenthesis, you will expect an exchange rate to follow. If the exchange rate is followed by a slash character, expect a currency name. Of course, we expect a closed parenthesis if we see an open one.

Table 6.1 shows a few valid `FCUR` external values (assuming that `baseCurrencyName` is "US\$"):

Table 6.1. Sample `FCUR` Values (in External Form)

External Form	Meaning
'1'	1 U.S. dollar
'1(.5)'	1 unit of <code>unknownCurrencyName</code> with an exchange rate of 0.5
'3(1/US\$)'	3 U.S. dollars
'5(.687853/GPB)'	–5 British pounds with an exchange rate of .687853 Pounds per 1 U.S. dollar
'10(7.2566/FRF)'	–10 French francs with an exchange rate of 7.2566 Francs per 1 U.S. dollar
'1.52(1.5702/CA\$)'	–1.52 Canadian dollars with an exchange rate of 1.5702 Canadian dollars per 1 U.S. dollar

The input function is named `fcur_in` (see Listing 6.10), and it converts from external (`FCUR`) form to internal (`fcur`) form. This function expects a single parameter: a pointer to a null-terminated string containing the external form of an `fcur` value.

Listing 6.10. `fcur.c` (Part 2)

Code View: [Scroll](#) / [Show All](#)

```
18 /*
19 ** Name: fcur_in()
20 **
21 ** Converts an fcur value from external form
22 ** to internal form.
23 */
24
25 PG_FUNCTION_INFO_V1(fcur_in);
26
27 Datum fcur_in(PG_FUNCTION_ARGS)
28 {
29     char * src      = PG_GETARG_CSTRING(0);
30     char * workStr = pstrdup( src );
31     char * units    = NULL;
32     char * name     = NULL;
33     char * xrate    = NULL;
34     fcur * result   = NULL;
35     char * endPtr   = NULL;
36
37     /* strtok() will find all of the components for us */
38
39     units = strtok( workStr, "(" );
40     xrate = strtok( NULL, "/" );
41     name  = strtok( NULL, ")" );
42
43     result = (fcur *)palloc( sizeof( fcur ) );
44
45     memset( result, 0x00, sizeof( fcur ) );
46
47     result->fcur_units = strtod( units, &endPtr );
48 }
```

```

49     if( xrate )
50     {
51         result->fcur_xrate = strtod( xrate, &endPtr );
52     }
53     else
54     {
55         result->fcur_xrate = 1.0;
56     }
57
58     if( name )
59     {
60         strncpy( result->fcur_name,
61                 name,
62                 sizeof( result->fcur_name ) );
63     }
64     else
65     {
66         strncpy( result->fcur_name,
67                 unknownCurrencyName,
68                 sizeof( result->fcur_name ) );
69     }
70
71     PG_RETURN_POINTER( result );
72 }
73

```

Notice that this looks suspiciously similar to the extension functions you saw earlier in this chapter. In particular, `fcur_in()` returns a `Datum` and uses `PG_FUNCTION_ARGS` to declare the parameter list. This similarity exists because `fcur_in()` is an extension function, so everything that you already know about writing extension functions applies to this discussion as well.

You use the `strtok()` function (from the C Runtime Library) to parse out the external form. `strtok()` is a destructive function; it modifies the string that you pass to it. So the first thing you need to do in this function is to make a copy of the input string. Use the `pstrdup()` function to make the copy. `pstrdup()` is similar to the `strdup()` function from the C Runtime Library, except that the memory that holds the copy is allocated using `palloc()` and must be freed using `pfree()`. You use `pstrdup()` to avoid any memory leaks should you forget to clean up after yourself.

Lines 39, 40, and 41 parse the input string into three components. Remember, you will accept input strings in any of the following forms:

```

units
units(exchange-rate)
units(exchange-rate/currency-name)

```

The `units` component must be a string representing a floating-point number. You will use the `strtod()` runtime function to convert `units` into a `float4`, so the format of the input string must meet the requirements of `strtod()`. Here is an excerpt from the Linux `strtod()` man page that describes the required form:

```

The expected form of the string is optional leading white
space as checked by isspace(3), an optional plus ('+')
or minus sign ('-') followed by a sequence of digits
optionally containing a decimal-point character, option-
ally followed by an exponent. An exponent consists of an
'E' or 'e', followed by an optional plus or minus
sign, followed by a non-empty sequence of digits. If the
locale is not "C" or "POSIX", different formats may be
used.

```

The optional `exchange-rate` component is also converted to a `float4` by `strtod()`.

The `currency-name` component is simply a three-character string. Values such as "US\$" (U.S. dollar), "GPB" (British pound), and "CA\$" (Canadian dollar) seem reasonable. In your sample data type, you won't do any validation on this string. In a real-world implementation, you would probably want to match the currency name with a table of valid (and standardized) spellings.

The first call to `strtok()` returns a null-terminated string containing all characters up to (but not including) the first `(` in `workStr`. If `workStr` doesn't contain a `(` character, `units` will contain the entire input string. The second call to `strtok()` picks out the optional `exchange-rate` component. The final call to `strtok()` picks out the optional `currency-name`.

After you have tokenized the input string into units, exchange rate, and currency name, you can allocate space for the internal form at line 43. Notice that `palloc()` is used here.

The rest of this function is pretty simple. You use `strtod()` to convert the units and exchange rate into the `fcur` structure. If the user didn't provide you with an exchange rate, assume that it must be 1.0. You finish building the `fcur` structure by copying in the first three characters of the currency name, or `unknownCurrencyName` if you didn't find a currency name in the input string.

Line 71 returns the `Datum` to the caller.

That's pretty simple! Of course, I omitted all the error-checking code that you would need in a real-world application.

Now, let's look at the output function. `fcur_out()`, shown in [Listing 6.11](#), converts an `fcur` structure from internal to external form.

Listing 6.11. `fcur.c` (Part 3)

Code View: [Scroll](#) / Show All

```
74 /*
75 ** Name: fcur_out()
76 **
77 **      Converts an fcur value from internal form
78 **      to external form.
79 */
80
81 PG_FUNCTION_INFO_V1(fcur_out);
82
83 Datum fcur_out(PG_FUNCTION_ARGS)
84 {
85     fcur * src = (fcur *)PG_GETARG_POINTER( 0 );
86     char * result;
87     char work[16+sizeof(src->fcur_name)+16+4];
88
89     sprintf( work, "%g(%g/%s)",
90             src->fcur_units,
91             src->fcur_xrate,
92             src->fcur_name );
93
94     result = (char *)palloc( strlen( work ) + 1 );
95
96     strcpy( result, work );
97
98     PG_RETURN_CSTRING( result );
99 }
100
101
```

This function is much shorter than the input function. That's typically the case because your code has far fewer decisions to make.

You format the `fcur` components into a work buffer at lines 89 through 92: `sprintf()` takes care of all the grunt work. Notice that you are formatting into an array of characters large enough to hold the largest result that you can expect (two 16-digit numbers, a function name, two parentheses, a slash, and a null terminator). Some of you might not like using a fixed-size buffer with `sprintf()`; use `snprintf()` if you have it and you are worried about buffer overflows.

After you have a formatted string, use `palloc()` to allocate the result string. (In case you were wondering, you format into a temporary buffer first so that you can allocate a result string of the minimum possible size.) At line 96, you copy the temporary string into the result string and then return that string at line 98.

I should point out an important consideration about the input and output functions that you have just written. It's very important that the format of the string produced by the output function match the format understood by the input function. When you back up a table using `pg_dump`, the archive contains the external form of each column. When you restore from the archive, the data must be converted from external form to internal form. If they don't match, you won't be able to restore your data.

Defining the Input and Output Functions in PostgreSQL

Now that you have created the input (external to internal) and output (internal to external) functions in C, you must compile them into a shared object module:

```
$ make -f makefile fcur.so
```

Next, create a symbolic link between `fcur.so` and PostgreSQL's preferred package directory so that PostgreSQL knows how to find out code:

```
$ ln -s `pwd`/fcur.so `pg_config --pkglibdir`
```

Now you can define the input and output functions in PostgreSQL:

```
movies=# CREATE OR REPLACE FUNCTION fcur_in( opaque )
movies-# RETURNS opaque
movies-# AS 'fcur.so' LANGUAGE 'C'
movies=# IMMUTABLE STRICT
CREATE
movies=# CREATE OR REPLACE FUNCTION fcur_out( opaque )
movies-# RETURNS opaque
movies-# AS 'fcur.so' LANGUAGE 'C'
movies=# IMMUTABLE STRICT
```

Notice that each of these functions expects an `opaque` parameter and returns an `opaque` value. You might be thinking that `fcur_in()` should take a null-terminated string and return a `FCUR`. That makes sense except for two minor problems: PostgreSQL doesn't have a SQL data type that represents a null-terminated string and PostgreSQL doesn't know anything about the `FCUR` data type yet. Okay, those aren't exactly minor problems. PostgreSQL helps you out a little here by letting you define these functions in terms of `opaque`. The `opaque` data type tells PostgreSQL that a SQL data type doesn't define the data that you are working with. One of the special properties of an `opaque` function is that you can't call it directly:

```
movies=# SELECT fcur_in( '5(1.3/GPB)' );
ERROR:  getTypeOutputInfo: Cache lookup of type 0 failed
```

This error message means, "don't try that again."

We've defined each of these functions with two additional attributes. The `IMMUTABLE` attribute tells PostgreSQL that calling this function twice with the same argument(s) is guaranteed to return the same result. If PostgreSQL knows that a function `IMMUTABLE`, it can optimize certain operations by computing the return value once and caching the result (hence the clever name).

Defining the Data Type in PostgreSQL

At this point, PostgreSQL knows about your input and output functions. Now you can tell PostgreSQL about your data type:

```
CREATE TYPE FCUR ( INPUT=fcur_in, OUTPUT=fcur_out, INTERNALLENGTH=12 );
```

This command creates a new data type (how exciting) named `FCUR`. The input function is named `fcur_in`, and the output function is named `fcur_out`. The `INTERNALLENGTH=12` clause tells PostgreSQL how much space is required to hold the internal value. I computed this value by hand—just add up the size of each member of the `fcur` structure and be sure that you account for any pad bytes. The safest way to compute the `INTERNALLENGTH` is to use your C compiler's `sizeof()` operator.

Let's create a table that uses this data type and insert a few values:

```
movies=# CREATE TABLE fcur_test( pkey INT, val FCUR );
CREATE
movies=# INSERT INTO fcur_test VALUES( 1, '1' );
INSERT
movies=# INSERT INTO fcur_test VALUES( 2, '1(.5)' );
INSERT
movies=# INSERT INTO fcur_test VALUES( 3, '3(1/US$)' );
INSERT
movies=# INSERT INTO fcur_test VALUES( 4, '5(.687853/GBP)' );
INSERT
movies=# INSERT INTO fcur_test VALUES( 5, '10(7.2566/FRF)' );
INSERT
movies=# INSERT INTO fcur_test VALUES( 6, '1(1.5702/CAS$)' );
INSERT
movies=# INSERT INTO fcur_test VALUES( 7, '1.5702(1.5702/CAS$)' );
INSERT
```

Now let's see what those values look like when you retrieve them:

```
movies=# SELECT * FROM fcur_test;
 pkey |      val
-----+-----
  1   | 1(1/???)
  2   | 1(0.5/???)
  3   | 3(1/US$)
  4   | 5(0.687853/GBP)
  5   | 10(7.2566/FRF)
  6   | 1(1.5702/CAS$)
  7   | 1.5702(1.5702/CAS$)
```

Not bad. The question marks are kind of ugly, but the data that you put in came back out.

At this point, you officially have a new data type. You can put values in and you can get values out. Let's add a few functions that make the `FCUR` type a little more useful.

It would be nice to know if two `FCUR` values represent the same amount of money expressed in your local currency. In other words, you want a function, `fcur_eq`, which you can call like this:

```
movies=# SELECT fcur_eq( '1', '1.5702(1.5702/CAS$)' );
 fcur_eq
-----
 t
(1 row)

movies=# SELECT fcur_eq( '1', '3(1.5702/CAS$)' );
 fcur_eq
-----
 f
(1 row)
```

The first call to `fcur_eq` tells you that 1.5702 Canadian dollars is equal to 1 U.S. dollar. The second call tells you that 3 Canadian dollars are not equal to 1 U.S. dollar.

To compare two `FCUR` values, you must convert them into a common currency. The `normalize()` function shown in [Listing 6.12](#) does just that.

Listing 6.12. `fcur.c` 0(Part 4)

```
102 /*
103 **   Name: normalize()
104 **
105 **       Converts an fcur value into a normalized
106 **       double by applying the exchange rate.
107 */
108
109 static double normalize( fcur * src )
110 {
111     return( src->fcur_units / src->fcur_xrate );
112 }
```

The `normalize()` function converts a given `FCUR` value into our local currency. You can use `normalize()` to implement the `fcur_eq()` function, shown in [Listing 6.13](#).

Listing 6.13. `fcur.c` (Part 5)

```
115 /*
116 **   Name: fcur_eq()
117 **
118 **       Returns true if the two fcur values
119 **       are equal (after normalization), otherwise
120 **       returns false.
121 */
122
123 PG_FUNCTION_INFO_V1(fcur_eq);
124
125 Datum fcur_eq(PG_FUNCTION_ARGS)
126 {
127     fcur * left    = (fcur *)PG_GETARG_POINTER(0);
128     fcur * right   = (fcur *)PG_GETARG_POINTER(1);
129
130     PG_RETURN_BOOL( normalize( left ) == normalize( right ) );
131 }
132
```

This function is straightforward. You normalize each argument, compare them using the `C ==` operator, and return the result as a `BOOL` Datum. You declare this function as `STRICT` so that you don't have to check for `NULL` arguments.

Now you can compile your code again and tell PostgreSQL about your new function (`fcur_eq()`):

```
$ make -f makefile fcur.so
$ psql -q
movies=# CREATE OR REPLACE FUNCTION fcur_eq( fcur, fcur )
movies=#     RETURNS bool
movies=#     AS 'fcur.so' LANGUAGE 'C'
movies=#     IMMUTABLE STRICT
```

Now you can call this function to compare any two `FCUR` values:

```
movies=# SELECT fcur_eq( '1', '1.5702(1.5702/CA$)' );
 fcur_eq
-----
t
(1 row)

movies=# SELECT fcur_eq( '1', NULL );
 fcur_eq
-----
(1 row)
```

The `fcur_eq` function is nice, but you really want to compare `FCUR` values using the `=` operator. Fortunately, that's easy to do:

```
movies=# CREATE OPERATOR =
movies=# (
movies=#     leftarg     = FCUR,
movies=#     rightarg    = FCUR,
movies=#     procedure   = fcur_eq,
movies=# );
```

This command creates a new operator named `=`. This operator has a `FCUR` value on the left side and a `FCUR` value on the right side. PostgreSQL calls the `fcur_eq` function whenever it needs to evaluate this operator.

Now you can evaluate expressions such as

```
movies=# SELECT * FROM fcur_test WHERE val = '1';
 pkey |      val
-----+-----
  1   | 1(1/???)
  7   | 1.5702(1.5702/CA$)
(2 rows)
```

The operator syntax is much easier to read than the functional syntax. Let's go ahead and add the other comparison operators: `<>`, `<`, `<=`, `>`, and `>=` (see [Listing 6.14](#)). They all follow the same pattern as the `=` operator: You normalize both arguments and then compare them as double values.

Listing 6.14. `fcur.c` (Part 6)

Code View: [Scroll](#) / Show All

```
133 /*
134 **   Name: fcur_ne()
135 **
136 **       Returns true if the two fcur values
137 **       are not equal (after normalization),
138 **       otherwise returns false.
139 */
140
141 PG_FUNCTION_INFO_V1(fcur_ne);
142
143 Datum fcur_ne(PG_FUNCTION_ARGS)
144 {
145     fcur * left    = (fcur *)PG_GETARG_POINTER(0);
146     fcur * right   = (fcur *)PG_GETARG_POINTER(1);
147
148     PG_RETURN_BOOL( normalize( left ) != normalize( right ));
149 }
150
151 /*
152 **   Name: fcur_lt()
153 **
154 **       Returns true if the left operand
155 **       is less than the right operand.
156 */
157
158 PG_FUNCTION_INFO_V1(fcur_lt);
159
160 Datum fcur_lt(PG_FUNCTION_ARGS)
161 {
162     fcur * left    = (fcur *)PG_GETARG_POINTER(0);
163     fcur * right   = (fcur *)PG_GETARG_POINTER(1);
164
165     PG_RETURN_BOOL( normalize( left ) < normalize( right ));
166 }
167
168 /*
169 **   Name: fcur_le()
170 **
171 **       Returns true if the left operand
172 **       is less than or equal to the right
173 **       operand.
174 */
175
176 PG_FUNCTION_INFO_V1(fcur_le);
177
178 Datum fcur_le(PG_FUNCTION_ARGS)
179 {
180     fcur * left    = (fcur *)PG_GETARG_POINTER(0);
181     fcur * right   = (fcur *)PG_GETARG_POINTER(1);
182
183     PG_RETURN_BOOL( normalize( left ) <= normalize( right ));
184 }
185
186 /*
187 **   Name: fcur_gt()
188 **
189 **       Returns true if the left operand
190 **       is greater than the right operand.
191 */
192
193 PG_FUNCTION_INFO_V1(fcur_gt);
194
195 Datum fcur_gt(PG_FUNCTION_ARGS)
196 {
197     fcur * left    = (fcur *)PG_GETARG_POINTER(0);
198     fcur * right   = (fcur *)PG_GETARG_POINTER(1);
199
200     PG_RETURN_BOOL( normalize( left ) > normalize( right ));
201 }
202
203 /*
204 **   Name: fcur_ge()
205 **
206 **       Returns true if the left operand
207 **       is greater than or equal to the right operand.
208 */
209
210 PG_FUNCTION_INFO_V1(fcur_ge);
211
212 Datum fcur_ge(PG_FUNCTION_ARGS)
213 {
214     fcur * left    = (fcur *)PG_GETARG_POINTER(0);
215     fcur * right   = (fcur *)PG_GETARG_POINTER(1);
216
217     PG_RETURN_BOOL( normalize( left ) >= normalize( right ));
218 }
```

Now you can tell PostgreSQL about these functions:

Code View: [Scroll](#) / Show All

```
movies=# CREATE OR REPLACE FUNCTION fcur_ne( fcur, fcur )
movies=# RETURNS boolean
movies=# AS 'fcur.so' LANGUAGE 'C'
movies=# IMMUTABLE STRICT
CREATE
movies=# CREATE OR REPLACE FUNCTION fcur_lt( fcur, fcur )
movies=# RETURNS boolean
movies=# AS 'fcur.so' LANGUAGE 'C'
movies=# IMMUTABLE STRICT
CREATE
movies=# CREATE OR REPLACE FUNCTION fcur_le( fcur, fcur )
movies=# RETURNS boolean
movies=# AS 'fcur.so' LANGUAGE 'C'
movies=# IMMUTABLE STRICT
CREATE
movies=# CREATE OR REPLACE FUNCTION fcur_gt( fcur, fcur )
movies=# RETURNS boolean
movies=# AS 'fcur.so' LANGUAGE 'C'
movies=# IMMUTABLE STRICT
CREATE
movies=# CREATE OR REPLACE FUNCTION fcur_ge( fcur, fcur )
movies=# RETURNS boolean
movies=# AS 'fcur.so' LANGUAGE 'C'
movies=# IMMUTABLE STRICT
CREATE
```

And you can turn each of these functions into an operator:

Code View: [Scroll](#) / Show All

```
movies=# CREATE OPERATOR <>
movies=# (
movies=# leftarg    = fcur,
movies=# rightarg   = fcur,
movies=# procedure  = fcur_ne,
movies=# commutator = <>
movies=# );
CREATE

movies=# CREATE OPERATOR <
movies=# (
movies=# leftarg    = fcur,
movies=# rightarg   = fcur,
movies=# procedure  = fcur_lt,
movies=# commutator = >
movies=# );
CREATE

movies=# CREATE OPERATOR <=
movies=# (
movies=# leftarg    = fcur,
movies=# rightarg   = fcur,
movies=# procedure  = fcur_le,
movies=# commutator = >=
movies=# );
CREATE

movies=# CREATE OPERATOR >
movies=# (
movies=# leftarg    = fcur,
movies=# rightarg   = fcur,
movies=# procedure  = fcur_gt,
movies=# commutator = <
movies=# );
CREATE

movies=# CREATE OPERATOR >=
movies=# (
movies=# leftarg    = fcur,
movies=# rightarg   = fcur,
movies=# procedure  = fcur_ge,
movies=# commutator = <=
movies=# );
CREATE
```

Notice that there is a `commutator` for each of these operators. The `commutator` can help PostgreSQL optimize queries that involve the operator.

For example, let's say that you have an index that covers the `balance` column. With a `commutator`, the query

```
SELECT * FROM customers WHERE balance > 10 and new_balance > balance;
```

can be rewritten as

```
SELECT * FROM customers WHERE balance > 10 and balance < new_balance;
```

This allows PostgreSQL to perform a range scan using the `balance` index. The `commutator` for an operator is the operator that PostgreSQL can use to swap the order of the operands. For example, `>` is the commutator for `<` because if $x > y$, $y < x$. Likewise, `<` is the commutator for `>`. Some operators are `commutators` for themselves. For example, the `=` operator is a commutator for itself. If $x = y$ is true, then $y = x$ is also true.

There are other optimizer hints that you can associate with an operator. See the `CREATE OPERATOR` section of the PostgreSQL Reference Manual for more information.

I'll finish up this chapter by defining one more operator (addition) and two functions that extend the usefulness of the `FCUR` data type.

First, let's look at a function that adds two `FCUR` values (see [Listing 6.15](#)):

Listing 6.15. `fcur.c` (Part 7)

Code View: [Scroll](#) / [Show All](#)

```
259 /*
260 ** Name: fcur_add()
261 **
262 ** Adds two fcur values, returning the result
263 ** If the operands are expressed in the same
264 ** currency (and exchange rate), the result
265 ** will be expressed in that currency,
266 ** otherwise, the result will be in normalized
267 ** form.
268 */
269
270 PG_FUNCTION_INFO_V1(fcur_add);
271
272 Datum fcur_add(PG_FUNCTION_ARGS)
273 {
274     fcur * left = (fcur *)PG_GETARG_POINTER(0);
275     fcur * right = (fcur *)PG_GETARG_POINTER(1);
276     fcur * result;
277
278     result = (fcur *)palloc( sizeof( fcur ) );
279
280     if( left->fcur_xrate == right->fcur_xrate )
281     {
282         if( strcmp( left->fcur_name, right->fcur_name ) == 0 )
283         {
284             /*
285             ** The two operands have a common currency - preserve
286             ** that currency by constructing a new fcur with the
287             ** same currency type.
288             */
289             result->fcur_xrate = left->fcur_xrate;
290             result->fcur_units = left->fcur_units + right->fcur_units;
291             strcpy( result->fcur_name, left->fcur_name );
292
293             PG_RETURN_POINTER( result );
294         }
295     }
296
297     result->fcur_xrate = 1.0;
298     result->fcur_units = normalize( left ) + normalize( right );
299     strcpy( result->fcur_name, baseCurrencyName );
300
301     PG_RETURN_POINTER( result );
302 }
303 }
304
```

This function returns a `FCUR` datum; at line 278, we use `palloc()` to allocate the return value. `fcur_add()` has a nice feature: If the two operands have a common currency and a common exchange rate, the result is expressed in that currency. If the operands are not expressed in a common currency, the result will be a value in local currency.

Lines 289 through 291 construct the result in a case where the operand currencies are compatible. If the currencies are not compatible, construct the result at lines 297 through 299.

Let's tell PostgreSQL about this function and make an operator (+) out of it:

```
movies=# CREATE OR REPLACE FUNCTION fcur_add( fcur, fcur )
movies=# RETURNS fcur
movies=# AS 'fcur.so' LANGUAGE 'C'
movies=# IMMUTABLE STRICT
CREATE
movies=# CREATE OPERATOR +
movies=# (
movies=# leftarg = fcur,
movies=# rightarg = fcur,
movies=# procedure = fcur_add,
```

```

movies=#   commutator = +
movies=# );
CREATE

```

Now, try it:

```

movies=# SELECT *, val + '2(1.5702/CA$)' AS result FROM fcur_test;
 pkey |      val      |      result
-----+-----+-----
    1 | 1(1/???)     | 2.27372(1/US$)
    2 | 1(0.5/???)   | 3.27372(1/US$)
    3 | 3(1/US$)     | 4.27372(1/US$)
    4 | 5(0.687853/GBP) | 8.54272(1/US$)
    5 | 10(7.2566/FRF) | 2.65178(1/US$)
    6 | 1(1.5702/CA$) | 3(1.5702/CA$)
    7 | 1.5702(1.5702/CA$) | 3.5702(1.5702/CA$)
(7 rows)

```

Notice that the `result` values for rows 6 and 7 are expressed in Canadian dollars.

Creating other arithmetic operators for the `FCUR` type is simple. If the operands share a common currency (and exchange rate), the result should be expressed in that currency. I'll let you add the rest of the arithmetic operators.

The last two functions that I wanted to show you (see [Listing 6.16](#)) will convert `FCUR` values to and from `REAL` values. Internally, the `REAL` data type is known as a `float4`.

Listing 6.16. `fcur.c` (Part 8)

```

220 /*
221 **   Name: fcur_to_float4()
222 **
223 **       Converts the given fcur value into a
224 **       normalized float4.
225 */
226
227 PG_FUNCTION_INFO_V1(fcur_to_float4);
228
229 Datum fcur_to_float4(PG_FUNCTION_ARGS)
230 {
231     fcur * src = (fcur *)PG_GETARG_POINTER(0);
232
233     PG_RETURN_FLOAT4( normalize( src ));
234 }
235 }

```

The `fcur_to_float4()` function converts an `FCUR` value into a normalized `FLOAT4` (that is, `REAL`) value. There isn't anything fancy in this function; let `normalize()` do the heavy lifting.

[Listing 6.17](#) shows the `float4_to_fcur()` function:

Listing 6.17. `fcur.c` (Part 9)

```

237 /*
238 **   Name: float4_to_fcur()
239 **
240 **       Converts the given float4 value into an
241 **       fcur value
242 */
243
244 PG_FUNCTION_INFO_V1(float4_to_fcur);
245
246 Datum float4_to_fcur(PG_FUNCTION_ARGS)
247 {
248     float4 src = PG_GETARG_FLOAT4(0);
249     fcur * result = (fcur *)palloc( sizeof( fcur ));
250
251     result->fcur_units = src;
252     result->fcur_xrate = 1.0;
253
254     strcpy( result->fcur_name, baseCurrencyName );
255
256     PG_RETURN_POINTER( result );
257 }

```

The `float4_to_fcur()` function is a bit longer, but it's not complex. You allocate space for the `result` using `palloc()`; then create the `result` as a value expressed in your local currency.

When you tell PostgreSQL about these functions, you won't follow the same form that you have used in earlier examples:


```

movies=# CREATE OR REPLACE FUNCTION FCUR( FLOAT4 )
movies=# RETURNS FCUR
movies=# AS 'fcur.so','float4_to_fcur'
movies=# LANGUAGE 'C'
movies=# IMMUTABLE STRICT
CREATE

```

Notice that the internal (C) name for this function is `float4_to_fcur()`, but the external (PostgreSQL) name is `FCUR`. Older versions of PostgreSQL (release 7.2 or older) know that the `FCUR` function can be used to implicitly convert a `FLOAT4` (or `REAL`) value into a `FCUR` value. PostgreSQL considers a function to be a conversion function if all of the following are true:

- The name of the function is the same as the name of a data type.
- The function returns a value whose type is the same as the function's name.
- The function takes a single argument of some other data type.

You can see that the `FCUR` function meets these criteria. Let's create the `FLOAT4` function along the same pattern:

```

movies=# CREATE OR REPLACE FUNCTION FLOAT4( FCUR )
movies=# RETURNS FLOAT4
movies=# AS 'fcur.so','fcur_to_float4'
movies=# LANGUAGE 'C'
movies=# IMMUTABLE STRICT
CREATE

```

If you're using PostgreSQL version 7.3 or later, you must explicitly tell the PostgreSQL server that `FLOAT4(FCUR)` and `FCUR(FLOAT4)` are conversion functions. To create a `CAST` that will convert a `FLOAT4` value to an `FCUR` value, execute the following command:

Code View: [Scroll](#) / [Show All](#)

```

movies=# CREATE CAST( FLOAT4 AS FCUR ) WITH FUNCTION FCUR( float4 ) AS IMPLICIT;

```

The `CREATE CAST` command specified the source type (`FLOAT4`), the target type (`FCUR`) and the signature of the conversion function (`FCUR(float4)`). The `AS IMPLICIT` clause tells PostgreSQL that it can silently convert `FLOAT4` values to `FCUR` values whenever it needs to; you don't have to write things like `CAST(4.0 AS FCUR)` once you've defined an `IMPLICIT CAST`.

Don't forget to create a `CAST` that will convert values in the other direction:

Code View: [Scroll](#) / [Show All](#)

```

movies=# CREATE CAST( FCUR AS FLOAT4 ) WITH FUNCTION FLOAT4( FCUR ) AS IMPLICIT;

```

Now PostgreSQL knows how to (implicitly) convert between `FLOAT4` values and `FCUR` values. Why is that so important? You can now use a `FCUR` value in any context in which a `FLOAT4` value is allowed. If you haven't defined a particular function (or operator), PostgreSQL will implicitly convert the `FCUR` value into a `FLOAT4` value and then choose the appropriate function (or operator).

For example, you have not defined a multiplication operator for your `FCUR` data type, but PostgreSQL knows how to multiply `FLOAT4` values:

```

movies=# SELECT *, (val * 5) as "Result" FROM fcur_test;

```

pkey	val	Result
1	1(1/???)	5
2	1(0.5/???)	10
3	3(1/US\$)	15
4	5(0.687853/GBP)	36.3449764251709
5	10(7.2566/FRF)	6.89027905464172
6	1(1.5702/CAS)	3.18430781364441
7	1.5702(1.5702/CAS)	5

You can now multiply `FCUR` values. Notice that the `Result` column does not contain `FCUR` values. PostgreSQL converted the `FCUR` values into `FLOAT4` values and then performed the multiplication. Of course, you can cast the result back to `FCUR` form. Here, we use the `@` (absolute value) operator to convert from `FCUR` to `FLOAT4` form and then cast the result back into `FCUR` form:

```

movies=# SELECT *, CAST( abs(val) AS FCUR ) FROM fcur_test;

```

pkey	val	fcu
1	1(1/???)	1(1/US\$)
2	1(0.5/???)	2(1/US\$)
3	3(1/US\$)	3(1/US\$)
4	5(0.687853/GBP)	7.269(1/US\$)
5	10(7.2566/FRF)	1.37806(1/US\$)
6	1(1.5702/CAS)	0.636862(1/US\$)
7	1.5702(1.5702/CAS)	1(1/US\$)

(7 rows)

Notice that all the result values have been normalized into your local currency.

Indexing Custom Data Types

At this point, you have a reasonably complete custom data type. You can create and display `FCUR` values, store them in a table, compare two `FCUR` values, and convert them to (and from) other data types. But you're missing one important feature: You can't create an index that includes an `FCUR` value. Once you have all of the comparison operators (`<`, `<=`, `=`, `>=`, and `>`) in place you are two short steps away.

To index values of a given data type, you must create an operator class that tells PostgreSQL which operators it should use for that type. You may recall from [Chapter 3, "PostgreSQL Syntax and Use,"](#) that PostgreSQL supports a number of index types (B-tree, hash, R-tree, and GiST). Each index type requires a different set of operators. For example, to build a B-tree index, PostgreSQL can make use of five different operators: `<`, `<=`, `=`, `>=`, and `>`. Before you can create an operator class that PostgreSQL can use to build B-tree indexes over `FCUR` values, you'll need one more function.

[Listing 6.18](#) shows the `fcur_cmp()` function that compares two `FCUR` values:

Listing 6.18. `fcur.c` (Part 10)

```
305 PG_FUNCTION_INFO_V1(fcur_cmp);
306
307 Datum fcur_cmp(PG_FUNCTION_ARGS)
308 {
309     fcur * left      = (fcur *)PG_GETARG_POINTER(0);
310     fcur * right     = (fcur *)PG_GETARG_POINTER(1);
311     double left_dbl  = normalize( left );
312     double right_dbl = normalize( right );
313
314     if( left_dbl > right_dbl )
315         PG_RETURN_INT32( 1 );
316     else if( left_dbl < right_dbl )
317         PG_RETURN_INT32( -1 );
318     else
319         PG_RETURN_INT32( 0 );
320 }
```

PostgreSQL will call `fcur_cmp()` repeatedly as it builds a B-tree index. `fcur_cmp()` expects two arguments, both of type `fcur`. After normalizing the values, `fcur_cmp()` returns `+1` if the first argument is greater than the second, `-1` if the second argument is greater than the first, or `0` if the arguments are equal.

Don't forget to tell PostgreSQL how to find this function:

```
CREATE OR REPLACE FUNCTION FCUR_CMP( FCUR, FCUR )
    RETURNS INT4
    AS 'fcur.so'
    LANGUAGE 'C'
    IMMUTABLE STRICT;
```

Now you have all of the pieces in place; you can create an operator class with the following command:

```
CREATE OPERATOR CLASS fcur_ops
    DEFAULT FOR TYPE fcur USING BTREE as
    OPERATOR 1 <,
    OPERATOR 2 <=,
    OPERATOR 3 =,
    OPERATOR 4 >=,
    OPERATOR 5 >,
    FUNCTION 1 fcur_cmp( fcur, fcur );
```

That's it; once you've created an operator class for type `FCUR`, you can create an index that includes values of that type. To create an operator class for the other index types (hash, R-tree, and GiST), you'll have to create a few more support functions. See section 31 ("Extending SQL") of the PostgreSQL reference documentation for more details.

Chapter 7. PL/ pgSQL

PL/pgSQL (Procedural Language/PostgreSQL) is a language that combines the expressive power of SQL with the more typical features of a programming language. PL/pgSQL adds control structures such as conditionals, loops, and exception handling to the SQL language. When you write a PL/pgSQL function, you can include any and all SQL commands, as well as the procedural statements added by PL/pgSQL.

Functions written in PL/pgSQL can be called from other functions. You can also define a PL/pgSQL function as a trigger. A trigger is a procedure that executes when some event occurs. For example, you might want to execute a PL/pgSQL function that fires when a new row is added to a table—that's what a trigger is for. You can define triggers for the `INSERT`, `UPDATE`, and `DELETE` commands.

Installing PL/ pgSQL

PostgreSQL can support a variety of procedural languages. Before you can use a procedural language, you have to install it into the database. Fortunately, this is a simple procedure.

The `createlang` shell script installs PL/pgSQL into a database. If you install PL/pgSQL in the `template1` database, it will automatically be installed in all databases created from that template. The format for `createlang` is

```
createlang plpgsql database-name
```

To install PL/pgSQL in the `movies` database, execute the following command:

```
$ createlang plpgsql movies
```

Notice that this is a command-line utility, not a `psql` command.

Language Structure

PL/pgSQL is termed a block-structured language. A block is a sequence of statements between a matched set of `DECLARE/BEGIN` and `END` statements. Blocks can be nested—meaning that one block can entirely contain another block, which in turn can contain other blocks, and so on. For example, here is a PL/pgSQL function:

Code View: [Scroll](#) / Show All

```
1 --
2 -- ch07.sql
3 --
4
5 CREATE OR REPLACE FUNCTION my_factorial(value INTEGER) RETURNS INTEGER AS $$
6   DECLARE
7     arg INTEGER;
8   BEGIN
9
10    arg := value;
11
12    IF arg IS NULL OR arg < 0 THEN
13      RAISE NOTICE 'Invalid Number';
14      RETURN NULL;
15    ELSE
16      IF arg = 1 THEN
17        RETURN 1;
18      ELSE
19        DECLARE
20          next_value INTEGER;
21        BEGIN
22          next_value := my_factorial(arg - 1) * arg;
23          RETURN next_value;
24        END;
25      END IF;
26    END IF;
27  END;
28 $$ LANGUAGE 'plpgsql';
```

The body of `my_factorial()` is actually the string between the opening dollar quotes (following the word `AS`) and the closing dollar quotes (just before the word `LANGUAGE`).

This function contains two blocks of code. The first block starts at line 6 and ends at line 27. The second block, which is nested inside the first, starts at line 19 and ends at line 24. The first block is called an outer block because it contains the inner block.

I'll talk about variable declarations in more detail in a moment, but I want to point out a few things here. At line 7, we declare a variable named `arg`. This variable has a well-defined lifetime. `arg` comes into existence when the function reaches the first `DECLARE` statement and goes out of existence as soon as the function reaches the `END` statement at line 27. The lifetime of a variable is also referred to as its scope. You can refer to a variable in any statement within the block that defines the scope of the variable. If you try to refer to a variable outside of its scope, you will receive a compilation error. Remember that you have two (nested) blocks in this function: the outer block and the inner block. Variables declared in an outer block can be used in inner blocks, but the reverse is not true. At line 22 (which is in the inner block), we use the `arg` variable, which was declared in the outer block. The variable `next_value` is declared within the inner block: If you try to use `next_value` in the outer block, you'll get an error.

This function (`my_factorial()`) contains two blocks, one nested within the other. You can nest blocks as deeply as you need to. You can also define blocks that are not nested. Here is the `my_factorial()` function again, but this time, I've included a few more blocks:

Code View: [Scroll](#) / Show All

```
1 --
2 -- ch07.sql
3 --
4
5 CREATE FUNCTION my_factorial( value INTEGER ) RETURNS INTEGER AS $$
6   DECLARE
7     arg INTEGER;
8   BEGIN
9
10    arg := value;
11
12    IF arg IS NULL OR arg < 0 THEN
13      BEGIN
14        RAISE NOTICE 'Invalid Number';
15        RETURN NULL;
16      END;
17    ELSE
18      IF arg = 1 THEN
19        BEGIN
20          RETURN 1;
21        END;
22      ELSE
23        DECLARE
24          next_value INTEGER;
25        BEGIN
26          next_value := my_factorial(arg - 1) * arg;
```

```

27         RETURN next_value;
28     END;
29 END IF;
30 END IF;
31 END;
32 $$ LANGUAGE 'plpgsql';

```

This version still has an outer block (lines 6 through 31), but you have multiple inner blocks: lines 13 through 16, lines 19 through 21, and lines 23 through 28. As I said earlier, variables declared in an outer block can be used in inner blocks but the reverse is not true. If you had declared any variables in the block starting at line 19, you could not use any of those variables past the end of the block (at line 21).

Notice that you can indicate the beginning of a block with a `DECLARE` statement or with a `BEGIN` statement. If you need to declare any variables within a block, you must include a `DECLARE` section. If you don't need any local variables within a block, the `DECLARE` section is optional (an empty `DECLARE` section is perfectly legal).

Quoting Embedded Strings

Prior to version 8.0, including string literals in a PL/pgSQL function was difficult and error prone. Because the body of a PL/pgSQL function is itself a string, you had to double up the quote characters around any string literals within the function.

Take a close look at line 14 in the previous example:

```
RAISE NOTICE 'Invalid Number';
```

Notice that the string literal `Invalid Number` is surrounded by a set of single quotes. You can write an embedded string value that way because the body of the function is defined in a string delimited by PostgreSQL's new dollar-quoting mechanism. If you don't use dollar-quoting to define function body, you must double up the quotes, like this:

```
RAISE NOTICE ''Invalid Number'';
```

If you're using a version of PostgreSQL older than 8.0, you can't use dollar-quoting and you'll have to write embedded string literals in one of the other forms described in [Chapter 2](#), "Working with Data in PostgreSQL." You could have written the embedded string in any of the three following forms:

```

RAISE NOTICE ''Invalid Number'';
RAISE NOTICE \'Invalid Number\';
RAISE NOTICE \047Invalid Number\047;

```

CREATE FUNCTION

Now, let's go back and look at the components of a function in more detail.

You define a new PL/pgSQL function using the `CREATE FUNCTION` command. The `CREATE FUNCTION` command comes in two forms. The first form is used for language interpreters that are embedded into the PostgreSQL server—PL/pgSQL functions fall into this category:

```

CREATE [OR REPLACE] FUNCTION name ( [[argname] argtype [, ...] ] )
    RETURNS return_type
    AS $$definition$$
    LANGUAGE langname
    [ WITH ( attribute [, ...] ) ]

```

The second form is used to define functions that are defined in an external language and compiled into a dynamically loaded object module:

```

CREATE [OR REPLACE] FUNCTION name ( [[argname] argtype [, ...] ] )
    RETURNS return_type
    AS $$obj_file$$, $$link_symbol$$
    LANGUAGE langname
    [ WITH ( attribute [, ...] ) ]

```

I covered compiled functions in more detail in [Chapter 6](#), "Extending PostgreSQL." For this chapter, I'll focus on the first form. Don't forget, if you're using a version of PostgreSQL older than 8.0, you can't use `$$` to delimit string values and you'll have to carefully quote embedded strings as described earlier in this chapter.

Each function has a name. However, the name alone is not enough to uniquely identify a PostgreSQL function. Instead, the function name and the data types of each argument (if any) are combined into a signature. A function's signature uniquely identifies the function within a database. This means that you can define many `my_factorial()` functions:

```

CREATE FUNCTION my_factorial( INTEGER )...
CREATE FUNCTION my_factorial( REAL )...
CREATE FUNCTION my_factorial( NUMERIC )...

```

Each of these functions is uniquely identified by its signature. When you call one of these functions, you provide the function name and an argument; PostgreSQL determines which function to use by comparing the data type of the arguments that you provide with the function signatures. If an exact match is found, PostgreSQL uses that function. If PostgreSQL can't find an exact match, it tries to find the closest match.

When you create a new function, you specify a list of arguments required by that function. In most programming languages, you would declare a name and a type for each function argument. In PL/pgSQL, you declare only the data type. The first argument is automatically named "\$1", the second argument is named "\$2", and so forth, up to a maximum of 32 arguments (if you're using a version of PostgreSQL older than 8.0, you're limited to 16 arguments per function). Starting with PostgreSQL version 8.0, you can include argument names in the `CREATE FUNCTION` command. That means that you can define the `my_factorial()` function like this:

```
CREATE FUNCTION my_factorial( inputArgument INTEGER )...
CREATE FUNCTION my_factorial( inputArgument REAL )...
CREATE FUNCTION my_factorial( inputArgument NUMERIC )...
```

Inside of `my_factorial()`, you can refer to the first argument as `$1` or as `inputArgument`. If you include argument names in the `CREATE FUNCTION` command, the names are not considered to be part of the function signature. If you define a function such as

```
CREATE FUNCTION my_factorial( inputArgument INTEGER )...
```

the function's signature is `my_factorial(INTEGER)`—you can `DROP` the function without specifying argument names (in fact, if you do specify argument names in a `DROP FUNCTION` command, the names are ignored).

You can use predefined data types, user-defined data types, and arrays of those types in a PL/pgSQL function.

It is important to remember that PL/pgSQL does not support default parameters. If you define a function that requires three parameters, you cannot call that function with fewer (or more) parameters. If you find that you need a function with a variable argument list, you can usually overload your function to obtain the same effect. When you overload a function, you define two (or more) functions with the same name but different argument lists. For example, let's define a function to compute the due date for a tape rental:

```
1 --
2 -- ch07.sql
3 --
4
5 CREATE FUNCTION compute_due_date( DATE ) RETURNS DATE AS $$
6   DECLARE
7
8     due_date      DATE;
9     rental_period INTERVAL := '7 days';
10
11 BEGIN
12
13   due_date := $1 + rental_period;
14
15   RETURN due_date;
16
17 END;
18 $$ LANGUAGE 'plpgsql';
```

This function takes a single parameter, a `DATE` value, and returns the date one week later. You might want a second version of this function that expects the rental date and a rental period:

```
20 -- ch07.sql
21 --
22 CREATE FUNCTION compute_due_date( DATE, INTERVAL ) RETURNS DATE AS $$
23 BEGIN
24
25   RETURN( $1 + $2 );
26
27 END;
28 $$ LANGUAGE 'plpgsql';
```

Now you have two functions named `compute_due_date()`. One function expects a `DATE` value, and the other expects a `DATE` value and an `INTERVAL` value. The first function `compute_due_date(DATE)`, provides the equivalent of a default parameter. If you call `compute_due_date()` with a single argument, the `rental_period` defaults to seven days.

I'd like to point out two things about the `compute_due_date(DATE, INTERVAL)` function.

First, a stylistic issue—the `RETURN` statement takes a single argument, the value to be returned to the caller. You can `RETURN` any expression that evaluates to the `return_type` of the function (we'll talk more about a function's `return_type` in a moment). I find it easier to read a `RETURN` statement if the expression is enclosed in parentheses (see line 25).

Second, you'll notice that I did not `DECLARE` any local variables. You can treat parameter variables just like any other variable—I used them in an expression in line 25. It's a rare occasion when you should settle for the automatic variable names supplied for function parameters. The name "\$1" doesn't convey much meaning beyond telling you that this variable happens to be the first parameter. You should really provide a meaningful name for each parameter; this gives the reader some idea of what you intended to do with each parameter.

If you're using an older version of PostgreSQL (or you're writing code that must work on an older version), you can use the `ALIAS` statement to give a second, more meaningful name to a parameter. Here is the `compute_due_date(DATE, INTERVAL)` function again, but this time I

have given alternate names to the parameters:

```
20 -- ch07.sql
21 --
22 CREATE FUNCTION compute_due_date(DATE, INTERVAL) RETURNS DATE AS '
23 DECLARE
24     rental_date    ALIAS FOR $1;
25     rental_period  ALIAS FOR $2;
26 BEGIN
27
28     RETURN( rental_date + rental_period );
29
30 END;
31 ' LANGUAGE 'plpgsql';
```

`ALIAS` gives you an alternate name for a parameter: you can still refer to an aliased parameter using the `$n` form, but I don't recommend it. Why bother to give a meaningful name to a parameter and then ignore it?

Starting with PostgreSQL version 8.0, you can skip the `ALIAS` commands and simply name the arguments in the `CREATE FUNCTION` command, like this:

```
CREATE FUNCTION compute_due_date(rental_date DATE, rental_period INTERVAL)
    RETURNS DATE AS ...
```

When you create a function, you must declare the data type of the return value. Our `compute_due_date()` functions return a value of type `DATE`. A value is returned from a function using the `RETURN` expression statement. Keep in mind that PL/pgSQL will try to convert the returned expression into the type that you specified when you created the function. If you tried to `RETURN('Bad Value')` from the `compute_due_date()` function, you would get an error (`Bad Date External Representation`). We'll see a special data type a little later (`TRIGGER`, or in versions older than 8.0, `OPAQUE`) that can be used only for trigger functions.

If you're writing a PL/pgSQL function that you want to run in a version of PostgreSQL older than 7.3, you must ensure that the function returns a value, even if it only returns `NULL`. Starting with version 7.3, you can define functions that return type `void`. A function that returns type `void` doesn't actually return a value—you would call such a function for the side effects provided by the function.

I'll skip over the function body^[1] for the moment and look at the final component^[2] required to define a new function. PostgreSQL functions can be written in a variety of languages. When you create a new function, the last component that you specify is the name of the language in which the body of the function is written. All the functions that you will see in this chapter are written in PL/pgSQL, which PostgreSQL knows as `LANGUAGE 'plpgsql'`.

^[1] The function body is everything between the `AS` keyword and the `LANGUAGE` keyword. The function body is specified in the form of a string.

^[2] When you create a function, you can also specify a set of optional attributes that apply to that function. These attributes tell PostgreSQL about the behavior of the function so that the query optimizer can know whether it can take certain shortcuts when evaluating the function. See the `CREATE FUNCTION` section in the PostgreSQL Programmer's Guide for more information.

DROP FUNCTION

Before you experiment much more with PL/pgSQL functions, it might be useful for you to know how to replace the definition of a function.

If you are using PostgreSQL 7.2 or later, you can use the `CREATE OR REPLACE FUNCTION ...` syntax. If a function with the same signature already exists, PostgreSQL will silently replace the old version of the function; otherwise, a new function is created.

If you are using a version of PostgreSQL older than 7.2, you will have to `DROP` the old function before you can create a new one. The syntax for the `DROP FUNCTION` command is

```
DROP FUNCTION name( [[argname] argtype [, ...] ] );
```

Notice that you have to provide the complete signature when you drop a function; otherwise, PostgreSQL would not know which version of the function to remove.

Of course, you can use the `DROP FUNCTION` command to simply remove a function—you don't have to replace it with a new version.

Function Body

Now that you have an overview of the components of a PL/pgSQL function, let's look at the function body in greater detail. I'll start by showing you how to include documentation (that is, comments) in your PL/pgSQL functions. Next, I'll look at variable declarations. Finally, I'll finish up this section by describing the different kinds of statements that you can use inside of a PL/pgSQL function.

Comments

There are two comment styles in PL/pgSQL. The most frequently seen comment indicator is the double dash: `--`. A double dash introduces a comment that extends to the end of the current line. For example:

```
-- This line contains a comment and nothing else
DECLARE
    customer_id  INTEGER;      -- This is also a comment
--   due_date    DATE;         -- This entire line is a comment
--                               -- because it begins with a '--'
```

PL/pgSQL understands C-style comments as well. A C-style comment begins with the characters `/*` and ends with the characters `*/`. A C-style comment can span multiple lines:

```
/*
    NAME: compute_due_date()

    DESCRIPTION:  This function will compute the due date for a tape
                  rental.

    INPUT:
                  $1 -- Date of original rental

    RETURNS:      A date indicating when the rental is due.
*/

CREATE FUNCTION compute_due_date( DATE ) RETURNS DATE
...
```

Choosing a comment style is purely a matter of personal preference. Of course, the person choosing the style may not be you—you may have to conform to coding standards imposed by your customer (and/or employer). I tend to use only the double-dash comment style in PL/pgSQL code. If I want to include a multi-line comment, I start each line with a double dash:

```
-----
-- NAME: compute_due_date()
--
-- DESCRIPTION:  This function will compute the due date for a tape
--               rental.
--
-- INPUT:
--               $1 -- Date of original rental
--
-- RETURNS:      A date indicating when the rental is due.
--
CREATE FUNCTION compute_due_date( DATE ) RETURNS DATE
...
```

I find that the double-dash style looks a little cleaner.

Variables

The variable declarations that you've seen up to this point have all been pretty simple. There are actually five ways to introduce a new variable (or at least a new variable name) into a PL/pgSQL function.

- Each parameter defines a new variable.
- You can declare new variables in the `DECLARE` section of a block.
- You can create an alternate name for a function parameter using the `ALIAS` statement.
- You can define a new name for a variable (invalidating the old name) using the `RENAME` statement.
- The iterator variable for an integer-based `FOR` loop is automatically declared to be an integer.

Let's look at these variables one at a time.

Function Parameters

I mentioned earlier in this chapter that each parameter in a PL/pgSQL function is automatically assigned a name. The first parameter (in left-to-right order) is named `$1`, the second parameter is named `$2`, and so on. You define the data type for each parameter in the function definition—for example:

```
CREATE FUNCTION write_history( DATE, rentals )...
```

This function expects two parameters. The first parameter is named \$1 and is of type DATE. The second parameter is named \$2 and is of type rentals. If you're using a newer version of PostgreSQL (8.0 or later), you can also define your own names for function parameters:

```
CREATE FUNCTION write_history( historyDate DATE, rentalRecord rentals )...
```

In this case, you've given two names to each parameter. You can refer to the first parameter as historyDate or as \$1 and the second parameter as rentalRecord or \$2. When you include parameter names in a CREATE FUNCTION command, you're assigning aliases for the parameters without explicitly writing ALIAS commands.

Notice that the write_history() function (in the preceding code line) expects an argument of type rentals. In the sample database, 'rentals' is actually the name of a table. Inside of the write_history() function, you can use the rentalRecord parameter (also known as \$2) as if it were a row in the rentals table. That means that you can work with rentalRecord.tape_id, rentalRecord.customer_id, rentalRecord.rental_date, or \$2.tape_id, \$2.customer_id, and \$2.rental_date.

When you call this function, you need to pass a row from the rentals table as the second argument. For example:

```
SELECT write_history( CURRENT_DATE, rentals ) FROM rentals;
```

DECLARE

The second way to introduce a new variable into a PL/pgSQL function is to list the variable in the DECLARE section of a block. The name of a non-parameter variable can include alphabetic characters (A-Z), underscores, and digits. Variable names must begin with a letter (A-Z or a-z) or an underscore. Names are case-insensitive: my_variable can also be written as My_Variable, and both still refer to the same variable.

The PL/pgSQL documentation mentions that you can force a variable name to be case-sensitive by enclosing it in double quotes. For example, "pi". As of PostgreSQL 7.1.3, this does not seem to work. You can enclose a variable name within double quotes if you need to start the name with a digit.

Oddly enough, you can actually DECLARE a variable whose name starts with a '\$', \$3, for example, but I wouldn't recommend it; I would expect that this feature (bug?) may be removed (fixed?) at some point in the future.

The complete syntax for a variable declaration is

```
var-name [CONSTANT] var-type [NOT NULL] [{ DEFAULT | := } expression];
```

Some of the examples in this chapter have declared variables using the most basic form:

```
due_date          DATE;
rental_period      INTERVAL := '7 days';
```

The first line creates a new variable named due_date. The data type of due_date is DATE. Because I haven't explicitly provided an initial value for due_date, it will be initialized to NULL.

The second line defines a new INTERVAL variable named rental_period. In this case, I have provided an initial value, so rental_period will be initialized to the INTERVAL value '7 days'. I could have written this declaration as

```
rental_period      INTERVAL DEFAULT '7 days';
```

In the DECLARE section of a block, DEFAULT is synonymous with ':='.

The initializer expression must evaluate to a value of the correct type. If you are creating an INTEGER variable, the initializer expression must evaluate to an INTEGER value or to a type that can be coerced into an INTEGER value.

In newer versions of PostgreSQL, you can declare array variables by writing a set of square brackets (and an optional element count) following the data type. For example, the declaration

```
monthly_balances NUMERIC(7,2) [12] := '{}';
```

defines a variable named monthly_balances as an array of 12 numeric values. There is one counter-intuitive quirk that you should know about when you declare an array variable. If you define an array without an initializer, the array is NULL—apparently, that's not the same thing as saying that the array is full of NULL values. You can't insert individual values into a NULL array. That means that code such as the following will silently fail:

```
DECLARE
    monthly_balances NUMERIC(7,2) [12];
BEGIN
    monthly_balances[1] := 10;
    monthly_balances[2] := monthly_balances[1] * 1.10;
    ...
```

You can't insert a value into a `NULL` array, but you can copy an entire array over the top of a `NULL` array:

```
DECLARE
    new_balances NUMERIC(7,2) [12];
    old_balances NUMERIC(7,2) [12] := '{}';
BEGIN
    new_balances := old_balances;
    ...
```

So you can only put a value into an array by initializing it or by copying another array over the top of it.

You can define PL/pgSQL functions that take array values as arguments, and you can return array values from PL/pgSQL functions.

Prior to PostgreSQL version 8.0, the `DECLARE` section had a couple of surprises up its sleeve. First, you could use any of the function parameters in the initializer expression, even if you `ALIASED` them. The following is illegal:

```
CREATE FUNCTION compute_due_date(DATE) RETURNS DATE AS '
DECLARE
    due_date DATE := $1 + '7 days':INTERVAL;
    ...
```

ERROR: Parameter \$1 is out of range

The second issue was that once you created a variable in a `DECLARE` section, you could not use that variable later within the same `DECLARE` section. That meant that you couldn't do something like

```
CREATE FUNCTION do_some_geometry(REAL) RETURNS REAL AS '
DECLARE
    pi CONSTANT REAL := 3.1415926535;
    radius REAL := 3.0;
    diameter REAL := pi * ( radius * radius );
    ...
```

ERROR: Attribute 'pi' not found

Both of these problems have been fixed in release 8.0.

Notice in the previous example that I declared `pi` to be a `'CONSTANT REAL'`. When you define a variable as `CONSTANT`, you prevent assignment to that variable. You must provide an initializer for a `CONSTANT`.

The final modifier for a variable declaration is `NOT NULL`. Defining a variable to be `NOT NULL` means that you will receive an error if you try to set that variable to `NULL`. You must provide an initializer when you create a `NOT NULL` variable^[3].

^[3] This makes perfect sense if you think about it. If you don't provide an initializer, PL/pgSQL will initialize each variable to `NULL`—you can't do that if you have declared the variable to be `NOT NULL`.

Now you can put all these pieces together. The following declarations are identical in function:

```
pi CONSTANT REAL NOT NULL DEFAULT 3.1415926535;
pi CONSTANT REAL NOT NULL := 3.1415926535;
pi CONSTANT REAL := 3.1415926535;
```

Each declares a `REAL` variable named `pi`, with an initial value of 3.14159265. The `NOT NULL` clause is superfluous here because we have declared `pi` to be a constant and we have given it a non-null initial value; it's not a bad idea to include `NOT NULL` for documentation purposes.

The default value for a variable is computed each time you enter the block that declares it. If you define a default value in terms of an expression, the variables and functions within that expression can change value from one execution to the next. For example, if an inner block declares a variable whose default value is defined by a variable in an outer block, the default value will vary with the outer variable.

Pseudo Data Types—`%TYPE` and `%ROWTYPE`

When you create a PL/pgSQL variable, you must declare its data type. Before moving on to the `ALIAS` command, there are a few pseudo data types that you should know about.

`%TYPE` lets you define one variable to be of the same type as another. Quite often, you will find that you need to temporarily store a value that you have retrieved from a table, or you might need to make a copy of a function parameter. Let's say that you are writing a function to process a rentals record in some way:

```
CREATE FUNCTION process_rental( rentals ) RETURNS BOOLEAN AS $$
DECLARE
    original_tape_id      CHAR(8);
    original_customer_id  INTEGER;
    original_rental_row   ALIAS FOR $1;

BEGIN

    original_tape_id      := original_rental_row.tape_id;
    original_customer_id := original_rental_row.customer_id;
    ...
```

In this snippet, you are making a local copy of the `rentals.tape_id` and `rentals.customer_id` columns. Without `%TYPE`, you have to ensure that you use the correct data types when you declare the `original_tape_id` and `original_customer_id` variables.

That might not sound like such a big deal now, but what about six months later when you decide that eight characters isn't enough to hold a tape ID?

Instead of doing all that maintenance work yourself, you can let PL/pgSQL do the work for you. Here is a much better version of the `process_rental()` function:

```
CREATE FUNCTION process_rental( rentals ) RETURNS BOOLEAN AS $$
DECLARE
    original_tape_id      rentals.tape_id%TYPE;
    original_customer_id  rentals.customer_id%TYPE;
    original_rental_row   ALIAS FOR $1;

BEGIN

    original_tape_id      := original_rental_row.tape_id;
    original_customer_id := original_rental_row.customer_id;
    ...
```

By using `%TYPE`, I've told PL/pgSQL to create the `original_tape_id` variable using whatever type `rentals.tape_id` is defined to be. I've also created `original_customer_id` with the same data type as the `rentals.customer_id` column.

This is an extremely powerful feature. At first blush, it may appear to be just a simple timesaving trick that you can use when you first create a function. The real power behind `%TYPE` is that your functions become self-maintaining. If you change the data type of the `rentals.tape_id` column, the `process_rentals()` function will automatically inherit the change. You won't have to track down all the places where you have made a temporary copy of a `tape_id` and change the data types.

You can use the `%TYPE` feature to obtain the type of a column or type of another variable (as shown in the code that follows). You cannot use `%TYPE` to obtain the type of a parameter. Starting with PostgreSQL version 7.2, you can use `%TYPE` in the argument list for a function. For example:

```
CREATE FUNCTION process_rental( rentals, rentals.customer_id%TYPE )
RETURNS BOOLEAN AS '
DECLARE
    original_tape_id      rentals.tape_id%TYPE;
    original_customer_id  rentals.customer_id%TYPE;
    original_rental_row   ALIAS FOR $1;
    ...
```

`%TYPE` lets you access the data type of a column (or variable). `%ROWTYPE` provides similar functionality. You can use `%ROWTYPE` to declare a variable that has the same structure as a row in the given table. For example:

```
CREATE FUNCTION process_rental( rentals ) RETURNS BOOLEAN AS $$
DECLARE
    original_tape_id      rentals.tape_id%TYPE;
    original_customer_id  rentals.customer_id%TYPE;
    original_rental_row   rentals%ROWTYPE;
    ...
```

The `original_rental_row` variable is defined to have the same structure as a row in the `rentals` table. You can access columns in `original_rental_row` using the normal dot syntax: `original_rental_row.tape_id`, `original_rental_row.rental_date`, and so on.

Using `%ROWTYPE`, you can define a variable that has the same structure as a row in a specific table. A bit later in this chapter, I'll show you how to process dynamic queries (see the section "[EXECUTE](#)"); that is, a query whose text is not known at the time you are writing your function. When you are processing dynamic queries, you won't know which table to use with `%ROWTYPE`.

Other Pseudo Types

The `RECORD` data type is used to declare a composite variable whose structure will be determined at execution time. I'll describe the `RECORD` type in more detail a bit later (see the section "[Loop Constructs](#)").

PostgreSQL version 7.3 introduced a new pseudo type named `TRIGGER`. A function defined with a return type of `TRIGGER` can only be used as a trigger function. I'll describe trigger functions later in this chapter (see the section titled "[Triggers](#)").

The final pseudo data type is `OPAQUE`. The `OPAQUE` type can be used only to define the return type of a function^[4]. You cannot declare a variable (or parameter) to be of type `OPAQUE`. In fact, you can use `OPAQUE` only to define the return type of a trigger function only. `OPAQUE` is an obsolete name; you should define trigger functions using the `TRIGGER` type instead.

^[4] You can use `OPAQUE` to define the data type of a function argument, but not when you are creating a PL/pgSQL function. Remember, functions can be defined in a number of different languages.

ALIAS and RENAME

Now, let's move on to the next method that you can use to define a new variable, or a least a new name for an existing variable. You've already seen the `ALIAS` statement earlier in this chapter. The `ALIAS` statement creates an alternative name for a function parameter. You cannot `ALIAS` a variable that is not a function parameter. Using `ALIAS`, you can define any number of names that equate to a parameter:

```
CREATE FUNCTION foo( INTEGER ) RETURNS INTEGER AS '
DECLARE
    param_1 ALIAS FOR $1;
    my_param ALIAS FOR $1;
    arg_1 ALIAS FOR $1;
BEGIN
    $1 := 42;
    -- At this point, $1, param_1, my_param and arg_1
    -- are all set to 42.
    ...

```

As we've mentioned already, if you're using PostgreSQL version 8.0 or later, you can skip the `ALIAS` commands and simply name each parameter in the argument list.

The `RENAME` statement is similar to `ALIAS`; it provides a new name for an existing variable. Unlike `ALIAS`, `RENAME` invalidates the old variable name. You can `RENAME` any variable, not just function parameters. The syntax for the `RENAME` statement is

```
RENAME old-name TO new-name
```

Here is an example of the `RENAME` statement:

```
CREATE FUNCTION foo( INTEGER ) RETURNS INTEGER AS '
DECLARE
    RENAME $1 TO param1;
BEGIN
    ...

```

Important Note

The `RENAME` statement does not work in PostgreSQL versions 7.1.2 through at least 7.4, but it appears to function correctly in version 8.0.

`RENAME` and `ALIAS` can be used only within the `DECLARE` section of a block.

FOR Loop Iterator

So far, you have seen four methods for introducing a new variable or a new variable name. In each of the preceding methods, you explicitly declare a new variable (or name) in the `DECLARE` section of a block and the scope of the variable is the block in which it is defined. The final method is different.

One of the control structures that you will be looking at soon is the `FOR` loop. The `FOR` loop comes in two flavors—the first flavor is used to execute a block of statements some fixed number of times; the second flavor executes a statement block for each row returned by a query. In this section, I will talk only about the first flavor.

Here is an example of a `FOR` loop:

```
FOR i IN 1 .. 12 LOOP
    balance := balance + customers.monthly_balances[i];
END LOOP;
```

In this example, you have defined a loop that will execute 12 times. Each statement within the loop (you have only a single statement) will be executed 12 times. The variable `i` is called the iterator for the loop (you may also see the term loop index to describe the iterator). Each time you go through this loop, the iterator (`i`) is incremented by 1.

The iterator for an integer `FOR` loop is automatically declared for you. The type of the iterator is `INTEGER`. It is important to remember that the iterator for an integer `FOR` loop is a new variable. If you have already declared a variable with the same name as the iterator, the original variable will be hidden for the remainder of the loop. For example:

```
...
DECLARE
    i REAL = 0;
    balance NUMERIC(9,2) = 0;
BEGIN

    --
    -- At this point, i = 0
    --

    FOR i IN 1 .. 12 LOOP

        --
        -- we now have a new copy of i, it will vary from 1 to 12
        --

        balance := balance + customers.monthly_balances[i];
    END LOOP;

    --
    -- Now, if we access i, we will find that it is

```

```
-- equal to 0 again
--
```

Notice that while you are inside the loop, there are two variables named `i`—the inner variable is the loop iterator, and the outer variable was declared inside of this block. If you refer to `i` inside the loop, you are referring to the inner variable. If you refer to `i` outside the loop, you are referring to the outer variable. A little later, I'll show you how to access the outer variable from within the loop.

Now that you have seen how to define new variables, it's time to move on. This next section explains each type of statement that you can use in the body of a PL/pgSQL function.

PL/ pgSQL Statement Types

At the beginning of this chapter, I said that PL/pgSQL adds a set of procedural constructs to the basic SQL language. In this next section, I'll examine the statement types added by PL/pgSQL. PL/pgSQL includes constructs for looping, exception and error handling, simple assignment, and conditional execution (that is, IF/THEN/ELSE). Although I don't describe them here, it's important to remember that you can also include any SQL command in a PL/pgSQL function.

Assignment

The most commonly seen statement in many programs is the assignment statement. Assignment lets you assign a new value to a variable. The format of an assignment statement should be familiar by now; you've already seen it in most of the examples in this chapter:

```
target := expression;
```

`target` should identify a variable, a function parameter, a column, or in some cases, a row. If `target` is declared as `CONSTANT`, you will receive an error. When PL/pgSQL executes an assignment statement, it starts by evaluating the `expression`. If `expression` evaluates to a value whose data type is not the same as the data type of `target`, PL/pgSQL will convert the value to the `target` type. (In cases where conversion is not possible, PostgreSQL will reward you with an error message.)

The `expression` is actually evaluated by the PostgreSQL server, not by PL/pgSQL. This means that `expression` can be any valid PostgreSQL expression. [Chapter 2](#), "Working with Data in PostgreSQL," describes PostgreSQL expressions in more detail.

SELECT INTO

The assignment statement is one way to put data into a variable; `SELECT INTO` is another. The syntax for a `SELECT INTO` statement is

```
SELECT INTO destination [, ...] select-list FROM ...;
```

A typical `SELECT INTO` statement might look like this:

```
...
DECLARE
    customer    customers%ROWTYPE;
BEGIN
    SELECT INTO customer * FROM customers WHERE customer_id = 10;
...

```

When this statement is executed, PL/pgSQL sends the query "SELECT * FROM customers WHERE customer_id = 10" to the server. This query should not return more than one row. The results of the query are placed into the `customer` variable. Because I specified that `customer` is of type `customers%ROWTYPE`, the query must return a row shaped exactly like a `customers` row; otherwise, PL/pgSQL signals an error.

I could also `SELECT INTO` a list of variables, rather than into a single composite variable:

```
DECLARE
    phone      customers.phone%TYPE;
    name       customers.customer_name%TYPE;
BEGIN
    SELECT INTO name, phone
        customer_name, customers.phone FROM customers
        WHERE customer_id = 10;
...

```

Notice that I had to explicitly request `customers.phone` in this query. If I had simply requested `phone`, PL/pgSQL would have assumed that I really wanted to execute the query:

```
SELECT customer_name, NULL FROM customers where customer_id = 10;
```

Why? Because I have declared a local variable named `phone` in this function, and PL/pgSQL would substitute the current value of `phone` wherever it occurred in the query. Because `phone` (the local variable) is initialized to `NULL`, PL/pgSQL would have stuffed `NULL` into the query. You should choose variable names that don't conflict with column names, or fully qualify column name references.

Of course, you can also `SELECT INTO` a `RECORD` variable and the `RECORD` will adapt its shape to match the results of the query.

I mentioned earlier that the query specified in a `SELECT INTO` statement must return no more than one row. What happens if the query returns no data? The variables that you are selecting into are set to `NULL`. You can also check the value of the predefined variable `FOUND` (described

later in this chapter) to determine whether a row was actually retrieved. What happens if the query returns more than one row? If you're using an older version of PostgreSQL, PL/pgSQL will throw an error at you. If you're using PostgreSQL version 8.0 or later, the target variables are filled in with values from the first row returned by the `SELECT` command.

A bit later in this chapter, you'll see the `FOR-IN-SELECT` loop that can handle an arbitrary number of rows (see the section "[Loop Constructs](#)").

Conditional Execution

Using the `IF` statement, you can conditionally execute a section of code. The most basic form of the `IF` statement is

```
IF expression THEN
    statements
END IF;
```

The expression must evaluate to a `BOOLEAN` value or to a value that can be coerced into a `BOOLEAN` value. If expression evaluates to `TRUE`, the statements between `THEN` and `END IF` are executed. If expression evaluates to `FALSE` or `NULL`, the statements are not executed.

Here are some sample `IF` statements:

```
IF ( now() > rentals.rental_date + rental_period ) THEN
    late_fee := handle_rental_overdue();
END IF;

IF ( customers.balance > maximum_balance ) THEN
    PERFORM customer_over_balance( customers );
    RETURN( FALSE );
END IF;
```

In each of these statements, the condition expression is evaluated by the PostgreSQL server. If the condition evaluates to `TRUE`, the statements between `THEN` and `END IF` are executed; otherwise, they are skipped and execution continues with the statement following the `END IF`.

You can also define a new block within the `IF` statement:

```
IF ( tapes.dist_id IS NULL ) THEN
    DECLARE
        default_dist_id CONSTANT integer := 0;
    BEGIN
        ...
    END;
END IF;
```

The obvious advantage to defining a new block within an `IF` statement is that you can declare new variables. It's usually a good idea to declare variables with the shortest possible scope; you won't pollute the function's namespace with variables that you need in only a few places, and you can assign initial values that may rely on earlier computations.

The next form of the `IF` statement provides a way to execute one section of code if a condition is `TRUE` and a different set of code if the condition is not `TRUE`. The syntax for an `IF-THEN-ELSE` statement is

```
IF expression THEN
    statements_1
ELSE
    statements_2
END IF;
```

In this form, `statements_1` will execute if expression evaluates to `TRUE`; otherwise, `statements_2` will execute. Note that `statements_2` will not execute if the expression is `TRUE`. Here are some sample `IF-THEN-ELSE` statements:

```
IF ( now() > rentals.rental_date + rental_period ) THEN
    late_fee := handle_rental_overdue();
ELSE
    late_fee := 0;
END IF;

IF ( customers.balance > maximum_balance ) THEN
    PERFORM customer_over_balance( customers );
    RETURN( FALSE );
ELSE
    rental_ok = TRUE;
END IF;
```

An `IF-THEN-ELSE` is almost equivalent to two `IF` statements. For example, the following

```
IF ( now() > rentals.rental_date + rental_period ) THEN
    statements_1
ELSE
    statements_2
END IF;
```

is nearly identical to

```
IF ( now() > rentals.rental_date + rental_period ) THEN
    statements_1
END IF;

IF ( now() <= rentals.rental_date + rental_period ) THEN
    statements_2
END IF;
```

The difference between these two scenarios is that using `IF-THEN-ELSE`, the condition expression is evaluated once; but using two `IF` statements, the condition expression is evaluated twice. In many cases, this distinction won't be important; but in some circumstances, the condition expression may have side effects (such as causing a trigger to execute), and evaluating the expression twice will double the side effects.

You can nest `IF-THEN-ELSE` statements:

```
IF ( today > compute_due_date( rentals ) ) THEN
    --
    -- This rental is past due
    --
    ...
ELSE
    IF ( today = compute_due_date( rentals ) ) THEN
        --
        -- This rental is due today
        --
        ...
    ELSE
        --
        -- This rental is not late and it's not due today
        --
        ...
    END IF;
END IF;
```

PostgreSQL versions 7.2 and later support a more convenient way to nest `IF-THEN-ELSE-IF` statements:

```
IF ( today > compute_due_date( rentals ) ) THEN
    --
    -- This rental is past due
    --
    ...
ELSIF ( today = compute_due_date( rentals ) ) THEN
    --
    -- This rental is due today
    --
    ...
ELSE
    --
    -- This rental is not late and it's not due today
    --
    ...
END IF;
```

The `ELSIF` form is functionally equivalent to a nested `IF-THEN-ELSE-IF` but you need only a single `END IF` statement. Notice that the spelling is `ELSIF`, not `ELSE IF`. You can include as many `ELSIF` sections as you like.

Loop Constructs

Next, let's look at the loop constructs offered by PL/pgSQL. Using a loop, you can repeat a sequence of statements until a condition occurs. The most basic loop construct is the `LOOP` statement:

```
[<<label>>]
LOOP
    statements
END LOOP;
```

In this form, the statements between `LOOP` and `END LOOP` are repeated until an `EXIT` or `RETURN` statement exits the loop. If you don't include an `EXIT` or `RETURN` statement, your function will loop forever. I'll explain the optional `<<label>>` in the section that covers the `EXIT` statement.

You can nest loops as deeply as you need:

Code View: [Scroll](#) / Show All

```
1 row := 0;
2
3 LOOP
4     IF( row = 100 ) THEN
5         EXIT;
6     END IF;
7
8     col := 0;
```



```

9
10 LOOP
11     IF( col = 100 ) THEN
12         EXIT;
13     END IF;
14
15     PERFORM process( row, col );
16
17     col := col + 1;
18
19 END LOOP;
20
21 row := row + 1;
22 END LOOP;
23
24 RETURN( 0 );

```

In the preceding code snippet, there are two loops. Because the inner loop is completely enclosed within the outer loop, the inner loop executes each time the outer loop repeats. The statements in the outer loop execute 100 times. The statements in the inner loop (lines 10 through 19) execute 100 x 100 times.

The `EXIT` statement at line 5 causes the outer `LOOP` to terminate; when you execute that statement, execution continues at the statement following the `END LOOP` for the enclosing loop (at line 24). The `EXIT` statement at line 12 will change the point of execution to the statement following the `END LOOP` for the enclosing loop (at line 21).

I'll cover the `EXIT` statement in more detail in the next section.

The next loop construct is the `WHILE` loop. The syntax for a `WHILE` loop is

```

[<<label>>]
WHILE expression LOOP
    statements
END LOOP;

```

The `WHILE` loop is used more frequently than a plain `LOOP`. A `WHILE` loop is equivalent to

```

[<<label>>]
LOOP

    IF( NOT ( expression )) THEN
        EXIT;
    END IF;

    statements

END LOOP;

```

The condition `expression` must evaluate to a `BOOLEAN` value or to a value that can be coerced to a `BOOLEAN`. The `expression` is evaluated each time execution reaches the top of the loop. If `expression` evaluates to `TRUE`, the statements within the loop are executed. If `expression` evaluates to `FALSE` or `NULL`, execution continues with the statement following the `END LOOP`.

Here is the nested loop example again, but this time, I have replaced the `IF` tests with a `WHILE` loop:

```

1 row := 0;
2
3 WHILE ( row < 100 ) LOOP
4
5     col := 0;
6
7     WHILE ( col < 100 ) LOOP
8
9         PERFORM process( row, col );
10
11         col := col + 1;
12
13     END LOOP;
14
15     row := row + 1;
16 END LOOP;
17
18 RETURN( 0 );

```

You can see that the `WHILE` loop is much neater and easier to understand than the previous form. It's also a lot easier to introduce a bug if you use a plain `LOOP` and have to write the `IF` tests yourself.

The third loop construct is the `FOR` loop. There are two forms of the `FOR` loop. In the first form, called the `integer-FOR` loop, the loop is controlled by an integer variable:

```

[<<label>>]
FOR iterator IN [ REVERSE ] start-expression .. end-expression LOOP
    statements

```

```
END LOOP;
```

In this form, the statements inside the loop are repeated while the `iterator` is less than or equal to `end-expression` (or greater than or equal to if the loop direction is `REVERSE`). Just before the first iteration of the loop, `iterator` is initialized to `start-expression`. At the bottom of the loop, `iterator` is incremented by 1 (or -1 if the loop direction is `REVERSE`); and if within the `end-expression`, execution jumps back to the first statement in the loop.

An integer-FOR loop is equivalent to:

```
[<<label>>]
DECLARE
    iterator INTEGER;
    increment INTEGER;
    end_value INTEGER;
BEGIN
    IF( loop-direction = REVERSE ) THEN
        increment := -1;
    ELSE
        increment := 1;
    END IF;

    iterator := start-expression;
    end_value := end-expression;

    LOOP
        IF( iterator >= end_value ) THEN
            EXIT;
        END IF;

        statements

        iterator := iterator + increment;

    END LOOP;
END;
```

The `start-expression` and `end-expression` are evaluated once, just before the loop begins. Both expressions must evaluate to an `INTEGER` value or to a value that can be coerced to an `INTEGER`.

Here is the example code snippet again, this time written in the form of an integer-FOR loop:

```
1 FOR row IN 0 .. 99 LOOP
2
3   FOR col in 0 .. 99 LOOP
4
5       PERFORM process( row, col );
6
7   END LOOP;
8
9 END LOOP;
10
11 RETURN( 0 );
12
```

This version is more readable than the version that used a `WHILE` loop. All the information that you need in order to understand the loop construct is in the first line of the loop. Looking at line 1, you can see that this loop uses a variable named `row` as the iterator; and unless something unusual happens inside the loop, `row` starts at 0 and increments to 99.

There are a few points to remember about the integer-FOR loop. First, the `iterator` variable is automatically declared—it is defined to be an `INTEGER` and is local to the loop. Second, you can terminate the loop early using the `EXIT` (or `RETURN`) statement. Third, you can change the value of the `iterator` variable inside the loop: Doing so can affect the number of iterations through the loop.

You can use this last point to your advantage. In PL/pgSQL, there is no way to explicitly specify a loop increment other than 1 (or -1 if the loop is `REVERSED`). But you can change the effective increment by modifying the iterator within the loop. For example, let's say that you want to process only odd numbers inside a loop:

```
1 ...
2 FOR i IN 1 .. 100 LOOP
3   ...
4   i := i + 1;
5   ...
6 END LOOP;
7 ...
```

The first time you go through this loop, `i` will be initialized to 1. At line 4, you increment `i` to 2. When you reach line 6, the `FOR` loop will increment `i` to 3 and then jump back to line 3 (the first line in the loop). You can, of course, increment the loop iterator in whatever form you need. If you fiddle with the loop iterator, be sure to write yourself a comment that explains what you're doing.

The second form of the `FOR` loop is used to process the results of a query. The syntax for this form is

```
[<<label>>]
FOR iterator IN query LOOP
    statements
END LOOP;
```

In this form, which I'll call the `FOR-IN-SELECT` form, the statements within the loop are executed once for each row returned by the query. query must be a SQL `SELECT` command. Each time through the loop, `iterator` will contain the next row returned by the query. If the query does not return any rows, the statements within the loop will not execute.

The iterator variable must either be of type `RECORD` or of a `%ROWTYPE` that matches the structure of a row returned by the query. Even if the query returns a single column, the iterator must be a `RECORD` or a `%ROWTYPE`.

Here is a code snippet that shows the `FOR` statement:

```
1 DECLARE
2   rental   rentals%ROWTYPE;
3 BEGIN
4
5   FOR rental IN SELECT * FROM rentals ORDER BY rental_date LOOP
6     IF( rental_is_overdue( rental )) THEN
7       PERFORM process_late_rental( rental );
8     END IF;
9   END LOOP;
10
11 END;
```

A `%ROWTYPE` iterator is fine if the query returns an entire row. If you need to retrieve a partial row, or you want to retrieve the result of a computation, declare the iterator variable as a `RECORD`. Here is an example:

```
1 DECLARE
2   my_record RECORD;
3 BEGIN
4
5   FOR my_record IN
6     SELECT tape_id, compute_due_date(rentals) AS due_date FROM rentals
7   LOOP
8     PERFORM
9       check_for_late_rental( my_record.tape_id, my_record.due_date );
10  END LOOP;
11
12 END;
```

A `RECORD` variable does not have a fixed structure. The fields in a `RECORD` variable are determined at the time that a row is assigned. In the previous example, you assign a row returned by the `SELECT` to the `my_record` `RECORD`. Because the query returns two columns, `my_record` will contain two fields: `tape_id` and `due_date`. A `RECORD` variable can change its shape. If you used the `my_record` variable as the iterator in a second `FOR-IN-SELECT` loop in this function, the field names within the `RECORD` would change. For example:

```
1 DECLARE
2   my_record RECORD;
3 BEGIN
4
5   FOR my_record IN SELECT * FROM rentals LOOP
6     -- my_record now holds a row from the rentals table
7     -- I can access my_record.tape_id, my_record.rental_date, etc.
8   END LOOP;
9
10  FOR my_record IN SELECT * FROM tapes LOOP
11    -- my_record now holds a row from the tapes table
12    -- I can now access my_record.tape_id, my_record.title, etc.
13  END LOOP;
14
15 END;
```

You also can process the results of a dynamic query (that is, a query not known at the time you write the function) in a `FOR` loop. To execute a dynamic query in a `FOR` loop, the syntax is a bit different:

```
[<<label>>]
FOR iterator IN EXECUTE query-string LOOP
  statements
END LOOP;
```

Notice that this is nearly identical to a `FOR-IN` loop. The `EXECUTE` keyword tells PL/pgSQL that the following string may change each time the statement is executed. The query-string can be an arbitrarily complex expression that evaluates to a string value; of course, it must evaluate to a valid `SELECT` statement. The following function shows the `FOR-IN-EXECUTE` loop:

```
1 CREATE OR REPLACE FUNCTION my_count( VARCHAR ) RETURNS INTEGER AS '
2   DECLARE
3     query           ALIAS FOR $1;
4     count           INTEGER := 0;
5     my_record       RECORD;
6   BEGIN
7     FOR my_record IN EXECUTE query LOOP
8       count := count + 1;
9     END LOOP;
10    RETURN count;
11  END;
12 ' LANGUAGE 'plpgsql';
```

EXIT

An `EXIT` statement (without any operands) terminates the enclosing block, and execution continues at the statement following the end of the block.

The full syntax for the `EXIT` statement is

```
EXIT [label] [WHEN boolean-expression];
```

All the `EXIT` statements that you have seen in this chapter have been simple `EXIT` statements. A simple `EXIT` statement unconditionally terminates the most closely nested block.

If you include `WHEN boolean-expression` in an `EXIT` statement, the `EXIT` becomes conditional—the `EXIT` occurs only if `boolean-expression` evaluates to `TRUE`. For example:

```
1 FOR i IN 1 .. 12 LOOP
2   balance := customer.customer_balances[i];
3   EXIT WHEN ( balance = 0 );
4   PERFORM check_balance( customer, balance );
5 END LOOP;
6
7 RETURN( 0 );
```

When execution reaches line 3, the `WHEN` expression is evaluated. If the expression evaluates to `TRUE`, the loop will be terminated and execution will continue at line 7.

This statement should really be named `EXIT...IF`. The `EXIT...WHEN` expression is not evaluated after each statement, as the name might imply.

Labels—EXIT Targets and Name Qualifiers

Now let's turn our attention to the subject of labels. A label is simply a string of the form

```
<<label>>
```

You can include a label prior to any of the following:

- A `DECLARE` section
- A `LOOP`
- A `WHILE` loop
- An integer `FOR` loop
- A `FOR...SELECT` loop

A label can perform two distinct functions. First, a label can be referenced in an `EXIT` statement. For example:

```
1 <<row_loop>>
2 FOR row IN 0 .. 99 LOOP
3
4   <<column_loop>>
5   FOR col in 0 .. 99 LOOP
6
7     IF( process( row, col ) = FALSE ) THEN
8       EXIT row_loop;
9     END IF;
10
11   END LOOP;
12
13 END LOOP;
14
15 RETURN( 0 );
```

Normally, an `EXIT` statement terminates the most closely nested block (or loop). When you refer to a label in an `EXIT` statement, you can terminate more than one nested block. When PL/pgSQL executes the `EXIT` statement at line 8, it will terminate the `<<column_loop>>` block and the `<<row_loop>>` block. You can't `EXIT` a block unless it is active: In other words, you can't `EXIT` a block that has already ended or that has not yet begun.

The second use for a label has to do with variable scoping. Remember that an `integer-FOR` loop creates a new copy of the iterator variable. If you have already declared the iterator variable outside of the loop, you can't directly access it within the loop. Consider the following example:

```
1 <<func>>
2 DECLARE
3   month_num INTEGER := 6;
4 BEGIN
5   FOR month_num IN 1 .. 12 LOOP
```

```

6     PERFORM compute_monthly_info( month_num );
7 END LOOP;
8 END;

```

Line 2 declares a variable named `month_num`. When execution reaches line 4, PL/pgSQL will create a second variable named `month_num` (and this variable will vary between 1 and 12). Within the scope of the new variable (between lines 4 and 6), any reference to `month_num` will refer to the new variable created at line 4. If you want to refer to the outer variable, you can qualify the name as `func.month_num`. In general terms, you can refer to any variable in a fully qualified form. If you omit the label qualifier, a variable reference refers to the variable with the shortest lifetime (that is, the most recently created variable).

RETURN

Every PL/pgSQL function must terminate with a `RETURN` statement. There are two forms for the `RETURN` statement:

```

RETURN expression;
RETURN;

```

Use the first form when you're writing a PL/pgSQL function that returns a simple value and the second form when you're writing a function returns a `SETOF` values. If your function returns a `SETOF` values, you'll use the `RETURN NEXT` statement (described in the next section) to build up a result set as you go.

When a `RETURN` statement executes, four things happen:

1. The expression (if any) is evaluated and, if necessary, coerced into the appropriate data type. The `RETURN` type of a function is declared when you create the function. In the example "`CREATE FUNCTION func() RETURNS INTEGER ...`", the `RETURN` type is declared to be an `INTEGER`. If the `RETURN` expression does not evaluate to the declared `RETURN` type, PL/pgSQL will try to convert it to the required type. If you are writing a function that returns a `SETOF` values, you should omit the expression.
2. The current function terminates. When a function terminates, all code blocks within that function terminate, and all variables declared within that function are destroyed.
3. The return value (obtained by evaluating expression or executing some number of `RETURN NEXT` statements) is returned to the caller. If the caller assigns the return value to a variable, the assignment completes. If the caller uses the return value in an expression, the caller uses the return value to evaluate the expression. If the function was called by a `PERFORM` statement, the return value is discarded.
4. The point of execution returns to the caller.

If you fail to execute a `RETURN` statement, you will receive an error (control reaches end of function without `RETURN`). You can include many `RETURN` statements in a function, but only one will execute: whichever `RETURN` statement is reached first.

RETURN NEXT

If you've defined a function that returns a `SETOF` values, you don't use the `RETURN` statement to give a value to the caller. Instead, you execute a series of zero or more `RETURN NEXT` statements. The syntax for a `RETURN NEXT` statement is

```

RETURN NEXT expression;

```

Each time you execute a `RETURN NEXT` statement, PL/pgSQL evaluates the expression and adds the result to the function's result set. If you are returning a `SETOF` rows, expression must evaluate to a row value. If you are returning a `SETOF` arrays, each expression must evaluate to an array (of the proper type). If you are returning a `SETOF` simple values, each expression must evaluate to a simple value of the appropriate type. If you are returning a `SETOF` anyarray or anyelement, see the discussion of polymorphic functions later in this chapter.

When you have finished building the result set, simply `RETURN` from the function. The following example defines a function that returns the monthly balances for a given customer in the form of a `SETOF NUMERIC` values:

Code View: [Scroll](#) / [Show All](#)

```

CREATE OR REPLACE FUNCTION getBalances( id INTEGER ) RETURNS SETOF NUMERIC AS $$
DECLARE
    customer    customers%ROWTYPE;
BEGIN

    SELECT * FROM customers INTO customer WHERE customer_id = id;

    FOR month IN 1..12 LOOP

        IF customer.monthly_balances[month] IS NOT NULL THEN
            RETURN NEXT customer.monthly_balances[month];
        END IF;

    END LOOP;

    RETURN;

END;
$$ LANGUAGE 'plpgsql';

```

Notice that this function will execute the `RETURN NEXT` statement anywhere from 0 to 12 times—that means that the result set built by this function may contain anywhere from 0 to 12 rows. If you don't execute a `RETURN NEXT` statement, the result set built by the function will be empty.

A function that returns a `SETOF` values acts like a table. That means that a `SETOF` function is typically written to the right of the `FROM` in a `SELECT` command. For example, to call the `getBalances()` function you just saw, you would write a query such as the following. (Note: These queries won't work for you unless you've added a `monthly_balances` array to the `customers` table):

```
movies=# SELECT customer_id, customer_name, balance, monthly_balances
movies=# FROM customers;
customer_id | customer_name | balance | monthly_balances
-----+-----+-----+-----
          1 | Jones, Henry |    0.00 |
          4 | Wonderland, Alice N. |    3.00 |
          2 | Rubin, William |   15.00 |
          3 | Panky, Henry |    0.00 | {5.00,52.20}
(4 rows)

movies=# SELECT * FROM getBalances( 3 );
getbalances
-----
          5.00
         52.20
(2 rows)

movies=# SELECT * FROM getBalances( 2 );
getbalances
-----
(0 rows)
```

Notice that the first call to `getBalances(3)` returned two rows because there are two entries in the `monthly_balances` column for customer number 3. The second call returned zero rows.

PERFORM

A function written in PL/pgSQL can contain SQL commands intermingled with PL/pgSQL-specific statements. Remember, a SQL command is something like `CREATE TABLE`, `INSERT`, `UPDATE`, and so on; whereas PL/pgSQL adds procedural statements such as `IF`, `RETURN`, or `WHILE`. If you want to create a new table within a PL/pgSQL function, you can just include a `CREATE TABLE` command in the code:

```
CREATE FUNCTION process_month_end( ) RETURNS BOOLEAN AS '
BEGIN
    ...
    CREATE TABLE temp_data ( ... );
    ...
    DROP TABLE temp_data;
    ...
END;
' LANGUAGE 'plpgsql';
```

You can include almost any SQL command just by writing the command inline. The exception is the `SELECT` command. A `SELECT` command retrieves data from the server. If you want to execute a `SELECT` command in a PL/pgSQL function, you normally provide variables to hold the results:

```
DECLARE
    Customer    customers%ROWTYPE;
BEGIN
    ...
    SELECT INTO customer * FROM customers WHERE( customer_id = 1 );
    --
    -- The customer variable will now hold the results of the query
    --
    ...
END;
```

On rare occasions, you may need to execute a `SELECT` statement, but you want to ignore the data returned by the query. Most likely, the `SELECT` statement that you want to execute will have some side effect, such as executing a function. You can use the `PERFORM` statement to execute an arbitrary `SELECT` command without using the results. For example:

```
...
PERFORM SELECT my_function( rentals ) FROM rentals;
...
```

You can also use `PERFORM` to evaluate an arbitrary expression, again discarding the results:

```
...
PERFORM record_timestamp( timeofday() );
...
```

EXECUTE

The `EXECUTE` statement is similar to the `PERFORM` statement. Although the `PERFORM` statement evaluates a SQL expression and discards the

results, the `EXECUTE` statement executes a dynamic SQL command, and then discards the results. The difference is subtle but important. When the PL/pgSQL processor compiles a `PERFORM expression` statement, the query plan required to evaluate the `expression` is generated and stored along with the function. This means that `expression` must be known at the time you write your function. The `EXECUTE` statement, on the other hand, executes a SQL statement that is not known at the time you write your function. You may, for example, construct the text of a SQL statement within your function, or you might accept a string value from the caller and then execute that string.

Here is a function that uses the `EXECUTE` command to time the execution of a SQL command:

```
1 CREATE FUNCTION time_command( VARCHAR ) RETURNS INTERVAL AS '
2 DECLARE
3     beg_time  TIMESTAMP;
4     end_time  TIMESTAMP;
5 BEGIN
6
7     beg_time := timeofday( );
8     EXECUTE $1;
9     end_time := timeofday( );
10
11     RETURN( end_time - beg_time );
12 END;
13 ' LANGUAGE 'plpgsql';
```

You would call the `time_command()` function like this:

```
movies=# SELECT time_command( 'SELECT * FROM rentals' );
time_command
-----
00:00:00.82
(1 row)
```

With the `EXECUTE` statement, you can execute any SQL command (including calls to PL/pgSQL functions) and the results will be discarded, except for the side effects.

GET DIAGNOSTICS

PL/pgSQL provides a catch-all statement that gives you access to various pieces of result information: `GET DIAGNOSTICS`. Using `GET DIAGNOSTICS`, you can retrieve a count of the rows affected by the most recent `UPDATE` or `DELETE` command and the `object-ID` of the most recently inserted row. The syntax for the `GET DIAGNOSTICS` statement is

```
GET DIAGNOSTICS variable = [ROW_COUNT|RESULT_OID], ...;
```

`ROW_COUNT` is meaningless until you have executed an `UPDATE` or `DELETE` command. Likewise, `RESULT_OID` is meaningless until you execute an `INSERT` command.

Error Handling

PostgreSQL version 8.0 introduced a new error-handling scheme to PL/pgSQL. Prior to version 8.0, any error that occurred during a PL/pgSQL function would abort the function and the transaction that called the function. Beginning with version 8.0, you can intercept error conditions (PL/pgSQL calls them exceptions) and handle them gracefully.

To trap an exception, include an `EXCEPTION` section just before the `END` of a block. The syntax for an `EXCEPTION` section is

```
EXCEPTION
    WHEN condition [OR condition...] THEN
        statements
    [ WHEN condition [OR condition...] THEN
        statements
    ...
    ]
```

The `condition` is derived from the error descriptions listed in Appendix A of the PostgreSQL reference documentation. Table 7.1 shows an excerpt from Appendix A. To convert one of these errors into a `condition`, just find the error code that you want to trap and write the error description, replacing each space with an underscore.

Table 7.1. Sample PostgreSQL Error Codes

Error Code	Description
Class 08	Connection Exception
08000	CONNECTION EXCEPTION
08003	CONNECTION DOES NOT EXIST
08006	CONNECTION FAILURE
08001	SQLCLIENT UNABLE TO ESTABLISH SQLCONNECTION
08004	SQLSERVER REJECTED ESTABLISHMENT OF SQLCONNECTION

For example, to trap error 08006, you would write an `EXCEPTION` section like this:

```
BEGIN
...
EXCEPTION
    WHEN connection_failure THEN
        RAISE ERROR 'Connection To Server Lost';
END;
```

If any of the statements between `BEGIN` and `EXCEPTION` throws a `connection_failure` error, PL/pgSQL immediately jumps to the first statement in the exception handler (in this case, the `RAISE ERROR` statement), bypassing the rest of the statements in the block.

You can't trap every condition listed in Appendix A; in particular, you can't trap `successful_completion`, any of the conditions listed in the `WARNING` category, or any of the conditions listed in the `NO DATA` category.

You can trap a whole category of error conditions by writing an `EXCEPTION` handler for that category. You can distinguish between errors and categories by looking at the last digit of the error code. If the last digit is a 0, you're looking at a category. To trap any of the errors in the `connection_exception` class, just write an `EXCEPTION` section like this:

```
BEGIN
...
EXCEPTION
    WHEN connection_exception THEN
        RAISE ERROR 'Something went wrong with the server connection';
END;
```

That sequence is equivalent to:

```
BEGIN
...
EXCEPTION
    WHEN
        connection_does_not_exist OR
        connection_failure OR
        sql_client_unable_to_establish_sql_connection OR
        sql_server_rejected_establishment_of_sql_connection OR
        transaction_resolution_unknown OR
        protocol_violation
    THEN
        RAISE ERROR 'Something went wrong with the server connection';
END;
```

PL/pgSQL defines a catch-all condition, named `others`, that you can use to trap any exceptions not trapped by another handler.

A single exception may match multiple exception handlers. For example, consider the following `EXCEPTION` section:

```
BEGIN
...
EXCEPTION
    WHEN connection_failure THEN
        RAISE ERROR 'Connection Lost';
    WHEN connection_exception THEN
        RAISE ERROR 'Something went wrong with the server connection';
    WHEN others THEN
        RAISE ERROR 'Something broke';
END;
```

If a `connection_failure` occurs, all three handlers match the exception: The `connection_failure` handler matches exactly; the `connection_exception` handler matches because a `connection_failure` is a member of the `connection_exception` category; and the `others` handler matches because `others` will match any exception. Which handler executes? The first one that matches. That means that you should always list the handlers from most-specific to most-general. If you were to write the `others` handler first, the `connection_failure` and `connection_exception` handlers could never execute.

Remember that you can nest blocks within a single PL/pgSQL function. Each block can have its own `EXCEPTION` section. When an exception occurs, PL/pgSQL searches through the currently active blocks to find a handler for that exception. If the first (most deeply nested) block hasn't defined a handler for the exception, PL/pgSQL aborts the first block and looks at the surrounding block. If that block hasn't defined a handler for the exception, PL/pgSQL aborts the second block as well and continues to the next block. If PL/pgSQL can't find a handler, it aborts the entire function and reports the exception to the caller of the function.

When a PL/pgSQL function enters a block that includes an `EXCEPTION` section, it creates a "subtransaction" by executing the internal equivalent of a `SAVEPOINT` command. If you have one block nested within another (and each block defines exception handlers), you have two subtransactions, one nested within the other. If an exception occurs, PL/pgSQL rolls back nested subtransactions as it searches for an exception handler. When PL/pgSQL finds an exception handler, it executes the handler and rolls back that subtransaction as well. Consider the following code snippet:

Code View: [Scroll](#) / [Show All](#)


```

...
FOR tape IN SELECT * FROM tapes LOOP
  BEGIN

    update_tape( tape );

    FOR rental IN SELECT * FROM rentals WHERE rentals.tape_id = tape.tape_id LOOP

      BEGIN

        update_rental_1( rental );
        update_rental_2( rental );

      EXCEPTION
        WHEN insufficient_privilege THEN
          RAISE NOTICE 'Privilege denied';
      END;

    END LOOP;
  EXCEPTION
    WHEN others THEN
      RAISE NOTICE 'Unable to process all tapes';
  END;
END LOOP;
...

```

This snippet contains two loops, one nested within the other. The outer loop reads through the `tapes` table and, for each tape, calls a function named `update_tape()` (presumably another PL/pgSQL function). The inner loop reads each `rentals` record for the current tape and calls two functions with each `rental`.

Every time the PL/pgSQL interpreter executes the first `BEGIN` statement, it creates a new subtransaction which we'll call `Touter`. Likewise, every time PL/pgSQL executes the second `BEGIN` statement, it creates a new subtransaction, `Tinner`, nested within `Touter`. Now consider what happens when an exception occurs.

If the `update_rental_2()` function throws an exception, PL/pgSQL aborts `Tinner` (rolling back any changes made by `update_rental_1()` and `update_rental_2()`) and then searches for a handler that matches the exception. If `update_rental_2()` throws an `insufficient_privilege` exception, PL/pgSQL finds the inner-most exception handler, jumps to the first `RAISE NOTICE` statement, and then moves on to the statement following the inner-most block. If `update_rental_2()` throws any other exception, PL/pgSQL ignores the inner-most exception handler (because it doesn't match the exception), aborts `Touter` (rolling back any changes made by `update_tape()`, `update_rental_1()`, and `update_rental_2()`), jumps to the second `RAISE NOTICE` statement, and moves on to the statement following the outer-most block.

If the `update_tape()` function throws an exception, PL/pgSQL aborts `Touter` (rolling back any changes made by `update_tape()`), jumps to the second `RAISE NOTICE` statement, and then moves on to the statement following the outer-most block.

Notice that an exception always aborts the inner-most subtransaction. PL/pgSQL will continue aborting nested subtransactions until it finds a handler for the exception. If no handler is found, the entire transaction is aborted. By using nested subtransactions (and nested exception handlers) in this way, the inner subtransaction contains all of the updates for a single `rental`. If the inner subtransaction aborts, only those changes made to the current `rental` are rolled back. The outer subtransaction contains all of the updates for a single `tape` (including all of the updates for all `rentals` of that `tape`). If you abort the outer subtransaction all changes made to the `tape` are rolled back and all changes made to the `rentals` of that `tape` are rolled back as well.

RAISE

Even though PL/pgSQL doesn't offer a way to intercept errors, it does provide a way to generate an error: the `RAISE` statement. Exceptions are usually generated when an error occurs while executing an SQL (or PL/pgSQL) statement, but you can explicitly raise an exception using the `RAISE` statement. The syntax for a `RAISE` statement is

```
RAISE severity 'message' [, variable [...]];
```

The `severity` determines how far the error message will go and whether the error should abort the current transaction.

Valid values for `severity` are

- **DEBUG**— The message is written to the server's log file and otherwise ignored. The function runs to completion, and the current transaction is not affected.
- **NOTICE**— The message is written to the server's log file and sent to the client application. The function runs to completion, and the current transaction is not affected.
- **EXCEPTION**— The message is written to the server's log file and PL/pgSQL throws a `raise_exception` exception that you can trap with an `EXCEPTION` handler as described in the previous section.

The `message` string must be a literal value—you can't use a PL/pgSQL variable in this slot, and you cannot include a more complex expression. If you need to include variable information in the error message, you can sneak it into the message by including a `%` character wherever you want the variable value to appear. For example:

```
rentals.tape_id := 'AH-54706';
RAISE DEBUG 'tape_id = %', rentals.tape_id;
```

When these statements are executed, the message `tape_id = AH-54706` will be written to the server's log file. For each (single) `%` character in the message string, you must include a variable. If you want to include a literal percent character in the message, write it as `%%`. For example:

```
percentage := 20;
RAISE NOTICE 'Top (%)%%', percentage;
```

translates to `Top (20)%`.

The `RAISE` statement is useful for debugging your PL/pgSQL code; it's even better for debugging someone else's code. I find that the `DEBUG` severity is perfect for leaving evidence in the server log. When you ship a PL/pgSQL function to your users, you might want to leave a few `RAISE DEBUG` statements in your code. This can certainly make it easier to track down an elusive bug (remember, users never write down error messages, so you might as well arrange for the messages to appear in a log file). I use the `RAISE NOTICE` statement for interactive debugging. When I am first building a new PL/pgSQL function, the chances are very slim that I'll get it right the first time. (Funny, it doesn't seem to matter how trivial or complex the function is.) I start out by littering my code with `RAISE NOTICE` statements; I'll usually print the value of each function parameter as well as key information from each record that I `SELECT`. As it becomes clearer that my code is working, I'll either remove or comment out (using `--`) the `RAISE NOTICE` statements. Before I send out my code to a victim, er, user, I'll find strategic places where I can leave `RAISE DEBUG` statements. The `RAISE DEBUG` statement is perfect for reporting things that should never happen. For example, because of the referential integrity that I built into the `tapes`, `customers`, and `rentals` tables, I should never find a `rentals` record that refers to a nonexistent customer. I'll check for that condition (a missing customer) and report the error with a `RAISE DEBUG` statement. Of course, in some circumstances, a missing customer should really trigger a `RAISE EXCEPTION`—if I just happen to notice the problem in passing and it really doesn't affect the current function, I'll just note it with a `RAISE DEBUG`. So, the rule I follow is: if the condition prevents further processing, I `RAISE an EXCEPTION`; if the condition should never happen, I `RAISE a DEBUG message`; if I am still developing my code, I `RAISE a NOTICE`.

Cursors

Direct cursor support is new in PL/pgSQL version 7.2. Processing a result set using a cursor is similar to processing a result set using a `FOR` loop, but cursors offer a few distinct advantages that you'll see in a moment.

You can think of a cursor as a name for a result set. You must declare a cursor variable just as you declare any other variable. The following code snippet shows how you might declare a cursor variable:

```
...
DECLARE
    rental_cursor      CURSOR FOR SELECT * FROM rentals;
...
```

`rental_cursor` is declared to be a cursor for the result set of the query `SELECT * FROM rentals`. When you declare a variable of type `CURSOR`, you must include a query. The cursor variable is said to be bound to this query, and the variable is a bound cursor variable.

Before you can use a bound cursor, you must open the cursor using the `OPEN` statement:

```
...
DECLARE
    rental_cursor      CURSOR FOR SELECT * FROM rentals;
BEGIN

    OPEN rental_cursor;

...
```

If you try to `OPEN` a cursor that is already open, you will receive an error message (cursor "name" already in use). If you try to `FETCH` (see the section that follows) from a cursor that has not been opened, you'll receive an error message (cursor "name" is invalid). When you use a cursor, you first `DECLARE` it, then `OPEN` it, `FETCH` from it, and finally `CLOSE` it, in that order. You can repeat the `OPEN`, `FETCH`, `CLOSE` cycle if you want to process the cursor results again.

FETCH

After a bound cursor has been opened, you can retrieve the result set (one row at a time) using the `FETCH` statement. When you fetch a row from a cursor, you have to provide one or more destination variables that PL/pgSQL can stuff the results into. The syntax for the `FETCH` statement is

```
FETCH cursor-name INTO destination [ , destination [...]];
```

The destination (or destinations) must match the shape of a row returned by the cursor. For example, if the cursor `SELECTS` a row from the `rentals` table, there are three possible destinations:

- A variable of type `rentals%ROWTYPE`
- Three variables: one of type `rentals.tape_id%TYPE`, one of type `rentals.customer_id%TYPE`, and the last of type `rentals.rental_date%TYPE`
- A variable of type `RECORD`

Let's look at each of these destination types in more detail.

When you `FETCH` into a variable of some `%ROWTYPE`, you can refer to the individual columns using the usual `variable.column` notation. For example:

```
...
DECLARE
    rental_cursor      CURSOR FOR SELECT * FROM rentals;
    rental              rentals%ROWTYPE;
BEGIN

    OPEN rental_cursor;

    FETCH rental_cursor INTO rental;
    --
    -- I can now access rental.tape_id,
    -- rental.customer_id, and rental.rental_date
    --
    IF ( overdue( rental.rental_date ) ) THEN
        ...
    
```

Next, I can `FETCH` into a comma-separated list of variables. In the previous example, the `rental_cursor` cursor will return rows that each contain three columns. Rather than fetching into a `%ROWTYPE` variable, I can declare three separate variables (of the appropriate types) and `FETCH` into those instead:

```
...
DECLARE
```

```

rental_cursor    CURSOR FOR SELECT * FROM rentals;
tape_id          rentals.tape_id%TYPE;
customer_id      rentals.customer_id%TYPE;
rental_date      rentals.rental_date%TYPE;
BEGIN

    OPEN rental_cursor;

    FETCH rental_cursor INTO tape_id, customer_id, rental_date;

    IF ( overdue( rental_date ) ) THEN
        ...

```

You are not required to use variables declared with %TYPE, but this is the perfect place to do so. At the time you create a function, you usually know which columns you will be interested in, and declaring variables with %TYPE will make your functions much less fragile in cases where the referenced column types might change.

You cannot combine composite variables and scalar variables in the same `FETCH` statement^[5]:

^[5] This seems like a bug to me. You may be able to combine composite and scalar variables in a future release.

```

...
DECLARE
rental_cursor CURSOR FOR SELECT *, now() - rental_date FROM rentals;
rental        rentals%ROWTYPE;
elapsed       INTERVAL;
BEGIN

    OPEN rental_cursor;

    FETCH rental_cursor INTO rental, elapsed; -- WRONG! Can't combine
                                           -- composite and scalar
                                           -- variables in the same
                                           -- FETCH

    IF ( overdue( rental.rental_date ) ) THEN
        ...

```

The third type of destination that you can use with a `FETCH` statement is a variable of type `RECORD`. You may recall from earlier in this chapter that a `RECORD` variable is something of a chameleon—it adjusts to whatever kind of data that you put into it. For example, the following snippet uses the same `RECORD` variable to hold two differently shaped rows:

```

...
DECLARE
rental_cursor CURSOR FOR SELECT * FROM rentals;
customer_cursor CURSOR FOR SELECT * FROM customers;
my_data       RECORD;
BEGIN
    OPEN rental_cursor;
    OPEN customer_cursor;

    FETCH rental_cursor INTO my_data;
    -- I can now refer to:
    --   my_data.tape_id
    --   my_data.customer_id
    --   my_data.rental_date

    FETCH customer_cursor INTO my_data;
    -- Now I can refer to:
    --   my_data.customer_id
    --   my_data.customer_name
    --   my_data.phone
    --   my_data.birth_date
    --   my_data.balance
...

```

After you have executed a `FETCH` statement, how do you know whether a row was actually retrieved? If you `FETCH` after retrieving the entire result, no error occurs. Instead, each PL/pgSQL function has an automatically declared variable named `FOUND`. `FOUND` is a `BOOLEAN` variable that is set by the PL/pgSQL interpreter to indicate various kinds of state information. Table 7.2 lists the points in time where PL/pgSQL sets the `FOUND` variable and the corresponding values.

Table 7.2. `FOUND` Events and Values

Event	Value
Start of each function	FALSE
Start of an integer—FOR loop	FALSE
Within an integer—FOR loop	TRUE
Start of a FOR...SELECT loop	FALSE
Within a FOR...SELECT loop	TRUE
Before SELECT INTO statement	FALSE

After <code>SELECT INTO</code> statement	TRUE (if rows are returned)
Before <code>FETCH</code> statement	FALSE
After <code>FETCH</code> statement	TRUE (if a row is returned)

So, you can see that `FOUND` is set to `TRUE` if a `FETCH` statement returns a row. Let's see how to put all the cursor related statements together into a single PL/pgSQL function:

```
...
DECLARE
    next_rental    CURSOR FOR SELECT * FROM rentals;
    rental         rentals%ROWTYPE;
BEGIN
    OPEN next_rental;

    LOOP
        FETCH next_rental INTO rental;
        EXIT WHEN NOT FOUND;
        PERFORM process_rental( rental );
    END LOOP;

    CLOSE next_rental;
END;
...
```

The first thing you do in this code snippet is `OPEN` the cursor. Next, you enter a `LOOP` that will process every row returned from the cursor. Inside of the `LOOP`, you `FETCH` a single record, `EXIT` the loop if the cursor is exhausted, and call another function (`process_rental()`) if not. After the loop terminates, close the cursor using the `CLOSE` statement.

So far, it looks like a cursor loop is pretty much the same as a `FOR-IN-SELECT` loop. What else can you do with a cursor?

Parameterized Cursors

You've seen that you must provide a `SELECT` statement when you declare a `CURSOR`. Quite often, you'll find that you don't know the exact values involved in the query at the time you're writing a function. You can declare a parameterized cursor to solve this problem.

A parameterized cursor is similar in concept to a parameterized function. When you define a function, you can declare a set of parameters (these are called the formal parameters, or formal arguments); those parameters can be used within the function to change the results of the function. If you define a function without parameters, the function will always return the same results (unless influenced by global, external data). Each language imposes restrictions on where you can use a parameter within a function. In general, function parameters can be used anywhere that a value-yielding expression can be used. When you make a call to a parameterized function, you provide a value for each parameter: The values that you provide (these are called the actual parameters, or actual arguments) are substituted inside of the function wherever the formal parameters appear.

When you define a cursor, you can declare a set of formal parameters; those parameters can be used with the cursor to change the result set of the query. If you define a cursor without parameters, the query will always return the same result set, unless influenced by external data. PL/pgSQL restricts the places that you can use a parameter within a cursor definition. A cursor parameter can be used anywhere that a value-yielding expression can be used. When you open a cursor, you must specify values for each formal parameter. The actual parameters are substituted inside of the cursor wherever the formal parameters appear.

Let's look at an example:

```
1 ...
2 DECLARE
3     next_customer    CURSOR (ID INTEGER) FOR
4         SELECT * FROM customers WHERE
5             customer_id = ID;
6     customer         customers%ROWTYPE;
7     target_customer  ALIAS FOR $1;
8 BEGIN
9
10    OPEN next_customer( target_customer );
11 ...
```

Lines 3, 4, and 5 declare a parameterized cursor. This cursor has a single formal parameter; an `INTEGER` named `ID`. Notice (at the end of line 5), that I have used the formal parameter within the cursor definition. When I open this cursor, I'll provide an `INTEGER` value for the `ID` parameter. The actual parameter that I provide will be substituted into the query wherever the formal parameter is used. So, if `target_customer` is equal to, say, 42, the cursor opened at line 10 will read:

```
SELECT * FROM customers WHERE customer_id = 42;
```

The full syntax for a cursor declaration is

```
variable-name CURSOR
[ (param-name param-type [, param-name param-type ...] ) ]
FOR select-query;
```

The full syntax for an OPEN statement is

```
OPEN cursor-name [ ( actual-param-value [, actual-param-value...] ) ];
```

You would parameterize a cursor for the same reasons that you would parameterize a function: you want the results to depend on the actual arguments. When you parameterize a cursor, you are also making the cursor more reusable. For example, I might want to process all the rentals in my inventory, but I want to process the rentals one customer at a time. If I don't use a parameterized cursor, I have to declare one cursor for each of my customers (and I have to know the set of customers at the time I write the function). Using a parameterized cursor, I can declare the cursor once and provide different actual arguments each time I open the cursor:

Code View: [Scroll](#) / Show All

```
1 CREATE OR REPLACE FUNCTION process_rentals_by_customer( ) RETURNS void AS $$
2   DECLARE
3     next_customer    CURSOR FOR SELECT * FROM customers;
4     next_rental      CURSOR( ID integer ) FOR
5                       SELECT * FROM rentals WHERE customer_id = ID;
6     customer         customers%ROWTYPE;
7     rental           rentals%ROWTYPE;
8   BEGIN
9
10    OPEN next_customer;
11
12    LOOP
13      FETCH next_customer INTO customer;
14      EXIT WHEN NOT FOUND;
15
16      OPEN next_rental( customer.customer_id );
17
18      LOOP
19        FETCH next_rental INTO rental;
20        EXIT WHEN NOT FOUND;
21
22        PERFORM process_rental( customer, rental );
23
24      END LOOP;
25
26      CLOSE next_rental;
27    END LOOP;
28
29    CLOSE next_customer;
30
31    RETURN;
32
33  END;
34
35 $$ LANGUAGE 'plpgsql';
```

Notice that you can OPEN and CLOSE a cursor as often as you like. A cursor must be closed before it can be opened. Each time you open a parameterized cursor, you can provide new actual parameters.

Cursor References

Now, let's turn our attention to another aspect of cursor support in PL/pgSQL—cursor references.

When you declare a CURSOR variable, you provide a SELECT statement that is bound to the cursor. You can't change the text of the query after the cursor has been declared. Of course, you can parameterize the query to change the results, but the shape of the query remains the same: If the query returns rows from the tapes table, it will always return rows from the tapes table.

Instead of declaring a CURSOR, you can declare a variable to be of type REFCURSOR. A REFCURSOR is not actually a cursor, but a reference to a cursor. The syntax for declaring a REFCURSOR is

```
DECLARE
  ref-name REFCURSOR;
  ...
```

Notice that you do not specify a query when creating a REFCURSOR. Instead, a cursor is bound to a REFCURSOR at runtime. Here is a simple example:

```
1 ...
2 DECLARE
3   next_rental CURSOR FOR SELECT * FROM rentals;
4   next_tape   CURSOR FOR SELECT * FROM tapes;
5   rental      rentals%ROWTYPE;
6   tape        tape%ROWTYPE;
7   next_row    REFCURSOR;
8 BEGIN
9   OPEN next_rental;
10  next_row := next_rental;
11  FETCH next_rental INTO rental;
12  FETCH next_row INTO rental;
```

```

13  CLOSE next_rental;
14
15  next_row := next_tape;
16  OPEN next_tape;
17  FETCH next_row INTO tape;
18  CLOSE next_row;
19  ...

```

In this block, I've declared two cursors and one cursor reference. One of the cursors returns rows from the `rentals` table, and the other returns rows from the `tapes` table.

At line 9, the `next_rental` cursor opens. At line 10, I give a value to the `next_row` cursor reference. We now have two ways to access the `next_rental` cursor: through the `next_rental` cursor variable and through the `next_row` cursor reference. At this point, `next_row` refers to the `next_rental` cursor. You can see (at lines 11 and 12) that you can `FETCH` a row using either variable. Both `FETCH` statements return a row from the `rentals` table.

At line 14, the `next_row` cursor reference points to a different cursor. Now, when you `FETCH` from `next_row`, you'll get a row from the `tapes` table. Notice that you can point `next_row` to a cursor that has not yet been opened. You can `CLOSE` a cursor using a cursor reference, but you can't `OPEN` a cursor using a cursor reference.

Actually, you can open a cursor using a `REFCURSOR`; you just can't open a named cursor. When you declare a `CURSOR` variable, you are really creating a PostgreSQL cursor whose name is the same as the name of the variable. In the previous example, you created one cursor (not just a cursor variable) named `next_rental` and a cursor named `next_tape`. PL/pgSQL allows you to create anonymous cursors using `REFCURSOR` variables. An anonymous cursor is a cursor that doesn't have a name^[6]. You create an anonymous cursor using the `OPEN` statement, a `REFCURSOR`, and a `SELECT` statement:

^[6] An anonymous cursor does in fact have a name, but PostgreSQL constructs the name, and it isn't very reader-friendly. An anonymous cursor has a name such as `<unnamed cursor 42>`.

```

1  ...
2  DECLARE
3    next_row REFCURSOR;
4  BEGIN
5    OPEN next_row FOR SELECT * FROM customers;
6  ...

```

At line 5, you are creating an anonymous cursor and binding it to the `next_row` cursor reference. After an anonymous cursor has been opened, you can treat it like any other cursor. You can `FETCH` from it, `CLOSE` it, and lose it. That last part might sound a little fishy, so let me explain further. Take a close look at the following code fragment:

Code View: [Scroll](#) / [Show All](#)

```

1  CREATE FUNCTION leak_cursors( INTEGER ) RETURNS INTEGER AS '
2  DECLARE
3    next_customer CURSOR FOR SELECT * FROM customers;
4    next_rental   REFCURSOR;
5    customer      customers%ROWTYPE;
6    rental        rentals%ROWTYPE;
7    count         INTEGER := 0;
8  BEGIN
9
10   OPEN next_customer;
11
12   LOOP
13     FETCH next_customer INTO customer;
14     EXIT WHEN NOT FOUND;
15     OPEN next_rental FOR
16       SELECT * FROM rentals
17       WHERE rentals.customer_id = customer.customer_id;
18
19     LOOP
20       FETCH next_rental INTO rental;
21       EXIT WHEN NOT FOUND;
22
23       RAISE NOTICE 'customer_id = %, rental_date = %',
24         customer.customer_id, rental.rental_date;
25
26       count := count + 1;
27     END LOOP;
28
29     next_rental := NULL;
30
31   END LOOP;
32   CLOSE next_customer;
33   RETURN( count );
34 END;
35 ' LANGUAGE 'plpgsql';

```

This function contains two loops: an outer loop that reads through the `customers` table and an inner loop that reads each rental for a given customer. The `next_customer` cursor is opened (at line 10) before the outer loop begins. The `next_rental` cursor is bound and opened (at lines 15, 16, and 17) just before the inner loop begins. After the inner loop completes, I set the `next_rental` cursor reference to `NULL` and

continue with the outer loop. What happens to the cursor that was bound to `next_rental`? I didn't explicitly close the cursor, so it must remain open. After executing the assignment statement at line 29, I have no way to access the cursor again—remember, it's an anonymous cursor, so I can't refer to it by name. This situation is called a resource leak. A resource leak occurs when you create an object (in this case, a cursor) and then you lose all references to that object. If you can't find the object again, you can't free the resource. Avoid resource leaks; they're nasty and can cause performance problems. Resource leaks will also cause your code to fail if you run out of a resource (such as memory space). We can avoid the resource leak shown in this example by closing the `next_rental` before setting it to `NULL`.

You've seen what not to do with a cursor reference, but let's see what cursor references are really good for. The nice thing about a cursor reference is that you can pass the reference to another function, or you can return a reference to the caller. These are powerful features. By sharing cursor references between functions, you can factor your PL/pgSQL code into reusable pieces.

One of the more effective ways to use cursor references is to separate the code that processes a cursor from the code that creates the cursor. For example, you may find that we need a function to compute the total amount of money that we have received from a given customer over a given period of time. I might start by creating a single function that constructs a cursor and processes each row in that cursor:

```
...
OPEN next_rental FOR
SELECT * FROM rentals WHERE
    customer_id = $1 AND
    rental_date BETWEEN $2 AND $3;

LOOP
    FETCH next_rental INTO rental;
    -- accumulate rental values here
...

```

This is a good start, but it works only for a single set of conditions: a given customer and a given pair of dates. Instead, you can factor this one function into three separate functions.

The first function creates a cursor that, when opened, will return all `rentals` records for a given customer within a given period; the cursor is returned to the caller:

```
CREATE FUNCTION
select_rentals_by_customer_interval( INTEGER, DATE, DATE )
RETURNS REFCURSOR AS '
DECLARE
    next_rental REFCURSOR;
BEGIN
    OPEN next_rental FOR
    SELECT * FROM rentals WHERE
        customer_id = $1 AND
        rental_date BETWEEN $2 AND $3;
    RETURN( next_rental );
END;
' LANGUAGE 'plpgsql';

```

The second function, given a cursor that returns `rentals` records, computes the total value of the `rentals` accessible through that cursor:

```
CREATE FUNCTION
compute_rental_value( REFCURSOR )
RETURNS NUMERIC AS '
DECLARE
    total NUMERIC(7,2) := 0;
    rental rentals%ROWTYPE;
    next_rental ALIAS FOR $1;
BEGIN
    LOOP
        FETCH next_rental INTO rental;
        EXIT WHEN NOT FOUND;
        -- accumulate rental values here
        --
        -- pretend that this is a complex
        -- task which requires loads of amazingly
        -- clever code
        ...
    END LOOP;
    RETURN( total );
END;
' LANGUAGE 'plpgsql';

```

The last function invokes the first two:

```
CREATE FUNCTION
compute_value_by_customer_interval( INTEGER, DATE, DATE )
RETURNS NUMERIC AS '
DECLARE
    curs REFCURSOR;
    total NUMERIC(7,2);
BEGIN
    curs := select_rentals_by_customer_interval( $1, $2, $3 );
    total := compute_rental_value( curs );
    CLOSE curs;
    RETURN( total );
END;
' LANGUAGE 'plpgsql';

```


The advantage to this approach is that you can construct a cursor using different selection criteria and call `compute_total_value()`. For example, you might want to compute the total values of all `rentals` of a given tape:

```
CREATE FUNCTION compute_tape_value( VARCHAR )
RETURNS NUMERIC AS '
DECLARE
    curs REFCURSOR;
    total NUMERIC(7,2);
BEGIN
    OPEN curs FOR SELECT * FROM rentals WHERE tape_id = $1;
    total := compute_rental_value( curs );
    CLOSE curs;
    RETURN( total );
END;
' LANGUAGE 'plpgsql';
```

Triggers

So far, all the functions that defined in this chapter have been called explicitly, either by using a `SELECT function()` command or by using the function within an expression. You can also call certain PL/pgSQL functions automatically. A trigger is a function that is called whenever a specific event occurs in a given table. An `INSERT` command, an `UPDATE` command, or a `DELETE` command can cause a trigger to execute.

Let's look at a simple example. You currently have a `customers` table defined like this:

```
CREATE TABLE customers
(
    customer_id    integer primary key,
    customer_name  character varying(50) not null,
    phone          character(8),
    birth_date     date,
    balance        decimal(7,2)
);
```

You want to create a new table that you can use to archive any rows that are deleted from the `customers` table. You also want to archive any updates to the `customers` table. Name this table `customer_archive`:

```
CREATE TABLE customer_archive
(
    customer_id    integer,
    customer_name  character varying(50) not null,
    phone          character(8),
    birth_date     date,
    balance        decimal(7,2),
    user_changed   varchar,
    date_changed   date,
    operation      varchar
);
```

Each row in the `customer_archive` table contains a complete `customers` record plus a few pieces of information about the modification that took place.

Now, let's create a trigger function that executes whenever a change is made to a row in the `customers` table. A trigger function is a function that takes no arguments and returns a special data type—`TRIGGER`. (I'll talk more about the information returned by a trigger in a moment.)

```
CREATE FUNCTION archive_customer() RETURNS TRIGGER AS '
BEGIN
    INSERT INTO customer_archive
    VALUES
    (
        OLD.customer_id,
        OLD.customer_name,
        OLD.phone,
        OLD.birth_date,
        OLD.balance,
        CURRENT_USER,
        now(),
        TG_OP
    );
    RETURN NULL;
END;
' LANGUAGE 'plpgsql';
```

Notice that I am using a variable in this function that I have not declared: `OLD`. Trigger functions have access to several predefined variables that make it easier to find information about the context in which the trigger event occurred. The `OLD` variable contains a copy of the original row when a trigger is executed because of an `UPDATE` or `DELETE` command. The `NEW` variable contains a copy of the new row when a trigger is executed for an `UPDATE` or `INSERT` command.

When this trigger executes, it creates a new row in the `customer_archive()` table. The new row will contain a copy of the original `customers` row, the name of the user making the modification, the date that the modification was made, and the type of operation: `TG_OP` will be set to `'UPDATE'`, `'INSERT'`, or `'DELETE'`.

Table 7.3 contains a complete list of the predefined variables that you can use inside of a trigger function:

Table 7.3. Predefined Trigger Variables

Name	Type	Description
NEW	%ROWTYPE	New values (for UPDATE and INSERT)
OLD	%ROWTYPE	Old values (for UPDATE and DELETE)
TG_NAME	name	Name of trigger
TG_WHEN	text	BEFORE or AFTER
TG_LEVEL	text	ROW or STATEMENT ^[7]
TG_OP	text	INSERT, UPDATE, or DELETE
TG_RELID	Oid	Object ID of trigger table
TG_RELNAME	name	Name of trigger table
TG_NARGS	integer	Count of the optional arguments given to the CREATE TRIGGER command
TG_ARGV[]	text[]	Optional arguments given to the CREATE TRIGGER command

^[7] Statement triggers are not supported in PostgreSQL, so `TG_LEVEL` will always be set to `ROW`.

Now that you have created a function, you have to define it as a trigger function. The `CREATE TRIGGER` command associates a function with an event (or events) in a given table. Here is the command that you use for the `archive_customer()` function:

```
1 CREATE TRIGGER archive_customer
2   AFTER DELETE OR UPDATE
3   ON customers
4   FOR EACH ROW
5   EXECUTE PROCEDURE archive_customer();
```

This is a rather unwieldy command, so let's look at it one line at a time.

The first line tells PostgreSQL that you want to create a new trigger—each trigger has a name—in this case, `archive_customer`. Trigger names must be unique within each table (in other words, I can have two triggers named `foo` as long as the triggers are defined for two different tables). Inside the trigger function, the `TG_NAME` variable holds the name of the trigger.

Line 2 specifies the event (or events) that cause this trigger to fire. In this case, I want the trigger to occur `AFTER` a `DELETE` command or an `UPDATE` command. Altogether, PostgreSQL can fire a trigger `BEFORE` or `AFTER` an `UPDATE` command, an `INSERT` command, or a `DELETE` command. In the trigger function, `TG_WHEN` is set to either `BEFORE` or `AFTER`, and `TG_OP` is set to `INSERT`, `UPDATE`, or `DELETE`.

Line 3 associates this trigger with a specific table. This is not an optional clause; each trigger must be associated with a specific table. You can't, for example, define a trigger that will execute on every `INSERT` statement regardless of the table involved. You can use the `TG_RELNAME` variable in the trigger function to find the name of the associated table. `TG_RELID` holds the object-ID (`OID`) of the table.

A single `DELETE` or `UPDATE` statement can affect multiple rows. The `FOR EACH` clause determines whether a trigger will execute once for each row or once for the entire statement. PostgreSQL does not support statement-level triggers at the moment, so the only choice is `FOR EACH ROW`. Inside of the trigger function, `TG_LEVEL` can contain either `ROW` or `STATEMENT`; but the only value currently implemented is `ROW`.

Line 5 finally gets around to telling PostgreSQL which function you actually want to execute when the specified events occur.

The full syntax for the `CREATE TRIGGER` command is

```
CREATE TRIGGER trigger-name
[BEFORE | AFTER] [ INSERT | DELETE | UPDATE [OR ...]]
ON table-name FOR EACH ROW
EXECUTE PROCEDURE function-name [(args)];
```

TRIGGER Return Values

A trigger function can return a value just like any other function, but the value that you return can have far-reaching consequences. If you return `NULL` from a row-level `BEFORE` trigger, PostgreSQL cancels the rest of the operation for that

row—that means that PostgreSQL won't fire any subsequent triggers and the `INSERT`, `UPDATE`, or `DELETE` won't occur for that row. If you return a non-`NULL` value from a row-level `BEFORE` trigger, the value that you return must match the structure of the table that you're modifying. If PostgreSQL is executing an `UPDATE` or `INSERT` command, the row value that you return from the trigger function is used in place of the original value.

PostgreSQL ignores the return value of an `AFTER` trigger. PostgreSQL also ignores the return value of a statement-level `BEFORE` trigger.

TRIGGER Function Arguments

Notice that the `CREATE TRIGGER` command allows you to specify optional arguments (indicated by `args` in the preceding syntax diagram). You can include a list of string literals when you create a trigger (any arguments that are not of string type are converted into strings). The arguments that you specify are made available to the trigger function through the `TG_NARGS` and `TG_ARGV` variables. `TG_NARGS` contains an integer count of the number of arguments. `TG_ARGV` contains an array of strings corresponding to the values that you specified when you created the trigger: `TG_ARGV[0]` contains the first argument, `TG_ARGV[1]` contains the second argument, and so on. You can use the optional trigger arguments to pass extra information that might help the trigger function know more about the context in which the trigger has executed. You might find this useful when using the same function as a trigger for multiple tables; although in most situations, the `TG_NAME`, `TG_RELNAME`, and `TG_OP` variables provide enough context information.

Polymorphic Functions

Starting with PostgreSQL version 8.0, you can write polymorphic functions in PL/pgSQL. A polymorphic function is a function with at least one parameter of type `ANYELEMENT` or `ANYARRAY`. The types `ANYELEMENT` and `ANYARRAY` are called polymorphic types because they can assume different "shapes" at run-time.

Here's a simple polymorphic function that will return the greater of two arguments:

```
-- ch07.sql
CREATE OR REPLACE FUNCTION max( arg1 ANYELEMENT, arg2 ANYELEMENT )
    RETURNS ANYELEMENT AS $$
BEGIN

    IF( arg1 > arg2 ) THEN
        RETURN( arg1 );
    ELSE
        RETURN( arg2 );
    END IF;

END;
$$ LANGUAGE 'plpgsql';
```

When you call this function with two `INTEGER` values, PL/pgSQL treats the function as if you had defined it as

```
CREATE OR REPLACE FUNCTION max( arg1 INTEGER, arg2 INTEGER )
    RETURNS INTEGER AS $$
```

The polymorphic arguments `arg1` and `arg2` are assumed to be of type `INTEGER`.

If you call this function with two `TEXT` values, `arg1` and `arg2` are considered to be of type `TEXT` and the return value is also assumed to be of type `TEXT`. In fact, you can call this function with two arguments of almost any type. The only restriction is that the function must compile properly for a given type. In the case of the `max()` function, that means that there must be a `>` operator that compares two values of that type (since the function compares `arg1` and `arg2` using the `>` operator).

When you call a polymorphic function, the actual values that you provide for polymorphic parameters must all be of the same type. You can't call the `max()` function with an `INTEGER` and a `TEXT` argument because `arg1` and `arg2` are both defined as `ANYELEMENT` parameters. You can mix polymorphic arguments with other data types, you just have to ensure that all polymorphic arguments are of the same type. If you define `ANYARRAY` arguments, the elements within those arrays must match the type of other polymorphic parameters.

You can also write functions that return a value of type `ANYELEMENT` or `ANYARRAY`. When you call such a function, PostgreSQL infers the data type of the return value from the data type of the polymorphic arguments. You can't write a function that returns a polymorphic value unless the function expects at least one `ANYELEMENT` (or `ANYARRAY`) argument.

Here's a function that returns a polymorphic value. `firstSmaller()` finds the first element in `arg2` that's smaller than `arg1`. `arg2` must be a one-dimensional array:

```
-- ch07.sql
CREATE OR REPLACE FUNCTION firstSmaller( arg1 ANYELEMENT, arg2 ANYARRAY )
    RETURNS ANYELEMENT AS $$
BEGIN

    FOR i IN array_lower( arg2, 1 ) .. array_upper( arg2, 1 ) LOOP

        IF arg2[i] < arg1 THEN
            RETURN( arg2[i] );
        END IF;

    END LOOP;

    RETURN NULL;

END;
$$ LANGUAGE 'plpgsql';
```

You can call this function with an `INTEGER` value and array of `INTEGER`s, or a `TEXT` value and array of `TEXT` values, or a `NUMERIC` value and an array of `NUMERIC` values, and so on. If the polymorphic arguments (`arg1` and `arg2`) are of type `INTEGER`, the return value will be of type `INTEGER`. If you call `firstSmaller()` with `NUMERIC` values, the return value will be of type `NUMERIC`.

A function that returns a polymorphic value automatically inherits an extra variable named `$0`. You can `ALIAS $0` to a more descriptive name, such as `result`, to make it easier to read your code. The type of `$0` is the same as the type of the return value; in other words, the data type of `$0` matches the data type of the polymorphic arguments.

The `sum()` function, shown here, returns a polymorphic value.

```
-- ch07.sql
CREATE OR REPLACE FUNCTION sum( arg1 ANYARRAY ) RETURNS ANYELEMENT AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN

    result := 0;

    FOR i IN array_lower( arg1, 1 ) .. array_upper( arg1, 1 ) LOOP

        IF arg1[i] IS NOT NULL THEN
            result := result + arg1[i];
        END IF;

    END LOOP;

    RETURN( result );

END;

$$ LANGUAGE 'plpgsql';
```

The data type for `$0` is inferred from the type of `arg1`. PL/pgSQL always initializes the return value to `NULL`—because this function accumulates `result` as it progresses through the `arg1` array, you must re-initialize `result` to 0 before you can add to it (remember, `NULL + 5` is not the same as `0 + 5`—`NULL + 5 = NULL`).

Note that you can't call the `sum()` function with an array of `TEXT` values because PostgreSQL doesn't define a `TEXT + TEXT` operator. You can call `sum()` with an array of any numeric type (`INTEGER`, `REAL`, `NUMERIC`, and so on).

PL/ pgSQL and Security

By default, a PL/pgSQL function executes with the privileges of the user that calls it. That's safe because an unprivileged user won't gain extra privileges simply by calling a PL/pgSQL function. However, there are times when you may want to convey extra privileges to a function. For example, you might hide sensitive information (such as payroll data) from a clerical user, but you want that user to "close the books" at the end of each month. Presumably, the `close_the_books()` function can do its work without exposing secret data to the user. If that's the case, you can tell PostgreSQL that you want the `close_the_books()` function to inherit the privileges of the author of the function. To convey extra privileges to a function, just add a `SECURITY` clause to the function definition. The `SECURITY` clause follows the function body and can precede or follow the `LANGUAGE` clause:

```
CREATE [OR REPLACE] FUNCTION name ( [[argname] argtype [, ...] ] )
    RETURNS return_type
    AS $$definition$$
    LANGUAGE langname | [ SECURITY INVOKER | SECURITY DEFINER ]
```

For example:

```
CREATE OR REPLACE FUNCTION close_the_books( ) RETURNS void AS $$
    BEGIN
        ...
    END;
$$ LANGUAGE 'plpgsql' SECURITY DEFINER;
```

If you don't include a `SECURITY` clause, PostgreSQL assumes `SECURITY INVOKER` (meaning that the function executes with the privileges of the invoker). Be aware that when you convey extra privileges to `close_the_books()`, you are also conveying extra privileges to any `SECURITY INVOKER` functions called by `close_the_books()`.

Chapter 15. Using PostgreSQL with PHP

PHP is a general-purpose programming language, but PHP is most commonly used to build dynamic web pages. A dynamic web page is a document that is regenerated each time it is displayed. For example, each time you point your web browser to `cnn.com`, you see the latest news. PHP is useful for building dynamic web pages because you can embed PHP programs within HTML documents. In fact, you can produce HTML documents from a PHP script.

PHP Architecture Overview

The job of a web server (such as Apache or Microsoft's IIS) is to reply to requests coming from a client (usually a web browser). When a browser connects to a web server, it requests information by sending a URL (Uniform Resource Locator). For example, if you browse to the URL <http://www.postgresql.org/software.html>, your web browser connects to the server at www.postgresql.org and requests a file named `software.html`.

After the web server has received this request, it must decide how to reply. If the requested file cannot be found, you'll see the all too familiar HTTP 404 - File not found. Most web servers will choose a response based on the extension of the requested file. A filename ending with `.html` (or `.htm`) is usually associated with a text file containing a HTML document.

Occasionally, you'll see a URL that ends in the suffix `.php`. A `.php` file is a script that is executed by a PHP processor embedded within the web server. The script is executed each time a client requests it. The web browser never sees the `.php` script; only the web server sees it. As the `.php` script executes, it sends information back to the browser (usually in the form of an HTML document).

Listing 15.1 shows a simple PHP script.

Listing 15.1. Simple.php

```
1 <?php
2 # Filename: Simple.php
3 echo "Hey there, I'm a PHP script!";
4 ?>
```

When you run this script (I'll show you how in a moment), the PHP interpreter will send the string "Hey there, I'm a PHP script!" to the browser.

PHP syntax might look a little strange at first, so here's a quick explanation. The script starts with the characters `<?php:` This tells the web server that everything that follows, up to the next `?>`, is a PHP script and should be interpreted by the PHP processor. The next line is treated as a comment because it starts with a `#` character (PHP understands other comment characters, such as `//"` as well). The third line is where stuff happens—this is a call to PHP's `echo()` function. `echo()` is pretty easy to understand; it just sends a string to the web browser. The characters on line 4 (`?>`) mark the end of the script.

Web browsers don't understand how to interpret PHP scripts; they prefer HTML documents. If you can use PHP to send textual data from the server to the browser, you can also send HTML documents (because an HTML document is textual data). This next PHP script (see Listing 15.2) will create an HTML document (and send it to the browser) as it executes.

Listing 15.2. SimpleHTML.php

```
1 <?php
2 # Filename: SimpleHTML.php
3 echo "<HTML>\n";
4 echo "    <HEAD>\n";
5 echo "        <TITLE>SimpleHTML</TITLE>\n";
6 echo "    <BODY>\n";
7 echo "        <CENTER>I'm another simple PHP script</CENTER>\n";
8 echo "    </BODY>\n";
9 echo "</HTML>";
10 ?>
```

When you use a web browser to request this file (`SimpleHTML.php`), the server will execute the script and send the following text to the browser:

```
<HTML>
<HEAD>
<TITLE>SimpleHTML</TITLE>
<BODY>
<CENTER>I'm another simple PHP script</CENTER>
```



```
</BODY>
</HTML>
```

The web browser interprets this as an HTML document and displays the result, as shown in [Figure 15.1](#).

Figure 15.1. SimpleHTML.php in a browser.



Of course, if you want to display static HTML pages, PHP doesn't really offer any advantages—we could have produced this HTML document without PHP's help. The power behind a PHP script is that it can produce a different page each time it executes. [Listing 15.3](#) shows a script that displays the current time (in the server's time zone).

Listing 15.3. Time.php

```
1 <?php
2 //Filename: Time.php
3
4 $datetime = date( "Y-m-d H:i:s (T)" );
5
6 echo "<HTML>\n";
7 echo " <HEAD>\n";
8 echo " <TITLE>Time</TITLE>\n";
9 echo " <BODY>\n";
10 echo " <CENTER>";
11 echo "The current time " . $datetime;
12 echo " </CENTER>\n";
13 echo " </BODY>\n";
14 echo " </HTML>";
15 ?>
```

Line 4 retrieves the current date and time, and assigns it to the variable `$datetime`. Line 11 appends the value of `$datetime` to a string literal and echoes the result to the browser. When you request this PHP script from within a browser, you see a result such as that shown in [Figure 15.2](#).

Figure 15.2. Time.php in a browser.



If you request this document again (say by pressing the `Refresh` button), the web server will execute the script again and display a different result.

Prerequisites

To try the examples in this chapter, you will need access to a web server that understands PHP. I'll be using the Apache web server with PHP installed, but you can also use PHP with Microsoft's IIS, Netscape's web server, and many other servers. To find out if the PostgreSQL interface is available in your copy of PHP, call the `phpinfo()` function and look for a section titled "PostgreSQL Support." If you see that section, you're ready to go. If you don't you might have to compile PHP from source code. You can learn how to compile PHP for your platform at the Zend website (<http://www.zend.com>). Be sure to add the `--with-pgsql` option when you configure the PHP source code.

I'll assume that you are comfortable reading simple HTML documents and have some basic familiarity with PHP in general. Most of this chapter focuses on the details of interacting with a PostgreSQL database from PHP. If you need more information regarding general PHP programming, visit <http://www.zend.com>.

Client 1—Connecting to the Server

The first PHP/PostgreSQL client establishes a connection to a PostgreSQL server and displays the name of the database to which you connect. [Listing 15.4](#) shows the `client1a.php` script.

Listing 15.4. `client1a.php`

```
1 <?php
2 //Filename: client1a.php
3
4 $connect_string = "dbname=movies user=bruce";
5
6 $db_handle = pg_connect( $connect_string );
7
8 echo "<HTML>\n";
9 echo "<HEAD>\n";
10 echo "<TITLE>client1</TITLE>\n";
11 echo "<BODY>\n";
12 echo "<CENTER>";
13 echo "Connected to " . pg_dbname( $db_handle );
14 echo "</CENTER>\n";
15 echo "</BODY>\n";
16 echo "</HTML>";
17 ?>
```

This script connects to a database whose name is hard-coded in the script (at line 4). At line 6, you attempt to make a connection by calling the `pg_connect()` function. `pg_connect()` returns a database handle (also called a database resource). Many of the PostgreSQL-related functions require a database handle, so you need to capture the return value in a variable (`$db_handle`).

When you call `pg_connect()`, you supply a connection string that contains a list of `property=value` pairs^[1]. [Table 15.1](#) lists some of the properties that can appear in a `pg_connect()` connection string. In `client1.php`, you specified two properties: `dbname=movies` and `user=bruce`.

^[1] When you call `pg_connect()` with a single argument, PHP calls the `PQconnectdb()` function from PostgreSQL's libpq API. PHP is yet another PostgreSQL API implemented in terms of libpq.

Table 15.1. Connection Attributes

Connect-string Property	Environment Variable	Example
user	PGUSER	user=korry
password	PGPASSWORD	password=cows
dbname	PGDATABASE	dbname=accounting
host	PGHOST	host=jersey
hostaddr	PGHOSTADDR	hostaddr=127.0.0.1
port	PGPORT	port=5432

If you don't specify one or more of the connect-string properties, default values are derived from the environment variables shown in [Table 15.1](#). If necessary, `pg_connect()` will use hard-coded default values for the `host(localhost)` and `port(5432)` properties. See the section titled "[Connection Properties](#)" in [Chapter 5](#), "Introduction to PostgreSQL Programming," for a complete description of the connection properties that you can use when you call `pg_connect()`.

I'm not very comfortable with the idea of leaving usernames and passwords sitting around in the web server's document tree. It's just too easy to make a configuration error that will let a surfer grab your PHP script files in plain-text form. If that happens, you've suddenly exposed your PostgreSQL password to the world.

A better solution is to factor the code that establishes a database connection into a separate PHP script and then move that script outside the web server's document tree. [Listing 15.5](#) shows a more secure version of your basic PostgreSQL/PHP script.

Listing 15.5. `client1b.php`

```
1 <?php
2 //Filename: client1b.php
3
4 include( "secure/my_connect_pg.php" );
5
6 $db_handle = my_connect_pg( "movies" );
7
```

```

8  echo "<HTML>\n";
9  echo "<HEAD>\n";
10 echo "<TITLE>client1</TITLE>\n";
11 echo "<BODY>\n";
12 echo "<CENTER>";
13 echo "Connected to " . pg_dbname( $db_handle );
14 echo "</CENTER>\n";
15 echo "</BODY>\n";
16 echo "</HTML>";
17 ?>

```

If you compare this to `client1a.php`, you'll see that I've replaced the call to `pg_connect()` with a call to `my_connect_pg()`. I've also added a call to PHP's `include()` directive. The `include()` directive is similar to the `#include` directive found in most C programs: `include(filename)` inlines the named file into the PHP script (`.php`). Now let's look at the `my_connect_pg.php` file (see [Listing 15.6](#)).

Listing 15.6. `connect_pg.php`

```

1  <?php
2  // File:  my_connect_pg.php
3
4  function my_connect_pg( $dbname )
5  {
6      $connect_string = "user=korry password=cows dbname=";
7      $connect_string .= $dbname;
8
9      return( pg_connect( $connect_string ) );
10 }
11 ?>

```

This script defines a function, named `my_connect_pg()`, which you can call to create a PostgreSQL connection. `my_connect_pg()` expects a single string argument, which must specify the name of a PostgreSQL database.

Notice that the username and password are explicitly included in this script. Place this script outside of the web server's document tree so that it can't fall into the hands of a web surfer. The question is: Where should you put it? When you call the `include()` directive (or the related `require()` function), you can specify an absolute path or a relative path. An absolute path starts with a `/` (or drive name or backslash in Windows). A relative path does not. The PHP interpreter uses a search path (that is, a list of directory names) to resolve relative pathnames. You can find the search path using PHP's `ini_get()` function:

```

...
echo "Include path = " . ini_get( "include_path" );
...

```

The `ini_get()` function returns a variable defined in PHP's initialization file^[2]; in this case, the value of `include_path`. On my system, `ini_get("include_path")` returns `./usr/local/php`. PHP searches for `include` files in the current directory (that is, the directory that contains the including script), and then in `/usr/local/php`. If you refer back to [Listing 15.5](#), you'll see that I am including `secure/my_connect_pg.php`. Combining the search path and relative pathname, PHP will find my `include` file in `/usr/local/php/secure/my_connect_pg.php`. The important detail here is that `/usr/local/php` is outside the web server's document tree (`/usr/local/htdocs`).

^[2] You can find the PHP's initialization file using `echo get_cfg_var("cfg_file_path")`.

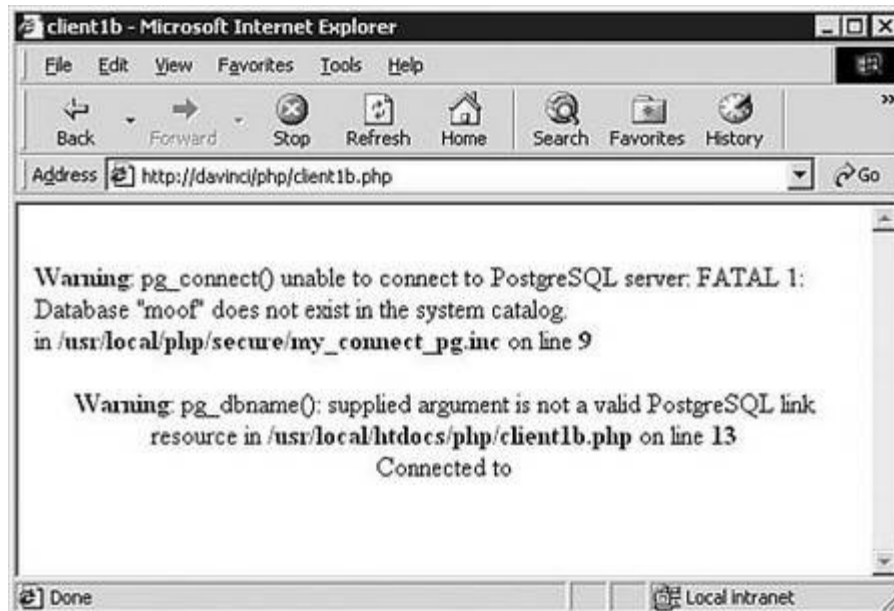
The `my_connect_pg.php` script not only secures the PostgreSQL password, it also gives you a single connection function that you can call from any script—all you need to know is the name of the database that you want.

If everything goes well, the user will see the message "Connected to movies."

Let's see what happens when you throw a few error conditions at this script. First, try to connect to a nonexistent database (see [Figure 15.3](#)).

Figure 15.3. Connecting to a nonexistent database.

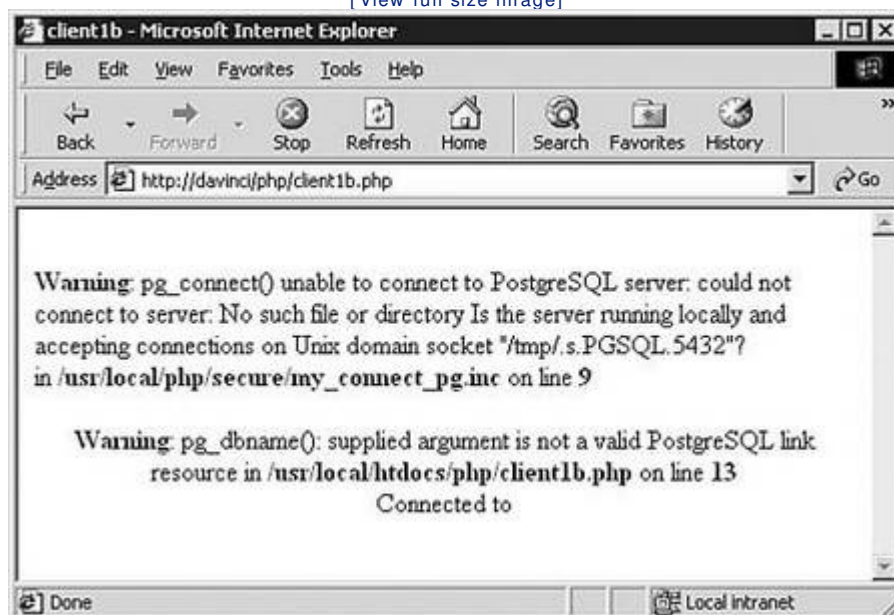
[\[View full size image\]](#)



That's not a friendly error message. Let's see what happens when you try to connect to a database that does exist, but where the PostgreSQL server has been shut down (see [Figure 15.4](#)).

Figure 15.4. Connecting to a database that has been shut down.

[\[View full size image\]](#)



Again, not exactly the kind of message that you want your users to see. In the next section, I'll show you how to intercept this sort of error and respond a little more gracefully.

Client 2—Adding Error Checking

You've seen that PHP will simply dump error messages into the output stream sent to the web browser. That makes it easy to debug PHP scripts, but it's not particularly kind to your users.

There are two error messages displayed in [Figure 15.4](#). The first error occurs when you call the `pg_connect()` function. Notice that the error message includes the name of the script that was running at the time the error occurred. In this case, `my_connect_pg.php` encountered an error on line 9—that's the call to `pg_connect()`. The second error message comes from line 13 of `client1b.php`, where you try to use the database handle returned by `my_connect_pg()`. When the first error occurred, `pg_connect()` returned an invalid handle and `my_connect_pg()` returned that value to the caller.

[Listing 15.7](#) shows a new version of the client script that intercepts both error messages.

Listing 15.7. `client2a.php`

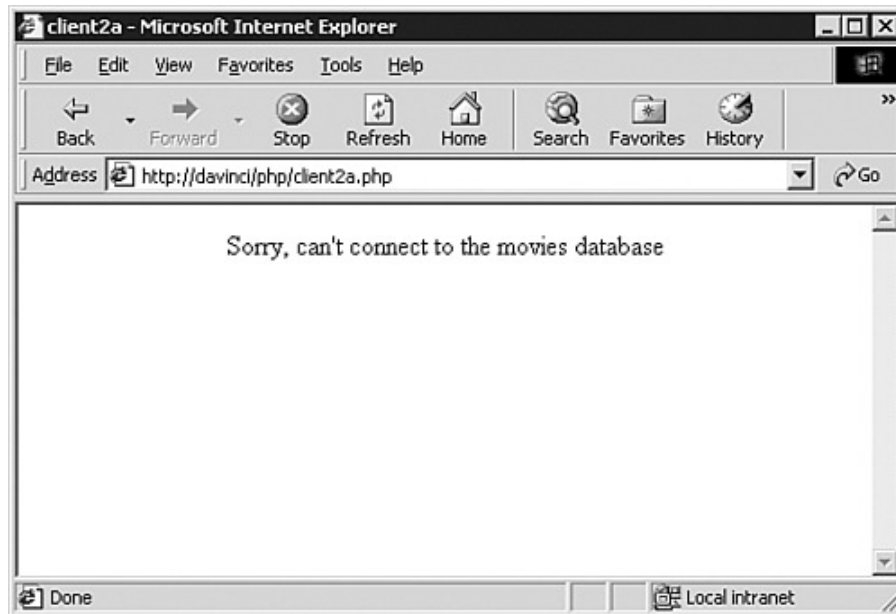
Code View: [Scroll](#) / [Show All](#)

```
1 <?php
2 //Filename: client2a.php
3
4 include( "secure/my_connect_pg.php" );
5
6 $db_handle = @my_connect_pg( "movies" );
7
8 echo "<HTML>\n";
9 echo "<HEAD>\n";
10 echo "<TITLE>client1b</TITLE>\n";
11 echo "<BODY>\n";
12 echo "<CENTER>";
13
14 if( $db_handle == FALSE )
15     echo "Sorry, can't connect to the movies database";
16 else
17     echo "Connected to " . pg_dbname( $db_handle );
18
19 echo "</CENTER>\n";
20 echo "</BODY>\n";
21 echo "</HTML>";
22 ?>
```

If you compare this script with `client1b.php`, you'll see that they are very similar. The first change is at line 6—I've added a `@` character in front of the call to `my_connect_pg()`. The `@` character turns off error reporting for the expression that follows. The next change is at line 14. Rather than blindly using the database handle returned by `my_connect_pg()`, you should first ensure that it is a valid handle. `pg_connect()` (and therefore `my_connect_pg()`) will return `FALSE` to indicate that a connection could not be established. If you find that `$db_handle` is `FALSE`, `client2a` displays a friendly error message; otherwise, it displays the name of the database to which you are connected (see [Figure 15.5](#)).

Figure 15.5. A friendlier error message.

[\[View full size image\]](#)



This looks much nicer, but now we've lost the details that we need to debug connection problems. What we really want is a friendly error message for the user, but details for the administrator.

You can achieve this using a custom-written error handler. [Listing 15.8](#) shows a custom error handler that emails the text of any error messages to your administrator.

Listing 15.8. `my_error_handler.php`

```
1 <?php
2
3 // Filename: my_handler.inc
4
5 function my_handler( $errno, $errmsg, $fname, $lineno, $context )
6 {
7
8     $err_txt = "At " . date("Y-m-d H:i:s (T)");
9     $err_txt .= " an error occurred at line " . $lineno;
10    $err_txt .= " of file " . $fname . "\n\n";
11    $err_txt .= "The text of the error message is:\n";
12    $err_txt .= $errmsg;
13
14    mail( "administrator", "Website error", $err_txt );
15 }
16 ?>
```

In a moment, you'll modify the `client2a.php` script so that it installs this error handler before connecting to PostgreSQL.

An error handler function is called whenever a PHP script encounters an error. The default error handler writes error messages into the output stream sent to the web browser. The custom error handler shown in [Listing 15.8](#) builds an email message from the various error message components and then uses PHP's `mail()` function to send the message to an address of your choice.

Now, let's modify the client so that it uses `my_handler()` (see [Listing 15.9](#)).

Listing 15.9. `client2b.php`

Code View: [Scroll](#) / Show All

```
1 <?php
2 //Filename: client2b.php
3
4 include( "secure/my_connect_pg.php" );
5 include( "my_handler.php" );
6
7 set_error_handler( "my_handler" );
8
9 $db_handle = my_connect_pg( "movies" );
10
11 echo "<HTML>\n";
```



```

12 echo "<HEAD>\n";
13 echo "<TITLE>client2b</TITLE>\n";
14 echo "<BODY>\n";
15 echo "<CENTER>";
16
17 if( $db_handle == FALSE )
18 echo "Sorry, can't connect to the movies database";
19 else
20 echo "Connected to " . pg_dbname( $db_handle );
21
22 echo "</CENTER>\n";
23 echo "</BODY>\n";
24 echo "</HTML>";
25
26 restore_error_handler();
27 ?>

```

You'll make four minor changes to `client2a.php`. First, include() `my__handler.php`. Next, call `set_error_handler()` to direct PHP to call `my_handler()` rather than the default error handler (see line 7). Third, remove the @ from the call to `my_connect_pg()`—you want errors to be reported now; you just want them reported through `my_handler()`. Finally, at line 26, restore the default error handler (because this is the last statement in your script, this isn't strictly required).

Now, if you run `client2b.php`, you'll see a user-friendly error message, and you should get a piece of email similar to this:

```

From daemon Sat Jan 12 09:15:59 2002
Date: Sat, 12 Jan 2002 09:15:59 -0400
From: daemon <daemon@davinci>
To: bruce@virtual_movies.com
Subject: Website error

```

```

At 2002-02-12 09:15:59 (EDT) an error occurred at line 9
of file /usr/local/php/secure/my_connect_pg.php

```

```

The text of the error message is:
pg_connect() unable to connect to PostgreSQL server: could
not connect to server: No such file or directory

```

```

Is the server running locally and accepting
connections on Unix domain socket "/tmp/.s.PGSQL.5432"?

```

Now, you know how to suppress error messages (using the @ operator) and how to intercept them with your own error handler.

In the remaining samples in this chapter, I will omit most error handling code so that you can see any error messages in your web browser; that should make debugging a little easier.

Now, it's time to move on to the next topic—query processing.

Search

Entire Site

☒ All Content☐ Current Book Only

GO >

Advanced Search

Table of Contents



Book

PostgreSQL, Second Edition

Copyright

The Real Value in Free Software

About the Authors

Acknowledgments

We Want to Hear from You!

Reader Services

Introduction

▶ General PostgreSQL Use

▼ Programming with PostgreSQL

▶ Introduction to PostgreSQL Programming

▶ Extending PostgreSQL

▶ PL/pgSQL

▶ The PostgreSQL C API—libpq

▶ A Simpler C API—libpqeasy

▶ The New PostgreSQL C++ API—libpqxx

▶ Embedding SQL Commands in C Programs—ecpg

▶ Using PostgreSQL from an ODBC Client Application

▶ Using PostgreSQL from a Java Client Application

▶ Using PostgreSQL with Perl

▼ Using PostgreSQL with PHP

PHP Architecture Overview

Prerequisites

Client 1—Connecting to the Server

Client 2—Adding Error Checking

Client 3—Query Processing

Client 4—An Interactive Query Processor

Other Features

Summary

▶ Using PostgreSQL with Tcl and Tcl/Tk

▶ Using PostgreSQL with Python

▶ Npgsql: The .NET Data Provider

▶ Other Useful Programming Tools

< Return to Search Results • Print

PostgreSQL, Second Edition

Table of Contents • Index

PRINT FIDELITY VIEW

HTML VIEW

- TEXT ZOOM +

< PREVIOUS

NEXT >

Search Terms: Show/Hide

Using PostgreSQL with PHP > Client 3—Query Processing

Client 3—Query Processing

The tasks involved in processing a query (or other command) using PHP are similar to those required in other PostgreSQL APIs. The first step is to execute the command; then you can (optionally) process the metadata returned by the command; and finally, you process the result set.

We're going to switch gears here. So far, we have been writing PHP scripts that are procedural—one PHP command follows the next. We've thrown in a couple of functions to factor out some repetitive details (such as establishing a new connection). For the next example, you'll create a PHP *class*, named `my_table`, that will execute a command and process the results. You can reuse this class in other PHP scripts; and each time you extend the class, all scripts automatically inherit the changes.

Let's start by looking at the first script that uses the `my_table` class and then we'll start developing the class. Listing 15.10 shows `client3a.php`.

Listing 15.10. `client3a.php`Code View: [Scroll](#) / Show All

```
1 <HTML>
2 <HEAD>
3   <TITLE>client3a</TITLE>
4   <BODY>
5
6 <?php
7   //Filename: client3a.php
8
9   include( "secure/my_connect_pg.php" );
10  include( "my_table_a.php" );
11
12  $db_handle = my_connect_pg( "movies" );
13
14  $table = new my_table( $db_handle, "SELECT * FROM customers;" );
15  $table->finish();
16
17  pg_close( $db_handle );
18
19 ?>
20
21 </BODY>
22 </HTML>
```

I rearranged the code in this client so that the static (that is, unchanging) HTML code is separated from the PHP script; that makes it a little easier to discern the script.

At line 10, `client3a` includes() the `my_table_a.php` file. That file contains the definition of the `my_table` class, and we'll look at it in greater detail in a moment. Line 14 creates a new `my_table` object named `$table`. The constructor function for the `my_table` class expects two parameters: a database handle and a command string. `my_table()` executes the given command and formats the results into an HTML table. At line 15, the call to `my_table->finish()` completes the HTML table. Finally, you call `pg_close()` to close the database connection; that's not strictly necessary, but it's good form.

Listing 15.11 shows `my_table_a.php`.

Browse by Category

- Applied Sciences
- Artificial Intelligence
- Business
- Certification
- Computer Science
- ▼ Databases
 - Access
 - Administration
 - Berkeley DB
 - ColdFusion
 - Data Mining
 - Data Warehousing
 - DB2
 - Database Design
 - Database Management
 - Filemaker Pro
 - FoxPro
 - Informix
 - Introduction
 - MySQL
 - Object Model
 - Object Oriented Database
 - Oracle
 - Performance Tuning
 - PostgreSQL**
 - Relational Database
 - SQL
 - SQL Server
 - Visual Basic
 - Web/Internet Database
- Desktop Publishing
- Desktop Applications
- E-Business
- E-Commerce
- Enterprise Computing
- Graphics
- Human-Computer Interaction
- Hardware
- Internet/Online
- IT Management
- Markup Languages
- Multimedia
- Networking
- Operating Systems
- Programming
- Security
- Software Engineering

[View All Titles >](#)

Code View: [Scroll](#) / [Show All](#)

```
1 <?php
2
3 // Filename: my_table_a.php
4
5 class my_table
6 {
7     var $result;
8     var $columns;
9
10    function my_table( $db_handle, $command )
11    {
12        $this->result = pg_query( $db_handle, $command );
13        $this->columns = pg_num_fields( $this->result );
14        $row_count    = pg_num_rows( $this->result );
15
16        $this->start_table();
17
18        for( $row = 0; $row < $row_count; $row++ )
19            $this->append_row( $this->result, $row );
20    }
21
22    function start_table()
23    {
24        echo '<TABLE CELLPADDING="2" CELSPACING="0" BORDER=1>';
25        echo "\n";
26    }
27
28    function finish()
29    {
30        print( "</TABLE>\n" );
31
32        pg_free_result( $this->result );
33    }
34
35    function append_row( $result, $row )
36    {
37        echo( "<TR>\n" );
38
39        for( $col = 0; $col < $this->columns; $col++ )
40        {
41            echo "    <TD>";
42            echo pg_fetch_result( $result, $row, $col );
43            echo "</TD>\n";
44        }
45
46        echo( "</TR>\n" );
47    }
48 }
49
50 ?>
```

my_table.php defines a single class named `my_table`. At lines 7 and 8, you see the two instance variables for this class. `$this->$result` contains a handle to a result set. `$this->$columns` stores the number of columns in the result set.

The constructor for `my_table` (lines 10 through 20) expects a database handle and a command string. At line 12, the constructor calls the `pg_query()` function to execute the given command. `pg_query()` returns a result set handle if successful, and returns `FALSE` if an error occurs. You'll see how to intercept `pg_query()` errors in a moment. After you have a result set, you can call `pg_num_fields()` to determine the number of columns in the result set and `pg_num_rows()` to find the number of rows.

pg_query() in Earlier PHP Versions

In older versions of PHP, the `pg_query()` function was named `pg_exec()`, `pg_num_fields()` was named `pg_numfields()`, and `pg_num_rows()` was named `pg_numrows()`. If you run into complaints about invalid function names, try the old names.

At line 16, the call to the `start_table()` member function prints the HTML table header. Finally, at lines 18 and 19, the constructor iterates through each row in the result set and calls `append_row()` to create a new row in the HTML table. We'll look at `append_row()` shortly.

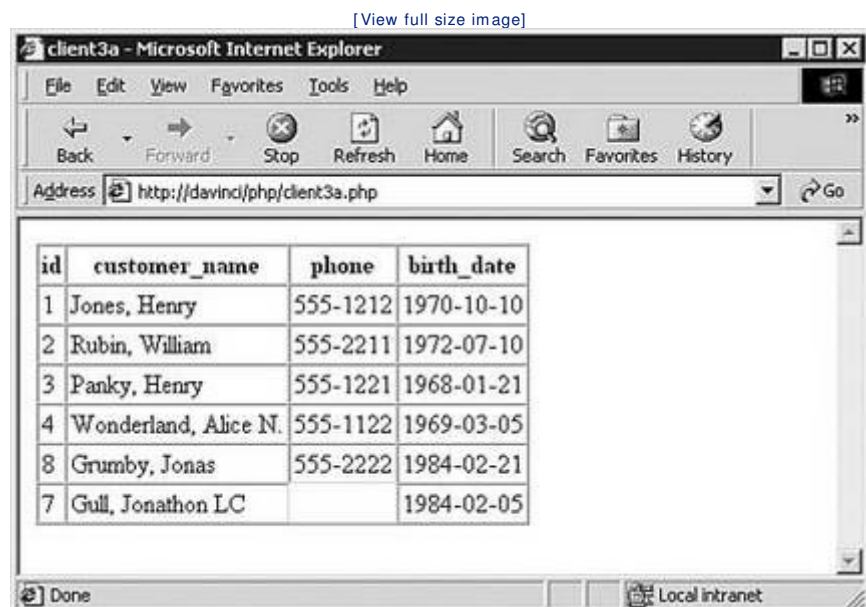
The `start_table()` and `finish_table()` member functions create the HTML table header and table footer, respectively. `finish_table()` also frees up the resources consumed by the result set by calling `pg_free_result()`.

The `append_row()` member function starts at line 35. `append_row()` expects two parameters: a result set handle (`$result`) and a row number (`$row`). At line 37, `append_row()` writes the HTML table-row tag (`<TR>`). The loop at lines 39 through 44 processes each column in the given row. For each column, `append_row()` writes the HTML table-data tag (`<TD>`) and the table-data closing tag (`</TD>`). In-between these tags, you see a call to `pg_fetch_result()` that retrieves a single value from the result set. When you call `pg_fetch_result()`, you provide three parameters: a result set handle, a row number, and a column number. `pg_fetch_result()` returns `NULL` if the requested value is `NULL`^[3]. If not `NULL`, `pg_fetch_result()` will return the requested value in the form of a string. Note that the PHP/PostgreSQL documentation states numeric values are returned as `float` or `integer` values. This appears not to be the case; all values are returned in string form.

^[3] In PHP 4.0 and above, `NULL` is equal to `FALSE`, but not identical to `FALSE`. This means that `NULL == FALSE` evaluates to `TRUE`, but `NULL === FALSE` does not.

Now if you load `client3a.php` in your web browser, you'll see a table similar to that shown in Figure 15.6.

Figure 15.6. `client3a.php` loaded into your web browser.



Other Ways to Retrieve Result Set Values

Besides `pg_fetch_result()`, PHP provides a number of functions that retrieve result set values.

The `pg_fetch_row()` function returns an array of values that correspond to a given row. `pg_fetch_row()` requires two parameters: a result resource (also known as a result set handle) and a row number.

```
pg_fetch_row( resource result, int row_number )
```

Listing 15.12 shows the `my_table.append_row()` member function implemented in terms of `pg_fetch_row()`.

Listing 15.12. `append_row()` Using `pg_fetch_row()`

```
...
1 function append_row( $result, $row )
2 {
3     echo( "<TR>\n" );
4
5     $values = pg_fetch_row( $result, $row );
6
7     for( $col = 0; $col < count( $values ); $col++ )
```

```

8  {
9      echo " <TD>";
10     echo $values[$col];
11     echo "</TD>\n";
12 }
13 echo( "</TR>\n" );
14 }
...

```

In this version, you fetch the requested row at line 5. When the call to `pg_fetch_row()` completes, `$values` will contain an array of column values. You can access each array element using an integer index, starting at element 0.

The next function, `pg_fetch_array()`, is similar to `pg_fetch_row()`. Like `pg_fetch_row()`, `pg_fetch_array()` returns an array of columns values. The difference between these functions is that `pg_fetch_array()` can return a normal array (indexed by column *number*), an associative array (indexed by column *name*), or both. `pg_fetch_array()` expects one, two, or three parameters:

```
pg_fetch_array( resource result [, int row [, int result_type ]] )
```

The third parameter can be `PGSQL_NUM`, `PGSQL_ASSOC`, or `PGSQL_BOTH`. When you specify `PGSQL_NUM`, `pg_fetch_array()` operates identically to `pg_fetch_row()`; the return value is an array indexed by column number. When you specify `PGSQL_ASSOC`, `pg_fetch_array()` returns an associative array indexed by column name. If you specify `PGSQL_BOTH`, you will get back an array that can be indexed by column number as well as by column name. Listing 15.13 shows the `append_row()` function rewritten to use `pg_fetch_array()`.

Listing 15.13. `append_row()` Using `pg_fetch_array()`

```

...
1 function append_row( $result, $row )
2 {
3     echo( "<TR>\n" );
4
5     $values = pg_fetch_array( $result, $row, PGSQL_ASSOC );
6
7     foreach( $values as $column_value )
8     {
9         echo " <TD>";
10        echo $column_value;
11        echo "</TD>\n";
12    }
13
14    echo( "</TR>\n" );
15 }
...

```

You should note that this version of `append_row()` misses the point of using `PGSQL_ASSOC`. It ignores the fact that `pg_fetch_array()` has returned an *associative* array. Associative arrays make it easy to work with a result set if you know the column names ahead of time (that is, at the time you write your script), but they really don't offer much of an advantage for ad hoc queries. To really take advantage of `pg_fetch_array()`, you would write code such as

```

...
$result = pg_query( $dbh, "SELECT * FROM customers;" );

for( $row = 0; $row < pg_num_rows( $result ); $row++ )
{
    $customer = pg_fetch_array( $result, $row, PGSQL_ASSOC );

    do_something_useful( $customer["customer_name"] );

    do_something_else( $customer["id"], $customer["phone"] );
}
...

```

You can also obtain an associative array by calling `pg_fetch_assoc(resource result [, int row])`. Calling `pg_fetch_assoc()` is equivalent to calling `pg_fetch_array(..., PGSQL_ASSOC)`.

Another function useful for static queries is `pg_fetch_object()`. `pg_fetch_object()`

returns a single row in the form of an object. The object returned has one field for each column, and the name of each field will be the same as the name of the column. For example:

```
...
$result    = pg_query( $dbhhandle, "SELECT * FROM customers;" );

for( $row = 0; $row < pg_num_rows( $result ); $row++ )
{
    $customer = pg_fetch_object( $result, $row, PGSQL_ASSOC );

    do_something_useful( $customer->customer_name );

    do_something_else( $customer->id, $customer->phone );
}
...
```

There is no significant difference between an object returned by `pg_fetch_object()` and an associative array returned by `pg_fetch_array()`. With `pg_fetch_array()`, you reference a value using `$array[$column]` syntax. With `pg_fetch_object()`, you reference a value using `$object->$column` syntax. Choose whichever syntax you prefer.

One warning about `pg_fetch_object()` and `pg_fetch_array(..., PGSQL_ASSOC)`—if your query returns two or more columns with the same column name, you will lose all but one of the columns. You can't have an associative array with duplicate index names, and you can't have an object with duplicate field names.

Metadata Access

You've seen that `pg_fetch_object()` and `pg_fetch_array()` expose column names to you, but the PHP/PostgreSQL API lets you get at much more metadata than just the column names.

The PHP/PostgreSQL interface is written using libpq (PostgreSQL's C-language API). Most of the functions available through libpq can be called from PHP, including the libpq metadata functions. Unfortunately, this means that PHP shares the limitations that you find in libpq.

In particular, the `pg_field_size()` function returns the size of a field. `pg_field_size()` expects two parameters:

```
int pg_field_size( resource $result, int $column_number )
```

The problem with this function is that the size reported is the number of bytes required to store the value *on the server*. It has nothing to do with the number of bytes seen by the client (that is, the number of bytes seen by your PHP script). For variable-length data types, `pg_field_size()` will return -1. If you're using a newer version of PHP (at least version 4.2.0) you can call `pg_field_prtlen()` to find the string length of a given value. You can call `pg_field_prtlen()` in either of the following forms:

Code View: [Scroll](#) / Show All

```
int pg_field_prtlen( resource $result, int $row_number, int $column_number )
int pg_field_prtlen( resource $result, int $row_number, string $column_name )
```

The `pg_field_type()` function returns the name of the data type for a given column. `pg_field_type()` requires two parameters:

```
int pg_field_type( resource $result, int $column_number )
```

The problem with `pg_field_type()` is that it is not 100% accurate. `pg_field_type()` knows nothing of user-defined types or domains. Also, `pg_field_type()` won't return details about parameterized data types. For example, a column defined as `NUMERIC(7,2)` is reported as type `NUMERIC`. *Note:* `pg_field_type()` has been improved in PHP version 5; it now queries the server to retrieve the name of the column's data type so it will return the correct name for user-defined types and domains (but it still doesn't return details about parameterized types).

Having conveyed the bad news, let's look at the metadata functions that are a little more useful for most applications.

You've already seen `pg_num_rows()` and `pg_num_fields()`. These functions return the number of rows and columns (respectively) in a result set.

The `pg_field_name()` and `pg_field_num()` functions are somewhat related. `pg_field_name()` returns the name of a column, given a column number index. `pg_field_num()` returns the column number index of a field given the field's name.

Let's enhance the `my_table` class a bit by including column names in the HTML table that we produce. Listing 15.14 shows a new version of the `start_table()` member function.

Listing 15.14. `my_table.start_table()`

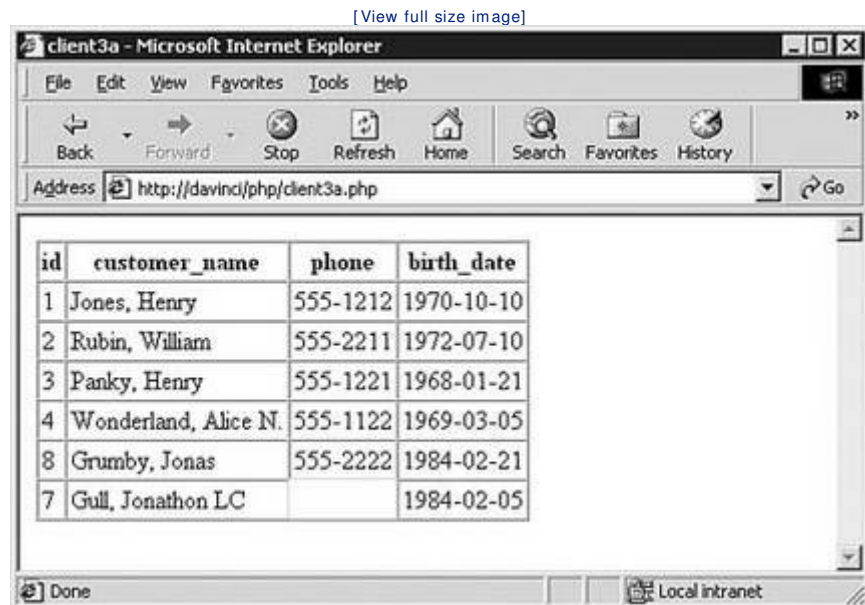
```
1 function start_table()
2 {
3     echo '<TABLE CELLPADDING="2" CELSPACING="0" BORDER="1">';
4
5     for( $col = 0; $col < $this->columns; $col++ )
6     {
7         echo "    <TH>";
8         echo pg_field_name( $this->result, $col );
9         echo "</TH>\n";
10    }
11    echo "\n";
12 }
```

I used the `<TH>` tag here instead of `<TD>`, so that the browser knows that these are table header cells (table header cells are typically bolded and centered).

Now when you browse to `client3a.php`, you see a nice set of column headers as shown in Figure 15.7.

Figure 15.7. `client3a.php`—with column headers.

[View full size image]



id	customer_name	phone	birth_date
1	Jones, Henry	555-1212	1970-10-10
2	Rubin, William	555-2211	1972-07-10
3	Panky, Henry	555-1221	1968-01-21
4	Wonderland, Alice N.	555-1122	1969-03-05
8	Grumby, Jonas	555-2222	1984-02-21
7	Gull, Jonathon LC		1984-02-05

Let's fix one other problem as long as we are fiddling with metadata. You may have noticed that the last row in Figure 15.7 looks a little funky—the `phone` number cell has not been drawn the same as the other cells. That happens when we try to create a table cell for a `NULL` value. If you look at the code that you built for the HTML table, you'll see that the last row has an empty `<TD></TD>` cell. For some reason, web browsers draw an empty cell differently.

To fix this problem, you can modify `append_row()` to detect `NULL` values (see Listing 15.15).

Listing 15.15. `my_table.append_row()`

```
1 function append_row( $result, $row )
2 {
3     echo( "<TR>\n" );
```

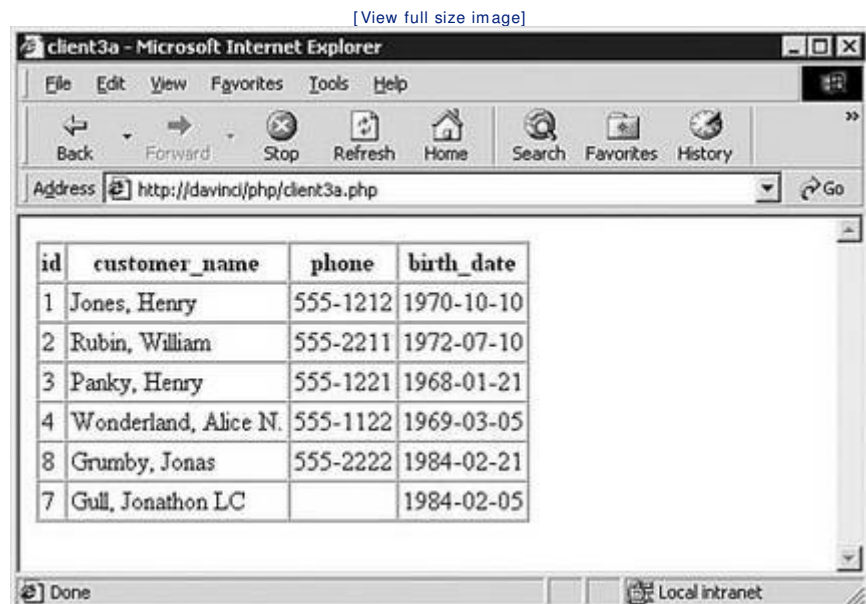
```

4
5     for( $col = 0; $col < $this->columns; $col++ )
6     {
7         echo "    <TD>";
8
9         if( pg_field_is_null( $result, $row, $col ) == 1 )
10            echo "&nbsp;";
11        elseif( strlen( pg_result( $result, $row, $col ) ) == 0 )
12            echo "&nbsp;";
13        else
14            echo pg_result( $result, $row, $col );
15        echo "</TD>\n";
16    }
17
18    echo( "</TR>\n" );
19 }

```

At line 9, you detect `NULL` values using the `pg_field_is_null()` function. If you encounter a `NULL`, you echo a non-breaking space character (` `) instead of an empty string. You have the same problem (a badly drawn border) if you encounter an empty string, and you fix it the same way (lines 11 and 12). Now, when you display a table, all the cells are drawn correctly, as shown in Figure 15.8.

Figure 15.8. `client3a.php`—final version.



There are a few more metadata functions that you can use in PHP, and you will need those functions in the next client that you write.

PHP, PostgreSQL, and Associative Functions

One of the more interesting abstractions promised (but not yet offered) by PHP and the PHP/PostgreSQL API is the *associative function*. An associative function gives you a way to execute a SQL command without having to construct the entire command yourself. Let's say that you need to `INSERT` a new row into the `customers` table. The most obvious way to do this in PHP is to build up an `INSERT` command by concatenating the new values and then executing the command using `pg_query()`. Another option is to use the `pg_insert()` function. With `pg_insert()`, you build an associative array. Each element in the array corresponds to a column. The key for a given element is the name of the column, and the value for the element is the value that you want to insert. For example, you can add a new row to the `customers` table with the following code:

```

...
$customer["id"]           = 8;
$customer["customer_name"] = "Smallberries, John";
$customer["birth_date"]    = "1985-05-14";

pg_insert( $db_handle, "customers", $customer );
...

```


In this code snippet, you have created an associative array with three entries. When you execute the call to `pg_insert()`, PHP will construct the following `INSERT` command:

```
INSERT INTO customers
(
    id,
    customer_name,
    birth_date
)
VALUES
(
    8,
    'Smallberries, John',
    '1985-05-14'
);
```

PHP knows the name of the table by looking at the second argument to `pg_insert()`. The column names are derived from the keys in the `$customers` array, and the values come from the values in the associative array.

Besides `pg_insert()`, you can call `pg_delete()` to build and execute a `DELETE` command. When you call `pg_delete()`, you provide a database handle, a table name, and an associative array. The associative array is used to construct a `WHERE` clause for the `DELETE` command. The values in the associative array are `ANDed` together to form the `WHERE` clause.

You can also use `pg_select()` to construct and execute a `SELECT *` command. `pg_select()` is similar to `pg_delete()`—it expects a database handle, a table name, and an associative array. Like `pg_delete()`, the values in the associative array are `ANDed` together to form a `WHERE` clause.

Finally, the `pg_update()` function expects two associative arrays. The first array is used to form a `WHERE` clause, and the second array should contain the data (column names and values) to be updated.

As of PHP version 5.0, the associative functions are documented as experimental and are likely to change. Watch for these functions in a future release.

PostgreSQL, Second Edition

[Table of Contents](#) • [Index](#)

[PRINT FIDELITY VIEW](#)

[HTML VIEW](#)

[- TEXT ZOOM +](#)

[< PREVIOUS](#)

[NEXT >](#)

Top of Page

Additional Reading

Hide 

Safari has identified sections in other books that relate directly to this selection using Self-Organizing Maps (SOM), a type of neural network algorithm. SOM enables us to deliver related sections with higher quality results than traditional query-based approaches allow.

1. [Using the PHP SQLite Extension](#)
From [SQLite](#) by [Chris Newman](#)
2. [The PHP API for MySQL](#)
From [Sams Teach Yourself MySQL in 21 Days, Second Edition](#) by [Tony Butcher](#)
3. [PHP Language Structure](#)
From [Sams Teach Yourself PHP, MySQL and Apache: All in One, Third Edition](#) by [Julie Meloni](#)
4. [The PHP API](#)
From [MySQL® Phrasebook](#) by [Zak Greant](#); [Chris Newman](#)
5. [The Process](#)
From [Learning PHP and MySQL](#) by [Jon A. Phillips](#); [Michele E. Davis](#)
6. [Quick Start, Quick Reference](#)
From [PHP and MySQL by Example](#) by [Ellie Quigley](#); [Marko Gargenta](#)
7. [Accessing MySQL Using PHP](#)
From [Learning MySQL](#) by [Seyed M.M. "Saied" Tahaghoghi](#); [Hugh E. Williams](#)
8. [Untainting User Data](#)
From [Learning MySQL](#) by [Seyed M.M. "Saied" Tahaghoghi](#); [Hugh E. Williams](#)
- 9.

Writing an Object-Oriented MySQL Interface for PHP
From MySQL Cookbook by Paul DuBois

10. Querying the Database with PHP Functions
From Learning PHP & MySQL, 2nd Edition by Michele E. Davis; Jon A. Phillips

URL <http://proquest.safaribooksonline.com/0672327562/ch15lev1sec5>

[Company](#) | [Terms of Service](#) | [Privacy Policy](#) | [Contact Us](#) | [Help](#) | [508 Compliance](#)
Copyright © 2007 Safari Books Online. All rights reserved.

Client 4—An Interactive Query Processor

You now have most of the pieces that you need to build a general-purpose query processor within a web browser. Our next client simply prompts the user for a SQL command, executes the command, and displays the results.

If you want to try this on your own web server, be sure that you understand the security implications. If you follow the examples in this chapter, your PHP script will use a hard-coded username to connect to PostgreSQL. Choose a user with very few privileges. In fact, most PHP/PostgreSQL sites should probably define a user account specifically designed for web access. If you're not careful, you'll grant John Q. Hacker permissions to alter important data.

We'll start out with a simple script and then refine it as we discover problems.

First, you need an HTML page that displays a welcome and prompts the user for a SQL command. Listing 15.16 shows the `client4.html` document.

Listing 15.16. `client4.html`

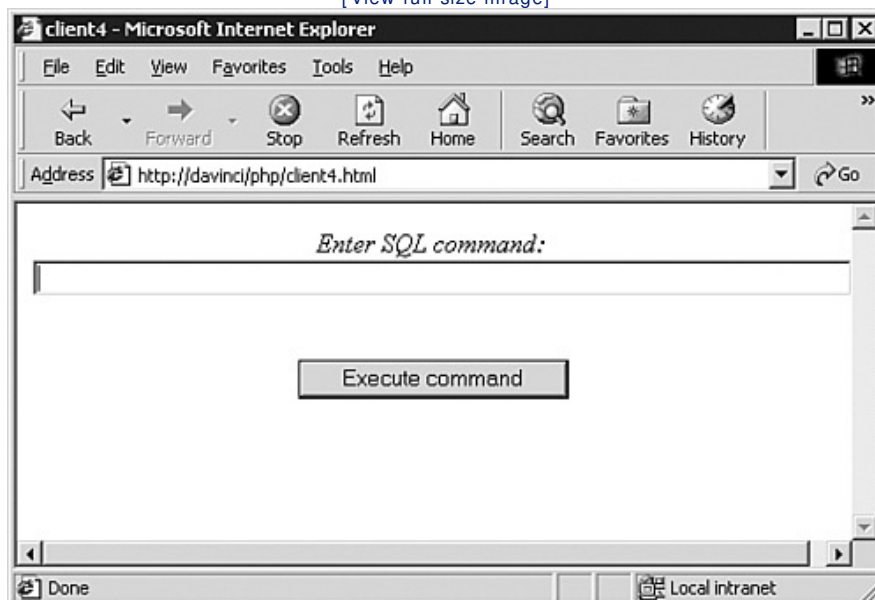
Code View: [Scroll](#) / [Show All](#)

```
1 <HTML>
2
3 <!-- Filename: client4.html>
4
5 <HEAD>
6 <TITLE>client4a</TITLE>
7 <BODY>
8 <CENTER>
9 <FORM ACTION="client4a.php" METHOD="POST">
10 <I>Enter SQL command:</I><br>
11
12 <INPUT TYPE="text"
13 NAME="query"
14 SIZE="80"
15 ALIGN="left"
16 VALUE="">
17
18 <BR><BR>
19 <INPUT TYPE="submit" VALUE="Execute command">
20 </FORM>
21 </CENTER></BODY>
22 </HTML>
```

This HTML document defines a form that will be posted to the server (see line 9). After the user enters a command and presses the `Execute Command` button, the browser will request the file `client4a.php`. We'll look at `client4a.php` in a moment. When you request this page in a web browser, you will see a form similar to that shown in Figure 15.9.

Figure 15.9. `client4.html`.

[\[View full size image\]](#)



Now let's look at the second half of the puzzle—`client4a.php` (see [Listing 15.17](#)).

Listing 15.17. `client4a.php`

Code View: [Scroll](#) / Show All

```
1 <HTML>
2 <HEAD>
3 <TITLE>Query</TITLE>
4 <BODY>
5 <?php
6
7     # Filename: client4a.php
8
9     include( "secure/my_connect_pg.php" );
10    include( "my_table_e.php" );
11
12    $command_text = $HTTP_POST_VARS[ "query" ];
13
14    if( strlen( $command_text ) == 0 )
15    {
16        echo "You forgot to enter a command";
17    }
18    else
19    {
20        $db_handle = my_connect_pg( "movies" );
21
22        $table = new my_table( $db_handle, $command_text );
23        $table->finish();
24
25        pg_close( $db_handle );
26    }
27    ?>
28 </BODY>
29 </HTML>
```

Most of this script should be pretty familiar by now. You include `secure/my_connect_pg.php` to avoid embedding a username and password inline. Next, include `my_table_e.php` so that you can use the `my_table` class (`my_table_e.php` includes all the modifications you made to the original version of `my_table_a.php`).

At line 12, `client4a` retrieves the command entered by the user from the `$HTTP_POST_VARS[]` variable. Look back at lines 12 through 16 of [Listing 15.16](#) (`client4.html`). You are defining an `INPUT` field named `query`. When the user enters a value and presses the `Execute Command` button, the browser posts the `query` field to `client4a.php`. PHP marshals all the posted values into a single associative array named `$HTTP_POST_VARS[]` (also known as `$_POST` starting in PHP version 5). The key for each value in this array is the name of the posted variable. So, you defined a field named `query`, and you can find the value of that field in `$HTTP_POST_VARS["query"]`.

If you try to execute an empty command using `pg_query()`, you'll be rewarded with an ugly error message. You can be a little nicer to your users by intercepting empty commands at lines 14 through 16 and displaying a less intimidating error message.

The remainder of this script is straightforward: simply establish a database connection and use the `my_table` class to execute the given command and display the result.

Let's run this script to see how it behaves (see [Figures 15.10](#) and [15.11](#)).

Figure 15.10. Submitting a query with `client4.html`.

[\[View full size image\]](#)

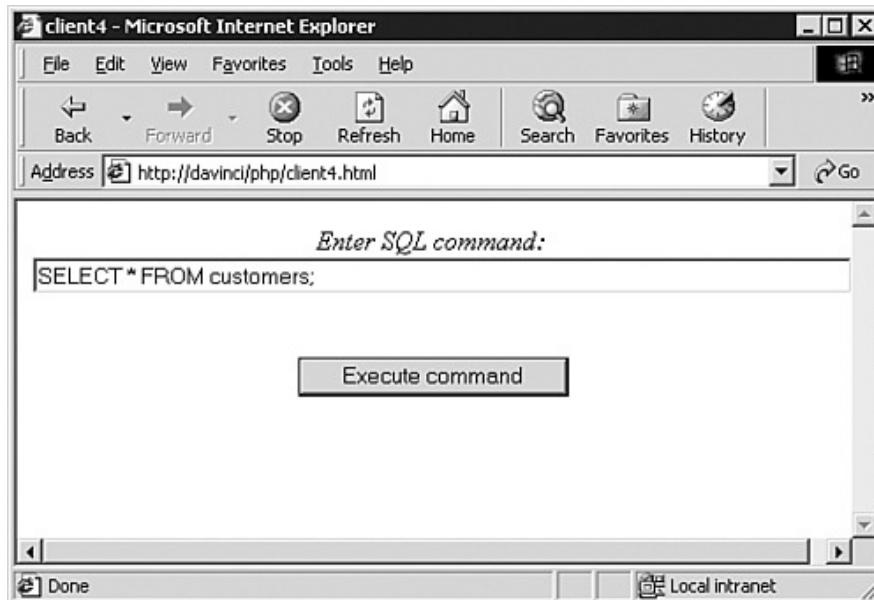
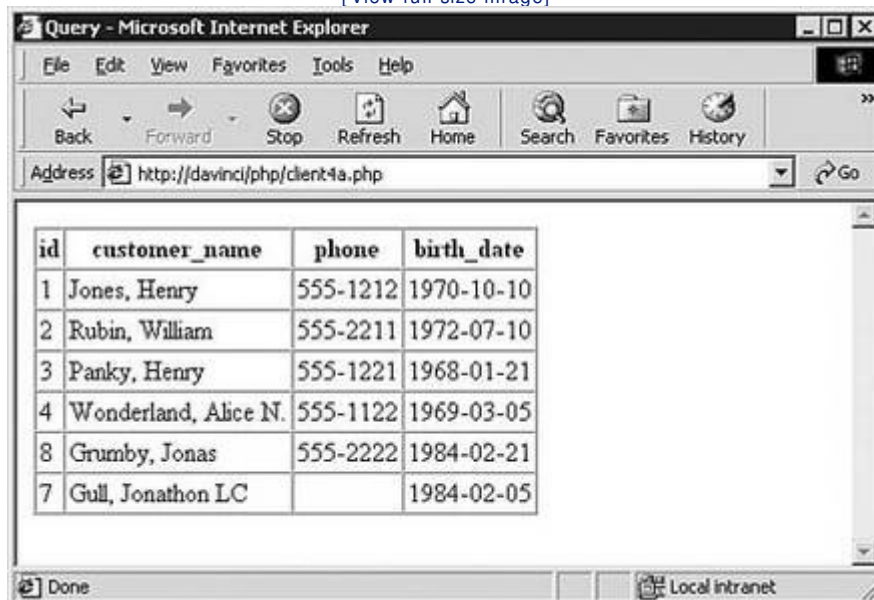


Figure 15.11. Submitting a query with client4.html—result.

[\[View full size image\]](#)



That worked nicely. Let's try another query (see [Figures 15.12](#) and [15.13](#)).

Figure 15.12. Causing an error with client4.html.

[\[View full size image\]](#)

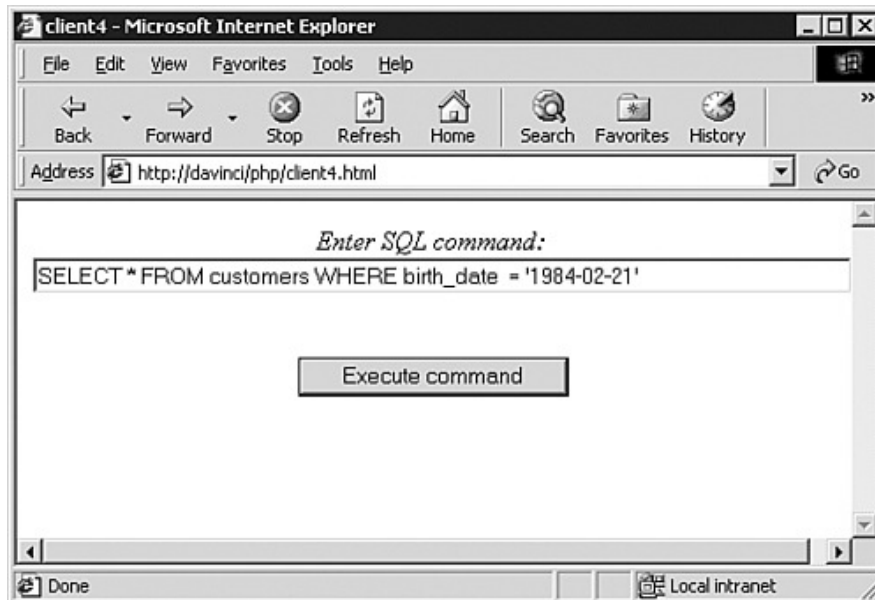
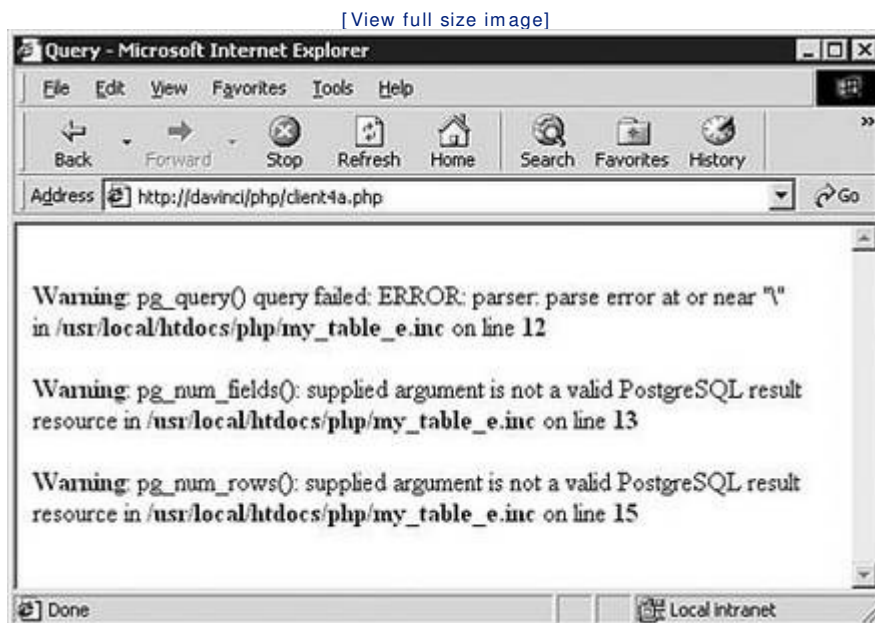


Figure 15.13. Causing an error with client4.html—result.



Hmmm...that's not what we were hoping for. What went wrong? Actually, there are several problems shown here. First, PHP is reporting that we have an erroneous backslash on line 12 of `my_table_e.php`. Line 12 is inside of the `my_table` constructor and it sends the following command to the server:

```
$this->result = pg_query( $db_handle, $command );
```

There are no backslashes on that line; there are no backslashes in the command that you entered. Where are the backslashes coming from? If you echo `$HTTP_POST_VARS["query"]`, you'll see that PHP has added escape characters to the command entered by the user. You entered `SELECT * FROM customers WHERE birth_date = '1984-02-21'`, and PHP changed this to `SELECT * FROM customers WHERE birth_date = \'1984-02-21\'`. According to the PHP manual, all single-quotes, double-quotes, backslashes, and NULLs are escaped with a backslash when they come from a posted value.^[4]

^[4] You can disable the automatic quoting feature by setting the `magic_quotes_gpc` configuration variable to `no`. I would not recommend changing this value—you're likely to break many PHP scripts.

This is easy to fix. You can simply strip the escape characters when you retrieve the command text from `$HTTP_VARS[]`. Changing `client4a.php`, line 12, to

```
if( get_magic_quotes_gpc() )
    $command_text = stripslashes( $HTTP_POST_VARS[ "query" ] );
```

will make it possible to execute SQL commands that contain single-quotes.

That was the first problem. The second problem is that you don't want the end-user to see these nasty-looking PHP/PostgreSQL error messages. To fix this problem, you need to intercept the error message and display it yourself. Listing 15.18 shows a new version of the `my_table` constructor.

Listing 15.18. `my_table.my_table()`

```
1 function my_table( $db_handle, $command )
2 {
3     $this->result = @pg_query( $db_handle, $command );
4
5     if( $this->result == FALSE )
6     {
7         echo pg_last_error( $db_handle );
8     }
9     else
10    {
11        $this->columns = pg_num_fields( $this->result );
12        $row_count    = pg_num_rows( $this->result );
13
14        $this->start_table( $command );
15
16        for( $row = 0; $row < $row_count; $row++ )
17            $this->append_row( $this->result, $row );
18    }
19 }
```

We've restructured this function a bit. Because the goal is to intercept the default error message, we suppress error reporting by prefixing the call to `pg_query()` with an `@`. At line 5, determine whether `pg_query()` returned a valid result set resource. If you are used to using PostgreSQL with other APIs, there is an important difference lurking here. In other PostgreSQL APIs, you get a result set even when a command fails—the error message is part of the result set. In PHP, `pg_query()` returns `FALSE` when an error occurs. You must call `pg_last_error()` to retrieve the text of the error message (see line 7).

If you have succeeded in executing the given command, you can build an HTML table from the result set as before.

Now, if you run into an error condition, the result is far more palatable (see Figures 15.14 and 15.15).

Figure 15.14. Causing an error with `client4.html`—part 2.

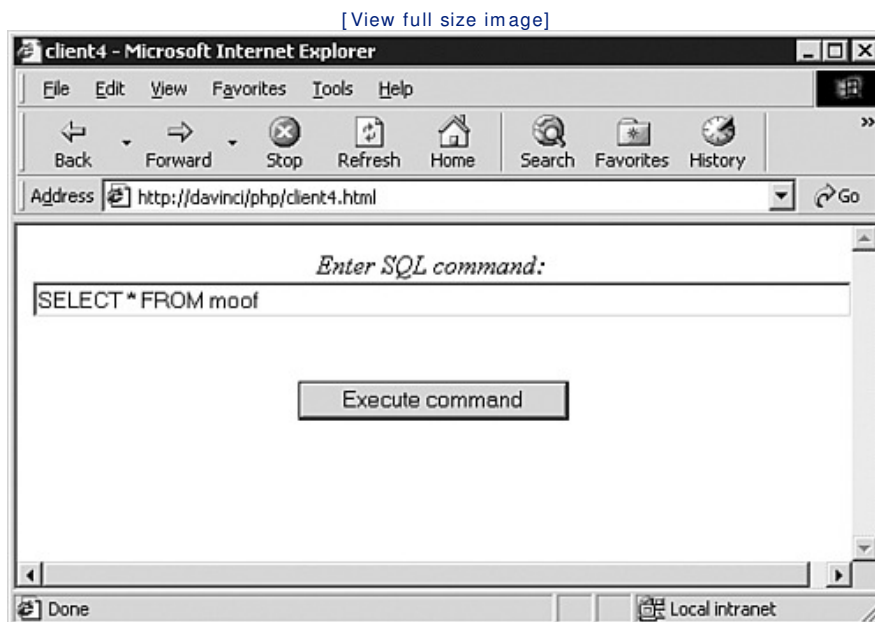
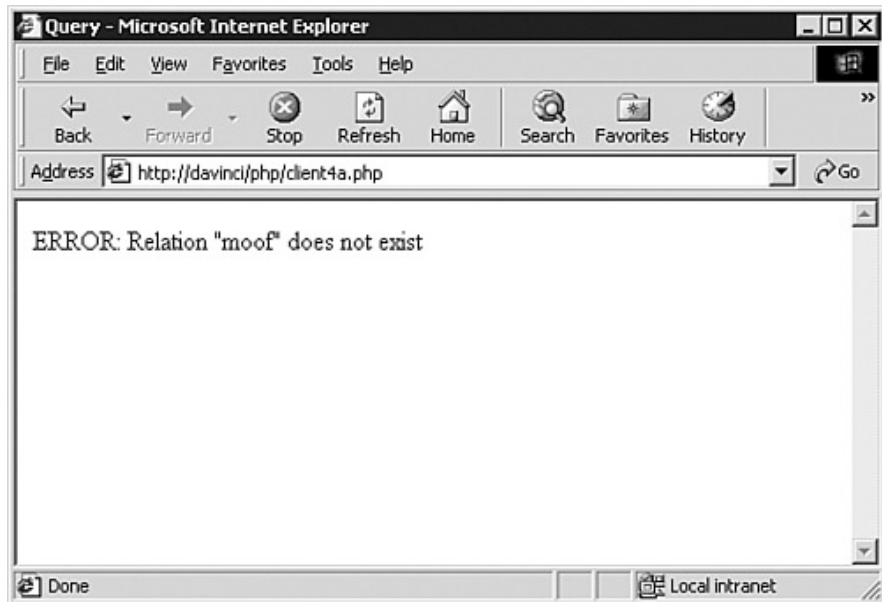


Figure 15.15. Causing an error with `client4.html`—part 2, result.

[\[View full size image\]](#)



Notice that you see only one error message this time. In [Figure 15.13](#), you saw multiple error messages. Not only had `client4a` failed to intercept the original error, but it went on to use an invalid result set handle; when you fix the first problem, the other error messages go away.

At this point, you can execute queries and intercept error messages. Let's see what happens when you execute a command other than `SELECT`. First, enter the command shown in [Figure 15.16](#).

Figure 15.16. Executing an `INSERT` command.

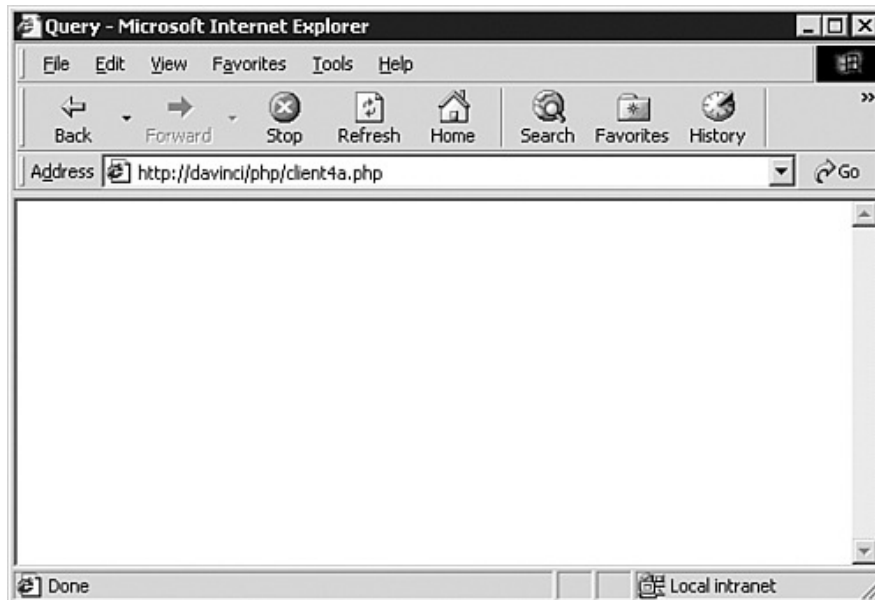
[\[View full size image\]](#)



After clicking on the `Execute Command` button, you see the result displayed in [Figure 15.17](#).

Figure 15.17. Executing an `INSERT` command—result.

[\[View full size image\]](#)



Hmmm...that's a bit minimalist for my taste. You should at least see a confirmation that something has happened. When you execute a non-SELECT command, the `pg_query()` function will return a result set resource, just like it does for a SELECT command. You can differentiate between SELECT and other commands by the fact that `pg_num_fields()` always returns 0 for non-SELECT commands.

Let's make one last modification to the `my_table` constructor, in Listing 15.19, so that it gives feedback regardless of which type of command executed.

Listing 15.19. `my_table.my_table()`—Final Form

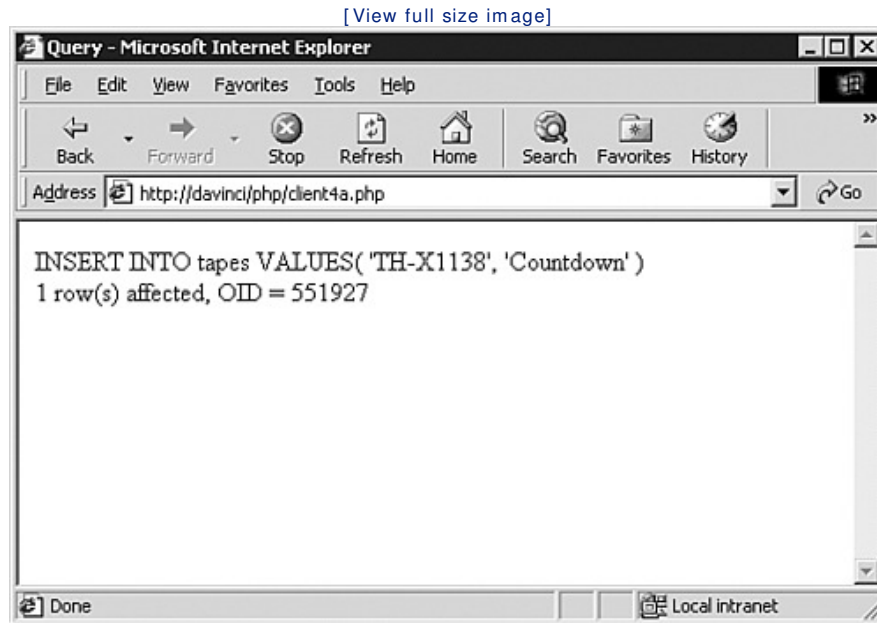
Code View: [Scroll](#) / [Show All](#)

```
1 function my_table( $db_handle, $command )
2 {
3     $this->result = @pg_query( $db_handle, $command );
4
5     if( $this->result == FALSE )
6     {
7         echo pg_last_error( $db_handle );
8     }
9     else
10    {
11        $this->columns = pg_num_fields( $this->result );
12
13        if( $this->columns == 0 )
14        {
15            echo $command;
16            echo "<BR>";
17            echo pg_affected_rows( $this->result );
18            echo " row(s) affected";
19
20            if( pg_last_oid( $this->result ) != 0 )
21                echo ", OID = ". pg_last_oid( $this->result );
22        }
23        else
24        {
25            $row_count = pg_num_rows( $this->result );
26
27            $this->start_table( $command );
28
29            for( $row = 0; $row < $row_count; $row++ )
30                $this->append_row( $this->result, $row );
31        }
32    }
33 }
```

This version checks the result set column count at line 13. If you find that the result set contains 0 columns, echo the command text and the number of rows affected by the command (that gives you feedback similar to what you would see using the `psql` client). You can also call the `pg_last_oid()` function. `pg_last_oid()` returns the OID (object ID) of the most recently inserted row. `pg_last_oid()` returns 0 if the command was not an `INSERT` or if more than one row was inserted.

The final results are shown in [Figure 15.18](#).

Figure 15.18. Executing an `INSERT` command—final result.



Now that you know how to write an interactive query processor using PHP, don't forget the security implications of doing so. Make sure that you connect to the PostgreSQL server using an account with very few privileges: If you give a visitor lots of privileges and a way to execute any command he wants, you're inviting disaster.

More typically, you ask a visitor to your site to type in a value that you use to construct an SQL command. For example, you may prompt the user for a "Customer Name" and retrieve the matching record with a query such as

```
$query = "SELECT * FROM customers WHERE customer_name = '";  
$query .= $_POST['customer_name'] . "'";
```

That works, sometimes. What happens if the user types in a value such as "Freddy's Fudge Factory"? Then your query becomes

```
SELECT * FROM customers WHERE customer_name = 'Freddy's Fudge Factory'.
```

See the problem? As far as PostgreSQL is concerned, a string literal is enclosed in a pair of single quotes. The PostgreSQL parser thinks the customer name ends at `Freddy`—the rest of the customer name is just a syntax error. To fix that sort of problem, just quote any string values with the `pg_escape_string()` function, like this:

```
$customer_name = pg_escape_string( $_POST['customer_name'] );  
$query = "SELECT * FROM customers WHERE customer_name = '$customer_name'";
```

Other Features

There are a number of PostgreSQL-related PHP functions that I have not covered in this chapter.

Newer versions of PHP have added support for asynchronous query processing (see `pg_send_query()`, `pg_connection_busy()`, and `pg_get_result()`). Asynchronous query processing probably won't be of much use when you are constructing dynamic web pages, but clever coders can use asynchronous queries to provide intermediate feedback for long-running operations (sorry, I'm not that clever).

PHP offers a set of functions that can give you information about a database connection. We used the `pg_dbname()` function in the first client (see [Listing 15.4](#)) to display the name of the database to which we were connected. You can also use the `pg_port()` and `pg_options()` function to retrieve the port number and options associated with a database connection. PHP provides a `pg_host()` function that is supposed to return the name of the host where the server resides. Be very careful calling `pg_host()`; if you have established a local connection (that is, using a Unix-domain socket), calling `pg_host()` may crash your web server because of a bug in the PHP/PostgreSQL interface.

Another function offered by PHP is `pg_pconnect()`. The `pg_pconnect()` function establishes a persistent connection to a PostgreSQL database. Persistent connections are cached by the web server and can be reused the next time a browser requests a document that requires access to the same database. See the PHP manual for information about the pros and cons of persistent connections.

Finally, PHP supports the PostgreSQL large-object interface. You can use the large-object interface to read (or write) large data items such as images or audio files.

Part III: PostgreSQL Administration

- 20 Introduction to PostgreSQL Administration
- 21 PostgreSQL Administration
- 22 Internationalization and Localization
- 23 Security
- 24 Replicating PostgreSQL Data with Slony
- 25 Contributed Modules

Chapter 20. Introduction to PostgreSQL Administration

This book is divided into three parts. The first part of the book was designed as a guide to new PostgreSQL users. The middle section covered PostgreSQL programming. The third section is devoted to the topic of PostgreSQL administration. These three parts correspond to the real-world roles that we play when using PostgreSQL.

Users are concerned mostly with getting data into the database and getting it back out again. Programmers try to provide users with the functionality that they need. Administrators are responsible for ensuring that programmers and end users can perform their jobs. Quite often, one person will fill two or three roles at the same time.

When you wear the hat of an administrator, you ensure that your users can store their data in a secure, reliable, high-availability, high-performance database.

Secure means that your data is safe from intruders. You must ensure that authorized users can do the things they need to do. You also need to ensure that users cannot gain access to data that they should not see.

Reliable means the data that goes into a database can be retrieved without corruption. Any data transformations should be expected, not accidental.

High-availability means that the database is available when needed. Your users should expect that the database is ready to use when they log in. Routine maintenance should follow a predictable schedule and should not interfere with normal use. High-availability may also affect your choice of operating system and hardware. You may want to choose a cluster configuration to prevent problems in the event of a single point of failure.

High-performance means that a user should be able to perform required tasks within an acceptable amount of time. A high-performance database should also feel responsive.

In this chapter, I'll introduce you to some of the tasks that a PostgreSQL administrator must perform. The remaining chapters cover each topic in greater detail.

Security

A PostgreSQL administrator is responsible for ensuring that authorized users can do what they need to do. An administrator is also responsible for making sure that authorized users can do only what they need to do. Another critical job is to keep intruders away from the user's data.

There are two aspects to PostgreSQL security—authentication and access. Authentication ensures that a user is in fact who he claims to be. After you are satisfied that a user has proven his identity, you must ensure that he can access the data that he needs.

Each user (or group) requires access to a specific set of resources. For example, an accounting clerk needs access to vendor and customer records, but may not require access to payroll data. A payroll clerk, on the other hand, needs access to payroll data, but not to customer records. One of your jobs as an administrator is to grant the proper privileges to each user.

Another aspect of security in general is the problem of securing PostgreSQL's runtime environment. Depending on your security requirements (that is, the sensitivity of your data), it may be appropriate to install network firewalls, secure routers, and possibly even biometric access controls. Securing your runtime environment is a problem that is not unique to PostgreSQL, and I won't explore that topic further in this book.

Chapter 23, "Security," shows you how to grant and revoke user privileges and also covers how to prevent tampering by intruders. I'll show you how to secure PostgreSQL data, configuration, and program files on Linux/Unix systems and on Windows hosts.

Chapter 25. Contributed Modules

Most of the software that I've described in this book is considered to be part of the core PostgreSQL distribution. The core distribution is managed by the core development team—a small, well-organized, and highly-dedicated team of professional developers and designers. But there are a huge number of developers that contribute software to the PostgreSQL community. Some of the contributed packages are included in the core distribution (in the `contrib` directory) and you can find many others at the PgFoundry and Gborg web sites (www.pgfoundry.org and gborg.postgresql.org).

Contributed software is a broad term that describes open-source software designed to work with PostgreSQL. That includes everything from graphical SQL client applications and graphical management applications to procedural languages such as PL/Java and PL/perl. You can also find contributed packages that will help you convert data and programs from other systems (such as MySQL, Oracle, and mSQL) into PostgreSQL. At the Gborg web site, you can find an ODBC driver, a JDBC driver (for Java applications), a DBD:: driver (for Perl applications), and interfaces for applications written in C, C++, Visual Basic, C#, Tcl/Tk, Ruby, Python, and maybe even Cobol. You'll find database design tools, monitoring tools, administrator tools, developer tools, even complete business applications.

In this chapter, I'll describe two of the contribute packages that come in the core PostgreSQL distribution: `xml2` and `tsearch2`. The `xml2` package lets you store XML documents inside a PostgreSQL database, query those documents using XPath queries, and convert XML documents using XSLT stylesheets. `tsearch2` is a full-text indexing and searching package that lets you turn your PostgreSQL server into a search engine.

Exchanging PostgreSQL Data with XML

XML is the wave of the future. Well, it's a wave in some future anyway. XML was designed to let you and I write applications that can exchange structured data. An XML document is a self-describing textual representation of data, often structured in a hierarchical form. In this section, I'll assume that you have some knowledge of XML, XPath queries, and XSLT stylesheets. If you aren't familiar with those technologies, read on—I'll show you a few examples that should help you understand the basic concepts.

You can store XML data in a PostgreSQL database without any help from third-party software. For example, let's say that one of your distributors offers a new service to the video store that you're running. Every so often the distributor sends you an XML document that describes a number of films. A typical document is shown in [Listing 25.1](#).

Listing 25.1. `films.xml`

Code View: [Scroll](#) / [Show All](#)

```
<films>
  <film>
    <name>Casablanca</name>
    <year>1942</year>
    <writers>
      <writer>Murray Burnett</writer>
      <writer>Joan Alison</writer>
      <writer>Julius J. Epstein</writer>
      <writer>Philip G. Epstein</writer>
      <writer>Howard Koch</writer>
    </writers>
    <leads>
      <lead>Humphrey Bogart</lead>
      <lead>Ingrid Bergman</lead>
      <lead>Peter Lorre</lead>
    </leads>
    <directors>
      <director>Michael Curtiz</director>
    </directors>
  </film>

  <film>
    <name>Rear Window</name>
    <year>1954</year>
    <writers>
      <writer>Cornell Woolrich</writer>
      <writer>John Michael Hayes</writer>
    </writers>
    <leads>
      <lead>James Stewart</lead>
      <lead>Grace Kelly</lead>
      <lead>Raymond Burr</lead>
    </leads>
    <directors>
      <director>Alfred Hitchcock</director>
    </directors>
  </film>

  <film>
    <name>The Godfather</name>
    <year>1972</year>
    <writers>
      <writer>Mario Puzo</writer>
      <writer>Francis Ford Coppola</writer>
    </writers>
    <leads>
      <lead>Marlon Brando</lead>
      <lead>Al Pacino</lead>
      <lead>James Caan</lead>
      <lead>Robert Duvall</lead>
      <lead>Diane Keaton</lead>
      <lead>Talia Shire</lead>
    </leads>
  </film>
</films>
```

```

    <directors>
      <director>Francis Ford Coppola</director>
    </directors>
  </film>
</films>

```

This document (`films.xml`) describes three films: *Casablanca*, *Rear Window*, and *The Godfather*. Each description contains a name, a year (the year that the film was released), a collection of writers, a collection of leads (leading actors and actresses), and a collection of directors.

To store this document in a PostgreSQL database, you could simply `INSERT` the whole thing into a `TEXT` column. In practice, you'd probably want to split the document into separate records (one for each film) and store each description in a separate row. Let's do that. First, create a table (`filminfo`) that will hold the film descriptions like this:

```

$ psql movies
Welcome to psql 8.0.0, the PostgreSQL interactive terminal.
...
movies=# CREATE TABLE filminfo
movies=# (
movies(#   film_name VARCHAR PRIMARY KEY,
movies(#   description TEXT
movies(# );
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit
        index "filminfo_pkey" for table "filminfo"
CREATE TABLE

```

Now you have a container, but how do you get the XML objects into the `filminfo` table? The distributor has given you an XML document; you want to split that document into separate objects and then `INSERT` those objects into the `filminfo` table.

Sounds like a perfect job for XSLT (Extensible Stylesheet Language Transformations). [Listing 25.2](#) shows an XSLT document (`splitFilms.xsl`) that will do the trick.

Listing 25.2. `splitFilms.xsl`

```

1 <xsl:stylesheet
2   xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
3   <xsl:output method="xml" omit-xml-declaration="yes"/>
4
5   <xsl:template match="/">
6     <xsl:for-each select="films/film">
7       <xsl:text>INSERT INTO filminfo VALUES('<xsl:text>
8         <xsl:value-of select="name"/>
9         <xsl:text>','</xsl:text>
10        <xsl:copy-of select="."/>
11        <xsl:text>');
12      </xsl:text>
13    </xsl:for-each>
14  </xsl:template>
15 </xsl:stylesheet>

```

`splitFilms.xsl` will parse through a list of films and create an `INSERT` command for each film that it finds. Given the first film described in `films.xml` (see [Listing 25.1](#)), `splitFilms.xsl` will produce an `INSERT` command that looks like this:

```

INSERT INTO filminfo VALUES('Casablanca','<film>
  <name>Casablanca</name>
  <year>1942</year>
  <writers>
    <writer>Murray Burnett</writer>
    <writer>Joan Alison</writer>
    <writer>Julius J. Epstein</writer>
    <writer>Philip G. Epstein</writer>
    <writer>Howard Koch</writer>
  </writers>
  <leads>
    <lead>Humphrey Bogart</lead>
    <lead>Ingrid Bergman</lead>
    <lead>Peter Lorre</lead>
  </leads>
  <directors>
    <director>Michael Curtiz</director>
  </directors>
</film>');

```

You can save the `INSERT` commands to a text file or, better yet, just pipe the output produced by `splitFilms.xsl` directly into the `psql` command. If you're using the `libxslt` package, you can execute `splitFilms.xsl` like this:

```
$ xsltproc splitFilms.xsl films.xml | psql movies
INSERT 846648 1
INSERT 846649 1
INSERT 846650 1
```

There are three films described in `films.xml` and `psql` reports three `INSERT` commands—that's a good sign.

Now that you have XML documents in your database, you can use them just like any other `TEXT` value. You can search inside of an XML document using PostgreSQL's regular expression operators. For example, to find films starring Jimmy Stewart, you could execute the following query:

```
movies=# SELECT film_name FROM filminfo
movies=# WHERE description ~* '<leads>.*James Stewart.*</leads>';
 film_name
-----
Rear Window
(1 row)
```

You can treat an XML document just like any other `TEXT` value, but you can add a number of XML-specific features by installing the `xml2` contributed module.

XPath Queries

`xml2` (also known as `pgxml`) is a small collection of functions that let you execute XPath queries against XML documents stored in a PostgreSQL database. `xml2` does not turn your database into an XML database—you can't execute XPath queries in place of `SELECT` statements. Instead, `xml2` lets you include XPath queries inside of `SELECT` (and other statements).

`xml2` defines three functions that will return a single value from an XPath query:

```
xpath_string( document, query ) RETURNS TEXT
xpath_number( document, query ) RETURNS FLOAT4
xpath_bool( document, query ) RETURNS BOOL
```

When you call one of the `xml2` query functions, you provide an XML document and an XPath query. The query function returns the object (a number, string, Boolean value, list of values, or a `nodeset`) identified by the XPath. If that sounds confusing, it may help to look at an example. Here's a query that extracts the `year` node stored within a film description:

Code View: [Scroll](#) / Show All

```
movies=# SELECT film_name, xpath_string( description, 'year' ) FROM filminfo;
 film_name | xpath_string
-----+-----
Casablanca | 1942
Rear Window | 1954
The Godfather | 1972
(3 rows)
```

In this case, the `xpath_string()` function is invoked three times (because there are three rows in the `filminfo` table). In each invocation, the `description` column acts as an XML document and `'year'` is an XPath query that navigates through the document until it finds the `year` element.

You can also use the XPath query functions in other parts of the `SELECT` command, such as the `WHERE` clause. For example, to find all films released before 1960:

```
movies=# SELECT film_name FROM filminfo
movies=# WHERE xpath_number( description, '/film/year' ) < 1960;
 film_name
-----
Casablanca
Rear Window
(2 rows)
```

In fact, you can use the result of an XPath query as a table. (I'll show you how to do that in a moment.)

`xml2` defines five functions that return multiple values from an XPath query. The first set of functions return a `nodeset` in the form of a `TEXT` string (a `nodeset` is a collection of XML nodes):

```
xpath_nodeset( document, query, topTag, itemTag ) RETURNS TEXT
xpath_nodeset( document, query, itemTag ) RETURNS TEXT
xpath_nodeset( document, query ) RETURNS TEXT
```

Use the `nodeset` functions when you want to extract a set of values from an XML document and you want the result to retain the XML tags present in the document. For example, to find all `writers` for a given film:

```

movies=# SELECT xpath_nodeset( description, 'writers' ) FROM filminfo
movies=#   WHERE film_name = 'Rear Window';
        xpath_nodeset
-----
<writers>
  <writer>Cornell Woolrich</writer>
  <writer>John Michael Hayes</writer>
</writers>
(1 row)

```

If you include a `topTag` or `itemTag`, `xpath_nodeset()` will wrap the entire nodeset in the `topTag` and wrap each item inside of an `<itemTag>` `</itemTag>` pair. The extra tags are useful if you are building a new XML document out of data already in the database.

To convert a nodeset into a more conventional (and often more useful form), use the `xpath_list()` function.

```

movies=# SELECT xpath_list( description, 'leads/lead') FROM filminfo
movies=#   WHERE film_name = 'The Godfather';
        xpath_list
-----
Marlon Brando,Al Pacino,James Caan,Robert Duvall,Diane Keaton,Talia Shire
(1 row)

```

`xpath_list()` expects two or three arguments. If you call `xpath_list()` with three arguments, the last argument determines the string that separates each element in the list (the default separator is `" , "`).

The last XPath function provided by `xml2` is `xpath_table()` and it can be somewhat confusing. `xpath_table()` creates a tabular result set by executing an XPath query (or a series of queries separated by `"|"`) against a table (or view). Here's an example:

Code View: [Scroll](#) / Show All

```

movies=# SELECT * FROM
movies=#   xpath_table('film_name','description','filminfo','year|leads/*','1=1')
movies=#   AS t(film_name text, year text, leads text);
        film_name | year | leads
-----+-----+-----
Casablanca       | 1942 | Humphrey Bogart
Casablanca       |      | Ingrid Bergman
Casablanca       |      | Peter Lorre
Rear Window      | 1954 | James Stewart
Rear Window      |      | Grace Kelly
Rear Window      |      | Raymond Burr
The Godfather     | 1972 | Marlon Brando
The Godfather     |      | Al Pacino
The Godfather     |      | James Caan
The Godfather     |      | Robert Duvall
The Godfather     |      | Diane Keaton
The Godfather     |      | Talia Shire
(12 rows)

```

The `AS` clause tells PostgreSQL the shape of the resulting table. (`xpath_table()` is defined to return a `SETOF RECORDS`—since a `RECORD` has no predefined shape, you have to tell PostgreSQL what shape to expect.)

`xpath_table()` expects five arguments:

```

xpath_table( key, document, table, xpathQueries, condition )

```

`xpath_table()` creates a `SELECT` command (based on the arguments that you provide) and then executes that command. Next, `xpath_table()` reads through each row returned by the `SELECT` command and evaluates the XPath queries against the `document` column.

The `SELECT` command is constructed from the `key`, `document`, `table`, and `condition` arguments. After executing the `SELECT` command, `xpath_table()` verifies that the query returned exactly two columns (a `key` and a `document`) and then reads through each row in the result set, evaluating each XPath query that you provide. For each XPath query, `xpath_table()` evaluates that query and, if it returns a value, stores that value in an intermediate tuple. When the tuple is complete, `xpath_table()` adds its to the final result set. If you invoke `xpath_table()` with two or more XPath queries that return nodesets of differing sizes, the final result set will contain `NULL` values.

That's a rather complex description that might be better illustrated by walking through the process one step at a time.

Given the arguments

```

xpath_table('film_name','description','filminfo','year|leads/*','1=1')

```

`xpath_table()` starts by splitting the `xpathQueries` argument (`year|leads/*`) into individual queries. In this case, `xpath_table()` finds two queries: `year` and `leads/*`. Since you've supplied two queries, the result set produced by `xpath_table()` will contain three columns: the first column will contain the `key` field (`film_name`), the second column will contain the result of the first XPath query (`year`), and the third column will contain the result of the second XPath query (`leads/*`).

Next, `xpath_table()` pastes together a `SELECT` command that looks like this:

```
SELECT film_name, description FROM filminfo WHERE l=1;
```

`xpath_table()` always selects two columns from the table: the key and the document. The `condition` argument is tacked on to the end of the command in the form of a `WHERE` clause—you must provide a `condition` even if you want to process every row in the given table^[1].

^[1] The condition argument is just tacked onto the end of the `SELECT` command—you can include any text that can legally follow the word `WHERE` in a `SELECT` command. In fact, you can force `xpath_table()` to join two tables by listing them both in the `table` argument and specifying a join in the `condition` argument.

Next, `xpath_table()` executes the `SELECT` command and loops through each row that makes it through the `WHERE` clause. `xpath_table()` constructs one or more tuples out of each row returned by the `SELECT` command. To fill in the first column in each new tuple, `xpath_table()` simply copies the key column (`film_name`) from the row returned by the `SELECT` command. To fill in the remaining columns, `xpath_table()` evaluates each XPath query against the document column. In this example, the first document (the `description` column) returned by the `SELECT` command looks like this:

```
<film>
  <name>Casablanca</name>
  <year>1942</year>
  <writers>
    <writer>Murray Burnett</writer>
    <writer>Joan Alison</writer>
    <writer>Julius J. Epstein</writer>
    <writer>Philip G. Epstein</writer>
    <writer>Howard Koch</writer>
  </writers>
  <leads>
    <lead>Humphrey Bogart</lead>
    <lead>Ingrid Bergman</lead>
    <lead>Peter Lorre</lead>
  </leads>
  <directors>
    <director>Michael Curtiz</director>
  </directors>
</film>
```

The first XPath query (`year`) nabs a nodeset that contains a single value (1942). The second XPath query (`leads/*`) returns a nodeset containing three values (Humphrey Bogart, Ingrid Bergman, and Peter Lorre). For each row returned by the `SELECT` command, `xpath_table()` produces one or more tuples. The number of new tuples is determined by the largest nodeset returned by the XPath queries. In this case, the largest nodeset contains three nodes, so `xpath_table()` will add three tuples to the final result set. To form the new tuples, `xpath_table()` copies the key value ("Casablanca") into the first column of each new tuple, and then starts copying the nodes into the remaining columns. If `xpath_table()` runs out of nodes for a given column (and it will run out when the nodesets differ in length), it writes a `NULL` value into the tuple instead.

When it finishes with the first row, `xpath_table()` repeats the process for each of the remaining rows. When it hits the second row (Rear Window), `xpath_table()` again finds that the nodeset produced by the first XPath query contains a single value (1954) and the second nodeset contains three values—that means three more rows in the final result set. The last row returned by the `SELECT` command (The Godfather) produces one nodeset that contains a single value (1972) and second nodeset that contains six values, so `xpath_table()` adds six more rows to the final result set.

You can infer a few rules from this walk-through:

- The number of columns produced by `xpath_tables()` is always one more than the number of XPath queries that you specify
- The `AS` clause that you define must contain one more column than the number of XPath queries that you specify
- The first column in the result set always contains values from the `key` column that you specify
- The number of rows produced for any given row in the source table is determined by the largest nodeset extracted from that row
- For any given row in the source table, some columns will contain `NULL` values if the nodesets extracted from that row differ in size

You can see that the XPath query functions are powerful but they can also be unwieldy. You can simplify the `xml2` functions by wrapping them in custom-made views and functions.

For example, you can easily create a view that uses `xpath_table()` to extract leading actors and actresses like this:

Code View: [Scroll](#) / [Show All](#)

```
movies=# CREATE VIEW film_leads AS
movies=# SELECT * FROM
movies=#   xpath_table('film_name','description','filminfo','leads/*','l=1')
movies=#   AS t(name text, leads text);
CREATE VIEW

test=# SELECT * FROM film_leads WHERE name = 'Casablanca';
   name   |      leads
-----+-----
Casablanca | Humphrey Bogart
```

```
Casablanca | Ingrid Bergman
Casablanca | Peter Lorre
(3 rows)
```

That's much better. Of course, you can treat a view built from `xpath_table()` just like any other table or view. For example, you could join the `film_leads` view and the `tapes` table to produce a list of all leading actors and actresses starring in the films that you have in stock:

```
movies=# SELECT DISTINCT ON( title, leads ) tape_id, title, leads
movies-# FROM tapes, film_leads WHERE name = title;
 tape_id | title | leads
-----+-----+-----
MC-68873 | Casablanca | Humphrey Bogart
MC-68873 | Casablanca | Ingrid Bergman
MC-68873 | Casablanca | Peter Lorre
AH-54706 | Rear Window | Grace Kelly
AH-54706 | Rear Window | James Stewart
AH-54706 | Rear Window | Raymond Burr
AB-67472 | The Godfather | Al Pacino
AB-67472 | The Godfather | Diane Keaton
AB-67472 | The Godfather | James Caan
AB-67472 | The Godfather | Marlon Brando
AB-67472 | The Godfather | Robert Duvall
AB-67472 | The Godfather | Talia Shire
(12 rows)
```

(the `DISTINCT ON` clause weeds out any duplicates in case you have multiple copies of the same video, each with a different `tape_id`.)

You can also simplify the XPath query functions (`xpath_string()`, `xpath_node()`, `xpath_list()`, and so on) by wrapping them in more convenient forms. In fact, you don't even have to resort to a procedural language (such as PL/pgSQL or Java)—you can write the wrapper functions in SQL. For example, the script shown in [Listing 25.3](#) creates a function that returns a comma-separated list of the leads actors and actresses starring in a given film.

Listing 25.3. `starring.sql`

```
--
-- Filename: starring.sql
--
CREATE FUNCTION starring( title TEXT ) RETURNS TEXT AS
$$
    SELECT xpath_list( description, 'leads/*')
    FROM filminfo
    WHERE film_name = $1
$$
LANGUAGE 'SQL';
```

You can call this function in the select-list part of a `SELECT` command, in the `WHERE` clause, or in both parts:

```
movies=# SELECT tape_id, title, starring( title ) FROM tapes
movies-# WHERE starring( title ) LIKE '%James Stewart%';
 tape_id | title | starring
-----+-----+-----
AH-54706 | Rear Window | James Stewart,Grace Kelly,Raymond Burr
(1 row)
```

That query returns the leading actors and actresses who star in any video that you stock that features `James Stewart`.

Converting XML Data with XSLT

At the beginning of the previous section, I showed you how to use XSLT to translate an XML document into a sequence of `INSERT` commands. XSLT can convert any XML document into (just about) any other form.

The `xml2` contributed module includes an XSLT processor that you can invoke from within the PostgreSQL server. (Actually, `xml2` includes an interface to the `libxslt` package.) That means that you can use XSLT to convert XML documents stored inside of your database, without ever leaving the comfort of your favorite PostgreSQL client.

XSLT is often used to produce HTML web pages from XML documents, and in this section, I'll show you how to turn the XML documents stored in the `filminfo` table into user-friendly web pages.

To convert an XML document using an XSLT stylesheet, call the `xslt_process()` function. `xslt_process()` expects two arguments:

```
xslt_process( document TEXT, stylesheet TEXT ) RETURNS TEXT
```

It may seem obvious, but it's worth pointing out that you can provide either argument as a `TEXT` literal, as an expression that evaluates to a

TEXT value, or as a reference to a column in the database. The most convenient way to use `xslt_process()` is to store both the XML document and the XSLT stylesheet in a PostgreSQL table (probably in two separate tables).

To start this exercise, I'll create a table that will hold XSLT stylesheets:

```
movies=# CREATE TABLE transforms( name VARCHAR, stylesheet TEXT );
CREATE TABLE
```

Next, I'll create a function named `stylesheet()` that will retrieve a stylesheet from the transforms table, given the name of the desired stylesheet:

```
movies=# CREATE FUNCTION stylesheet( name VARCHAR ) RETURNS TEXT AS
movies-# $$
movies$#     SELECT stylesheet FROM transforms WHERE name = $1
movies$# $$ LANGUAGE 'SQL';
CREATE FUNCTION
```

At this point, the `movies` database stores XML documents inside of the `filminfo` table and stores XSLT stylesheets in the `transforms` table. Listing 25.4 shows a script that will add a new stylesheet to the `transforms` table.

Listing 25.4. `movieOfTheWeek.sql`

Code View: [Scroll](#) / Show All

```
INSERT INTO transforms VALUES (
'movieOfTheWeek',

$$<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:output method="html"/>
  <xsl:template match="/film">
    <HTML>
      <HEAD><TITLE>Movie of the week</TITLE></HEAD>
      <STYLE TYPE="text/css">
        BODY {font-family: sans-serif}
        H1 {font-size: 20pt}
        H2 {font-size: 22pt; font-style: italic}
        LU {font-size: 16pt}
      </STYLE>
      <BODY>
        <H1>This week's movie</H1>
        <H2><xsl:value-of select="name"/></H2>

        <H1>Starring:</H1>

        <UL>
          <xsl:for-each select="leads/lead">
            <LI><xsl:value-of select="."/></LI>
          </xsl:for-each>
        </UL>

        <H1>Directed by:</H1>

        <UL>
          <xsl:for-each select="directors/director">
            <LI><xsl:value-of select="."/></LI>
          </xsl:for-each>
        </UL>

        <P>Released in: <xsl:value-of select="year"/></P>

      </BODY>
    </HTML>
  </xsl:template>
</xsl:stylesheet>$$ );
```

When you execute this script, you're storing an XSLT stylesheet in the `transforms` table. I won't walk you through the workings of the stylesheet; XSLT is a powerful but unfriendly language and I can't really do it justice in this book, so pick up a good XSLT book if you need more information. I will point out a very important rule, though—the XSLT processor included with `xml2` examines the first character in the stylesheet to decide whether you've given it a real stylesheet or a reference to a remote stylesheet. If the first character is a "<", the XSLT processor assumes that the string is the stylesheet. If the first character is anything other than a "<", the XSLT processor assumes that you've given it a URI that identifies the real stylesheet (that is, the string is treated as if it were something like "`http://example.com/the-real-stylesheet.xml`"). When you `INSERT` (or `UPDATE`) a stylesheet into a table, make sure that the opening "<" immediately follows the quote character(s)—you don't want a newline between the "\$\$" and the "<" or you'll spend a great deal of time increasing your four-letter vocabulary.

To apply the stylesheet shown in Listing 25.4, choose a film that you want to feature (say, *Casablanca*) and execute the following

commands:

```
movies=# \t
Showing only tuples.
movies=# \o movieOfTheWeek.html
movies=# SELECT xslt_process( description, stylesheet( 'movieOfTheWeek' ))
movies-# FROM filminfo
movies-# WHERE film_name = 'Casablanca';
movies=# \o
movies=# \t
Tuples only is off
```

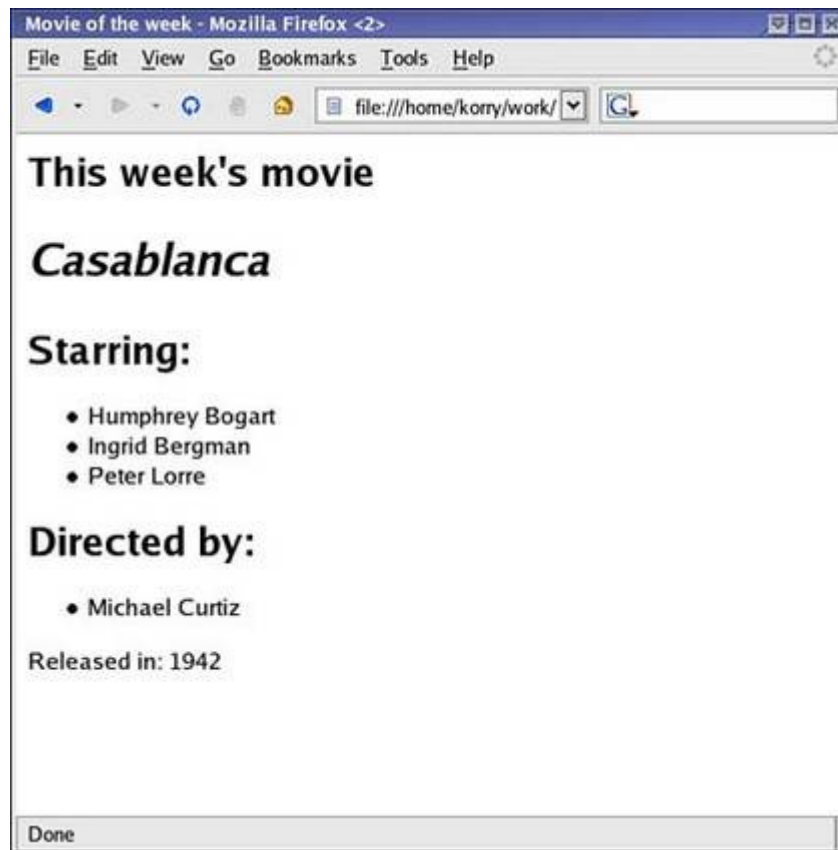
The first command (`\t`) turns on `psql`'s "tuple-only mode." In this mode, `psql` omits the column headers (and command responses) that you normally see when you execute a query—you only see the data values returned by each query. The second command (`\o movieOfTheWeek.html`) tells `psql` to create a new file named `movieOfTheWeek.html` (in the current directory) and write command results to that file. Once you've executed those two commands, `psql` will write raw query results (tuples only) to `movieOfTheWeek.html`.

As I mentioned earlier, `xslt_process()` expects two arguments. The first argument tells `xslt_process()` to convert the XML document found in `filminfo`'s `description` column. The second argument specifies the stylesheet that `xslt_process()` will use to control the conversion process. The `SELECT` command calls the `stylesheet()` function I defined earlier (at the beginning of this section) to retrieve the `movieOfTheWeek` stylesheet.

When the `SELECT` command completes, the `\o` command closes `movieOfTheWeek.html` and redirects `psql` output to your screen. The `\t` command turns tuple-only mode off again so that you'll see column headers (and command tags).

The overall result here is that you've produced an HTML document (`movieOfTheWeek.html`) by converting the an XML document using an XSLT stylesheet. If you view `movieOfTheWeek.html` in a web browser, you'll see a page similar to that shown in [Figure 25.1](#).

Figure 25.1. `movieOfTheWeek.html`.



Using Full-text Search

If you've ever used Google to search for a web site (and who hasn't), you've used a full-text search engine. A full-text search engine catalogs the words and phrases found in a set of documents and then lets you search for documents that match a given pattern. The `tsearch2` contributed module adds full-text search features to a PostgreSQL server. When you add the `tsearch2` package to a database, you get a small collection of new data types, a large collection of new functions (most of which you'll never invoke directly), and a new search operator. When you use `tsearch2`, the documents that you catalog and search are the values stored in your database. If you have `customers` table that stores miscellaneous notes about each customer, the `notes` column in each row might be a searchable document. If your database includes a `parts` list, you may want to catalog the description of each part. In `tsearch2` terms, a document is a string that you want to catalog and search. A pattern is a logical collection of words that you want to find. A pattern may be as simple as a single word. If a pattern contains multiple words, you can use pattern operators to define how the words relate to the documents that you want to search. For example, you can create a pattern that matches any document that contains every word in the pattern, a pattern that matches any document that contains any of the words in the pattern, a pattern that matches all documents that contain none of the words in the pattern, or any combination.

Before you try out any of the examples in this section, install `tsearch2` into your database and then execute the following command:

```
movies=# SELECT set_curcfg( 'default' );
set_curcfg
-----
(1 row)
```

If you don't call `set_curcfg()` first, the queries that I show you will fail with the error:

```
ERROR: could not find tsearch config by locale
```

I'll explain the `set_curcfg()` function in a moment, but let's look at a few sample queries first.

The `@@` operator is similar in concept to `LIKE`, `ILIKE`, or the regular-expression operators (`~`, `~*`, and so on): It compares a pattern against a string. To use the `@@` operator, you must convert the pattern into an object of type `tsquery` and the string you want to compare against into an object of type `tsvector`. For example, to search the `tapes` table for titles that include the word "Godfather":

```
movies=# SELECT tape_id, title FROM tapes WHERE
movies=#   to_tsvector( title ) @@ to_tsquery( 'Godfather' );
 tape_id | title
-----+-----
  AB-12345 | The Godfather
  AB-67472 | The Godfather
(2 rows)
```

If you want to search for all titles that include `Rear` and `Window`, the search pattern should look like this:

```
to_tsquery( 'Rear & Window' )
```

To search for all titles that include `Rear` or `Window`, separate the words with the `|` operator, like this:

```
to_tsquery( 'Rear | Window' )
```

To search for all titles that include the word `Window`, but not the word `Rear`, use the `!` operator:

```
to_tsquery( 'Rear & ! Window' )
```

You can combine operators to create complex patterns. Use parentheses to group expressions. For example, this pattern

```
to_tsquery( 'Island & (Earth | Gilligan)' )
```

will match `This Island Earth` and `Escape From Gilligan's Island`, but not `Escape From Devil's Island`.

It's easy to confuse the `tsvector` and `tsquery` data types since they appear to be so similar. They are, in fact, very different. A `tsvector` contains a catalog of the words that appear in a document. A `tsquery` contains a search pattern, not just a list of words. In fact, if you try to create a `tsquery` from a string that contains a list of words, `tsearch2` will reward you with an error message (`ERROR: syntax error`). Instead, you have to tell `tsearch2` how those words relate to the document that you're searching—you have to separate multiple words with a `tsquery` operator (`&`, `|`, or `!`).

Of course, if pattern matching was the only feature offered by `tsearch2`, it wouldn't be very exciting—after all, you can match patterns with `LIKE/ILIKE` or the regular-expression operators. `tsearch2` supports three distinct advantages over the other pattern matching mechanisms offered by PostgreSQL:

- Stop words
- Stemming

- Indexing

A stop word is a word that `tsearch2` automatically ignores when searching and indexing. If you've ever Googled for a phrase like `the world's best margarita recipe`, you may have noticed the following reply:

```
"the" is a very common word and was not included in your search.
```

"The" is a stop word—it's a word that would match just about every (English) document ever searched (other examples include "a," "and," "not," "some," and so on.). Stop words take up extra space (and slow down searches) without contributing to the task of identifying interesting documents. When you convert a search pattern into a `tsquery` object, `tsearch2` strips out any stop words that it finds in the pattern.

Stemming is the process of identifying word variations. When you create a search pattern into a `tsquery` object, `to_tsquery()` replaces each word in the pattern with its stem. For example, `donate`, `donation`, `donating`, `donates`, and `donated` are all variations of the same word. If `to_tsquery()` finds one of those variants in a search pattern, it replaces each occurrence with the stem: `donat`. That means that a search for `donate` will match documents that contain any variant of `donate`.

Of course, if you stem a search pattern, you must also stem the documents that you are searching through. `to_tsvector()` removes stops words and stems word variations using the same set of rules used by `to_tsquery()`.

The process of stemming, stopping, and cataloging the words in a document is expensive. When you execute a query that invokes `to_tsvector()` like this:

```
SELECT tape_id, title FROM tapes WHERE
    to_tsvector( title ) @@ to_tsquery( 'Godfather' );
```

the PostgreSQL server will stem, stop, and catalog every row in the table. If you execute the same query (or a similar query) again, the server has to stem, stop, and catalog every row a second time. You can greatly improve performance by building a `tsvector` for each document at the time you add the document to the database.

The `movies` sample database that we've been using in most of this book doesn't really contain enough data to thoroughly exercise `tsearch2`, but the `recalls` table presented in [Chapter 4](#), "Performance," does. The `recalls` table (in the `perf` database) contains 39,241 rows of information about automobile recalls. Each row contains three large `VARCHAR` fields that contain a description of a defect, the possible consequences of the defect, and the corrective action promised by the manufacturer.

To demonstrate the performance benefits offered by `tsearch2`, I'll use `tsearch2` to count the number of `recalls` that contain the word `hydraulic` (in the `desc_defect` column).

It takes my computer approximately 6.4 seconds to execute the following query:

```
SELECT COUNT(*) FROM recalls WHERE
    to_tsvector( desc_defect ) @@ to_tsquery( 'hydraulic' );
```

That query stems, stops, and catalogs every `desc_defect` in the `recalls` table, and identifies 808 rows that match the given pattern. If I execute the same query repeatedly, each iteration takes (approximately) the same amount of time.

Just for purposes of comparison, it takes approximately 2.7 seconds to perform a similar query using a regular-expression:

```
SELECT COUNT(*) FROM recalls WHERE
    desc_defect ~* 'hydraulic';
```

Again, if I execute the same query repeatedly, each iteration takes roughly the same amount of time.

Why does a `tsearch2`-based search take nearly two and a half times longer than a regular-expression search? Because the `to_tsvector()` function is stemming, stopping, and cataloging every word found in the `desc_defect` column. The regular-expression search simply scans through the `desc_defect` column and stops as soon as it finds the word `hydraulic`. In fact, the `tsearch2` and the regular-expression search identify a different set of matches (because of the stemming rules and parsing rules used by `tsearch2`).

It's usually a waste of time to stem, stop, and catalog every row for each `tsearch2` query, because the vast majority of the documents remain unchanged from query to query. Instead, I'll add a `tsvector` column to my `recalls` table and "precompute" the stem, stop, and catalog information required for a `tsearch2` query:

```
perf=# ALTER TABLE recalls
perf=#   ADD COLUMN fts_desc_defect TSVECTOR;
ALTER TABLE

perf=# UPDATE RECALLS SET
perf=#   fts_desc_defect = to_tsvector( desc_defect );
UPDATE 39241

perf=# VACUUM FULL ANALYZE recalls;
VACUUM
```

Now I can run a `tsearch2` query again, but this time, I search the new `fts_desc_defect` column instead:

```
SELECT COUNT(*) FROM recalls WHERE
fts_desc_defect @@ to_tsquery( 'hydraulic' );
```

Notice that I don't have to convert `fts_desc_defect` into a `tsvector` because it already is a `tsvector`. This query identifies the same set of rows selected by the first query, but this query runs in 0.22 seconds. That's a considerable improvement over the original query (6.7 seconds). But I can make it faster still.

`tsearch2` provides the infrastructure required to index `tsvector` values. To create an index that @@ can use:

Code View: [Scroll](#) / Show All

```
perf=# CREATE INDEX fti_desc_defect ON recalls USING GIST( fts_desc_defect );
CREATE INDEX

perf=# VACUUM FULL ANALYZE recalls;
VACUUM
```

Now when I search the `fts_desc_defect` column (using the same query), it takes less than .04 seconds (4/100ths of a second) to identify the same set of rows. PostgreSQL uses the index on `fts_desc_defect` to read only those rows that match the search pattern.

At this point, I've improved performance, but I've introduced a bug. If I search the `desc_defect` column, PostgreSQL has to stem, stop, and catalog every row in the `recalls` table (every time I search) and I get poor performance. If I search the pre-cataloged `fts_desc_defect` column, I get good performance. But what happens if I add a new row to the `recalls` table? Or `UPDATE` an existing row (and change the words in the `desc_defect` column)? Searching against `desc_defect` guarantees that I'll see the most recent data. Searching against `fts_desc_defect` guarantees that I'll see obsolete data. Fortunately, this problem is easy to fix—in fact, there are two different solutions.

The most obvious way to keep `fts_desc_defect` up-to-date is to create a `TRIGGER` that recomputes the `tsvector` whenever I add a new row or update an existing row. `tsearch2` even comes with a function that you can use to implement the trigger:

```
perf=# CREATE TRIGGER tg_fts_recalls
perf=#   BEFORE UPDATE OR INSERT ON RECALLS
perf=#   FOR EACH ROW
perf=#   EXECUTE PROCEDURE tsearch2( fts_desc_defect, desc_defect );
CREATE TRIGGER
```

The `tsearch2()` function expects two arguments: the name of a `tsvector` column and the name of a text (or other string-valued) column. The trigger will effectively call `ts_tovector(desc_defect)` and copy the result into the `fts_desc_defect` column.

I can test this trigger pretty easily:

Code View: [Scroll](#) / Show All

```
perf=# SELECT COUNT(*) FROM recalls fts_desc_defect @@ to_tsquery('hydraulic');
count
-----
      808
(1 row)

perf=# UPDATE recalls
perf=#   SET desc_defect = 'busted hydraulic line'
perf=#   WHERE record_id = 4909;
UPDATE 1

perf=# SELECT COUNT(*) FROM recalls fts_desc_defect @@ to_tsquery('hydraulic');
count
-----
      809
(1 row)
```

Another way to solve this problem is to drop the `fts_desc_defect` column—you don't need it to gain the benefits offered by pre-cataloging and indexing. Instead of adding an `fts_desc_defect` column and then creating an index that covers that column, just create a function-based index. First, I'll clean out the `tsvector` column that I added earlier:

```
perf=# DROP TRIGGER tg_fts_recalls ON recalls;
DROP TRIGGER

perf=# DROP INDEX fti_desc_defect;
DROP INDEX

perf=# ALTER TABLE recalls DROP COLUMN tg_fts_recalls;
ALTER TABLE
```

Now I'll create a new index function-based index:

```
perf=# CREATE INDEX fti_desc_defect ON recalls
perf=# USING GIST( to_tsvector( desc_defect ));
CREATE INDEX
```

As you might expect, it takes a while to create the function-based index. PostgreSQL reads through every row in the `recalls` table; invokes the `to_tsvector()` function to stem, stop, and catalog the `desc_defect` column; and then stores the result in the new index. Of course, if I add a new row to the `recalls` table (or update an existing row), PostgreSQL ensures that the index is kept up-to-date.

By creating a function-based index (or, more properly, an expression-based index), I eliminate the need for a trigger, I can get rid of the extra `tsvector` column (and save quite a bit of space), and I still get the performance boost offered by a pre-cataloged index.

Searching Multiple Columns

The `recalls` table contains three `VARCHAR` fields that we might want to search: `desc_defect` (a description of the defect), `con_defect` (possible consequences of the defect), and `cor_action` (the corrective action promised by the manufacturer). I could search all three columns using a query such as

```
perf=# SELECT COUNT(*) FROM recalls WHERE
perf=# to_tsvector(desc_defect) @@ to_tsquery('hydraulic' )
perf=# OR
perf=# to_tsvector(con_defect) @@ to_tsquery('hydraulic' )
perf=# OR
perf=# to_tsvector(cor_action) @@ to_tsquery('hydraulic' );
count
-----
902
(1 row)
```

That works, but it's not a simple query to write. Instead of searching through each column individually, I can string all three columns together and search through the concatenation:

```
perf=# SELECT COUNT(*) FROM recalls WHERE
perf=# to_tsvector(desc_defect || con_defect || cor_action)
perf=# @@ to_tsquery('hydraulic' )
count
-----
902
(1 row)
```

Unfortunately, the simplicity of this query is misleading: It works for the `recalls` table, but it won't produce correct results if your documents contain any `NULL` values. (The document columns in `recalls` contain no `NULL` values.) A `NULL` value works out this query because the concatenation operator (`||`) assumes that any string appended to a `NULL` results in a `NULL`. In short: `'hydraulic line' || NULL || 'hydraulic piston'` evaluates to `NULL`. That means that a `NULL` value in `desc_defect`, `con_defect`, or `cor_action` would effectively hide the other values in that row. To fix this problem, I can rewrite the query using the `coalesce()` function to map `NULL` values into some other value (in this case, an empty string):

```
perf=# SELECT COUNT(*) FROM recalls WHERE
perf=# to_tsvector(
perf=# COALESCE( desc_defect, '' ) || ' ' ||
perf=# COALESCE( con_defect, '' ) || ' ' ||
perf=# COALESCE( cor_action, '' ))
perf=# @@ to_tsquery('hydraulic' )
count
-----
902
(1 row)
```

This query takes about 15 seconds on my computer. (The `OR` version takes the same amount of time.) So much for simplicity—the `OR` version was easier to write and easier to understand.

However, I've been leading you down this tortuous path for a good reason. Remember that in the previous section I showed you how to create an expression-based index? I can create an index defined by the rather complex expression that I wrote in that last query, and PostgreSQL will use that index to search for patterns in `desc_defect`, `con_defect`, and `cor_action`.

```
perf=# CREATE INDEX fti_recalls ON recalls USING GIST(
perf=# to_tsvector(
perf=# COALESCE( desc_defect, '' ) || ' ' ||
perf=# COALESCE( con_defect, '' ) || ' ' ||
perf=# COALESCE( cor_action, '' ));
CREATE INDEX
```

Now, when I run the previous query (the one with all the `COALESCE` noise in it), I see the results in 0.44 seconds.

Simplifying tsearch2 with Customized Functions

`tsearch2` queries tend to be rather unwieldy. You can simplify `tsearch2` by creating a few wrapper functions that hide the details of the complicated queries. For example, I can create a function named `documents()` that will return the (properly coalesced) concatenation of `desc_defect`, `con_defect`, and `cor_action`:


```

perf=# CREATE FUNCTION documents( recall recalls ) RETURNS TSVECTOR AS
perf=# $$
perf$# SELECT
perf$# COALESCE( $1.desc_defect, '' ) || ' ' ||
perf$# COALESCE( $1.con_defect, '' ) || ' ' ||
perf$# COALESCE( $1.cor_action, '' );
perf$# $$ LANGUAGE 'SQL' IMMUTABLE;

```

I can also define a function named `document()` (singular this time) that converts `documents()` into a `tsvector`:

```

perf=# CREATE FUNCTION document( recall recalls ) RETURNS TSVECTOR AS
perf=# $$
perf$# SELECT to_tsvector( documents( $1 ));
perf$# $$ LANGUAGE 'SQL' IMMUTABLE;

```

Now I can use the `document()` function in conjunction with the `@@` operator:

```

perf=# SELECT COUNT(*) FROM recalls r
perf=# WHERE document(r) @@ to_tsquery( 'hydraulic' );
count
-----
      902
(1 row)

```

That's much easier to type in and much easier to read. I can even create a function-based index based on `document()`:

```

perf=# DROP INDEX fti_recalls;
DROP INDEX

perf=# CREATE INDEX fti_recalls USING GIST( document( recalls ));
CREATE INDEX

```

The `document()` function returns a `tsvector` based on `desc_defect`, `con_defect`, and `cor_action`. The only thing that I can do with a `tsvector` is search it, so I may as well add another function that simplifies the search:

Code View: [Scroll](#) / [Show All](#)

```

perf=# CREATE FUNCTION find_recalls( pattern text ) RETURNS SETOF RECALLS AS
perf=# $$
perf$# SELECT * FROM recalls WHERE
perf$# to_tsvector(
perf$# COALESCE( desc_defect, '' ) || ' ' ||
perf$# COALESCE( con_defect, '' ) || ' ' ||
perf$# COALESCE( cor_action, '' ))
perf$# @@ to_tsquery( $1 );
perf$# $$ LANGUAGE 'SQL';
CREATE FUNCTION

```

Now I can simply invoke `find_recalls()` to search for a pattern in `desc_defect`, `con_defect`, or `cor_action`:

```

perf=# SELECT COUNT(*) FROM find_recalls( 'hydraulic' );
count
-----
      902
(1 row)

```

Searching for Phrases

You can't use `tsearch2` to search for phrases—`tsearch2` catalogs the individual words in a document, but doesn't keep enough information to know when one word directly follows another. You can't use `tsearch2`, for example, to search for a phrase such as `hydraulic line`. (Enclosing the phrase in quotes won't help.) You can search for a pattern such as `hydraulic & line`, but that will match all documents that contain both words, even if `line` appears before `hydraulic` or if `hydraulic` is separated from `line` by a number of other words.

To search for a phrase, you have to use `LIKE`, `ILIKE`, or a regular-expression operator. For example

```

perf=# SELECT COUNT(*) FROM recalls
perf=# WHERE documents(recalls) ~* 'hydraulic line';
count
-----
      53
(1 row)

```

But you can still use `tsearch2` to speed up a phrase search. It stands to reason that any document that contains the phrase `hydraulic line` will contain the individual words `hydraulic` and `line`, right? Looking at it the other way around, a document cannot contain the phrase `hydraulic line` unless it contains the individual words `hydraulic` and `line`. You already know how to identify the set of rows that contain the

words hydraulic and line—just add `AND document(recalls) @@ to_tsquery('hydraulic & line')` to the `WHERE` clause.

The original version of this query (regular-expression only) takes about 8.2 seconds to run on my computer. By adding `tsearch2` to a regular-expression based phrase search, I can drastically reduce the number of rows that the server will have to search. The new query looks like this:

```
perf=# SELECT COUNT(*) FROM recalls
perf=#   WHERE documents(recalls) ~* 'hydraulic line'
perf=#   AND document(recalls) @@ to_tsquery('hydraulic & line');
count
-----
      53
(1 row)
```

The new version takes 0.12 seconds to identify the same set of rows. I can use the `timer` utility (described in [Chapter 4](#)) to see what the server does with each query. Here's the `timer` output from the first (regular-expression only) query:

Code View: [Scroll](#) / [Show All](#)

```
$ timer "SELECT COUNT(*) FROM recalls
>       WHERE documents(recalls) ~* 'hydraulic line'"
+-----+-----+-----+-----+-----+-----+-----+-----+
|               SEQUENTIAL I/O               |               INDEXED I/O               |
| scans| tuples | heap_blks | cached| scans| tuples | idx_blks | cached|
+-----+-----+-----+-----+-----+-----+-----+-----+
| recalls |      1|  39241 |      5399 |      0 |      0 |      0 |      0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

You can see that the server evaluated this query by scanning every one of the 39,241 rows in the table.

Code View: [Scroll](#) / [Show All](#)

```
$ timer "SELECT COUNT(*) FROM recalls
>       WHERE documents(recalls) ~* 'hydraulic line'"
>       AND document(recalls) @@ to_tsquery('hydraulic & line');
+-----+-----+-----+-----+-----+-----+-----+-----+
|               SEQUENTIAL I/O               |               INDEXED I/O               |
| scans| tuples | heap_blks | cached| scans| tuples | idx_blks | cached|
+-----+-----+-----+-----+-----+-----+-----+-----+
| recalls |      0 |      0 |      71 |      5 |      1 |     126 |     173 |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

The server first uses an index scan (on `fts_recalls`) to quickly identify the 126 rows that satisfy the `tsearch2` criteria and then matches each of those rows against the regular expression.

Configuring `tsearch2`

Each time you start a new client session, `tsearch2` tries to find a configuration that matches the server's locale. `tsearch2` configurations are stored in a small collection of tables: `pg_ts_cfg`, `pg_ts_cfgmap`, `pg_ts_parser`, and `pg_ts_dict`. `tsearch2` comes with three predefined configurations: `default`, `default_russian`, and `simple`:

```
perf=# SELECT * FROM pg_ts_cfg
      ts_name | prs_name | locale
+-----+-----+-----+
 default     | default  | C
 default_russian | default  | ru_RU.KOI8-R
 simple      | default  |
(3 rows)
```

To find the proper configuration, `tsearch2` searches `pg_ts_cfg` for a row where the `locale` column matches your server's locale. If it can't find a matching configuration, you'll see a message stating `ERROR: could not find tsearch config by locale`. You can find the locale used by your server with the following query:

```
perf=# SELECT setting FROM pg_settings WHERE name = 'lc_ctype';
setting
-----
en_US.UTF-8
(1 row)
```

If your server's locale doesn't match any of the locales in `pg_ts_cfg`, you have four options:

- Specify a configuration on every call to `to_tsvector()` and `to_tsquery()`

- Call the `set_curcfg()` function at the beginning of every client session
- Clone an existing configuration
- Create a new configuration from scratch

The first option is simple, but it complicates your code. The `to_tsvector()` function comes in two flavors^[2]. To use the first flavor, you invoke `to_tsvector()` with a single string argument and it converts that string into a `tsvector` using the "current" configuration. To use the second flavor, call `ts_vector()` with two arguments: the name of a configuration and a string. `to_tsvector()` will convert the string into a `tsvector` using the configuration that you specified in the first argument. `ts_toquery()` comes in two flavors as well.

^[2] `tsearch2` provides a third flavor for `to_tsvector()` and `to_tsquery()`. You must know the OID of a `pg_ts_cfg` row to use the third form.

The second option is inconvenient and somewhat dangerous. You have to call `set_curcfg()` in every client session that might use `tsearch2`—if you've created a trigger or index based on `tsearch2`, that means you have to call `set_curcfg()` in any session that could update a cataloged column.

Cloning an existing configuration is often the easiest and safest choice. Cloning an existing configuration is a two-step process. First, you add a new row to the `pg_ts_cfg` table, then you make a copy of the corresponding entries in the `pg_ts_cfgmap` table. For example, if your server's locale is `en_US.UTF8`, you can clone the `default` configuration with the following commands:

Code View: [Scroll](#) / [Show All](#)

```
perf=# INSERT INTO pg_ts_cfg VALUES( 'default_enUS', 'default', 'en_US.UTF8');
INSERT

perf=# INSERT INTO pg_ts_cfgmap
perf=#   SELECT 'default_enUS', tok_alias, dict_name
perf=#   FROM pg_ts_cfgmap WHERE ts_name = 'default';
INSERT
```

The first command creates a new (empty) configuration named `default_enUS`—`tsearch2` will select this configuration when the server's locale is `en_US.UTF8`. The second command clones the `default` entries in `pg_ts_cfgmap`, creating an identical set of entries that belong to `default_enUS`. Once you've created a clone that matches your server's locale, you should be able to use `tsearch2` without specifying an explicit configuration in each call to `to_tsquery()` and `to_tsvector()`.

Creating a new configuration from scratch is not too complex, but you'll need an understanding of the stemming, stopping, and cataloging process before you start.

When you create a `tsvector` from a text string, `tsearch2` starts by invoking a parser that picks apart the text string into its component parts. `tsearch2` comes with a single parser (named `default`), but you can write your own parser if you have special requirements. The `default` parser was designed to parse plain-text and HTML documents: It knows how to process HTML tags, HTTP headers, email and host addresses, and so on. The parser identifies and classifies each token in a text string. For example, given the string "send 42 messages to bruce@example.com", the default parser will identify four words (send, messages, and to), one unsigned integer (42), and an email address (bruce@example.com). The default parser classifies each token into one (or more) of the categories shown in [Table 25.1](#)^[3].

^[3] The default parser also defines three categories for words and word fragments composed of Cyrillic characters: `nlword`, `nlhword`, and `part-nlhword`.

Table 25.1. `tsearch2` Lexical Categories

Category	Description	Examples
lword	Any word composed entirely of alphabetic characters	bruce
word	Any word composed of alphabetic and numeric characters	bruce42, postgres81
email	An Internet email address (user@host)	bruce@example.com
url	An HTTP or FTP URL	http://www.postgresql.org/index.html ftp://ftp.postgresql.org/index.html
host	An Internet hostname	www.postgresq.orglocalhost.localdomain
sfloat	A floating point number in scientific notation	325.667E12 6.626E-34
version	A generic version number (a number with more than one decimal point)	8.0.0 2.6.9.1
part_hword	Parts of a hyphenated word	post-gres-sql8
lpart_hword	Latin parts of a hyphenated word	post-gre-sql8
blank	Whitespace and any characters not matched by other rules	(parens are considered blanks) \$so are other special characters!

tag	An HTML tag	<tr>
http	The protocol component of an HTTP URL	http://www.postgresql.org
hword	A hyphenated word	postgre-sql8
lhword	A hyphenated Latin word	postgre-sql
uri	A uniform resource identifier (usually the filename component of a URL)	http://www.postgresql.org/index.html
file	A relative or absolute Linux/Unix pathname	/tmp/README.txt../ README.txt
float	A floating-point number	3.14159 6.626
int	A signed integer	-32 +45
uint	An unsigned integer	32 45
entity	An HTML entity	 ,

When the parser finishes tokenizing and classifying the text string, it ends up with a collection of token values and each token is assigned to a category. Some of the "words" in the text string may result in multiple tokens. For example, the string <http://www.postgresql.org/index.html> produces four tokens (you can call the `ts_debug()` function to see the result of the parsing process):

```
perf=# SELECT token, tok_type
perf=# FROM ts_debug('http://www.postgresql.org/index.html');
-----+-----
http:// | http
www.postgresql.org/index.html | url
www.postgresql.org | host
/index.html | uri
(4 rows)
```

Next, the parser iterates through the list of tokens and weeds out any that are deemed uninteresting. To decide which tokens to discard, `to_tsvector()` uses the token type (and the configuration name) to locate a record in the `pg_ts_cfgmap` table. If `to_tsvector()` can't find a matching entry in `pg_ts_cfgmap`, it discards the token. For example, given the tokens parsed from <http://www.postgresql.org/index.html>, `to_tsvector()` finds:

```
perf=# SELECT * FROM pg_ts_cfgmap
perf=# WHERE ts_name = 'default'
perf=# AND tok_alias IN( 'http', 'url', 'host', 'uri' );
-----+-----+-----
ts_name | tok_alias | dict_name
-----+-----+-----
default | url       | {simple}
default | host      | {simple}
default | uri       | {simple}
(3 rows)
```

Notice that `tsearch2` won't find an entry for `ts_name = 'default'` and `tok_alias = 'http'`, so it discards that token (the `http://` header). The default configuration discards blank, tag, http, and entity tokens. Discarding a "word" based on its token classification is similar to stopping an entire category of words. Discarded tokens are not cataloged by `tsearch2`, so you won't be able to search for them. Of course, you can tell `tsearch2` that you want it to catalog a given category by adding that category to the `pg_ts_cfgmap` table. Similarly, you can tell `tsearch2` to ignore a given category (say, the `file` category) by removing that category from `pg_ts_cfgmap`.

For each token that makes it through the `pg_ts_cfgmap` filter, `tsearch2` starts the stemming and stopping process. When `to_tsvector()` finds an entry in `pg_ts_cfgmap` that matches the configuration name and token type, that entry identifies a dictionary processor. `to_tsvector()` feeds the token into that dictionary processor and adds the result (if any) to the `tsvector`. The dictionary processor may stem the token by translating it into a new token. The dictionary processor may instead stop the word by returning a `NULL` value. Or, the dictionary processor may pass the token through without modification.

The `tsearch2` package comes with five sample dictionary processors.

The `simple` dictionary processor converts each token into lowercase characters and the searches for the result in a list of stop words—if it finds the (lowercased) token in the list, it returns `NULL`, otherwise it returns the lowercased token to `to_tsvector()` (and `to_tsvector()` adds the token to the `tsvector` that it's building). `tsearch2` installs the simple dictionary processor with an empty stop word list (which means that every token makes it through the simple dictionary after it's been translated to lowercase). To add a stop word list (which is just a newline-separated list of words), save the name of your stop word file in the `dict_initoption` column of the `pg_ts_dict` row corresponding to the `simple` dictionary processor. For example, if you've stored a list of stopwords in a file named `/usr/share/stopwords.english`, execute the following command:

Code View: [Scroll](#) / [Show All](#)

```
perf=# UPDATE pg_ts_dict SET dict_initoption = '/usr/share/stopwords.english';
UPDATE
```

The `en_stem` dictionary processor handles stop words and stemming. `en_stem` searches for the token in a list of stop words and discards the

token if found. (Like the simple dictionary processor, `en_stem` finds the stop word list in its `pg_ts_dict.dict_initoption`.) If the token is not found in the stop word list, `en_stem` tries to convert the token into its root form by stripping off common English prefixes and suffixes. For example, `en_stem` converts `donate`, `donation`, `donating`, `donates`, and `donated` into the stem `donat`. `to_tsvector()` stores the stem in the `tsvector` that it's building. If you search for the word `donate`, `tsearch2` will match `donate`, `donation`, `donating`, `donates`, and `donated`.

The `ru_stem` dictionary processor is identical to the `en_stem` processor except it stems each token using rules designed for Russian text. (You would most likely use a different list of stop words too.)

The `synonym` dictionary processor doesn't do any stop word processing (or stemming). When you feed a token to the synonym processor, it searches for a match in a list of word-synonym pairs. If it finds a match, the processor returns the synonym. For example, given the list of synonyms:

```
zaurus    pda
newton    pda
pocketpc  pda
nokia     phone
treo      phone
```

The `synonym` processor will translate `zaurus`, `newton`, and `pocketpc` into `pda`, and will translate `nokia` and `treo` into `phone`. If `synonym` can't find a match in the list, it returns `NULL`.

The last dictionary processor is named `ispell_template`. `ispell_template` is based on the `ispell` program and it searches for each token in a separate dictionary file (not included with `tsearch2`). If `ispell_template` finds the token (or a variant of the token) in the dictionary, it returns the stemmed form of the word to `to_tsvector()`. If `ispell_template` can't find the token (or a variant of the token) in the dictionary, it returns `NULL` (and the token is discarded). `ispell_template` also uses a stop word list to filter out common words. There's an important difference between `ispell_template` and `en_stem`. Both dictionary processors convert tokens into stem form, but `ispell_template` will discard any token that it can't find in the dictionary: `en_stem`, on the other hand, simply passes through any token that it can't stem. The `ispell_template` processor won't work until you connect it to a dictionary—a process described in the "Tsearch Introduction" document that comes with `tsearch2`.

You can string multiple dictionary processors together by listing each one in the `pg_ts_cfgmap.dictname` column. For example, to apply the `synonym` processor and then the `en_stem` processor to every `lword` token:

```
perf=# UPDATE pg_ts_cfgmap
perf=#   SET dict_name = '{"synonym","en_stem"}
perf=#   WHERE ts_name = 'default_enUS' AND tok_alias = 'lword';
UPDATE
```

`tsearch2` tries each dictionary processor, in order, and stops as soon as a processor returns a non-`NULL` value.

Now you know how all of the pieces fit together. `tsearch2` uses the server's locale to find a configuration (in the `pg_ts_cfg` table). The configuration identifies a parser. `tsearch2` uses that parser to split a text string into a set of tokens and assigns a category to each token. The `pg_ts_cfgmap` maps each configuration/token category combination into the name of a dictionary processor. (If a combination is not found in `pg_ts_cfgmap`, `tsearch2` discards all tokens of that category.) The dictionary processor (typically) filters each token through a list of stop words and then stems anything that makes it through the filter.

To create a new configuration, you can write a new parser, change the `pg_ts_cfgmap` to include (or exclude) token categories, modify the `pg_ts_cfgmap` to apply a different dictionary processor to a token category, implement a new dictionary processor, or modify the list of stop words used by a dictionary. If you use the `synonym` dictionary processor, you can also modify the synonym map. In most cases, you won't need to write any code (unless you find that you have to implement a new parser or dictionary processor); just adjust a configuration table (or external file). If you do write a new parser or dictionary processor, consider donating it to the PostgreSQL community so other users can benefit from your efforts.

`tsearch2` offers a number of other features that I haven't described here. You know that `tsearch2` can identify the documents that match a given pattern—`tsearch2` can also rank the matches according to relevance. (Check out the `rank()` and `rank_cd()` functions.) When `tsearch2` finds a document that matches a pattern, you can ask the `headline()` function to produce a string that highlights the search words in context. See the `tsearch2` documentation for more details.

If `tsearch2` doesn't have what you need, check out the OpenFTS package. OpenFTS is a user-friendly wrapper around `tsearch2`. You can use OpenFTS to expose the documents in your database to users that may not know how to formulate SQL queries (and may not understand the results). You can find OpenFTS at openfts.sourceforge.net.