A Practical Guide to the Advanced Open Source Database

PostgreSQL

Up and Running

O'REILLY®

Regina Obe & Leo Hsu

PostgreSQL: Up and Running

If you're thinking about migrating to the PostgreSQL open source database system, this guide provides a concise overview to help you quickly understand and use PostgreSQL's unique features. Not only will you learn about the enterprise class features in the 9.2 release, you'll also discover that PostgeSQL is more than just a database system—it's also an impressive application platform.

With numerous examples throughout this book, you'll learn how to achieve tasks that are difficult or impossible in other databases. If you're an existing PostgreSQL user, you'll pick up gems you may have missed along the way.

- Learn basic administration tasks, such as role management, database creation, backup, and restore
- Apply the psql command-line utility and the pgAdmin graphical administration tool
- Explore PostgreSQL tables, constraints, and indexes
- Learn powerful SQL constructs not generally found in other databases
- Use several different languages to write database functions
- Tune your queries to run as fast as your hardware will allow
- Query external and variegated data sources with Foreign Data Wrappers
- Learn how to replicate data, using built-in replication features

Twitter: @oreillymedia
facebook.com/oreilly

US $19.99 CAN $20.99
ISBN: 978-1-449-32633-3

O'REILLY®
oreilly.com

PostgreSQL: Up and Running

Regina Obe and Leo Hsu

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

PostgreSQL: Up and Running
by Regina Obe and Leo Hsu

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Meghan Blanchette
Production Editor: Iris Febres
Proofreader: Iris Febres

Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

Revision History for the First Edition:

2012-07-02	First release
2012-09-07	Second release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449326333* for release details.

ISBN: 978-1-449-32633-3

[LSI]

1346956672

Table of Contents

Preface

PostgreSQL is an open source relational database management system that began as a University of California, Berkeley project. It was originally under the BSD license, but is now called the PostgreSQL License (TPL). For all intents and purposes, it's BSD licensed. It has a long history, almost dating back to the beginning of relational databases.

It has enterprise class features such as SQL windowing functions, the ability to create aggregate functions and also utilize them in window constructs, common table and recursive common table expressions, and streaming replication. These features are rarely found in other open source database platforms, but commonly found in newer versions of the proprietary databases such as Oracle, SQL Server, and IBM DB2. What sets it apart from other databases, including the proprietary ones we just mentioned, is the ease with which you can extend it without changing the underlying base—and in many cases, without any code compilation. Not only does it have advanced features, but it performs them quickly. It can outperform many other databases, including proprietary ones for many types of database workloads.

In this book, we'll expose you to the advanced ANSI-SQL features that PostgreSQL offers. and the unique features PostgreSQL has that you won't find in other databases. If you're an existing PostgreSQL user or have some familiarity with PostgreSQL, we hope to show you some gems you may have missed along the way; or features found in newer PostgreSQL versions that are not in the version you're using. If you have used another relational database and are new to PostgreSQL, we'll show you some parallels with how PostgreSQL handles tasks compared to other common databases, and demonstrate feats you can achieve with PostgreSQL that are difficult or impossible to do in other databases. If you're completely new to databases, you'll still learn a lot about what PostgreSQL has to offer and how to use it; however, we won't try to teach you SQL or relational theory. You should read other books on these topics to take the greatest advantage of what this book has to offer.

This book focuses on PostgreSQL versions 9.0 to 9.2, but we will cover some unique and advanced features that are also present in prior versions of PostgreSQL.

What Makes PostgreSQL Special and Why Use It?

PostgreSQL is special because it's not just a database: it's also an application platform —and an impressive one at that.

PostgreSQL allows you to write stored procedures and functions in several programming languages, and the architecture allows you the flexibility to support more languages. Example languages that you can write stored functions in are SQL (built-in), PL/pgSQL (built-in), PL/Perl, PL/Python, PL/Java, and PL/R, to name a few, most of which are packaged with many distributions. This support for a wide variety of languages allows you to solve problems best addressed with a domain or more procedural language; for example, using R statistics functions and R succinct domain idioms to solve statistics problems; calling a web service via Python; or writing map reduce constructs and then using these functions within an SQL statement.

You can even write aggregate functions in any of these languages that makes the combination more powerful than you can achieve in any one, straight language environment. In addition to these languages, you can write functions in C and make them callable, just like any other stored function. You can have functions written in several different languages participating in one query. You can even define aggregate functions with nothing but SQL. Unlike MySQL and SQL Server, no compilation is required to build an aggregate function in PostgreSQL. So, in short, you can use the right tool for the job even if each sub-part of a job requires a different tool; you can use plain SQL in areas where most other databases won't let you. You can create fairly sophisticated functions without having to compile anything.

The custom type support of PostgreSQL is sophisticated and very easy to use, rivaling and often outperforming most other relational databases. The closest competitor in terms of custom type support is Oracle. You can define new data types in PostgreSQL that can then be used as a table column. Every data type has a companion array type so that you can store an array of a type in a data column or use it in an SQL statement. In addition to the ability of defining new types, you can also define operators, functions, and index bindings to work with these. Many third-party extensions for PostgreSQL take advantage of these fairly unique features to achieve performance speeds, provide domain specific constructs to allow shorter and more maintainable code, and accomplish tasks you can only fantasize about in other databases.

If building your own types and functions is not your thing, you have a wide variety of extensions to choose from, many of which are packaged with PostgreSQL distros. PostgreSQL 9.1 introduced a new SQL construct, `CREATE EXTENSION`, which allows you to install the many available extensions with a single SQL statement for each in a specific database. With `CREATE EXTENSION`, you can install in your database any of the aforementioned PL languages and popular types with their companion functions and operators, like hstore, ltree, postgis, and countless others. For example, to install the popular PostgreSQL key-value store type and its companion functions and operators, you would type:

```
CREATE EXTENSION hstore;
```

In addition, there is an SQL command you can run—see "Extensions and Contribs" on page 18—to see the list of available and installed extensions.

Many of the extensions we mentioned, and perhaps even the languages we discussed, may seem like arbitrary terms to you. You may recognize them and think, "Meh, I've seen Python, and I've seen Perl... So what?" As we delve further, we hope you experience the same "WOW" moments we have come to appreciate with our many years of using PostgreSQL. Each update treats us to new features, eases usability, brings improvements in speed, and pushes the envelope of what is possible with a database. In the end, you will wonder why you ever used any other relational database, when PostgreSQL does everything you could hope for—and does it for free. No more reading the licensing cost fine print of those other databases to figure out how many dollars you need to spend if you have 8 cores on your server and you need X,Y, Z functionality, and how much it will cost you when you get 16 cores.

On top of this, PostgreSQL works fairly consistently across all supported platforms. So if you're developing an app you need to resell to customers who are running Linux, Mac OS X, or Windows, you have no need to worry, because it will work on all of them. There are binaries available for all if you're not in the mood to compile your own.

Why Not PostgreSQL?

PostgreSQL was designed from the ground up to be a server-side database. Many people do use it on the desktop similarly to how they use SQL Server Express or Oracle Express, but just like those it cares about security management and doesn't leave this up to the application connecting to it. As such, it's not ideal as an embeddable database, like SQLite or Firebird.

Sadly, many shared-hosts don't have it pre-installed, or have a fairly antiquated version of it. So, if you're using shared-hosting, you're probably better off with MySQL. This may change in the future. Keep in mind that virtual, dedicated hosting and cloud server hosting is reasonably affordable and getting more competitively priced as more ISPs are beginning to provide them. The cost is not that much more expensive than shared hosting, and you can install any software you want on them. Because of these options, these are more suitable for PostgreSQL.

PostgreSQL does a lot and a lot can be daunting. It's not a dumb data store; it's a smart elephant. If all you need is a key value store or you expect your database to just sit there and hold stuff, it's probably overkill for your needs.

For More Information on PostgreSQL

This book is geared at demonstrating the unique features of PostgreSQL that make it stand apart from other databases, as well as how to use these features to solve real world

problems. You'll learn how to do things you never knew were possible with a database. Aside from the cool "Eureka!" stuff, we will also demonstrate bread-and-butter tasks, such as how to manage your database, how to set up security, troubleshoot performance, improve performance, and how to connect to it with various desktop, command-line, and development tools.

PostgreSQL has a rich set of online documentation for each version. We won't endeavor to repeat this information, but encourage you to explore what is available. There are over 2,250 pages in the manuals available in both HTML and PDF formats. In addition, fairly recent versions of these online manuals are available for hard-copy purchase if you prefer paper form. Since the manual is so large and rich in content, it's usually split into a 3-4 volume book set when packaged in hard-copy form.

Below is a list of other PostgreSQL resources:

- *Planet PostgreSQL* is a blog aggregator of PostgreSQL bloggers. You'll find PostgreSQL core developers and general users show-casing new features all the time and demonstrating how to use existing ones.
- *PostgreSQL Wiki* provides lots of tips and tricks for managing various facets of the database and migrating from other databases.
- *PostgreSQL Books* is a list of books that have been written about PostgreSQL.
- *PostGIS in Action Book* is the website for the book we wrote on PostGIS, the spatial extender for PostgreSQL.

Conventions Used in This Book

The following typographical conventions are used in this book:

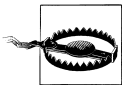
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*PostgreSQL: Up and Running* by Regina Obe and Leo Hsu (O'Reilly). Copyright 2012 Regina Obe and Leo Hsu, 978-1-449-32633-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://bit.ly/PostgreSQL_UR

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1
The Basics

In this chapter, we'll cover the basics of getting started with PostgreSQL. This includes where to get binaries and drivers, what's new and exciting in the latest 9.2 release, common administration tools, PostgreSQL nomenclature, and where to turn for help.

Where to Get PostgreSQL

Years ago, if you wanted PostgreSQL, you had to compile it from source. Thankfully, those days are gone. Granted, you can still compile should you so choose, but most users nowadays get their PostgreSQL with a prepackaged installer. A few clicks or keystrokes, and you're on your way in 10 minutes or less.

If you're installing PostgreSQL for the first time and have no existing database to upgrade, you should always install the latest stable release version for your OS. *http://www.postgresql.org/download* maintains a listing of places where you can download PostgreSQL binaries. In "Installation Guides and Distributions" on page 133, you'll find installation guides and some other additional custom distributions that people we've talked to seem to like.

Notable PostgreSQL Forks

The fact that PostgreSQL has MIT/BSD style licensing makes it a great candidate for forking. Various groups have done exactly that over the years. Some have contributed their changes. Netezza, a popular database choice for data warehousing workloads, in its inception was a PostgreSQL fork. GreenPlum, used for data warehousing and analyzing petabytes of information, was a spinoff of Bizgres, which was a community-driven spinoff of PostgreSQL focused on Big Data. PostgreSQL Advanced Plus by EnterpriseDb is a fork of the PostgreSQL codebase—it adds Oracle syntax and compatibility features to woo Oracle users. EnterpriseDb does provide funding to the PostgreSQL community, and for this we're grateful.

All the aforementioned are proprietary, closed source forks. tPostgres and Postgres-XC are two budding forks that we find interesting with open source licensing. tPostgres braches off PostgreSQL 9.2 and targets Microsoft SQL Server users. For instance, with tPostgres, you can write functions using T-SQL. Postgres-XC is a cluster server providing write-scalable, synchronous multi-master replication. What makes Postgres-XC special is that it supports distributed processing and replication. It is now at version 1.0.

Administration Tools

There are three popular tools for managing PostgreSQL and these are supported by PostgreSQL core developers; they tend to stay in synch with PostgreSQL versions. In addition, there are plenty of commercial offerings as well.

psql
: psql is a command-line interface for writing queries and managing PostgreSQL. It comes packaged with some nice extras, such as an import and export commands for delimited files, and a reporting feature that can generate HTML output. psql has been around since the beginning of PostgreSQL and is a favorite of hardcore PostgreSQL users. Newer converts who are more comfortable with GUI tools tend to favor pgAdmin.

pgAdmin
: This is the widely used, free, graphical administration tool for PostgreSQL. You can download it separately from PostgreSQL. pgAdmin runs on the desktop and can connect to multiple PostgreSQL servers regardless of version or OS. Even if you have your database server on a window-less Unix-based server, install pgAdmin and you'll find yourself armed with a fantastic GUI. pgAdmin is pictured in Figure 1-1.

  Some installers, such as those offered by EnterpriseDB, package pgAdmin with the database server install. If you're unfamiliar with PostgreSQL, you should definitely start with pgAdmin. You'll get a great overview and gain an appreciation of the richness of PostgreSQL just by exploring all the database objects in the main interface. If you're coming from SQL Server and used Management Studio, you'll feel right at home.

PHPPgAdmin
: PHPPgAdmin, pictured in Figure 1-2, is a free, web-based administration tool patterned after the popular PHPMyAdmin for MySQL. PostgreSQL has many more kinds of database objects than MySQL, as such PHPPgAdmin is a step up from PHPMyAdmin with additions to manage schemas, procedural languages, casts, operators, and so on. If you've used PHPMyAdmin, you'll find PHPPgAdmin to be nearly identical.

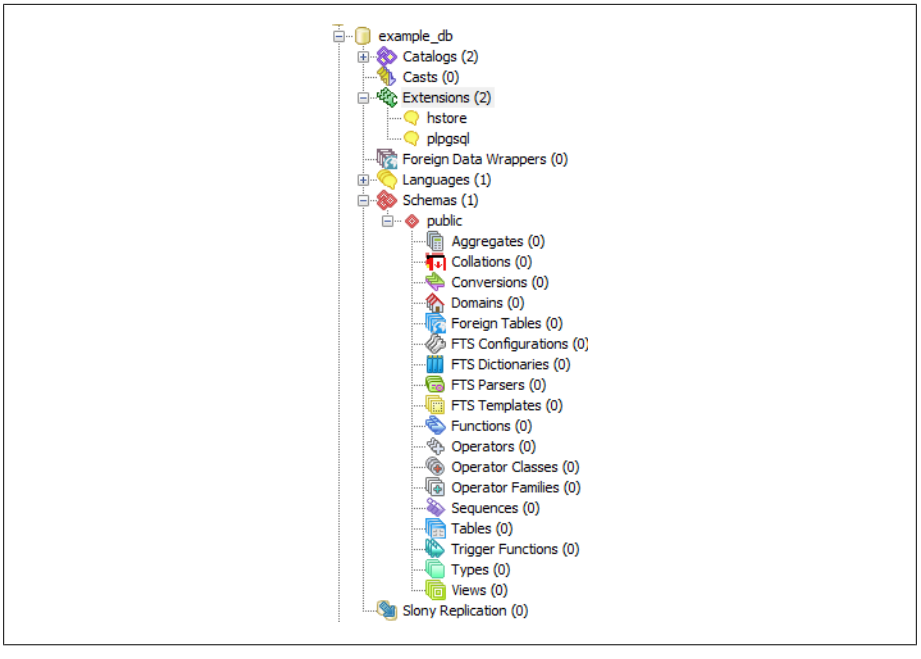


Figure 1-1. pgAdmin

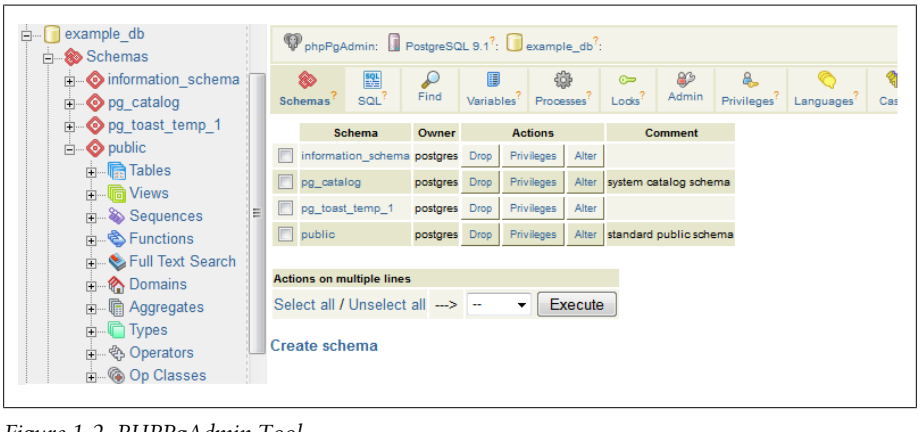


Figure 1-2. PHPPgAdmin Tool

What's New in Latest Versions of PostgreSQL?

The upgrade process gets simpler with each new version. There's no reason not to always keep in step with the latest version. PostgreSQL is the fastest growing database technology today. Major versions come out almost annually. Each new version adds enhancements to ease of use, stability, security, performance, and avant-garde features. The lesson here? Always upgrade, and do so often.

Why Upgrade?

If you're using PostgreSQL 8.2 or below: upgrade now! Enough said.

If you're using PostgreSQL 8.3: upgrade soon! 8.3 will be reaching end-of-life in early 2013. Details about PostgreSQL EOL policy can be found here: *PostgreSQL Release Support Policy*. EOL is not a place you want to be. New security updates and fixes to serious bugs will no longer be available. You'll need to hire specialized PostgreSQL core consultants to patch problems or to implement workarounds—probably not a cheap proposition, assuming you can even locate someone to begin with.

Regardless of which version you are using, you should always try to run the latest micro-versions for your version. An upgrade from say 8.4.8 to 8.4.11 requires just binary file replacement, which can be generally done with a quick restart after installing the upgrade. Only bug fixes are introduced in micro-versions, so there's little cause for concern and can in fact save you grief.

What to Look for in PostgreSQL 9.2

At time of writing, PostgreSQL 9.1 is the latest stable release, and 9.2 is waiting in the wings to strut its stuff. All of the anticipated features in 9.2 are already set in stone and available in the 9.2 beta release. The following list discusses the most notable features:

- Index-only scans. If you need to retrieve only columns that are already a part of an index, PostgreSQL will skip the need to go to the table. You'll see significant speed improvement in these queries as well as aggregates such as `COUNT(*)`.
- Sorting improvements that improve in-memory sort operations by as much as 20%.
- Improvements in *prepared statements*. A prepared statement is now parsed, analyzed, and rewritten, but not necessarily planned. It can also produce custom saved plans of a given prepared statement which are dependent on argument inputs. This reduces the chance that a prepared statement will perform worse than an equivalent ad-hoc query.
- Cascading streaming replication supports streaming from a slave to another slave.
- SP-GiST, another advance in GiST index technology using space filling trees. This should have great impact on the various extensions that rely on GiST for speed.
- `ALTER TABLE IF EXISTS` syntax for making changes to tables.
- Many new variants of `ALTER TABLE ALTER TYPE` commands that used to require whole table rewrites and rebuild of indexes. (More details are available at *More Alter Table Alter Types*.)
- Even more pg_dump and pg_restore options. (Read our article at *9.2 pg_dump Enhancements*.)
- plv8js is a new language handler that allows you to create functions in JavaScript.

- JSON built-in data type and companion functions `row_to_json()`, `array_to_json()`. This should be a welcome addition for web developers writing AJAX applications.
- New range type class of types where a pair of values in data type forms a range, eliminating the need to cludge range-like functionality.
- Allow SQL functions to reference arguments by name instead of by number.

PostgreSQL 9.1 Improvements

PostgreSQL 9.1 introduced enterprise features, making it an even more viable alternative to the likes of Microsoft SQL Server and Oracle:

- More built-in replication features including synchronous replication.
- Extensions management using the new `CREATE EXTENSION`, `ALTER EXTENSION`. Extensions make installing and removing add-ons a breeze.
- ANSI-compliant foreign data wrappers for querying disparate data sources.
- Writeable common table expressions (CTE). The syntactical convenience of CTEs now works for `UPDATE` and `INSERT` queries.
- Unlogged tables speeds up queries against tables where logging is unnecessary.
- Triggers on views. In prior versions, to make views updatable you used `DO INSTEAD` rules, which only supported SQL for programming logic. Triggers can be written in most procedural languages—except SQL—and opens the door for more complex abstraction using views.
- KNN GiST adds improvement to popular extensions like full-text search, trigram (for fuzzy search and case insensitive search), and PostGIS.

Database Drivers

If you are using or plan to use PostgreSQL, chances are that you're not going to use it in a vacuum. To have it interact with other applications, you're going to need database drivers. PostgreSQL enjoys a generous number of freely available database drivers that can be used in many programming languages. In addition, there are various commercial organizations that provide drivers with extra bells and whistles at modest prices. Below, we've listed a few popular, open source ones:

- PHP is a common language used to develop web applications, and most PHP distributions come packaged with at least one PostgreSQL driver. There is the older pgsql and the newer pdo_pgsql. You may need to enable them in your php.ini or do a yum install, but they are usually already there.
- Java. If you are doing Java development, there are always updated versions of JDBC that support the latest PostgreSQL, which you can download from *http://jdbc.postgresql.org*.

- For .NET. (Microsoft or Mono) you can use the Npgsql driver, which has source and binary versions for .NET Frameworks 3.5 and above, and Mono.NET.
- If you need to connect from MS Access or some other Windows Office productivity software, download ODBC drivers from *http://www.postgresql.org/ftp/odbc/versions/msi*. The link includes both 32-bit and 64-bit ODBC drivers.
- LibreOffice/OpenOffice. LibreOffice 3.5 (and above) comes packaged with a native PostgreSQL driver. For OpenOffice and older versions of LibreOffice, you can use a PostgreSQL JDBC driver or the SDBC driver. You can find details about connecting to these on our article *OO Base and PostgreSQL*.
- Python is a beautiful language and has support for PostgreSQL via various Python database drivers; at the moment, Psycopg is the most popular.
- Ruby. You can connect to PostgreSQL via rubypg.
- Perl. You'll find PostgreSQL connectivity support via DBI and the DBD:Pg driver or pure Perl DBD:PgPP driver from CPAN.

Server and Database Objects

So you installed PostgreSQL and open up pgAdmin. You expand the server tree. Before you is a bewildering array of database objects, some familiar and some completely foreign. PostgreSQL has more database objects than probably any other database, and that's without considering add-ons. You'll probably never touch many of these objects, but if you dream up a new functionality that you wish PostgreSQL would offer, more likely than not, it's already implemented using one of those esoteric objects that you've been ignoring. This book is not even going to attempt to describe all that you'll find in a PostgreSQL install. With PostgreSQL churning out features at breakneck speed, we can't imagine any book that could possibly itemize all that PostgreSQL has to offer. We'll now discuss the most commonly used database objects:

server service
: The PostgreSQL server service is often just called a PostgreSQL server, or daemon. You can have more than one a physical server as long as they listen on different ports or IPs and have different places to store their respective data.

database
: Each PostgreSQL server houses many databases.

table
: Table are the workhorses of any database. What is unique about PostgreSQL tables is the inheritance support and the fact that every table automatically begets an accompanying custom data type. Tables can inherit from other tables and querying can bring up child records from child tables.

schema
: Schemas are part of the ANSI-SQL standards, so you'll see them in other databases. Schemas are the logical containers of tables and other objects. Each database can have multiple schemas.

tablespace
: Tablespace is the physical location where data is stored. PostgreSQL allows tablespaces to be independently managed, which means you can easily move databases to different drives with just a few commands.

view
: Most relational databases have views for abstracting queries. In PostgreSQL, you can also have views that can be updated.

function
: Functions in PostgreSQL can return scalar value or sets of records. Aggregates are functions used with SQL constructs such as `GROUP BY` to summarize data. Most of the time, they return scalars but in PostgreSQL they can return composite objects.

operator
: These are symbolic functions that have backing of a function. In PostgreSQL, you can define your own.

cast
: Casts allow you to convert from one data type to another. They are supported by functions that actually perform the conversion. What is rare about PostgreSQL that you won't find with many other databases is that you can create your own casts and thus change the default behavior of casting. Casting can be implicit or explicit. Implicit casts are automatic and usually will expand from a more specific to a more generic type. When an implicit cast is not offered, you must cast explicitly.

sequence
: Sequence is what controls auto-incrementation in table definitions. They are usually automatically created when you define a serial column. Because they are objects in their own right, you could have multiple serial columns use the same sequence object, effectively achieveing uniqueness not only within the column but across them.

trigger
: Found in many databases, triggers detect data change events and can react before or after the actual data is changed. PostgreSQL 9.0 introduced some special twists to this with the `WHEN` clause. PostgreSQL 9.1 added the extra feature of making triggers available for views.

foreign data wrappers
: Foreign data wrappers allow you to query a remote data source whether that data source be another relational database server, flat file, a NoSQL database, a web service or even an application platform like SalesForce. They are found in SQL

Server as linked tables, but PostgreSQL implementation follows the *SQL/Management of External Data (MED)* standard, and is open to connect to any kind of data source.

row/record
: Rows and records generally mean the same thing. In PostgreSQL, rows can be treated independently from their respective tables. This distinction becomes apparent and useful when you write functions or use the row constructor in SQL.

extension
: This is a new feature introduced in 9.1 that packages a set of functions, types, casts, indexes, and so forth into a single unit for maintainability. It is similar in concept to Oracle packages and is primarily used to deploy add-ons.

Where to Get Help

There will come a day when you need additional help. Since that day always arrives earlier than expected, we want to point you to some resources now rather than later. Our favorite is the lively mailing list network specifically designed for helping new and old users with technical issues. First, visit *PostgreSQL Help Mailing Lists*. If you are new to PostgreSQL, the best newsgroup to start with is *PGSQL-General Mailing List*. Finally, if you run into what appears to be a bug in PostgreSQL, report it at *PostgreSQL Bug Reporting*.

CHAPTER 2

Database Administration

This chapter will cover what we feel are the most common activities for basic administration of a PostgreSQL server; namely: role management, database creation, add-on installation, backup, and restore. We'll assume you've already installed PostgreSQL and have one of the administration tools at your disposal.

Configuration Files

Three main configuration files control basic operations of a PostgreSQL server instance. These files are all located in the default PostgreSQL data folder. You can edit them using your text editor of choice, or using the admin pack that comes with pgAdmin ("Editing postgresql.conf and pg_hba.conf from pgAdmin" on page 47).

- *postgresql.conf* controls general settings, such as how much memory to allocate, default storage location for new databases, which IPs PostgreSQL listens on, where logs are stored, and so forth.
- *pg_hba.conf* controls security. It manages access to the server, dictating which users can login into which databases, which IPs or groups of IPs are permitted to connect and the authentication scheme expected.
- *pg_ident.conf* is the mapping file that maps an authenticated OS login to a PostgreSQL user. This file is used less often, but allows you to map a server account to a PostgreSQL account. For example, people sometimes map the OS root account to the postgre's super user account. Each authentication line in *pg_hba.conf* can use a different *pg_ident.conf* file.

If you are ever unsure where these files are located, run the Example 2-1 query as a super user while connected to any of your databases.

Example 2-1. Location of configuration files

```
SELECT name, setting
FROM pg_settings
WHERE category = 'File Locations';
```

```
name              |          setting
------------------+---------------------------
config_file       |E:/PGData91/postgresql.conf
data_directory    |E:/PGData91
external_pid_file |
hba_file          |E:/PGData91/pg_hba.conf
ident_file        |E:/PGData91/pg_ident.conf
```

The postgresql.conf File

postgresql.conf controls the core settings of the PostgreSQL server instance as well as default settings for new databases. Many settings—such as sorting memory—can be overriden at the database, user, session, and even function levels for PostgreSQL versions higher than 8.3.

Details on how to tune this can be found at *Tuning Your PostgreSQL Server*.

An easy way to check the current settings you have is to query the pg_settings view, as we demonstrate in Example 2-2. Details of the various columns of information and what they mean are described in *pg_settings*.

Example 2-2. Key Settings

```
SELECT name, context ❶, unit ❷
 , setting ❸, boot_val ❹, reset_val ❺
FROM pg_settings
WHERE name
in('listen_addresses','max_connections','shared_buffers','effective_cache_size',
'work_mem', 'maintenance_work_mem')
ORDER BY context,name;
```

```
name                 |  context   | unit | setting | boot_val  | reset_val
---------------------+------------+------+---------+-----------+-----------
listen_addresses     | postmaster |      | *       | localhost | *
max_connections      | postmaster |      | 100     | 100       | 100
shared_buffers       | postmaster | 8kB  | 4096    | 1024      | 4096
effective_cache_size | user       | 8kB  | 16384   | 16384     | 16384
maintenance_work_mem | user       | kB   | 16384   | 16384     | 16384
work_mem             | user       | kB   | 1024    | 1024      | 1024
```

❶ If `context` is set to postmaster, it means changing this parameter requires a restart of the postgresql service. If `context` is set to user, changes require a reload to take effect globally. Furthermore, user context settings can be overridden at the database, user, session, or function levels.

❷ `unit` tells you the unit of measurement that the setting is reported in. This is very important for memory settings since, as you can see, some are reported

in 8 kB and some in `kB`. In *postgresql.conf*, usually you explicitly set these to a unit of measurement you want to record in, such as 128 MB. You can also get a more human-readable display of a setting by running the statement: `SHOW effective_cache_size;`, which gives you 128 MB, or `SHOW maintenance_work_mem;`, which gives you 16 MB for this particular case. If you want to see everything in friendly units, use `SHOW ALL`.

❸❹❺ `setting` is the currently running setting in effect; `boot_val` is the default setting; `reset_val` is the new value if you were to restart or reload. You want to make sure that after any change you make to *postgresql.conf* the setting and `reset_val` are the same. If they are not, it means you still need to do a reload.

We point out the following parameters as ones you should pay attention to in *postgresql.conf*. Changing their values requires a service restart:

- `listen_addresses` tells PostgreSQL which IPs to listen on. This usually defaults to `localhost`, but many people change it to *, meaning all available IPs.
- `port` defaults to 5432. Again, this is often set in a different file in some distributions, which overrides this setting. For instance, if you are on a Red Hat or CentOS, you can override the setting by setting a `PGPORT` value in */etc/sysconfig/pgsql/your_service_name_here*.
- `max_connections` is the maximum number of concurrent connections allowed.
- `shared_buffers` defines the amount of memory you have shared across all connections to store recently accessed pages. This setting has the most effect on query performance. You want this to be fairly high, probably at least 25% of your on-board memory.

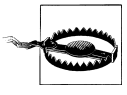
The following three settings are important, too, and take effect without requiring a restart, but require at least a reload, as described in "Reload the Configuration Files" on page 14.

- `effective_cache_size` is an estimate of how much memory you expect to be available in the OS and PostgreSQL buffer caches. It has no affect on actual allocation, but is used only by the PostgreSQL query planner to figure out whether plans under consideration would fit in RAM or not. If it's set too low, indexes may be underutilized. If you have a dedicated PostgreSQL server, then setting this to half or more of your on-board memory would be a good start.
- `work_mem` controls the maximum amount of memory allocated for each operation such as sorting, hash join, and others. The optimal setting really depends on the kind of work you do, how much memory you have, and if your server is a dedicated database server. If you have many users connecting, but fairly simple queries, you want this to be relatively low. If you do lots of intensive processing, like building a data warehouse, but few users, you want this to be high. How high you set this also depends on how much motherboard memory you have. A good article to read

on the pros and cons of setting `work_mem` is *Understanding postgresql.conf work_mem*.

- `maintenance_work_mem` is the total memory allocated for housekeeping activities like vacuuming (getting rid of dead records). This shouldn't be set higher than about 1 GB.

The above settings can also be set at the database, function, or user level. For example, you might want to set `work_mem` higher for a power user who runs sophisticated queries. Similarly, if you have a sort-intensive function, you could raise the `work_mem` just for it.



I edited my postgresql.conf and now my server is broken.

The easiest way to figure out what you did wrong is to look at the log file, which is located in the root of the data folder, or in the subfolder *pg_log*. Open up the latest file and read what the last line says. The error notice is usually self-explanatory.

A common culprit is that you set the `shared_buffers` too high. Another common cause of failures is that there is an old *postmaster.pid* hanging around from a failed shutdown. You can safely delete this file which is located in the data cluster folder and try to restart again.

The pg_hba.conf File

The *pg_hba.conf* controls which and how users can connect to PostgreSQL databases. Changes to the *pg_hba.conf* require a reload or a server restart to take effect. A typical *pg_hba.conf* looks like this:

```
# TYPE DATABASE USER ADDRESS METHOD
# IPv4 local connections:
host    all        all        127.0.0.1/32    ident ❶
# IPv6 local connections:
host    all        all    ::1/128 ❷    trust
host    all        all    192.168.54.0/24 ❸    md5
hostssl ❹ all    all    0.0.0.0/0    md5
# Allow replication connections from localhost, by a user with the ❺
# replication privilege.
#host    replication    postgres    127.0.0.1/32    trust
#host    replication    postgres    ::1/128    trust
```

❶ Authentication method. ident, trust, md5, password are the most common and available. In PostgreSQL 9.1, the additional peer authentication was introduced. The ident and peer options are only available on Linux/Unix like systems (not Windows). Others such as gss, radius, ldap, and pam, may not always be installed.

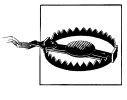
❸ IPv4 syntax for defining network range. The first part in this case 192.168.54.0 is the network address. The /24 is the bit mask. In this example, we are allowing anyone in our subnet of 192.168.54.0 to connect as long as they provide a valid md5 hashed password.

❷ IPv6 syntax for defining localhost. This only applies to servers with IPv6 support and may cause the configuration file to not load if you have it and don't have IPv6. For example, on a Windows XP or Windows 2003 machine, you shouldn't have this line.

❹ Users must connect through SSL. In our example, we allow anyone to connect to our server as long as they connect using SSL and have a valid md5-encrypted password.

❺ Defines a range of IPs allowed to replicate with this server. This is new in PostgreSQL 9.0+. In this example, we have the line remarked out.

For each connection request, postgres service checks the *pg_hba.conf* file in order from the top down. Once a rule granting access is encountered, processing stops and the connection is allowed. Should the end of the file be reached without any matching rules, the connection is denied. A common mistake people make is to not put the rules in order. For example, if you put 0.0.0.0/0 reject before you put 127.0.0.1/32 trust, local users won't be able to connect, even though you have a rule allowing them to do so.



I edited my pg_hba.conf and now my database server is broken.

This occurs quite frequently, but it's easily recoverable. This error is generally caused by typos, or by adding an unavailable authentication scheme. When the postgres service can't parse the *pg_hba.conf* file, it'll block all access or won't even start up. The easiest way to figure out what you did wrong is to read the log file. This is located in the root of the data folder or in the sub folder *pg_log*. Open up the latest file and read the last line. The error message is usually self-explanatory. If you're prone to slippery fingers, consider backing up the file prior to editing.

Authentication Methods

PostgreSQL has many methods for authenticating users, probably more than any other database. Most people stick with the four main ones: trust, ident, md5, and password. There is also a fifth one: `reject`. which performs an immediate deny. Authentication methods stipulated in *pg_hba.conf* serve as gatekeepers to the entire server. Users or devices must still satisfy individual role and database access restrictions after connecting.

We list the most commonly used authentication methods below. For more information on the various authentication methods, refer to *PostgreSQL Client Authentication*.

- `trust` is the least secure of the authentication schemes and means you allow people to state who they are and don't care about the passwords, if any, presented. As long as they meet the IP, user, and database criteria, they can connect. You really should use this only for local connections or private network connections. Even then it's possible to have IPs spoofed, so the more security-minded among us discourage its use entirely. Nevertheless, it's the most common for PostgreSQL in-

stalled on a desktop for single user local access where security is not as much of a concern. The user name defaults to the logged in user if not specified.

- md5 is the most common and means an md5-encrypted password is required.
- password means clear text password authentication.
- ident uses the *pg_ident.conf* to see if the OS account of the user trying to connect has a mapping to a PostgreSQL account. Password is not checked.

You can have multiple authentication methods, even for the same database; just keep in mind the top to bottom checking of *pg_hba.conf*.

Reload the Configuration Files

Many, but not all changes, to configuration files require restarting the postgres service. Many changes take effect by performing a reload of the configuration. Reloading doesn't affect active connections. Open up a command line and follow these steps to reload:

```
pg_ctl reload -D your_data_directory_here
```

If you have PostgreSQL installed as a service in Redhat EL or CentOS, you can do:

```
service postgresql-9.1 reload
```

where `postgresql-9.1` is the name of your service.

You can also log in as a super user on any database and run this SQL statement:

```
SELECT pg_reload_conf();
```

You can also do this from pgAdmin, refer to "Editing postgresql.conf and pg_hba.conf from pgAdmin" on page 47.

Setting Up Groups and Login Roles (Users)

In PostgreSQL, there is really only one kind of an account and that is a role. Some roles can log in; when they have login rights, they are called users. Roles can be members of other roles, and when we have this kind of relationship, the containing roles are called groups. It wasn't always this way, though: Pre-8.0 users and groups were distinct entities, but the model got changed to be `role`-centric to better conform to the ANSI-SQL specs.

For backward compatibility, there is still a `CREATE USER` and `CREATE GROUP`. For the rest of this discussion, we'll be using the more generic `CREATE ROLE` , which is used to create both users and groups.

If you look at fairly ANSI-SQL standard databases such as Oracle and later versions of SQL Server, you'll notice they also have a `CREATE ROLE` statement, which works similarly as the PostgreSQL one.

Creating an Account That Can Log In

`postgres` is an account that is created when you first initialize the PostgreSQL data cluster. It has a companion database called `postgres`. Before you do anything else, you should login as this user via psql or pgAdmin and create other users. pgAdmin has a graphical section for creating user roles, but if you were to do it using standard SQL data control language (DCL), you would execute an SQL command as shown in Example 2-3.

Example 2-3. User with login rights that can create database objects

```
CREATE ROLE leo LOGIN PASSWORD 'lion!king'
CREATEDB VALID UNTIL 'infinity';
```

The `'infinity'` is optional and assumed if not specified. You could instead put in a valid date at which you want the account to expire.

If you wanted to create a user with super rights, meaning they can cause major destruction to your database cluster and can create what we call untrusted language functions, you would create such a user as shown in Example 2-4. You can only create a super user if you are a super user yourself.

Example 2-4. User with login rights that can create database objects

```
CREATE ROLE regina LOGIN PASSWORD 'queen!penultimate'
 SUPERUSER VALID UNTIL '2020-10-20 23:00';
```

As you can see, we don't really want our queen to reign forever, so we put in a timestamp when her account will expire.

Creating Group Roles

Group roles are generally roles that have no login rights but have other roles as members. This is merely a convention. There is nothing stopping you from creating a role that can both login and can contain other roles.

We can create a group role with this SQL DCL statement:

```
CREATE ROLE jungle INHERIT;
```

And add a user or other group role to the group with this statement:

```
GRANT jungle TO leo;
```

Roles Inheriting Rights

One quirky thing about PostgreSQL is the ability to define a role that doesn't allow its member roles to inherit its rights. The concept comes into play when you define a role to have member roles. You can designate that members of this role don't inherit rights of the role itself. This is a feature that causes much confusion and frustration when

setting up groups, as people often forget to make sure that the group role is marked to allow its permissions as inheritable.

Non-Inheritable rights

Some permissions can't be inherited. For example, while you can create a group role that you mark as super user, this doesn't make its member roles super users; however, those users can impersonate their parent role, thus gaining super power rights for a brief period.

Databases and Management

The simplest create database statement to write is:

```
CREATE DATABASE mydb;
```

The owner of the database will be the logged in user and is a copy of `template1` database.

Creating and Using a Template Database

A template database is, as the name suggests, a database that serves as a template for other databases. In actuality, you can use any database as template for another, but PostgreSQL allows you to specifically flag certain databases as templates. The main difference is that a database marked as template can't be deleted and can be used by any user having `CREATEDB` rights (not just superuser) as a template for their new database. More details about template databases are described in the PostgreSQL manual *Managing Template Databases*.

The `template1` database that is used as the default when no template is specified, doesn't allow you to change encodings. As such, if you want to create a database with an encoding and collation different from your default, or you installed extensions in `template1` you don't want in this database, you may want to use `template0` instead.

```
CREATE DATABASE mydb TEMPLATE template0;
```

If we wanted to make our new database a template, we would run this SQL statement as a super user:

```
UPDATE pg_database SET datistemplate=true WHERE datname='mydb';
```

This would allow other users with `CREATEDB` rights to use this as a template. It will also prevent the database from being deleted.

Organizing Your Database Using Schemas

Schemas are a logical way of partitioning your database into mini-containers. You can divide schemas by functionality, by users, or by any other attribute. Aside from logical partitioning, they provide an easy way for doling out rights. One common practice is

to install all contribs and extensions, covered in "Extensions and Contribs" on page 18 into a separate schema and give rights to all users of a database.

To create a schema called contrib in a database, we connect to the database and run this SQL:

```
CREATE SCHEMA contrib;
```

The default `search_path` defined in *postgresql.conf* is `"$user",public`. This means that if there is a schema with the same name as the logged in user, then all non-schema qualified objects will first check the schema with the same name as user and then the public schema. You can override this behavior at the user level or the database level. For example, if we wanted all objects in contrib to be accessible without schema qualification, we would change our database as follows:

```
ALTER DATABASE mydb SET search_path="$user",public,contrib;
```

The `SET search_path` change will not take effect for existing connections. You'll need to reconnect to your database to experience the change.

Schemas are also used for simple abstraction. A table name only needs to be unique within the schema, so many applications exploit this by creating same named tables in different schemas and, depending on who is logging in, they will get their own version based on which is their primary schema.

Permissions

Permissions are one of the trickiest things to get right in PostgreSQL. This is one feature that we find more difficult to work with than other databases. Permission management became a lot easier with the advent of PostgreSQL 9.0+. PostgreSQL 9.0 introduced default permissions, which allowed for setting permissions on all objects of a particular schema or database as well as permissions on specific types of objects. More details on permissions management are detailed in the manual, in sections ALTER DEFAULT PRIVILEGES and GRANT.

Getting back to our contrib schema. Let's suppose we want all users of our database to have `EXECUTE` and `SELECT` access to any tables and functions we will create in the contrib schema. We can define permissions as shown in Example 2-5:

Example 2-5. Defining default permissions on a schema

```
GRANT USAGE ON SCHEMA contrib TO public;
ALTER DEFAULT PRIVILEGES IN SCHEMA contrib
GRANT SELECT, REFERENCES ON TABLES
    TO public;

ALTER DEFAULT PRIVILEGES IN SCHEMA contrib
```

```
    GRANT SELECT, UPDATE ON SEQUENCES
    TO public;

ALTER DEFAULT PRIVILEGES IN SCHEMA contrib
    GRANT EXECUTE ON FUNCTIONS
    TO public;

ALTER DEFAULT PRIVILEGES IN SCHEMA contrib
    GRANT USAGE ON TYPES
    TO public;
```

If you already have your schema set with all the tables and functions, you can retroactively set permissions on each object separately or do this for all existing tables, functions, and sequences with a `GRANT .. ALL .. IN SCHEMA`.

Example 2-6. Set permissions on existing objects of a type in a schema

```
GRANT USAGE ON SCHEMA contrib TO public;
GRANT SELECT, REFERENCES, TRIGGER
    ON ALL TABLES IN SCHEMA contrib
    TO public;

GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA contrib TO public;
GRANT SELECT, UPDATE ON ALL SEQUENCES IN SCHEMA contrib TO public;
```

If you find this all overwhelming for setting permissions, just use pgAdmin for permission management. pgAdmin provides a great interface for setting default permissions, as well as retroactively granting bulk permissions of selective objects. We'll cover this feature in "Creating Databases and Setting Permissions" on page 47.

Extensions and Contribs

Extensions and contribs are add-ons that you can install in a PostgreSQL database to extend functionality beyond the base offerings. They exemplify the best feature of open source software: people collaborating, building, and freely sharing new features. Prior to PostgreSQL 9.1, the add-ons were called contribs. Since PostgreSQL 9.1+, add-ons are easily installed using the new PostgreSQL extension model. In those cases, the term extension has come to replace the term contrib. For the sake of consistency, we'll be referring to all of them by the newer name of extension, even if they can't be installed using the newer extension model.

The first thing to know about extensions is that they are installed separately in each database. You can have one database with the fuzzy text support extension and another that doesn't. If you want all your databases to have a certain set of extensions installed in a specific schema, you can set up a template database as discussed in "Creating and Using a Template Database" on page 16 with all these installed, and then create all your databases using that template.

To see which extensions you have already installed, run the query in Example 2-7:

Example 2-7. List extensions installed

```
SELECT *
FROM pg_available_extensions
WHERE comment LIKE '%string%' OR installed_version IS NOT NULL
ORDER BY name;
```

```
name          | default_version | installed_version |    comment
--------------+-----------------+-------------------+-----------------------
citext        | 1.0             |                   | data type for case-insen..
fuzzystrmatch | 1.0             | 1.0               | determine simil.. and dist..
hstore        | 1.0             | 1.0               | data type for .. (key, value) ..
pg_trgm       | 1.0             | 1.0               | text similarity measur..index sear..
plpgsql       | 1.0             | 1.0               | PL/pgSQL procedural language
postgis       | 2.0.0           | 2.0.0             | geometry, geography,..raster ..
temporal      | 0.7.1           | 0.7.1             | temporal data type ..
```

To get details about a particular installed extension, enter the following command from psql:

```
\dx+ fuzzystrmatch
```

Or run this query:

```
SELECT pg_catalog.pg_describe_object(d.classid, d.objid, 0) AS description
FROM pg_catalog.pg_depend AS D
  INNER JOIN pg_extension AS E ON D.refobjid = E.oid
WHERE D.refclassid = 'pg_catalog.pg_extension'::pg_catalog.regclass
 AND deptype = 'e' AND E.extname = 'fuzzystrmatch';
```

Which outputs what is packaged in the extension:

```
description
------------------------------------------------------------------------
function dmetaphone_alt(text)
function dmetaphone(text)
function difference(text,text)
function text_soundex(text)
function soundex(text)
function metaphone(text,integer)
function levenshtein_less_equal(text,text,integer,integer,integer,integer)
function levenshtein_less_equal(text,text,integer)
function levenshtein(text,text,integer,integer,integer)
function levenshtein(text,text)
```

Installing Extensions

Regardless of how you install an extension in your database, you'll need to have gathered all the dependent libraries in your PostgreSQL *bin* and *lib*, or have them accessible via your system path. For small extensions, most of these libraries already come prepackaged with your PostgreSQL install so you don't have to worry. For others, you'll either need to compile your own, get them with a separate install, or copy the files from another equivalent setup.

The Old Way

Prior to PostgreSQL 9.1, the only way to install an extension was to manually run the requisite SQL scripts in your database. Many extensions still can only be installed this way.

By convention, add-ons scripts are automatically dumped into the *contrib* folder of your PostgreSQL if you use an installer. Where you'd find this folder will depend on your particular OS and distro. As an example, on a CentOS running 9.0, to install the pgAdmin pack, one would run the following from the command line:

```
psql -p 5432 -d postgres -f /usr/pgsql-9.0/share/contrib/adminpack.sql
```

The New Way

With PostgreSQL 9.1 and above, you can use the `CREATE EXTENSION` command. The two big benefits are that you don't have to figure out where the extension files are kept (they are kept in a folder *share/extension*), and you can uninstall just as easily with `DROP EXTENSION`. Most of the common extensions are packaged with PostgreSQL already, so you really don't need to do more than run the command. To retrieve extensions not packaged with PostgreSQL, visit the PostgreSQL Extension Network. Once you have downloaded, compiled, and installed (install just copies the scripts and .control to *share/extension*, and the respective binaries to *bin* and *lib*) the new extension, run `CREATE EXTENSION` *`extension_name`* to install in specific database. Here is how we would install the fuzzystrmatch extension in PostgreSQL 9.1+: the new way no longer requires psql since `CREATE EXTENSION` is part of the PostgreSQL's SQL language. Just connect to the database you want to install the extension and run the SQL command:

```
CREATE EXTENSION fuzzystrmatch;
```

If you wanted all your extensions installed in a schema called my_extensions, you would first create the schema, and install the extensions:

```
CREATE EXTENSION fuzzystrmatch SCHEMA my_extensions;
```

Upgrading from Old to New

If you've been using a version of PostgreSQL before 9.1 and restored your old database into a 9.1 during a version upgrade, all add-ons should continue to work untouched. For maintainability, you'll probably want to upgrade your old extensions in the *contrib* folder to use the new extensions approach. Many extensions, especially the ones that come packaged with PostgreSQL, have ability to upgrade pre-extension installs. Let's suppose you had installed the tablefunc extension (which provides cross tabulation functions) to your PostgreSQL 9.0 in a schema called contrib, and you've just restored your database to a PostgreSQL 9.1 server. Run the following command to upgrade the extension:

```
CREATE EXTENSION tablefunc SCHEMA contrib FROM unpackaged;
```

You'll notice that the old functions are still in the *contrib* schema, but moving forward they will no longer be backed up and your backups will just have a `CREATE EXTENSION ..` clause.

Common Extensions

Many extensions come packaged with PostgreSQL, but are not installed by default. Some past extensions have gained enough traction to become part of the PostgreSQL core, so if you're upgrading from an ancient version, you may not even have to worry about extensions.

Old Extensions Absorbed into PostgreSQL

Prior to PostgreSQL 8.3, the following extensions weren't part of core:

- PL/PgSQL wasn't always installed by default in every database. In old versions, you had to run `CREATE LANGUAGE plpgsql;` in your database. From around 8.3 on, it's installed by default, but you retain the option of uninstalling it.
- *tsearch* is a suite for supporting full-text searches by adding indexes, operators, custom dictionaries, and functions. It became part of PostgreSQL core in 8.3. You don't have the option to uninstall it. If you're still relying on old behavior, you can install the *tsearch2* extension, which retained old functions that are no longer available in the newer version. A better approach would be just to update where you're using the functions because compatiblity with the old tsearch could end at any time.
- *xml* is an extension that adds in support of XML data type and related functions and operators. As of version 8.3, XML became an integral part of PostgreSQL, in part to meet the ANSI-SQL XML standard. The old extension, now dubbed *xml2*, can still be installed and contains functions that didn't make it into the core. In particular, you need this extension if you relied on the `xlst_process()` function for processing XSL templates. There are also a couple of old XPath functions not found in the core.

Popular Extensions

In this section, we'll list and quickly describe the most popular, and some may say, must-have extensions, that aren't part of current core.

- *postgis* elevates PostgreSQL to a state-of-the-art spatial database outrivaling all commercial options. If you deal with standard OGC GIS data, demographic statistics data, or geocoding, you don't want to be without this one. You can learn more about PostGIS in our book, *PostGIS in Action*. Part of the book's proceeds will help fund the PostGIS project itself. PostGIS is a whopper of an extension, weighing in at over 800 functions, types, and spatial indexes.

- *fuzzystrmatch* is a lightweight extension with functions like `soundex`, `levenshtein`, and `metaphone` for fuzzy string matching. We discuss its use in *Where is Soundex and Other Warm and Fuzzy Things*.
- *hstore* is an extension that adds key-value pair storage and index support well-suited for storing pseudo-normalized data. If you are looking for a comfortable medium between relational and NoSQL, check out hstore.
- *pg_trgm* (trigram) is an extension that is another fuzzy string search library. It is often used in conjunction with fuzzystrmatch. In PostgreSQL 9.1, it takes on another special role in that it makes `ILIKE` searches indexable by creating a trigram index. Trigram can also index wild-card searches of the form `LIKE '%something%'`. Refer to *Teaching ILIKE and LIKE New Tricks* for further discussion.
- *dblink* is a module that allows you to query other PostgreSQL databases. This is currently the only supported mechanism of cross-database interaction for PostgreSQL. In PostgreSQL 9.3, foreign data wrapper for PostgreSQL is expected to hit the scene.
- *pgcrypto* provides various encryption tools including the popular PGP. We have a quick primer on using it available here: *Encrypting Data with pgcrypto*.

As of 9.1, less used procedural languages (PLs), index types, and foreign data wrappers (FDW) are also packaged as extensions.

Backup

PostgreSQL comes with two utilities for backup—*pg_dump* and *pg_dumpall*. You'll find both in the `bin` folder. You use *pg_dump* to backup specific databases, and *pg_dumpall* to backup all databases and server globals. *pg_dumpall* needs to run under a postgres super user account so it has access to backup all databases. You will notice that most of the commands for these tools will have both long names as well as equivalent short switches. You can use them interchangeably, even in the same command. We'll be covering just the basics here, but for a more in-depth discussion, refer to the PostgreSQL *Backup and Restore* section of the official manual.

We often specify the port and host in these commands because we often run them via scheduled jobs not on the same machine; or we have several instances of PostgreSQL running on the same box, each running on a different port. Sometimes specifying the `-h or --host` switch, for example, may cause problems if your service is set to only listen on local. You can safely leave it out if you are running from the server.

You may also want to employ the use of *~pgpass* since none of these command lines give you the option of specifying a password.

Selective Backup Using pg_dump

For day-to-day backup, *pg_dump* is generally more expeditious than *pg_dumpall* because it can selectively backup tables, schemas, databases. *pg_dump* backs up to plain SQL, but also compressed and TAR formats. Compressed and TAR backups can take advantage of the parallel restore feature introduced in 8.4. Refer to "Database Backup: pg_dump" on page 140 for a listing of *pg_dump* command options.

In this example, we'll show a few common backup scenarios and corresponding *pg_dump* switches. These examples should work for any version of PostgreSQL.

Example 2-8. pg_dump usage

Creates a compressed, single database backup:

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v -f mydb.backup mydb
```

Creates a plain-text single database backup, including Creates database:

```
pg_dump -h localhost -p 5432 -U someuser -C -F p -b -v -f mydb.backup mydb
```

Creates a compressed backup of tables with a name that starts with payments in any schema:

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v -t *.payments* -f payment_tables.backup
mydb
```

Creates a compressed backup of all objects in hr and payroll schemas:

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v -n hr -n payroll -f
hr_payroll_schemas.backup mydb
```

Creates a compressed backup of all objects in all schemas, excluding public schemas:

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v -N public -f
all_schema_except_public.backup mydb
```

Creates a plain-text SQL backup of select tables, useful for porting to lower versions of PostgreSQL or other database systems:

```
pg_dump -h localhost -p 5432 -U someuser -F p --column-inserts -f select_tables.backup mydb
```

If you have spaces in your file paths, you'll want to wrap the file path in double quotes: `"/path with spaces/mydb.backup"`. As a general rule, you can always use double quotes if you aren't sure.

The Directory format option was introduced in PostgreSQL 9.1. This option backs up each table as a separate file in a folder and gets around the problem where your file system has limitations on the size of each file. It is the only *pg_dump* backup format option that generates multiple files. An example of this is shown in Example 2-8. The directory backup first creates the directory to put the files in and errors out if the directory already exists.

Example 2-9. Directory format backup

The *a_directory* is created and in the folder, a separate gzipped file for each table and a file that has all the structures listed.

```
pg_dump -h localhost -p 5432 -U someuser -F d -f /somepath/a_directory mydb
```

Systemwide Backup Using pg_dumpall

The *pg_dumpall* utility is what you would use to backup all databases into a single plain-text file, along with server globals such as tablespace definitions and users. Refer to "Server Backup: pg_dumpall" on page 142 for listing of available *pg_dumpall* command options.

It's a good idea to backup globals such as roles and tablespace definitions on a daily basis. Although you can use *pg_dumpall* to backup databases as well, we generally don't bother or do it—at most, once a month—since it would take much longer to restore the plain text backup for large databases.

To backup roles and tablespaces:

```
pg_dumpall -h localhost -U postgres --port=5432 -f myglobals.sql --globals-only
```

If you only care about backing up roles and not tables spaces, you would use the roles only option:

```
pg_dumpall -h localhost -U postgres --port=5432 -f myroles.sql --roles-only
```

Restore

There are two ways of restoring in PostgreSQL:

- Using psql to restore plain text backups generated with *pg_dumpall* or *pg_dump*
- Using *pg_restore* utility for restoring compressed, tar and directory backups created with *pg_dump*

Terminating Connections

Before you can perform a full drop and restore of a database or restore a particular table that's in use, you'll need to kill connections. Every once in a while, someone else (never you) will execute a query that he or she didn't mean to and end up wasting resources. You could also run into a query that's taking much longer than what you have the patience for. Should these things happen, you'll either want to cancel the query on the connection or kill the connection entirely. To cancel running queries or to terminate connections, you elicit three administrative functions.

- `pg_stat_activity` (`SELECT * FROM pg_stat_activity;`) is a view that will list currently active connections and the process id. Additionally, it'll provide details of the active query running on each connection, the connected user (`usename`), the

database (`datname`) in use, and start times of query currently running. You need this view to obtain the proc ids of connections that you wish to terminate.

- `pg_cancel_backend(procid)` (`SELECT pg_cancel_backend(procid);`) will cancel all active queries on a connection, but doesn't terminate the connection.
- `pg_terminate_backend(procid)` (`SELECT pg_terminate_backend(procid);`) will kill a specific connection. All running queries will automatically cancel. This will be your weapon of choice prior to a restore to prevent an eager user from immediately restarting a cancelled query.

PostgreSQL, unlike some other databases, lets you embed functions that perform actions within a regular `SELECT` query. This means that though `pg_terminate_backend()` and `pg_cancel_backend()` can only act on one connection at a time, you can effectuate multiple connections by wrapping them in a `SELECT`. For example, let's suppose a user (Regina) was hogging up resources and had 100 connections going. We can kill all her connections by running this command:

Before 9.2:

```
SELECT pg_terminate_backend(procpid) FROM pg_stat_activity WHERE usename = 'regina';
```

9.2 and after:

```
SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE usename = 'regina';
```

`pg_stat_activity` has changed considerably in PostgreSQL 9.2 with renaming and addition of new columns. For example, `procpid` is now `pid`. More details about the changes and enhancements are detailed in *PostgreSQL 9.2 Monitoring Enhancements*.

Using psql to Restore Plain Text SQL backups

A plain SQL backup is nothing more than a text file of a huge SQL script. It's the least convenient of backups to have, but it's portable across different database systems. With SQL backup, you must execute the entire script, there's no partial restore, unless you're willing to manually edit the file. Since there are no options, the backups are simple to restore by using the `-f` psql switch as shown in Example 2-10. However, they are useful if you need to load data to another DBMS with some editing.

Example 2-10. Restores plain text SQL backups

Restores a full backup and ignore errors:

```
psql -U postgres -f myglobals.sql
```

Restores and stops on first error:

```
psql -U postgres --set ON_ERROR_STOP=on -f myglobals.sql
```

Restores a partial backup to a specific database:

```
psql -U postgres -d mydb -f select_objects.sql
```

Using pg_restore

If you backed up using *pg_dump*, and specified a non-plain text backup format like tar, custom, or directory, you can use the versatile *pg_restore* utility for the restore. *pg_restore* provides you with a dizzying array of options for restoration and far surpasses any restoration utility found in other database systems. Here are some of its outstanding features:

- As of 8.4, you can do parallel restores using the `-j` switch to control the number of threads to use. This allows each thread to be restoring a separate table simultaneously, which significantly speeds up restores.
- You can generate a plain text table of contents from your backup file to confirm what has been backed up. You have the ability to edit this table of contents and use the revision to control which database objects will be restored.
- Just as *pg_dump* allows you to do selective backups of objects to save time, *pg_restore* allows you to do selective restores even from a backup that contains a full database.
- For the most part, *pg_restore* and *pg_dump* are backward-compatible. You can backup a database on an older version and restore using a newer version.

Refer to "Database Backup: pg_restore" on page 143 for a listing of *pg_restore* command options.

A basic restore command of a compressed or TAR backup would be to first create the database in SQL:

```
CREATE DATABASE mydb;
```

and then restore:

```
pg_restore --dbname=mydb --jobs=4 --verbose mydb.backup
```

If the database is the same as the one you backed up, you can create the database in ones step with the following:

```
pg_restore --dbname=postgres --create --jobs=4 --verbose mydb.backup
```

If you use the `--create` switch, the `--dbname` switch needs to be different from the database being created, since you can't really run anything within the context of a database that has yet to be created. The downside of using --create is that the database name is always the name of the one you backed up and you can't change it during the restore.

If you are running 9.2, you can take advantage of the `--section` switch to restore just the table structure without the actual data. This is useful if you want to use an existing database as a template for a new one. To do so, we would first create the target database using psql or pgAdmin:

```
CREATE DATABASE mydb2;
```

and then use *pg_restore*:

```
pg_restore --dbname=mydb2 --section=pre-data --jobs=4 mydb.backup
```

Managing Disk Space with Tablespaces

PostgreSQL uses tablespaces to ascribe logical names to physical locations on disk. Initializing a PostgreSQL cluster automatically begets two tablespaces: *pg_default*, which stores for all user data and *pg_global*, which stores all system data. These are located in the same folder as your default data cluster. You're free to create tablespaces at will and house them on any server disks. You can explicitly assign default tablespaces for new objects by database. You can also move existing database objects to new ones.

Creating Tablespaces

To create a tablespace, you just need to denote a logical name and a physical folder. The postgres service account needs to have full access to this folder. If you are on a Windows server, use the following command (note the use of Unix-style slashes):

```
CREATE TABLESPACE secondary LOCATION 'C:/pgdata91_secondary';
```

For Unix-based systems, you first have to create the folder or define an fstab location then use this command:

```
CREATE TABLESPACE secondary LOCATION '/usr/data/pgdata91_secondary';
```

Moving Objects Between Tablespaces

You can shuffle database objects among different tablespaces. To move all objects in the database to our secondary tablespace:

```
ALTER DATABASE mydb SET TABLESPACE secondary;
```

To move just a table:

```
ALTER TABLE mytable SET TABLESPACE secondary;
```

Moving a table to another tablespace locks it for the duration of the move.

Verboten

We have seen so many ways that people manage to break their PostgreSQL server that we thought it best to end this chapter itemizing the most common mistakes that people make. For starters, if you don't know what you did wrong the log file could provide

clues. Look for the *pg_log* folder in your PostgreSQL data folder or the root of the PostgreSQL data folder for the log files. It's also quite possible that your server shutdown before a log entry could be written in which case the log won't help you. Should your server fail to restart, try the command line by using:

```
path/to/your/bin/pg_ctl -D your_postgresql_data_folder
```

Delete PostgreSQL Core System Files and Binaries

When people run out of disk space, the first thing they do is panic and start deleting files from the PostgreSQL data cluster folder because it's so big. Part of the reason why this mistake happens so frequently is that some folders such as *pg_log*, *pg_xlog*, and *pg_clog* sound like logging folders that you expect to build up and be safe to delete. There are some files you can safely delete, and some that will destroy your data if you do.

The *pg_log* folder often found in your data folder is a folder that tends to build up, especially if you have logging enabled. Files in this folder can always be safely deleted without issues. In fact, many people just schedule jobs to delete them.

Files in the other folders except for *pg_xlog* should never be deleted, even if they sound like logs. In particular, don't even think of touching `pg_clog`, the active commit log, without getting into trouble.

pg_xlog stores transaction logs. Some systems we've seen are configured to move processed transaction logs in a subfolder called *archive*. You'll often have an archive folder somewhere (not necessarily as a subfolder of *pg_xlog*) if you are running synchoronous replication, continuous archiving, or just keeping around logs if you need to revert to a different point in time. Deleting files in the root of *pg_xlog* will destroy data, however, deleting files in the archived folder will just prevent you from performing point-in-time recovery, or if a slave server hasn't played back the logs, prevent them from fetching them. If you aren't concerned about any of these scenarios, then it's safe to delete or move files in the archive folder.

Be weary of overzealous anti-virus programs, especially on Windows. We've seen cases where AV software removed important binaries in the PostgreSQL *bin* folder. Should PostgreSQL fail to start on a Windows system, the event viewer is the first place to look for clues as to why.

Giving Full Administrative Rights to the Postgres System (Daemon) Account

Many people are under the misconception that the postgres account needs to have full administrative rights to the server. In fact, depending on your PostgreSQL version, if you give the postgres account full administrative rights to the server, your database server may not even start.

The postgres system account should always be created as a regular system user in the OS with just rights to the data cluster and additional tablespace folders. Most installers

will set up the correct permissions for postgres. Don't try to do postgres any favors by giving it more rights than it needs. Granting unnecessary rights leaves your system vulnerable should you fall under an SQL injection attack. There are cases where you'll need to give the postgres account write/delete/read rights to folders or executables outside of the data cluster. With scheduled jobs that execute batch files, this need often arises. We advise you to practice restraint and only grant the minimum rights necessary to get the job done.

Setting shared_buffers Too High

Loading up your server with RAM doesn't mean you can set the `shared_buffers` as high as you'd like. Try it and your server may crash or refuse to start. If you are running PostgreSQL on 32-bit Windows, setting it higher than 512 MB often results in instability. With PostgreSQL 64-bit windows, you can push the envelop a bit higher and even exceed 1 GB without any issues. On some Linux systems, the compiled `SHMMAX` variable is low and `shared_buffers` can't be set higher. Details on how to remedy this issue are detailed in the manual, in the section *Kernel Resources*.

Trying to Start PostgreSQL on a Port Already in Use

If you do this, you'll see errors in your *pg_log* files of the form. Make sure PostgreSQL is not already running. Here are the common reasons why this happens:

- You've already started postgres service.
- You are trying to run it on a port already in use by another service.
- Your postgres service had a sudden shutdown and you have an orphan *postgresql.pid* file in the data folder. Just delete the file and try to start again.
- You have an orphaned PostgreSQL process. When all else fails, kill all running PostgreSQL processes and then start again.

CHAPTER 3
psql

psql is the de rigueur command-line utility packaged with PostgreSQL. Aside from its most common use of running queries, you can use psql as an automated scripting tool, as a tool for importing or exporting data, restoring, database administration, and even go so far as to use it as a minimalistic reporting tool. psql is easy to use. Like any other command-line tool, you just have to be familiar with the myriad of switches involved. If you only have access to a server's command line with no GUI, psql is pretty much your only choice for querying and managing PostgreSQL. If you fall into this category, we suggest that you print out the dump of psql help from the "psql: Interactive and Scriptable" on page 144 and frame it right above your workstation.

Just as the other command-line tools packaged with PostgreSQL, you can forgo explicitly specifying, host, port, user by setting the environment variables `PGHOST`, `PGPORT`, `PGUSER` as described in *Environment Variables* and setting `PGPASSWORD` or using a password file as described in *The Password File*. Should you omit the parameters without having set the environment variables, psql will use the standard defaults. For examples in this chapter, we'll assume you are using default values or have these variables set. If you're using pgAdmin as well, you can jump right to psql using the plugin interface, (see "Accessing psql from pgAdmin" on page 45). A console window will open with psql and already connected directly to the database in pgAdmin.

Interactive psql

To use psql, the first thing you'll want to know is what you can do interactively. You can get help with `psql \?`. For a thorough list of available interactive commands, refer to "psql Interactive Commands" on page 144.

While in psql, to get help on any SQL commands, type `\h` followed by the command as in the following example:

```
\h CREATE TABLE

            Command: CREATE TABLE
            Description: define a new table
Syntax:
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ]
table_name ( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]
:
```

Although you have many more interactive commands than non-interactive ones at your disposal, you can effectively use all interactive commands non-interactively by embedding them into scripts. We'll go into more detail on how to do this in the later sections of this chapter.

Non-Interactive psql

Non-interactive commands means that you ask psql to execute a script file composed of a mix of SQL statements and psql commands. You can alternatively pass one or more SQL statements. These methods are especially applicable to automated tasks. Once you have batched your commands into a file, you can schedule the job to run at regular intervals using a job scheduling agent like pgAgent (covered in "Job Scheduling with pgAgent" on page 55), Unix `crontab` or Windows scheduler. For situations where you have many commands that must be run in sequence or repeatedly, you're better off creating a script first and then running it using psql. There will be far fewer switches to worry about when running psql non-interactively since the details have been embedded in the script file. To execute a file simply use the `-f` switch as follows:

```
psql -f some_script_file
```

If you don't have your commands saved to a file, you can type them in using a `-c` switch. An example follows:

```
psql -d postgresql_book -c "DROP TABLE IF EXISTS dross; CREATE SCHEMA staging;"
```

Notice how you can have multiple SQL statements as long as you separate them with a semicolon. For a more detailed listing of switches you can include, refer to "psql Non-Interactive Commands" on page 146.

You can embed interactive commands inside script files (see Example 3-1).

Example 3-1. Script with psql interactive commands

Contents of *build_stage.psql*:

```
\a      ❶
\t
\g create_script.sql
SELECT 'CREATE TABLE staging.factfinder_import(geo_id varchar(255), geo_id2 varchar(255),
geo_display varchar(255)
 , '|| array_to_string(array_agg('s' || lpad(i::text,2, '0') || ' varchar(255), s' ||
lpad(i::text,2, '0') || '_perc varchar(255) ' ), ',') || ');' ❷
FROM generate_series(1,51) As i;
\o  ❸
\i create_script.sql ❹
```

❶ Since we want the output of our query to be saved as an executable statement, we need to remove the headers by using the `\t` switch and use the `\a` switch to get rid of the extra breaking elements that psql normally puts in. We then use the `\g` switch to force our query output to be redirected to a file.

❷ The use of `lpad` is so that each numbered column is left padded with 0s so we will have columns s01, s01_perc, s02, s02_perc, ... The `lpad` and similar functions are detailed in "String Functions" on page 63.

❸ We call the `\o` with no file arguments to stop redirection of query results to file.

❹ To do the actual execution of the `CREATE TABLE` statement we built, we use the `\i` followed by the generated script. The `\i` is the interactive version of the non-interactive `-f` switch.

To run Example 3-1, we would type:

```
psql -f build_stage.psql -d postgresql_book
```

Example 3-1 is an adaptation of an approach we described in *How to Create an N-column Table*. As noted in the article, you can perform this without an intermediary file by using the `DO` command introduced in PostgreSQL 9.0. The intermediary does have the benefit that you have an easy record of what was done.

Session Configurations

If you do use psql as your workhorse, consider customizing your psql environment. psql can read configuration settings from a file called *psqlrc*. When psql is launched, it searches for this file and runs any commands in the file to initialize the environment. On Unix-based systems, the file is generally named *.psqlrc* and searched for in the home directory. On Windows, this file is called *psqlrc.conf* and searched for in the *%APPDATA%\postgresql* folder, which usually resolves to *C:\Users\your_login\AppData\Roaming\postgresql*. Don't worry if you can't find the file; it usually doesn't appear on its own and you need to manually create it. Any settings in the file will override psql

defaults. More details about this file can be found in psql. You can find examples of *psqlrc* at *psqlrc File for DBAs* and *Silencing Commands in .psqlrc*.

If you wish to start psql without checking *psqlrc*, use the `-X` switch.

In PostgreSQL 9.2, psql understands two new OS environment variables:

- `PSQL_HISTORY` allows you to control where psql names and places the history file instead of using the default *~/.psql_history*.
- `PSQLRC` allows you to control the location of the startup file. Setting this before launching psql, or as part of your system environment settings, will make psql use this location.

The contents of a *psqlrc* file look as shown in Example 3-2. Pretty much any psql command can be added to it for execution at startup.

Example 3-2. Example .psqlrc or psqlrc.conf file

```
\pset null 'NULL'
\encoding latin1
\set PROMPT1 '%n@%M:%>%x %/# '
\set PROMPT2 ''
\timing on
\set qstats91 'SELECT usename, datname, substring(current_query, 1,100) || ''...'' As query
FROM pg_stat_activity WHERE current_query != ''<IDLE>'';'
\set qstats92 'SELECT usename, datname, left(query,100) || ''...'' As query FROM
pg_stat_activity WHERE state != ''idle'' ;'
\pset pager always
```

Each set command should be on a single line. For example, the `qstats91` statement and its value should be all on the same line.

Some commands only work on Unix-based systems and not on Windows, so our `psqlrc` is fairly generic.

When you launch psql now, you'll see the execution result of your `psqlrc` as follows:

```
Null display is "NULL".
Timing is on.
Pager is always used.
psql (9.2beta1)
Type "help" for help.
postgres@localhost:5442 postgresql_book#
```

We'll cover some popular settings found in *psqlrc* files. You can still set them during your session if you don't want them on or off by default.

Changing Prompts

If you do most of your work using psql and you connect to multiple databases and servers, chances are you'll be jumping around between them using `\connect`. Customizing your prompt to show which server and database and connected user you're on helps greatly. In our *psqlrc* file, we set our prompt to include who we are logged in as (`%n`), the host server (`%M`), the port `%>`, transaction status `%x`), and the database (`%/`).

The cryptic short-hand symbols we used to define our `PROMPT1` and `PROMPT2` in Example 3-2 are documented in the *psql Reference Guide*.

When we connect with psql to our database, our prompt looks like:

```
postgres@localhost:5442 postgresql_book#
```

If we change to another database say `\connect postgis_book`, our prompt changes to

```
postgres@localhost:5442 postgis_book#
```

Timing Details

You may find it instructive to have psql output the time it took for each query to execute. Use the `\timing` command to toggle it on and off.

When that is enabled, each query you run will include at the end, the amount of time taken, for example:

```
\timing on

SELECT COUNT(*) FROM pg_tables;
```

will output the following:

```
count
--------
73
(1 row)
Time: 18.650 ms
```

AUTOCOMMIT

By default, `AUTOCOMMIT` is on, meaning any SQL command you issue that changes data will immediately commit. Each command is its own transaction. If you are doing a large batch of precarious updates, you may want a safety net. Start by turning AUTOCOMMIT off:

```
\set AUTOCOMMIT off
```

Once `AUTOCOMMIT` is off, you'll have the option to rollback before you commit:

```
UPDATE census.facts SET short_name = 'this is a mistake';
```

To roll this back:

```
ROLLBACK;
```

To commit:

```
COMMIT;
```

Shortcuts

The `\set` command is also useful for defining user-defined shortcuts. You may want to store the shortcuts in your *psqlrc* file to have them available each time. For example, if you use `EXPLAIN ANALYZE VERBOSE` all the time and you're tired of typing it all out, you can define a variable as follows:

```
\set eav 'EXPLAIN ANALYZE VERBOSE'
```

Now whenever you want to do an `EXPLAIN ANALYZE VERBOSE` of a query, you prefix it with `:eav` (colon resolves the variable):

```
:eav SELECT COUNT(*) FROM pg_tables;
```

You can even save commonly used queries as strings in your *psqlrc* startup script as we did for `qstats91` and `qstats92`. So, if I am on a PostgreSQL 9.2 database, I can see current activity by just typing the following:

```
:qstats92
```

Retrieving Prior Commands

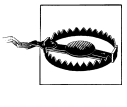
As with many command line tools, you can use the up arrows to access prior commands. The number of previous commands stored in the command history is controlled using the `HISTSIZE` variable. For example:

```
\set HISTSIZE 10
```

will allow you to recover the past ten commands and no more from the command history file.

You can also have psql pipe the history of commands into separate files for each database using a command like the following:

```
\set HISTFILE ~/.psql_history- :HOST - :DBNAME
```



The psql history feature generally doesn't work on Windows unless running under Cygwin. This feature relies on the readline library, which Windows distributions are generally not compiled with. For the same reason, tab completion also doesn't work.

Finally, to unset a variable in psql, simply issue the `\unset` command followed by the variable name. For example:

```
\unset qstats91
```

psql Gems

In this section, we cover some really helpful features that are buried inside psql help.

Executing Shell Commands

Although you normally use SQL and psql specific commands in psql, you can call out to the OS shell using the ! command. Let's say you're on Windows and need to get a list of all OS environment settings that start with A. Instead of exiting psql, you can just directly type the following:

```
\! set A

ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\Administrator\AppData\Roaming
```

Lists and Structures

There are various psql commands available to get lists of objects along with details. In Example 3-3, we demonstrate how to list all tables in schema pg_catalog that start with *pg_t*, along with their size.

Example 3-3. List tables with \dt+

```
\dt+ pg_catalog.pg_t*

Schema     |       Name       | Type  |  Owner   |  Size  | Description
-----------+------------------+-------+----------+--------+-------------
pg_catalog | pg_tablespace    | table | postgres | 40 kB  |
pg_catalog | pg_trigger       | table | postgres | 16 kB  |
pg_catalog | pg_ts_config     | table | postgres | 40 kB  |
pg_catalog | pg_ts_config_map | table | postgres | 48 kB  |
pg_catalog | pg_ts_dict       | table | postgres | 40 kB  |
pg_catalog | pg_ts_parser     | table | postgres | 40 kB  |
pg_catalog | pg_ts_template   | table | postgres | 40 kB  |
pg_catalog | pg_type          | table | postgres | 112 kB |
```

If we wanted detail about a particular object such as the pg_ts_config table, we would use the \d command, as shown in Example 3-4.

Example 3-4. Describe object with \d

```
\d+ pg_ts_dict

Table "pg_catalog.pg_ts_dict"
Column         | Type | Modifiers | Storage  | Stats target | Description
---------------+------+-----------+----------+--------------+-------------
dictname       | name | not null  | plain    |              |
dictnamespace  | oid  | not null  | plain    |              |
dictowner      | oid  | not null  | plain    |              |
dicttemplate   | oid  | not null  | plain    |              |
dictinitoption | text |           | extended |              |
Indexes:
```

```
"pg_ts_dict_dictname_index" UNIQUE, btree (dictname, dictnamespace)
"pg_ts_dict_oid_index" UNIQUE, btree (oid)
Has OIDs: yes
```

Importing and Exporting Data

psql has a command called `\copy` for both importing from and exporting to a delimited text file. The default delimiter is tab with rows separated by new line breaks. For our first example, we downloaded data from *US Census Fact Finder* covering racial demographics of housing in Massachusetts. You can download the file from *PostgreSQL Book Data*. Fact Finder is a treasure trove of data about the US; a statistician's dream land. We encourage you to explore it via the guided wizard. Our usual practice in loading denormalized or unfamiliar data is to create a separate schema to segregate it from production data. We then write a series of explorative queries to get a good sense of what we have on our hands. Finally, we distribute the data into various normalized production tables and delete the staging schema.

Before bringing the data into PostgreSQL, you must first create a table to hold the incoming data. The data must match the file both in the number of columns and data types. This could be an annoying extra step for a well-formed file, but does obviate the need for psql to guess at data types. psql processes the entire import as a single transaction; should it encounter any errors in the data, the entire import will fail. If you're unsure about the data contained in the file, we recommend setting up the table with the most accommodating data types and then recast later if necessary. For example, if you can't be sure that a column will just have numeric values, make it character varying to get the data in for inspection and then recast later.

Example 3-5. Importing data with psql

```
psql
\connect postgresql_book
\cd /postgresql_book/ch03
\copy staging.factfinder_import FROM DEC_10_SF1_QTH1_with_ann.csv CSV
```

In Example 3-5, we launch psql interactively, connect to our database, use `\cd` to change the current directory to the folder with our data and then import our data using the `\copy` command. Since default for copy is tab-delimited, we need to augment our statement with `CSV` to denote that our data is comma separated instead of tab delimited.

If you had data with non-standard delimiters like | delimited columns and you also wanted to replace blank data points with nulls, you would use a command:

```
\copy sometable FROM somefile.txt DELIMITER '|' NULL As '';
```

There is another COPY command, which is part of the SQL language (not to be confused with the `\copy` in psql) that requires the file be on the server. Since psql is a client utility, all path references are relative to the client while the SQL version is relative to the server and runs under the context of the postgres OS process account. We detail the differences between the two in *Import Fixed-width Data in PostgreSQL with just psql*.

Another favorite tool for loading data with more options for loading data from a variety of sources is *pgloader*. You will have much finer control over the import process, but Python and psychopg must be installed on your machine first.

Exporting data is even easier than importing data. You can even export a subset of a table. As mentioned, the psql `\copy` command and companion *SQL COPY* allow you to do just that. In Example 3-6, we demonstrate how to export the data we just loaded back to tab format.

Example 3-6. Export data with psql

```
\connect postgresql_book
\copy (SELECT * FROM staging.factfinder_import  WHERE s01 ~ E'^[0-9]+' ) TO /test.tab WITH
DELIMITER E'\t' CSV HEADER
```

The default behavior of exporting data without qualifications is to export as tab delimited. However the default doesn't export the header columns. In fact, as of the time of writing, you can only use the `HEADER` in conjunction with the `CSV` option.

Example 3-7. Export data with psql

```
\connect postgresql_book
\copy staging.factfinder_import TO /test.csv WITH CSV HEADER QUOTE '"' FORCE QUOTE *
```

The `FORCE QUOTE *` ensures that all columns are double quoted. For clarity, we also indicate the quoting character, though double quotes is assumed if omitted.

Basic Reporting

Believe it or not, psql is capable of doing basic HTML reports. Try the following and check out the HTML output.

```
psql -d postgresql_book -H -c "SELECT category, count(*) As num_per_cat FROM
pg_settings WHERE category LIKE '%Query%' GROUP BY category ORDER BY category;" -o
test.html
```

Not too shabby! The above, however, just outputs an HTML table instead of a fully qualified HTML document. To create a meatier report, you'd compose a script as shown in Example 3-8.

category	num_per_cat
Query Tuning / Genetic Query Optimizer	7
Query Tuning / Other Planner Options	5
Query Tuning / Planner Cost Constants	6
Query Tuning / Planner Method Configuration	11
Statistics / Query and Index Statistics Collector	6

(5 rows)

Figure 3-1. Minimalist HTML report

Example 3-8. Settings report

content of settings_report.psql

```
\o settings_report.html ❶
\T 'cellspacing=0 cellpadding=0' ❷
\qecho '<html><head><style>H2{color:maroon}</style>'  ❸
\qecho '<title>PostgreSQL Settings</title></head><body>' ❹
\qecho '<table><tr valign=''top''><td><h2>Planner Settings</h2>' ❺
\x on ❻
\t on ❼
\pset format html ❽
SELECT category, string_agg(name || '=' || setting, E'\n' ORDER BY name ❾ ) As settings  ❿
FROM pg_settings
WHERE category LIKE '%Planner%'
GROUP BY category
ORDER BY category;
\H
\qecho '</td><td><h2>File Locations</h2>' ⓫
\x off ⓬
\t on ⓭
\pset format html ⓮
SELECT name, setting FROM pg_settings WHERE category = 'File Locations' ORDER BY name;
\qecho '<h2>Memory Settings</h2>'
SELECT name, setting, unit FROM pg_settings WHERE category ILIKE '%memory%' ORDER BY name;
\qecho '</td></tr></table>'
\qecho '</body></html>'
\o
```

❶ Redirect query output to a file.

❷ HTML table settings for query output.

❸❹❺⓫ Write additional content beyond the query output to our output file.

❻ Set to expanded mode. The first query is output in expanded mode which means that the column headers are repeated for each row and the column of each row is output as a separate row.

❽⓮ Force the queries to output as HTML tables.

❿❾ We use the aggregate function `string_agg()`, which was introduced in PostgreSQL 9.0 to concatenate all properties in the same category into a single column. We are also taking advange of the new `ORDER BY` clause for aggregate functions introduced in 9.0 to sort properties by name.

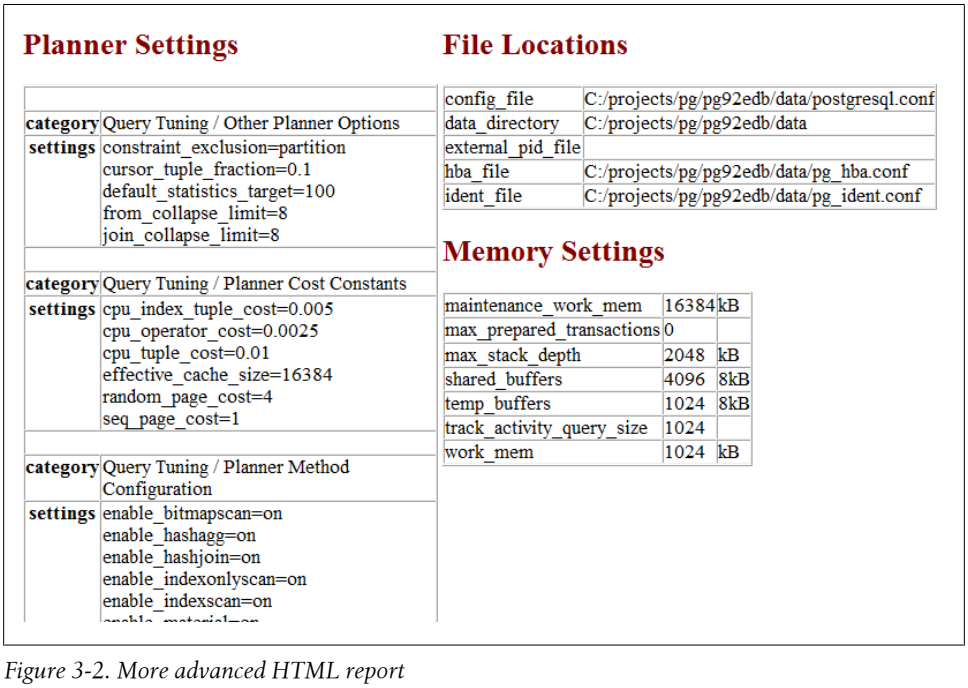
⓬ Set to expanded mode off. The second and third query are output in non-expanded mode which means that there is one output row per table row.

❼⓭ Set to tuples only mode. This causes queries to not have any column headers or row count.

Example 3-8 demonstrates that with the interspersing of SQL and some psql commands, we can create a fairly comprehensive tabular report consisting of many sub reports.

You run Example 3-8 by connecting interactively with psql and using the `\i settings_report.psql` command, or running on the command line using the `psql -f settings_report.psql`.

The generated output of *settings_report.html* is shown in Figure 3-2.



Planner Settings

category	Query Tuning / Other Planner Options
settings	constraint_exclusion=partition cursor_tuple_fraction=0.1 default_statistics_target=100 from_collapse_limit=8 join_collapse_limit=8
category	Query Tuning / Planner Cost Constants
settings	cpu_index_tuple_cost=0.005 cpu_operator_cost=0.0025 cpu_tuple_cost=0.01 effective_cache_size=16384 random_page_cost=4 seq_page_cost=1
category	Query Tuning / Planner Method Configuration
settings	enable_bitmapscan=on enable_hashagg=on enable_hashjoin=on enable_indexonlyscan=on enable_indexscan=on

File Locations

config_file	C:/projects/pg/pg92edb/data/postgresql.conf
data_directory	C:/projects/pg/pg92edb/data
external_pid_file	
hba_file	C:/projects/pg/pg92edb/data/pg_hba.conf
ident_file	C:/projects/pg/pg92edb/data/pg_ident.conf

Memory Settings

maintenance_work_mem	16384	kB
max_prepared_transactions	0	
max_stack_depth	2048	kB
shared_buffers	4096	8kB
temp_buffers	1024	8kB
track_activity_query_size	1024	
work_mem	1024	kB

Figure 3-2. More advanced HTML report

Having a script means that you can output many queries in one report, and of course, schedule it as a job via pgAgent or crontab.

CHAPTER 4

Using pgAdmin

pgAdmin (a.k.a. pgAdmin III or pgAdmin3) is the current rendition of the most commonly used graphical administration tool for PostgreSQL. Though it has its shortcomings, we are always encouraged by not only how quickly bugs are fixed, but also how quickly new features are added. Since it's accepted as the official graphical administration tool for PostgreSQL, and packaged with many binary distributions of PostgreSQL, pgAdmin has the responsibility to always be kept in sync with the latest PostgreSQL releases. Should a new release of PostgreSQL induct new features, you can count on the latest pgAdmin to let you manage it. If you're new to PostgreSQL, you should definitely start with pgAdmin before exploring other tools that could cost money. We should also mention that as of yet, we have not encountered a tool that's absolutely superior to pgAdmin.

Getting Started

Get pgAdmin at *http://www.pgadmin.org*. While on the site, you may opt to peruse one of the guides that'll introduce pgAdmin, but the tool is well-organized and, for the most part, guides itself quite well. For the adventurous, you can always try beta and alpha releases of pgAdmin. Your help in testing would be greatly appreciated by the community.

Overview of Features

To whet your appetite, here's a list of goodies found in *pgAdmin* that are our favorites. There are many more you can find listed on *pgAdmin Features*:

- Graphical `EXPLAIN` plan for your queries. This most awesome feature offers a pictorial insight into what the query planner is thinking. Gone are the days of trying to wade through the verbosity of text-based planner outputs.
- SQL pane. pgAdmin ultimately interacts with PostgreSQL via SQL; it's not shy about letting you see the generated SQL. When you use the graphical interface to

make changes to your database, the underlying SQL to perform the tasks automatically displays in the SQL pane. For SQL novices, studying the generated SQL is a great learning opportunity. For pros, taking advantage of the generated SQL is a great time-saver.

- Direct editing of configuration files such as *postgresql.conf* and *pg_hba.conf*. You no longer need to dig around for the files and use another editor.
- Data export. pgAdmin can easily export query results as CSV or other delimited format. It can even export as HTML, providing you with a turn-key reporting engine, albeit a bit crude.
- Backup and restore wizard. Can't remember the myriad of commands and switches to perform a backup or restore using *pg_restore* and *pg_dump*? *pgAdmin* has a nice interface that'll let you selectively back up and restore databases, schemas, single tables, and globals, and the message tab shows you the command line *pg_dump* or *pg_restore* it used to do it.
- Grant Wizard. This time-saver will allow you to change permissions on many database objects in one fell swoop.
- pgScript engine. This is a quick and dirty way to run scripts that don't have to complete as a transaction. With this you can run loops and so forth that commit on each SQL update, unlike stored functions that require all steps completed before the work is committed. Unfortunately, you can not use it outside of pgAdmin GUI.
- Plugin architecture. Newly developed add-ons are quickly accessible with a single mouse-click. You can even install your own. We have a description of this feature in *Change in pgAdmin Plugins and PostGIS*.
- *pgAgent* plugin. We'll be devoting an entire section to this cross-platform job scheduling agent which is similar in flavor to SQL Server's job scheduler (SQLAgent). pgAdmin provides a cool interface to it.

Connecting to a PostgreSQL server

Connecting to a PostgreSQL server with pgAdmin is fairly self-explanatory. The General and Advanced tabs are shown in Figure 4-1.

Navigating pgAdmin

pgAdmin's tree layout is intuitive to follow but does start off showing you every esoteric object found in the database. You can pare down the display tree by going into the Options tab and unchecking objects that you would rather not have to stare at every time you use *pgAdmin*.

To simplify the tree sections, go to Tools→Options→Browser, you will see a screen as shown in Figure 4-2.

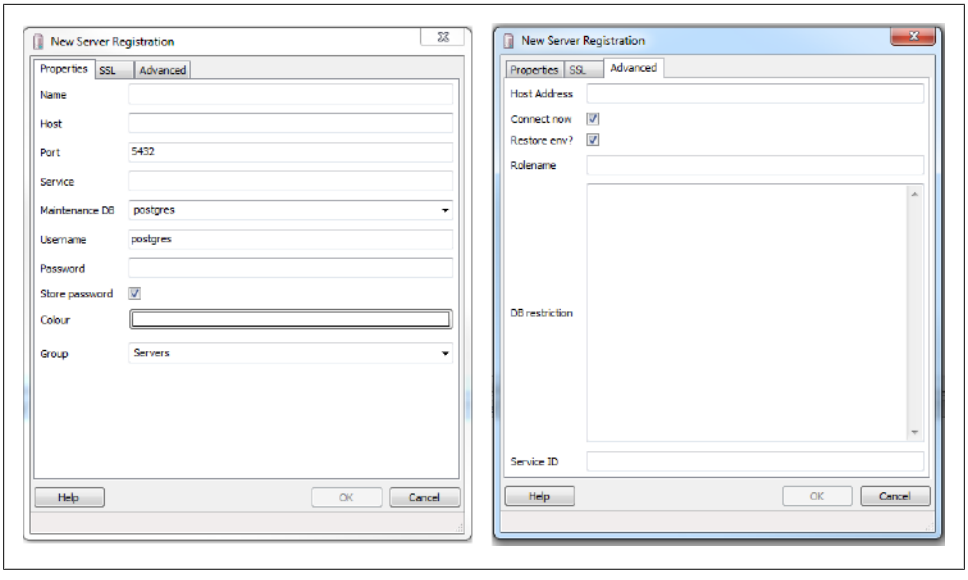
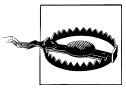


Figure 4-1. pgAdmin register server connection dialog

If you check the *Show System Objects* in the treeview check box, you'll see the guts of PostgreSQL consisting of internal functions, system tables, hidden columns in each table, and so forth. You will also see the metadata stored in the information_schema catalog and the pg_catalog PostgreSQL system catalog. information_schema is an ANSI-SQL standard catalog found in other databases such as MySQL and SQL Server. You may recognize some of the tables and columns from working with other databases and its superb for getting standard metadata in a cross database compatible way.



pgAdmin does not always keep the tree in sync, with current state of the database. For example, if one person alters a table, the tree for a second person will not automatically refresh. There is a setting in recent versions that forces an automatic refresh if you check it, but may slow things down a bit.

pgAdmin Features

pgAdmin is chock full of goodies. We won't have the space to bring them all to light so we'll just highlight the features that we use on a regular basis.

Accessing psql from pgAdmin

Although *pgAdmin* is a great tool, there are cases where psql does a better job. One of those cases is executing large SQL files such as those output by *pg_dump* and other dump tools. To do this, you'll want to use psql covered in Chapter 3. pgAdmin has a feature that makes jumping to psql easy and painless. If you click on the `plugin` menu

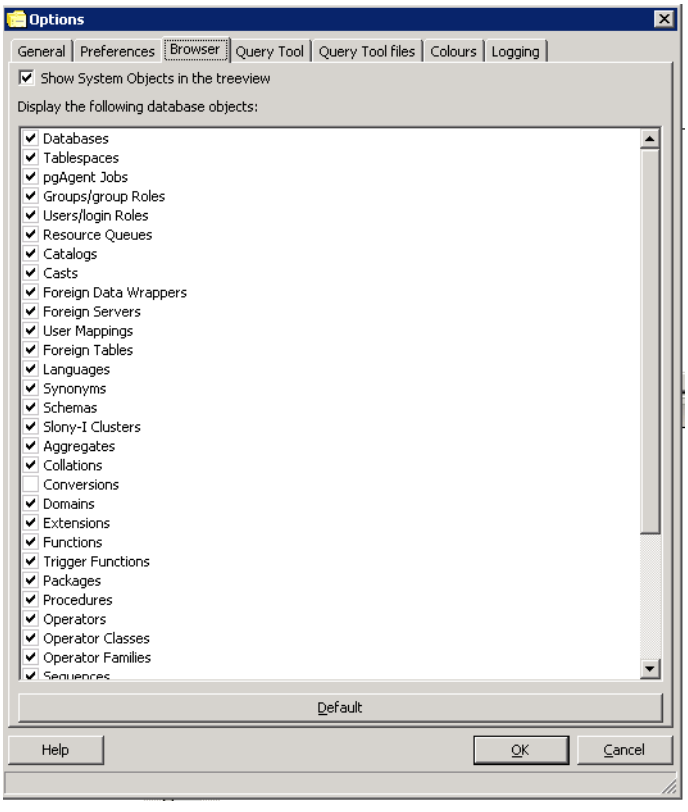


Figure 4-2. Hide or unhide database objects in pgAdmin browse tree

item as shown in Figure 4-3 and then psql, this will open a psql session connected to the database you are currently connected to in *pgAdmin*. You can use the `\cd` and `\i` psql commands to cd and run a psql script file.

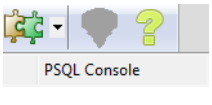


Figure 4-3. psql plugin

Since this feature relies on a database connection, you'll see it disabled until you're connected to a database.

Editing postgresql.conf and pg_hba.conf from pgAdmin

To edit configuration files directly from pgAdmin, you need to have the admin pack extension installed on your server. If you installed PostgreSQL using one of the one-click installers, you should see the menu enabled as shown in Figure 4-4.

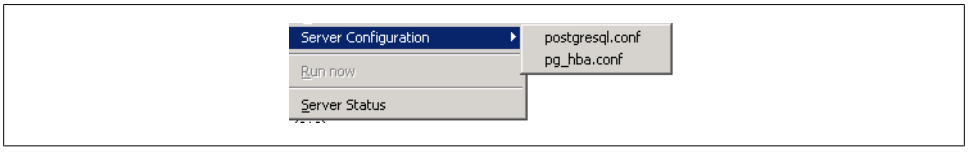


Figure 4-4. PgAdmin configuration file editor

If the menu is greyed out and you are connected to a PostgreSQL server, then you don't have the admin pack installed on that server or are not logged in as a superuser. To install the admin pack on a 9.0 or lower server, connect to the database named postgres as a superuser and run the file *share\contrib\adminpack.sql*. For PostgreSQL 9.1 or above, connect to the database named *postgres* and run the SQL statement `CREATE EXTENSION adminpack;`, or use the graphical interface for installing extensions as shown in Figure 4-5. Disconnect from the server and reconnect, and you should see the menu enabled.

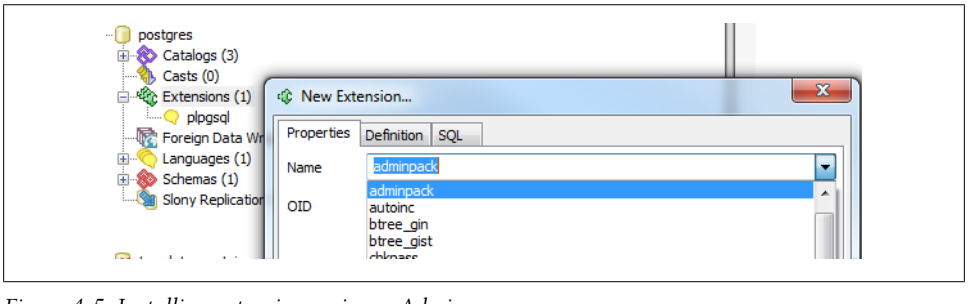


Figure 4-5. Installing extensions using pgAdmin

Creating Databases and Setting Permissions

Creating Databases and Other Objects

Creating a database in *pgAdmin* is simple. Just right-click on the database section of the tree and choose *New Database* as shown in Figure 4-6. The definition tab provides a drop down to use a specific template database, similar to what we did in "Creating and Using a Template Database" on page 16.

You'd follow the same steps to create roles, schemas, and other objects. Each will have its own relevant set of tabs for you to specify additional attributes.

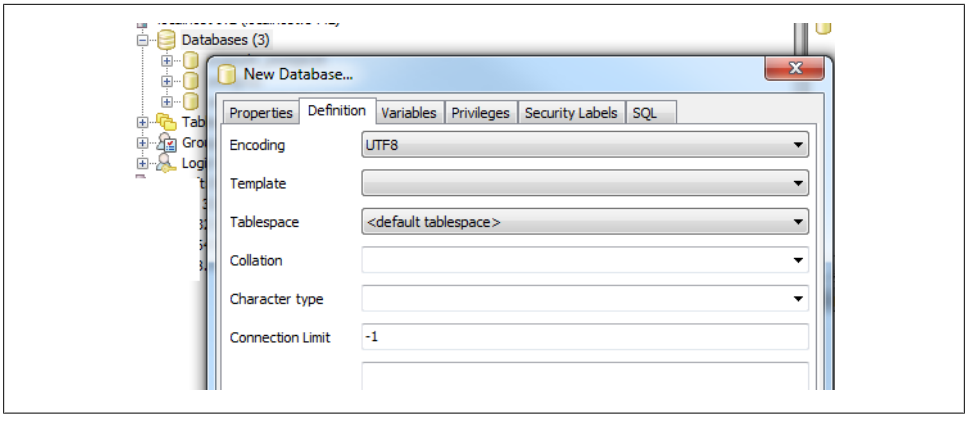


Figure 4-6. Creating a new database

Permission Management

For setting permissions on existing objects, nothing beats the *pgAdmin* Grant Wizard, which you can access from the *Tools→Grant Wizard* menu of *pgAdmin*. As with many other features, this option is greyed out unless you are connected to a database. It's also sensitive to the location in the tree you are on. For example, to set permissions in objects located in the census schema, we select the census and then choose the Grant Wizard. The grant wizard screen is shown in Figure 4-7. You can then selectively check all or some of the objects and switch to the `Privileges` tab to define the roles and permissions you want to grant.

More often than setting permissions on existing objects, you may want to set default privileges for new objects in a schema or database. To do, so right-click the schema or database, select `Properties`, and then go to the `Default Privileges` tab and set permissions for the desired object types as shown in Figure 4-8. The default privileges feature is only available if you are running PostgreSQL 9.0 or above.

When setting permissions for schema, make sure to also set the USAGE permission on the schema to the groups you will be giving access.

Backup and Restore

Most of the backup and restore features of *pg_dump* and *pg_restore* are accessible from *pgAdmin*. In this section, we'll repeat some of the examples we covered in "Backup" on page 22 and "Restore" on page 24, but using *pgAdmin*'s graphical interface instead of the command line. The backup and restore in pgAdmin are just GUIs to the underlying *pg_dump* and *pg_restore* utilities. If you have several versions of PostgreSQL or pgAdmin installed on your computer, it's a good idea to make sure that the *pgAdmin* version is using utilities versions that you expect. Check what the *bin* setting in *pgAdmin* is pointing to in order to ensure it's the latest available, as shown in Figure 4-9.

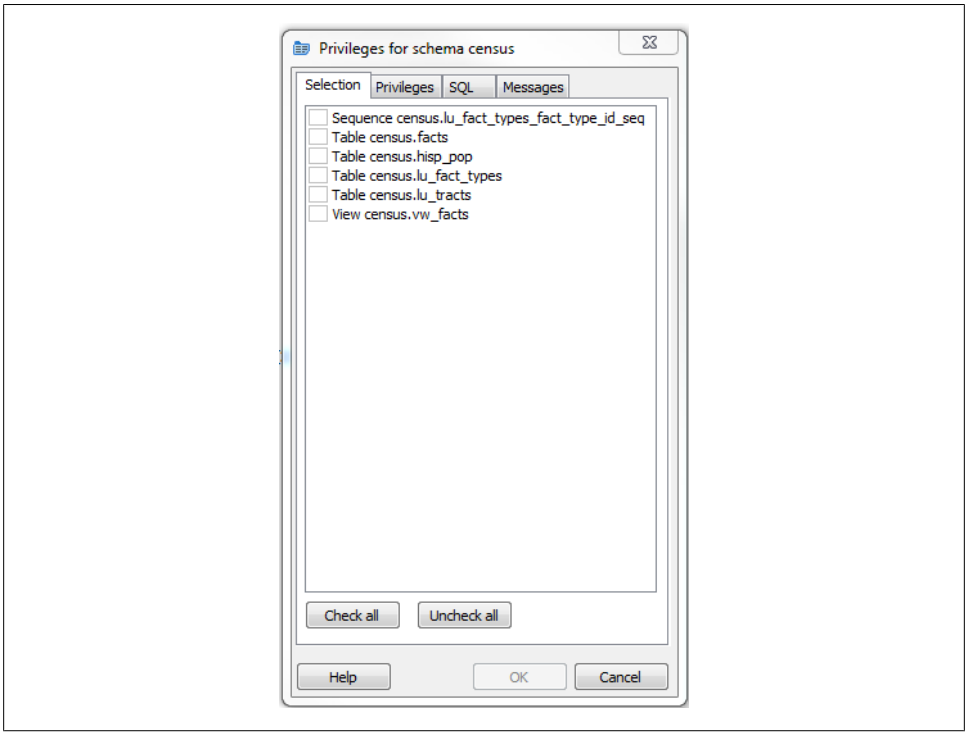
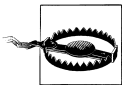


Figure 4-7. Grant Wizard



If your server is remote or your databases are huge, we recommend using the command-line tools for backup and restore instead of *pgAdmin* to avoid adding another layer of complexity in what could already be a pretty lengthy process. Also keep in mind that if you do a compressed/tar/directory backup with a newer version of *pg_dump*, then you also need to use the same or higher version of *pg_restore* because a newer *pg_dump* compressed or tar backup can not be restored with an older *pg_restore*.

Backing up a whole database

In Example 2-8, we demonstrated how to back up a database. To repeat the same steps using the *pgAdmin* interface, we would right click on the database we want to backup and choose `custom` for format as shown in Figure 4-10.

Backing up of System Wide Objects

pgAdmin provides a graphical interface to *pg_dumpall* for backing up system objects, which does much the same as what we covered in "Systemwide Backup Using

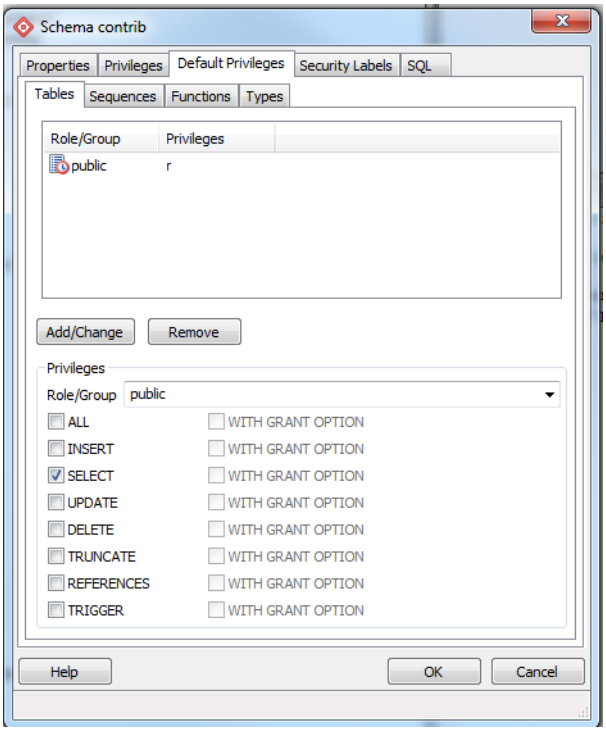


Figure 4-8. Grant Default Permissions

pg_dumpall" on page 24. To use, first connect to the server you want to backup from the Server tree and then from the top menu, choose Tools → Backup Globals.

Unfortunately, *pgAdmin* doesn't give you any options of what to backup as you get by using the command line interface, and instead will backup all table spaces and roles. If you want to backup the whole server, doing a *pg_dumpall*, then use the Tools → Backup Server option.

Selective Backup of Database Objects

pgAdmin provides a graphical interface to *pg_dump* that we covered in "Selective Backup Using pg_dump" on page 23 for doing selective backup of objects. To back up selective objects right-mouse click on the object you want to back up and select `Backup ...`. You can back up a whole database, schema, table, or anything else.

If all you wanted to back up was that one object, you can forgo the other tabs and just do as we did in Figure 4-10. However, you can selectively pick or uncheck some more items by clicking on the objects tab as shown in Figure 4-12.

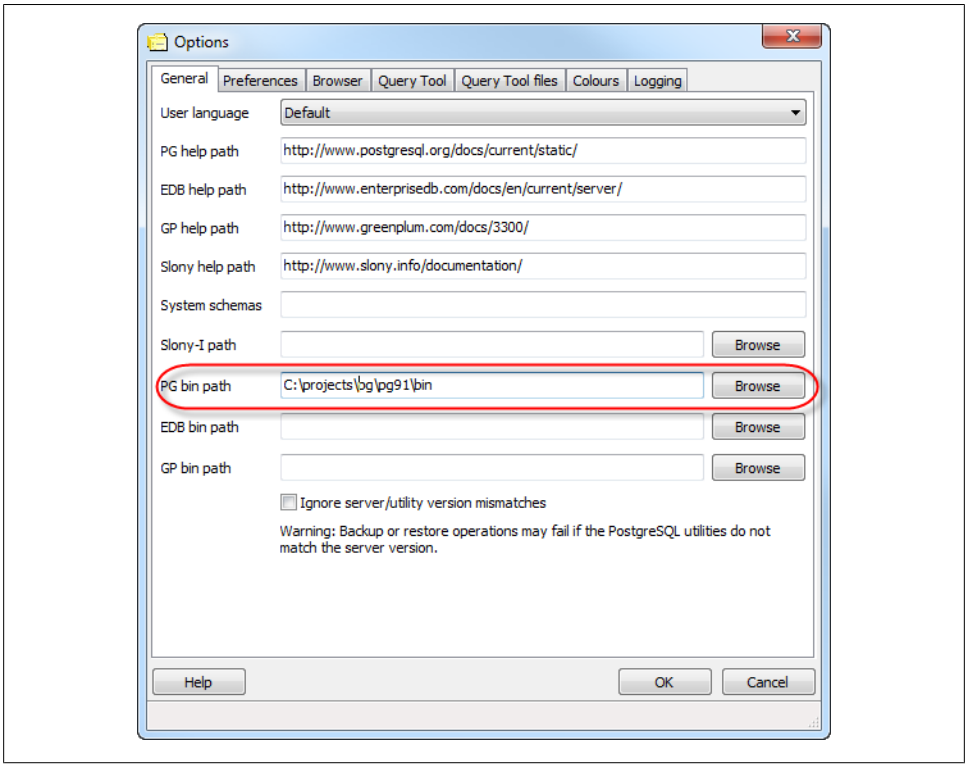


Figure 4-9. pgAdmin File→Options

pgAdmin behind the scenes just runs *pg_dump* to perform the backup. If ever you want to know the actual commands it's doing for later scripting, just look at the *Messages* tab of the backup screen after you click the *Backup* button, and you'll see the exact call with arguments to *pg_dump*.

pgScript

pgScript is a built-in scripting tool in *pgAdmin*. It's most useful for being able to run repetitive SQL tasks. Unlike PostgreSQL stored functions, *pgScript* commits data right away which makes it particularly handy for memory-hungry processes that you don't need completed as a single transaction. You can see an example of where we use it for batch geocoding here at *http://www.postgresonline.com/journal/archives/181-pgAdmin-pgScript.html*.

The underlying language is lazily typed and supports loops, data generators, macro replacement, basic print statements and record variables. The general syntax is similar to that of Transact SQL—the stored procedure language of Microsoft SQL Server. You

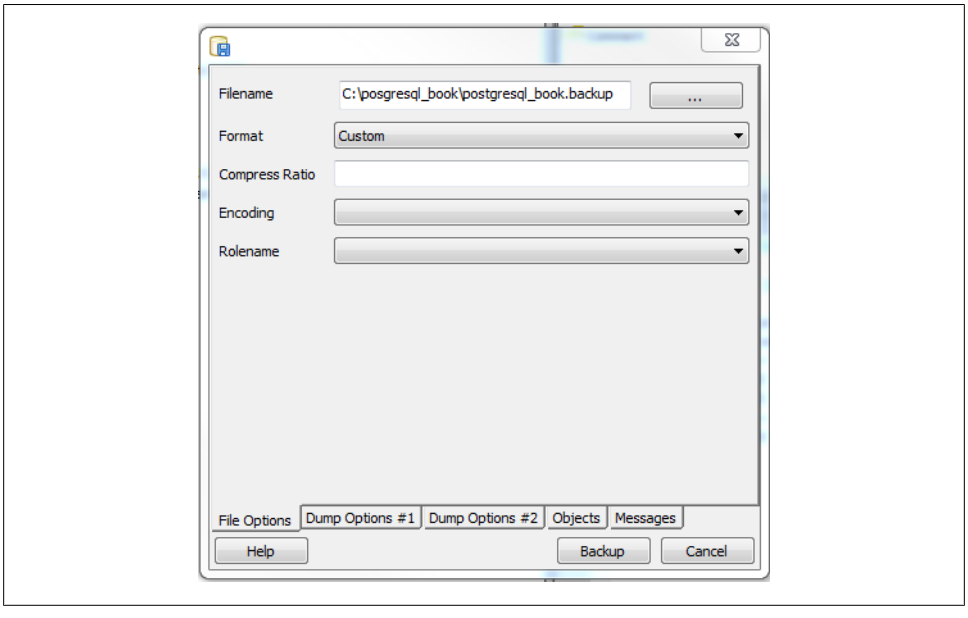


Figure 4-10. Backup database

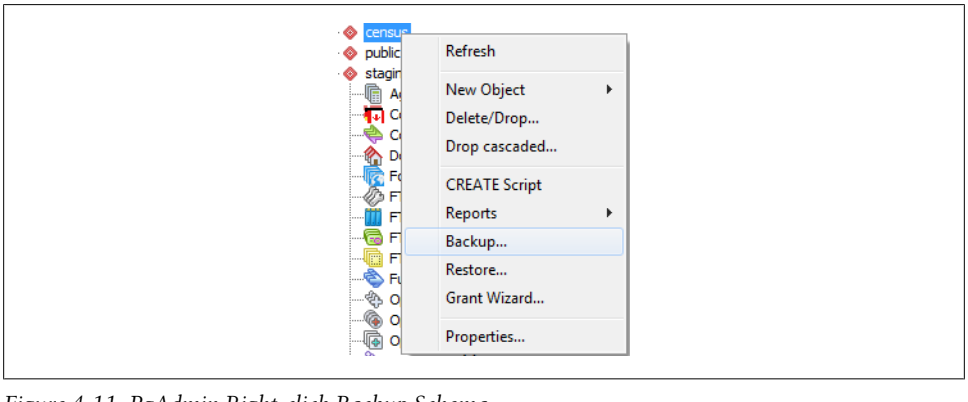


Figure 4-11. PgAdmin Right-click Backup Schema

launch *pgScript* by opening up a query window, typing in some pgScript specific syntax, and then click on the icon to execute it. We'll show you some examples.

Example 4-1 demonstrates how to use *pgScript* record variables and loops to build a cross tab table using the `lu_fact_types` table we create in Example 6-7. It creates an empty table called `census.hisp_pop` with numeric columns of `hispanic_or_latino`, `white_alone`, `black_or_african_american_alone`, and so on.

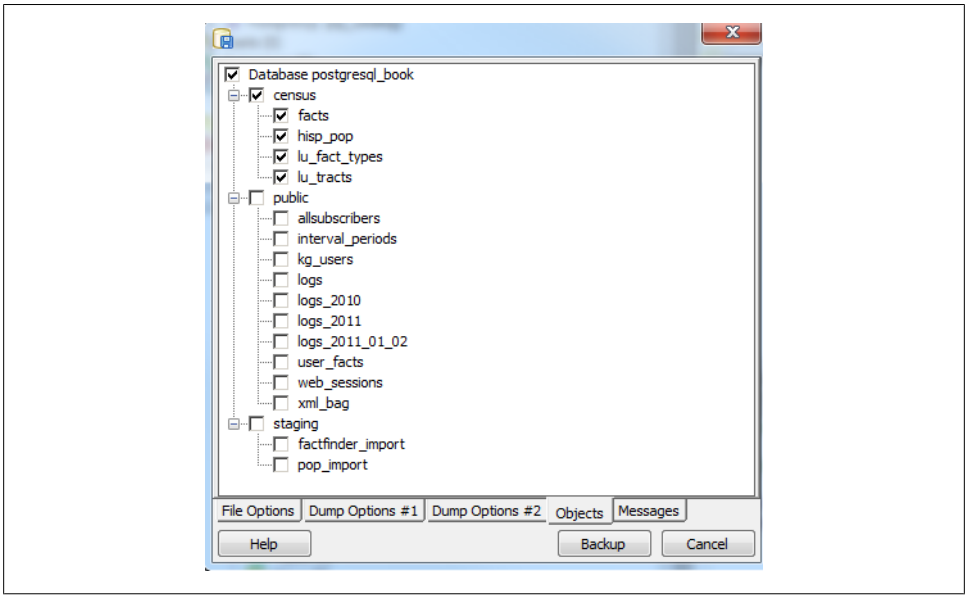


Figure 4-12. PgAdmin Right-click Backup Selective

Example 4-1. Create table using record variables in pgScript

```
DECLARE @I, @labels, @tdef;
SET @I = 0;
labels becomes a record variable
SET @labels = SELECT
quote_ident(replace(replace(lower(COALESCE(fact_subcats[4], fact_subcats[3])), ' ',
'_'),':',''')) As col_name,
fact_type_id
FROM census.lu_fact_types
WHERE category = 'Population' AND fact_subcats[3] ILIKE 'Hispanic or Latino%'
ORDER BY short_name;
SET @tdef = 'census.hisp_pop(tract_id varchar(11) PRIMARY KEY ';

Loop thru records using LINES function
WHILE @I < LINES(@labels)
BEGIN
SET @tdef = @tdef + ', ' + @labels[@I][0] + ' numeric(12,3) ';
SET @I = @I + 1;
END
SET @tdef = @tdef + ')';
print out table def
PRINT @tdef;
create the table
CREATE TABLE @tdef;
```

Although pgScript does not support the `EXECUTE` command like PL/pgSQL for running dynamically generated SQL, we demonstrated in Example 4-1 that it's still possible to

do so by using macro replacement in pgScript. Example 4-2 pushes the envelope a bit further by populating the `census.hisp_pop` table we just created.

Example 4-2. Dynamic Population with pgScript

```
DECLARE @I, @labels, @tload, @tcols, @fact_types;
SET @I = 0;
SET @labels = SELECT
quote_ident(replace(
replace(
lower(
COALESCE(fact_subcats[4], fact_subcats[3])), ' ', '_'),':'
,'')) As col_name, fact_type_id
FROM census.lu_fact_types
WHERE category = 'Population' AND fact_subcats[3] ILIKE 'Hispanic or Latino%'
ORDER BY short_name;
SET @tload = 'tract_id';
SET @tcols = 'tract_id';
SET @fact_types = '-1';
WHILE @I < LINES(@labels)
BEGIN
SET @tcols = @tcols + ', ' + @labels[@I][0] ;
SET @tload = @tload + ', MAX(CASE WHEN fact_type_id = ' + CAST(@labels[@I][1] AS STRING) +
' THEN val ELSE NULL END)' ;
SET @fact_types = @fact_types + ', ' + CAST(@labels[@I][1] As STRING);
SET @I = @I + 1;
END
INSERT INTO census.hisp_pop(@tcols)
SELECT @tload
FROM census.facts
WHERE fact_type_id IN(@fact_types) AND yr=2010
GROUP BY tract_id;
```

The lesson to take away from Example 4-2 is that as long as the beginning of your statement starts with SQL, you can dynamically inject SQL fragments into it anywhere.

Graphical Explain

One of the great gems in *pgAdmin* is its informative, at-a-glance graphical explain of the query plan. You can access the graphical explain plan by opening up an SQL query window, write some query and then clicking on the  icon.

If we run the query:

```
SELECT left(tract_id, 5) As county_code, SUM(hispanic_or_latino) As tot
 , SUM(white_alone) As tot_white
 , SUM(COALESCE(hispanic_or_latino,0) - COALESCE(white_alone,0)) AS non_white
FROM census.hisp_pop
GROUP BY county_code
ORDER BY county_code;
```

We will get a graphical explain as shown in Figure 4-13. The best tip we can give for reading the graphical explain plan is to follow the fatter arrows. The fatter the arrow, the more time consuming a step.

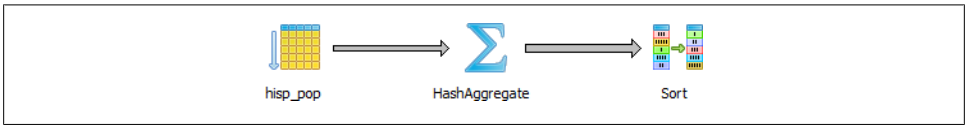


Figure 4-13. Graphical explain example

Graphical explain will be disabled if Query→Explain→Buffers is enabled. So make sure to uncheck buffers before trying a graphical explain. In addition to the graphical explain, the `Data Output`code> tab will show the textual explain plan which for this example looks like:

```
GroupAggregate  (cost=111.29..151.93 rows=1478 width=20)
  Output: ("left"((tract_id)::text, 5)), sum(hispanic_or_latino),
sum(white_alone), ...
  ->  Sort  (cost=111.29..114.98 rows=1478 width=20)
    Output: tract_id, hispanic_or_latino, white_alone, ("left"((tract_id)::text, 5))
    Sort Key: ("left"((tract_id)::text, 5))
    ->  Seq Scan on census.hisp_pop  (cost=0.00..33.48 rows=1478 width=20)
          Output: tract_id, hispanic_or_latino, white_alone, "left"((tract_id)::text,
5)
```

Job Scheduling with pgAgent

pgAgent is a handy utility for scheduling jobs. Since *pgAgent* can execute batch scripts in the OS, we use it for much more than scheduling PostgreSQL jobs. In fact, we don't recall the last time where we even touched crontab or the Windows task scheduler. *pgAgent* goes further, you can actually schedule jobs on any other server regardless of operating system. All you have to do is install the *pgAgent* service, PostgreSQL server itself is not required but the client connection libraries are. Since pgAgent is built atop of PostgreSQL, we are blessed with the added advantage of having access to all the tables controlling the agent. If we ever need to replicate a complicated job multiple times, we can just go into the database tables directly and insert records instead of using the interface to set up each new job. We'll get you started with *pgAgent* in this section, but please visit *Setting up pgAgent and Doing Scheduled Backups* to see more working examples and details of how to set it up.

Installing pgAgent

You can download *pgAgent* from pgAgent Download. The packaged SQL install script will create a new schema named *pgAgent* in the postgres database and add a new section to your *pgAgmin*.

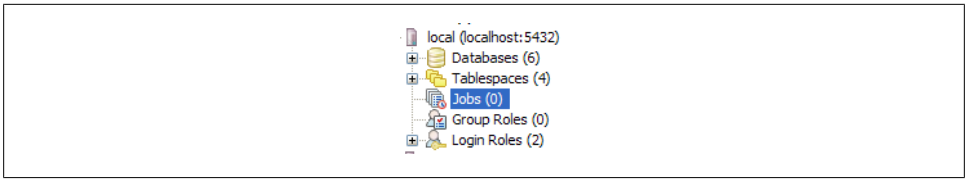
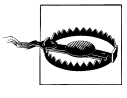


Figure 4-14. pgAdmin with pgAgent installed

Should you wish *pgAgent* to run batch jobs on additional servers, follow the same steps, except for the install of the pgagent SQL script. Pay particular attention to the permission settings of the *pgAgent* service. Make sure each agent has adequate permissions to execute the batch jobs that you will be scheduling.



Batch jobs often fail in *pgAgent* though they may run fine from the command line. This is often due to permission issues. *pgAgent* always runs under the context of the account the *pgAgent* service is running under. If this account doesn't have sufficient permissions or the necessary network path mappings, then it will fail.

Scheduling Jobs

Each scheduled job has two parts: the execution steps and the schedule to run. When you create a new job, you need to specify one or more steps. For each step, you can enter SQL to run, point to a shell script on the OS, or even cut and paste in a full shell script as we commonly do. The syntax for the SQL will not vary across OS and the PostgreSQL server it runs on is controlled by the Connection Type property of the step. The syntax for batch jobs should be specific to the OS running it. For example, if your pgAgent job agent is running on Windows, your batch jobs should have valid DOS commands. If you are on Linux, your batch jobs should have valid sh or bash commands. Steps run in alphabetical order and you can decide what kind of actions you wish to take upon success or failure of each individual step. You also have the option of disabling steps that should remain dormant but you don't want to delete because you may reactivate them later. Once you have the steps ready, go ahead and set up a schedule to run them. You can get fairly detailed with the scheduling screen. You can even set up multiple schedules.

By default, all job agents on other machines will execute all the jobs. If you want to only have the job run on one specific machine, you'll need to fill in the `host agent` field when creating the job. Agents running on other servers will skip the job if it doesn't match their host name.

pgAgent really consists of two parts: the data defining the jobs and storing the job logging, which resides in *pgAgent* schema, usually in postgres database; the job agents query the jobs for the next job to run and then insert relevant logging information in the database. Generally, both the PostgreSQL Server holding the data and the job agent executing the jobs reside on the same server, but in practice they are not required to. Additionally, you can have one PostgreSQL server servicing many job agents residing on different servers.

A fully formed job is shown in Figure 4-15.

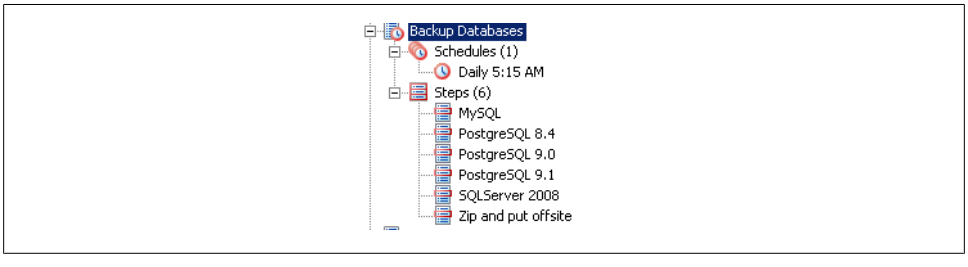


Figure 4-15. pgAgent job shown in pgAdmin

Helpful Queries

To get a glimpse inside the tables controlling all of your agents and jobs, connect to the postgres database and execute the query in Example 4-3:

Example 4-3. Description of pgAgent tables

```
SELECT c.relname As table_name, d.description
FROM pg_class As c
 INNER JOIN pg_namespace n ON n.oid = c.relnamespace
 INNER JOIN pg_description As d ON d.objoid = c.oid AND d.objsubid = 0
WHERE n.nspname = 'pgagent'
ORDER BY c.relname;

table_name     |       description
---------------+-------------------------
pga_job        | Job main entry
pga_jobagent   | Active job agents
pga_jobclass   | Job classification
pga_joblog     | Job run logs.
pga_jobstep    | Job step to be executed
pga_jobsteplog | Job step run logs.
pga_schedule   | Job schedule exceptions
```

As you can see, with your finely-honed SQL skills you can easily replicate jobs, delete jobs, edit jobs directly by messing with pgAgent packaged tables. Just be careful!

Although *pgAdmin* provides an intuitive interface to *pgAgent* scheduling and logging, you may find the need to run your own reports against the system tables. This is especially true if you have many jobs or you just want to do stats on your results. We'll demonstrate the one query we use often.

Example 4-4. List log step results from today

```
SELECT j.jobname, s.jstname, l.jslstart,l.jslduration, l.jsloutput
FROM pgagent.pga_jobsteplog As l
 INNER JOIN pgagent.pga_jobstep As s ON s.jstid = l.jsljstid
 INNER JOIN pgagent.pga_job As j ON j.jobid = s.jstjobid
WHERE jslstart > CURRENT_DATE
ORDER BY j.jobname, s.jstname, l.jslstart DESC;
```

We find it very useful for monitoring batch jobs because sometimes these show as having succeeded when they actually failed. *pgAgent* really can't discern success or failure of a shell script on several operating systems. The jsloutput field provides the shell output, which usually details about what went wrong.

CHAPTER 5
Data Types

PostgreSQL supports the workhorse data types of any database: numerics, characters, dates and times, booleans, and so on. PostgreSQL sprints ahead by adding support for dates and times with time zones, time intervals, arrays and XML. If that's not enough, you can even add your custom types. In this chapter, we're not going to dwell on the vanilla data types, but focus more on showing you ones that are unique to PostgreSQL.

Numeric Data Types

You will find your everyday integers, decimals, and floating point numbers in PostgreSQL. Of the numeric types, we just want to highlight the serial and bigserial data types and a nifty function to quickly generate arithmetic series of integers.

Serial

Strictly speaking, serial is not a data type in its own right. Serial and its bigger sibling bigserial are auto-incrementing integers. This data type goes by different names in different databases, autonumber being the most common alternative moniker. When you create a table and specify a column as type `serial`, PostgreSQL first creates a column of data type integer and then creates a sequence object in the background. It then sets the default of the new integer column to pull its value from the sequence. In PostgreSQL, sequence is a database object in its own right, and an ANSI-SQL standard feature you will also find in Oracle, IBM DB2, SQL Server 2012+, and some other relational databases. You can inspect and edit the object using *pgAdmin* or with `ALTER SEQUENCE`. You can edit its current value, where the sequence should begin and end, and even how many numbers to skip each time. Because sequences are independent objects, you can create them separate from a table using `CREATE SEQUENCE`, and you can share the same sequence among different tables. If you want two tables to never end up with a common identifier field, you could have both tables pull from the same sequence when creating new rows by setting the default value of the integer column to next sequence `nextval()` function.

Generate Series Function

PostgreSQL has a nifty function called `generate_series()` that we have yet to find in other leading databases. It's actually part of a family of functions for automatically creating sequential rows. What makes `generate_series()` such a great function is that it allows you perform a `FOR .. LOOP` like behavior in SQL. Suppose we want a list of the last day of each month for a particular date range. To do this in another language would either involve some procedural loop or creating a massive cartesian product of dates and then filtering. With generate_series, you can do it with a query as shown in Example 5-12.

Here's another example using integers with an optional step parameter:

Example 5-1. generate_series() with stepping of 13

```
SELECT x FROM generate_series(1,51,13) As x;

x
----
1
14
27
40
```

As shown in Example 5-1, you can pass in an optional step argument that defines how many steps to skip for each successive element. Leaving out the step will default it to `1`. Also note that the end value will never exceed our prescribed range, so although our range ends at 51, our last number is 40 because adding another 13 to our 40 exceeds the upper bound.

Arrays

Arrays play an important role in PostgreSQL. They are particularly useful in building aggregate functions, forming `IN` and `ANY` clauses, as well as holding intermediary value for morphing to other data types. In PostgreSQL, each data type, including custom types you build, has a companion array type. For example, integer has an integer array type `integer[]`, character has a character array type `character[]`, and so forth. We'll show you some useful functions to construct arrays short of typing them in manually. We will then point out some handy functions for array manipulations. You can get the complete listing of array functions and operators in the PostgreSQL reference Array Operators and Functions.

Array Constructors

The most rudimentary way to create an array is to simply type the elements:

```
SELECT ARRAY[2001, 2002, 2003] As yrs;
```

If the elements of your array can be extracted from a query, you can use the more sophisticated constructor function: `array()`:

```
SELECT array(SELECT DISTINCT date_part('year', log_ts)
 FROM logs ORDER BY date_part('year', log_ts));
```

Although `array()` has to be used with a query returning a single column, you can specify a composite type as the output, thus achieving multicolumn results. We demonstrate this in "Custom and Composite Data Types" on page 71.

You can convert delimited strings to an array with the `string_to_array()` function as demonstrated in Example 5-2:

Example 5-2. Converting a delimited string to an array

```
SELECT string_to_array('abc.123.z45', '.') As x;

x
---------------
{abc,123,z45}
```

`array_agg()` is a variant function that can take a set of any data type and convert it to an array. See this example Example 5-3:

Example 5-3. Using GROUP BY with array_agg()

```
SELECT array_agg(log_ts ORDER BY log_ts) As x
FROM logs
WHERE log_ts BETWEEN '2011-01-01'::timestamptz AND '2011-01-15'::timestamptz;

x
---------------
{'2011-01-01', '2011-01-13', '2011-01-14'}
```

Referencing Elements in An Array

Elements in arrays are most commonly referenced using the index of the element. PostgreSQL array index starts at 1. If you try to access an element above the upper bound, you won't get an error—only NULL will be returned. The next example grabs the first and last element of our array column.

```
SELECT fact_subcats[1] AS primero
  , fact_subcats[array_upper(fact_subcats, 1)] As ultimo
 FROM census.lu_fact_types;
```

We used `array_upper()` to get the upper bound of the array. The second, required parameter of the function indicates the dimension. In our case, our array is just one-dimensional, but PostgreSQL supports multi-dimensional arrays.

Array Slicing and Splicing

PostgreSQL also supports array slicing using the `start:end` syntax. What gets returned is another array that is a subset of the original. For example, if we wanted to return

from our table new arrays that just contain elements 2 through 4 of each original, we would type:

```
SELECT fact_subcats[2:4] FROM census.lu_fact_types;
```

And to glue two arrays together end to end, we simply use the concatenation operator as follows:

```
SELECT fact_subcats[1:2] || fact_subcats[3:4] FROM census.lu_fact_types;
```

Character Types

There are tree basic types of character types in PostgreSQL: `character` (a.k.a. `char`), `character varying` (a.k.a. `varchar`), and `text`. Unlike other databases you might have worked with, `text` is not stored any differently from `varchar`, and no performance difference for the same size data so PostgreSQL has no need for distinctions like mediumtext, bigtext, and so forth. Even if a type is text, you can still sort by it. Any data larger than what can fit in a record page gets pushed to `TOAST`. So how `text`/`varchar` are stored is only contingent on the actual size of the data in the field and PostgreSQL handles it all for you. When using `varchar`, there are still some gotchas when you try to enlarge the number of characters. If you try to expand the size of an existing `varchar` field for a table with many rows, the process could take a while. People have different opinions as to whether you should abandon the use of `varchar` and just stick with `text`. Rather than waste space arguing about it here, read the debate at In Defense of VarcharX.

The difference between `varchar` with no size modifier and `text` is subtle. `varchar` has a cap around 1 GB and `text` has no limit. In practice. you can do things like override the behavior of `varchar` operators. which you can't do easily with `text`. This override is particularly useful for cross-database compatibility. We demonstrate an example of this in *Using MS Access with PostgreSQL*, where we show how to make `varchar` behave without case sensitivity and still be able to use an index. `varchar` without a size modifier is essentially equivalent to SQL Server's `varchar(max)`.

Most people use `text` or `varchar` except for cases where a value should be exactly `n` characters long. The reason for this? `character` is right-padded with spaces out to the specified size for both storage and display; this is more storage costly, though more semantically meaningful for a key that should be a fixed length. For comparison the extra spaces are ignored for character, but not for `varchar`. Performance-wise, there is no speed benefit with using `character` over `varchar` in PostgreSQL.

PostgreSQL has an abundant number of functions for parsing strings. In this section, we'll give some common recipes we've found useful.

String Functions

The most common manipulations done to strings is to pad, trim off white space, and extract substrings. PostgreSQL has no shortage of these functions to aid you in these endeavors. In this section, we'll provide examples of these. These functions have been around since the age of dinosaurs, so regardless of which version of PostgreSQL you're using, you should have all these at your disposal. PostgreSQL 9.0 introduced a new string aggregate function called `string_agg()`, which we demonstrated in Example 3-8. `string_agg()` is equivalent in concept to MySQL's `group_concat()`.

Example 5-4. Using lpad() and rpad() to pad

```
SELECT lpad('ab', 4, '0') As ab_lpad, rpad('ab', 4, '0') As ab_rpad, lpad('abcde', 4, '0')
As ab_lpad_trunc;
```

```
ab_lpad | ab_rpad | ab_lpad_trunc
---------+---------+---------------
00ab    | ab00    | abcd
```

Observe that in Example 5-4, `lpad()` actually truncates instead of padding.

PostgreSQL has several functions for trimming text. These are `trim()` (a.k.a. `btrim()`), `ltrim()`, `rtrim()`. By default, all trim will remove spaces, but you can pass in an optional argument indicating other characters to trim.

Example 5-5. Using trims to trim space and characters

```
SELECT a As a_before, trim(a) As a_trim , rtrim(a) As a_rt, i As i_before, ltrim(i,'0') As
i_lt_0, rtrim(i,'0') As i_rt_0, trim(i,'0') As i_t_0
FROM (SELECT repeat(' ', 4) || i::text || repeat(' ', 4) As a, '0' || i::text As i
  FROM generate_series(0, 200, 50) As i) As x;
```

```
a_before| a_trim |  a_rt   | i_before | i_lt_0 | i_rt_0 | i_t_0
--------+--------+---------+----------+--------+--------+-------
0       | 0      |     0   | 00       |        |        |
50      | 50     |     50  | 050      | 50     | 05     | 5
100     | 100    |     100 | 0100     | 100    | 01     | 1
150     | 150    |     150 | 0150     | 150    | 015    | 15
200     | 200    |     200 | 0200     | 200    | 02     | 2
```

Splitting Strings into Arrays, Tables, or Substrings

There are a couple of functions useful in PostgreSQL for breaking strings apart.

The `split_part()` function is useful for getting an element of a delimited string.

Example 5-6. Get the nth element of a delimited string

```
SELECT split_part('abc.123.z45', '.', 2) As x;

x
--------
123
```

The `string_to_array()` is useful for creating an array of elements from a delimited string. By combining `string_to_array()` with `unnest()` function, you can expand the returned array into a set of rows.

Example 5-7. Convert delimited string to array to rows

```
SELECT unnest(string_to_array('abc.123.z45', '.')) As x;

x
--------
abc
123
z45
```

Regular Expressions and Pattern Matching

PostgreSQL's regular expression support is downright fantastic. You can return matches as tables, arrays, or do fairly sophisticated replace and updates. Back-referencing and other fairly advanced search patterns are also supported. In this section, we'll provide a short-sampling of these. For more information, refer to the official documentation, in the following sections: *Pattern Matching* and *String Functions*.

Our example shows you how to format phone numbers stored simply as contiguous digits:

Example 5-8. Reformat a phone number using back referencing

```
SELECT regexp_replace('6197256719', '([0-9]{3})([0-9]{3})([0-9]{4})', E'\(\\1\) \\2-\\3')
As x;

x
----------------
(619) 725-6719
```

The `\\1`, `\\2`, etc. refers to the elements in our pattern expression. We use the reverse solidus `\(` to escape the parenthesis. The `E'` is PostgreSQL syntax for denoting that a string is an expression so that special characters like `\` would be treated literally.

You might have a piece of text with phone numbers embedded; the next example shows how to extract the phone numbers and turn them into rows all in one step.

Example 5-9. Return phone numbers in piece of text as separate rows

```
SELECT unnest(regexp_matches('My work phone is (619)725-6719. My mobile is 619.852.5083.
 Mi número de casa es 619-730-6254. Call me.',
   E'[(]{0,1}[0-9]{3}[)-.]{0,1}[0-9]{3}[-.]{0,1}[0-9]{4}', 'g')) As x;

x
---------------
(619)725-6719
619.852.5083
619-730-6254
```

Below, we list the matching rules for Example 5-9:

- `[(]{0,1}`: Starts with 0 or 1 (.
- `[0-9]{3}`: Followed by 3 digits.
- `[)-.]{0,1}`: Followed by 0 or 1 of),-, or .
- `[0-9]{4}`: Followed by 4 digits.
- `regexp_matches()` returns a string array consisting of matches of a regular expression. If you don't pass in the `'g'` parameter, your array will just return the first match of the regular expression. The `'g'` stands for global and returns all matches of a regular expression as separate elements.
- `unnest()` is a function introduced in PostgreSQL 8.4 that explodes an array into a row set.

There are many ways to write the same regular expression. `\\d` is shorthand for `[0-9]`, for example. But given the few characters you'd save, we prefer the more descriptive long form.

In addition to the wealth of regular expression functions, you can use regular expressions with `SIMILAR TO` and `~` operators. In the next example, we'll return all description fields with embedded phone numbers.

```
SELECT description FROM mytable WHERE description ~  E'[(]{0,1}[0-9]{3}[)-.]{0,1}[0-9]
{3}[-.]{0,1}[0-9]{4}';
```

Temporal Data Types

PostgreSQL support for temporal data is the best of any database we've come across. In addition to the usual dates and times, PostgreSQL has support for time zones, enabling the automatic handling of DST conversions by region. Details of the various types and DST support is detailed in Data Types. Specialized data types such as interval allows for easy arithmetics using dates and times. Plus, PostgreSQL has the concept of infinity and negative infinity, saving us from explicitly having to create conventions that we'll forget. Finally, PostgreSQL 9.2 unveiled range types that provide support for

date ranges as well as numeric types with companion operators, index bindings and functions for working with them and ability to create new range types. There are nine data types available in a PostgreSQL database for working with temporal data and understanding the distinctions could be important to make sure you choose the right data type for the job. These data types are all defined in the ANSI-SQL 92 specs except for the PostgreSQL range types. Many other leading databases support some, but not all, these data types. Oracle has the most varieties of temporal types, MS SQL 2008+ comes in second, and MySQL of any version comes in last (with no support for time-zones in any version and in lower versions not even properly checking validity of dates).

- `date` just stores the month, day, and year, with no timezone awareness and no concept of hours, minutes, or seconds.
- `time` records hours, minutes, seconds with no awareness of time zone or calendar dates.
- `timestamp` records both calendar dates and time (hours, minutes, seconds) but does not care about the time zone. As such the displayed value of this data won't change when you change your server's time zone.
- `timestamptz` (a.k.a. `timestamp with time zone`) is a time zone-aware date and time data type. Internally, `timestamptz` is stored in Coordinated Universal Time (UTC), but display defaults to the time zone of the server (or database/user should you observe differing time zones at those levels). If you input a timestamp with no time zone and cast to one with time zone, PostgreSQL will assume the server's time zone. This means that if you change your server's time zone, you'll see all the displayed times change.
- `timetz` (a.k.a. `time with time zone`) is the lesser-used sister of timestamptz. It is time zone-aware but does not store the date. It always assumes DST of the current time. For some programing languages with no concept of time without date, it may map `timetz`to a timestamp with a time zone at the beginning of time (for example, Unix Epoch 1970, thus resulting in DST of year 1970 being used).
- `interval` is a duration of time in hours, days, months, minutes, and others. It comes in handy when doing date-time arithmetic. For example, if the world is supposed to end in exactly 666 days from now, all you have to do is add an interval of 666 days to the current time to get the exact moment when it'll happen (and plan accordingly).
- `tsrange` is new in PostgreSQL 9.2 and allows you to define opened and closed ranges of timestamp with no timezone. The type consists of two timestamps and opened/closed range qualifiers. For example `'[2012-01-01 14:00, 2012-01-01 15:00)'::tsrange` would define a period starting at 14:00 but ending before 15:00.
- `tstzrange` is new in PostgreSQL 9.2 and allows you to define opened and closed ranges of timestamp with timezone.
- `daterange` is new in PostgreSQL 9.2 and allows you to define opened and closed ranges of dates.

Time Zones: What It Is and What It Isn't

A common misconception of PostgreSQL time zone aware data types is that an extra time zone information is being stored along with the timestamp itself. This is incorrect. If you save 2012-2-14 18:08:00-8 (-8 being the Pacific offest from UTC), Postgresql internally works like this:

- Get the UTC time for 2012-02-14 18:08:00-8. This would be 2012-02-15 04:08:00-0.
- PostgreSQL stores the value 2012-02-15 04:08:00 in the database.

When you call the data back for display, PostgreSQL goes through the following steps:

- Find the time zone observed by the server or what was requested. Suppose it's "America/New_York" and get the offset for that period of time corresponding to the UTC of the date time in question. For things that just have a time element like `timetz`, the offset assumed—if not specified—is the current local time offset. Let's suppose it's -5. You can also directly specify an offset instead of a time zone to avoid the Daylight Savings check.
- Compute the date time 2012-02-15 04:08:00 with a -5 offset to get 2012-02-15 21:08:00, then display as 2012-02-15 21:08:00-5.

As you can see, PostgreSQL doesn't store the time zone but simply uses it to know how to convert it to UTC for storage. The main thing to remember is that the input time zone is only used to compute the UTC value and once stored, that input time zone information is gone. When PostgreSQL displays back the time, it always does so in the default time zone dictated by the session, user, database, or server and checks in that order. If you employed time zone aware data types, we implore you to consider the consequence of a server move from one time zone to another. Suppose you based a server in New York City, and subsequently restored the database in Los Angeles. All timestamp with time zone fields would suddenly display in Pacific time. This is fine as long as you anticipate this behavior.

Here's an example where something can go wrong. Suppose that McDonald's had their server on the East Coast and the opening time for stores is `timetz`. A new McDonald's opens up in San Francisco. The new franchisee phones McDonald's HQ to add their store to the master directory with an opening time of 7 a.m. The data entry dude entered the information as he is told—7 a.m. PostgreSQL inteprets this to mean 7 a.m. Eastern, and now people are waiting in line wondering why they can't get their breakfast sandwiches at 4 a.m. Being hungry is one thing, but we can imagine many situations where a screw-up with difference of three hours could mean life or death.

So why would anyone want to use time zone aware data types? First, it does save having to do time zone conversions manually. For example, if a flight leaves Boston at 8 a.m. and arrives in Los Angeles at 11 a.m., and your server is in Europe, you don't want to have to figure out the offset for each manually. You could just enter the data with the

Boston and Los Angeles offsets. There's another convincing reason to use time zone aware data types: the automatic handling of Daylight Savings Time. With countries deviating more and more from each other in DST observation schedules and even changing from year to year, manually keeping track of DST changes for a globally used database would almost require a dedicated programmer who does nothing but keep up to date with the latest DST schedules.

Here's an interesting example: A traveling sales person catches a flight home from San Francisco to nearby Oakland. When he boards the plane the clock at the terminal reads 2012-03-11 1:50 a.m. When he lands, the clock in the terminal reads 2012-03-11 3:10 a.m, How long was the flight? With time zone aware time stamps, you get 20 minutes, which is the plausible answer for a short flight across the Bay. We actually get the wrong answer if we don't use time zone aware timestamps.

```
SELECT '2012-03-11 3:10AM'::timestamptz - '2012-03-11 1:50AM'::timestamptz;
```

gives you 20 minutes, while

```
SELECT '2012-03-11 2:45AM'::timestamp - '2012-03-11 1:50AM'::timestamp;
```

gives you 1 hour and 20 minutes.

We should add that your server needs to be in the US for the above discrepancy to show up.

Let's drive the point home with more examples, using a Boston server.

Example 5-10. Inputting time in one time zone and output in another

```
SELECT '2012-02-28 10:00 PM America/Los_Angeles'::timestamptz;

2012-02-29 01:00:00-05
```

For Example 5-10, I input my time in Los Angeles local time, but since my server is in Boston, I get a time returned in Boston local time. Note that it does give me the offset, but that is merely display information. The timestamp is internally stored in UTC.

Example 5-11. Timestamp with time zone to timestamp at location

```
SELECT '2012-02-28 10:00 PM America/Los_Angeles'::timestamptz AT TIME ZONE 'Europe/Paris';

2012-02-29 07:00:00
```

In Example 5-11, we are getting back a timestamp without time zone. So you'll notice the answer you get when you run this same query will be the same as mine. The query is asking: What time is it in Paris if it's 2012-02-28 10:00 p.m. in Los Angeles? Note the absence of UTC offset in the result. Also, notice how I can specify time zone with its official names rather than just an offset, visit Wikipedia for a list of official time zone names (*http://en.wikipedia.org/wiki/Zoneinfo*).

Operators and Functions for Date and Time Data Types

The inclusion of a temporal interval data type greatly eases date and time arithmetics in PostgreSQL. Without it, we'd have to create another family of functions or use a nesting of functions as most other databases do. With intervals, we can add and subtract timestamp data simply by using the arithmetic operators we're intimately familiar with. Table 5-1 provides a listing of operators and functions used with date and time data types.

Table 5-1. Date and Timestamp Operators

Operator	Example
+ Adding an interval	`SELECT '2012-02-10 11:00 PM'::timestamp + interval '1 hour';` `2012-02-11 00:00:00`
- Subtracting an interval	`SELECT '2012-02-10 11:00 PM'::timestamptz - interval '1 hour';` `2012-02-10 22:00:00-05`

OVERLAPS Returns true or false if two temporal ranges overlap. This is an ANSI-SQL operator equivalent to the functional `overlaps()`. OVERLAPS takes four parameters, the first pair and the last pair constitute the two ranges.	`SELECT '2012-10-25 10:00 AM'::timestamp,'2012-10-25 2:00 PM'::timestamp OVERLAPS '2012-10-25 11:00 AM'::timestamp,'2012-10-26 2:00 PM'::timestamp AS x, '2012-10-25'::date,'2012-10-26'::date OVERLAPS '2012-10-26'::date,'2012-10-27'::date As y;`
	`x | y`
	`-------+--------`
	`t | f`
	Overlap considers the time periods to be half-open, meaning that the start is included but the end is not. This is slightly different behavior than when using the common `BETWEEN` operator, which considers both start and end to be included. The quirk with overlaps won't appear unless one of your ranges is a fixed point in time (a period where start and end are identical). Do watch out for this if you're a avid user of the the overlaps function.

In addition to the operators, PostgreSQL comes with functions with temporal types in mind. A full listing can be found here Date Time Functions and Operators. We'll demonstrate a sampling here.

Once again, we start with the versatile `generate_series` function. Above PostgreSQL 8.3 or above, you can use this function with temporal types and interval steps.

Example 5-12. Generate a time series using generate_series()

```
SELECT (dt - interval '1 day')::date As eom
FROM generate_series('2/1/2012', '6/30/2012', interval '1 month') As dt;

eom
------------
2012-01-31
2012-02-29
2012-03-31
2012-04-30
2012-05-31
```

As you can see in Example 5-12, we can express dates in our local date time format, or the more global ISO Y-M-D format. PostgreSQL automatically interprets differing in-

put formats. To be safe, we tend to stick with entering dates in ISO, because date formats vary from culture to culture, server to server, or even database to database.

Another popular activity is extracting or formatting parts of a complete date time. Here, the functions `date_part()` and `to_char()` come to the rescue. The next example will also drive home the abidance of DST for a time zone aware data type.

Example 5-13. Extracting elements of a date time

We intentionally chose a period that crosses a daylight savings switchover in US/East.

```
SELECT dt, date_part('hour',dt) As mh, to_char(dt, 'HH12:MI AM') As formtime
 FROM generate_series('2012-03-11 12:30 AM', '2012-03-11 3:00 AM', interval '15 minutes')
As dt

dt                     | mh | formtime
-----------------------+----+----------
2012-03-11 00:30:00-05 |  0 | 12:30 AM
2012-03-11 00:45:00-05 |  0 | 12:45 AM
2012-03-11 01:00:00-05 |  1 | 01:00 AM
2012-03-11 01:15:00-05 |  1 | 01:15 AM
2012-03-11 01:30:00-05 |  1 | 01:30 AM
2012-03-11 01:45:00-05 |  1 | 01:45 AM
2012-03-11 03:00:00-04 |  3 | 03:00 AM
```

By default, `generate_series()` will assume `timestamp with time zone` if you don't explicitly cast to `timestamp`. It will always return `timestamp with time zone`.

XML

The XML datatype is perhaps one of the more controversial types you'll find in a relational database. It violates principles of normalization and makes purists cringe. Nonetheless, all of the high-end proprietary relational databases support them (IBM DB2, Oracle, SQL Server). PostgreSQL jumped on the bandwagon and offers plenty of functions to work with data of XML type. We've also authored many articles on working with XML in the context of PostgreSQL. (For further reading, you can find these articles at *http://www.postgresonline.com/journal/index.php?/plugin/tag/xml*.) PostgreSQL comes packaged with various functions for generating data, concatenating, and parsing XML data. These are outlined in PostgreSQL XML Functions.

Loading XML Data

To start, we show you how to get XML data into a table:

Example 5-14. Populate XML field

```
INSERT INTO web_sessions(session_id, session_state)
 VALUES ('robe'
    , '<session><screen_properties>
 <prop><name>color</name><val>red</val></prop>
```

```
<prop><name>background</name><val>snoopy</val></prop>
</screen_properties></session>'::xml);
```

Querying XML Data

For querying XML, the `xpath()` function is really useful. The first argument is an XPath query statement, the second is an XML string. Output is an array of XML objects that satisfy the XPath query. In example Example 5-15, we'll combine XPath with `unnest()` to return all the screen property names. Remember that `unnest` unravels the array into a row set. We then cast the XML fragment to text:

Example 5-15. Query XML field

```
SELECT (xpath('/prop/name/text()', prop) )[1]::text As pname ❶
  , (xpath('/prop/val/text()', prop) )[1]::text As pval ❷
 FROM (❸SELECT unnest(xpath('/session/screen_properties/prop', session_state)) As prop
   FROM web_sessions WHERE session_id = 'robe') As X;

pname    |  pval
-----------+--------
color      | red
background | snoopy
```

❸ Unravel into `<prop>`, `<name>`, `</name>`, `<val>`, `</val>`, `</prop>` tags.

❶ Get text element in `name` and `val` tags of each `prop` element.

❷ We need to use array subscripting because XPath always returns an array, even if there's only one element to return.

Custom and Composite Data Types

In this section, we'll demonstrate how to define a simple custom type and use it. The `composite`, (a.k.a. `record`, `row`) object type is a special type in PostgreSQL because it's often used to build an object that is then cast to a custom type or as return types for functions needing to return multiple columns.

All Tables Are Custom

As mentioned earlier, PostgreSQL automatically creates custom type for all the tables. For all intents and purposes, you can use custom types just as you would any other built-in type. So, we could conceivably create a table that has as a column type that is of another table's custom type, and we can go even further and make an array of that type. We'll go ahead and demonstrate this table turducken:

```
CREATE TABLE user_facts(user_id varchar(30) PRIMARY KEY, facts census.facts[]);
```

We can create an instance of factoid composite type as follows:

```
ROW(86,'25001010206', 2012, 123, NULL)::census.facts
```

And then stuff this factoid into our table:

```
INSERT INTO user_facts(user_id, facts)
 VALUES('robe', ARRAY[ROW(86, '25001010206', 2012, 123, NULL)::census.facts]);
```

We can add more factoids to the same row using the array || (concatenation) operator and the array constructor `array()`:

```
UPDATE user_facts
 SET facts = facts || array(SELECT F FROM census.facts AS F WHERE fact_type_id = 86)
WHERE user_id = 'robe';
```

Finally, we can query our composite array column:

```
SELECT facts[5].*, facts[1].yr As yr_1 FROM user_facts WHERE user_id = 'robe';
```

```
fact_type_id |  tract_id   |  yr  |   val    | perc | yr_1
-------------+-------------+------+----------+------+------
86           | 25001010304 | 2010 | 2421.000 |      | 2012
```

Building Your Own Custom Type

Although you can easily create composite types just by creating a table, at some point, you'll probably wish to build your own from scratch. For example, let's build a complex number data type with the following statement:

```
CREATE TYPE complex_number AS (r double precision, i double precision);
```

We can then use this complex_number as a column type:

```
CREATE TABLE circuits(circuit_id text PRIMARY KEY, tot_volt complex_number);
```

We can then query our table with statements such as:

```
SELECT circuit_id, (tot_volt).*
 FROM circuits;
```

or an equivalent:

```
SELECT circuit_id, (tot_volt).r, (tot_volt).i
 FROM circuits;
```

People from other databases are a bit puzzled by the `(tot_volt)` syntax. If you leave out the `()` for a composite type that is not an array, you get an error of form *missing FROM-clause entry for table "tot_volt"*, which is caused because `tot_volt` could just as easily refer to a table called `tot_volt`.

Although we didn't show it here, you can also define operators that will work with your custom type. You could define the operator of + addition between two complex numbers or a complex number and a real number. Being able to build custom types and operators pushes PostgreSQL to the boundary of a full-fledged development environment, bringing us ever closer to our conception of an ideal world where everything is table-driven.

CHAPTER 6

Of Tables, Constraints, and Indexes

Tables

In addition to the run-of-the-mill data table, PostgreSQL offers several kinds of tables that are rather unique: temporary, unlogged (demonstrated in Example 6-3), inherited (demonstrated in Example 6-2), and typed tables (demonstrated in Example 6-4).

Table Creation

In this section, we'll demonstrate some common table creation examples. Most are similar to or exactly what you'll find in other databases.

Example 6-1. Basic table creation

```
CREATE TABLE logs(
 log_id serial PRIMARY KEY ❶, user_name varchar(50) ❷
 , description text ❸
 , log_ts timestamp with time zone NOT NULL DEFAULT current_timestamp); ❹
CREATE INDEX idx_logs_log_ts ON logs USING btree(log_ts);
```

❶ `serial` type is the data type you use when you want an incrementing auto number. It creates a companion `sequence` object and defines the new column as an integer with the default value set to the next value of the sequence object. It is often as a primary key.

❷ `varchar` is a variable length string similar to what you will find used in other databases. It can also be written as `character varying(50)`. If you don't specify a size, the size is unconstrained.

❸ `text` is an unconstrained string. It's never followed with a size.

❹ `timestamp with time zone` is a date and time data type always stored in UTC. It will, by default, always display date and time in the server's own time zone unless you tell it to otherwise. It's often written using the short-hand `timestamptz`. There is a

companion data type called `timestamp`, which lacks the time zone. As a result, the value of `timestamp` will not change if your server's time zone changes.

PostgreSQL is the only database that we know of that offers table inheritance. When you specify that a child table inherit from a parent, the child will be created with all the columns of the parent in addition to its own columns. All structural changes made to the parent will automatically propagate its child tables. To save you even more time, whenever you query the parent, all rows in the children are included as well. Not every trait of the parent passes down to the child, notably indexes and primary key constraints.

Example 6-2. Inherited table creation

```
CREATE TABLE logs_2011(PRIMARY KEY(log_id)) INHERITS (logs);
CREATE INDEX idx_logs_2011_log_ts ON logs USING btree(log_ts);
ALTER TABLE logs_2011
  ADD CONSTRAINT chk_y2011
   CHECK (log_ts BETWEEN '2011-01-01'::timestamptz AND '2012-1-1'::timestamptz);❶
```

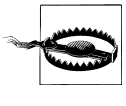
❶ We defined a check constraint to limit data to just year 2011 for our time zone. Since we didn't specify a time zone, our timestamp will default to the server's time zone. Having the check constraint in place allows the query planner to completely skip over inherited tables that do not satisfy a query condition.

For ephemeral data that could be rebuilt in event of a disk failure or don't need to be restored after a crash, you might prefer having more speed over redundancy. In 9.1, the `UNLOGGED` modifier allows you to create unlogged tables. These tables will not be part of any write-ahead logs. Should you accidentally unplug the power cord on the server, when you turn back the power, all data in your unlogged tables will be wiped clean during the roll-back process. You can find more examples and gotchas in *Depesz: Waiting for 9.1 Unlogged Tables*.

Example 6-3. Unlogged table creation

```
CREATE UNLOGGED TABLE web_sessions(session_id text PRIMARY KEY, add_ts timestamptz
 , upd_ts timestamptz, session_state xml);
```

The benefit of unlogged tables is that they can be 15 times or faster to write to than logged tables.



Unlogged tables are always truncated during crash recovery, so don't use them for data that is not derivable or ephemeral. They also don't support GIST indexes, and are therefore unsuitable for exotic data types that require such an index for speedy access. GIN indexes are supported, though.

PostgreSQL 9.0+ provides another way of table creation whereby the column structure is defined by a composite data type. When using this method, you can't add additional

columns directly to the table. The advantage of this approach is that if you have many tables sharing the same structure and you need to alter the column, you can do so by simply changing the underlying type.

We'll demonstrate by first creating a type with the definition:

```
CREATE TYPE app_user AS (user_name varchar(50), email varchar(75), pwd varchar(50));
```

We can then create a table that has rows that are instances of this type:

Example 6-4. Typed table Creation

```
CREATE TABLE super_users OF app_user(CONSTRAINT pk_super_users PRIMARY KEY (user_name));
```

Let's say we now need to add a phone number to all our tables. We simply have to run the following command to alter the underlying type:

```
ALTER TYPE app_user ADD ATTRIBUTE main_phone varchar(18) CASCADE;
```

Normally, you can't change the definition of a type if tables depend on that type. The `CASCADE` modifier allows you to override this restriction.

Multi-Row Insert

PostgreSQL syntax pretty much abides by the ANSI-SQL standards for adding data, but it does have some lagniappes not always found in many other databases, one of which is a multi-row constructor that can be used to insert more than one record at a time. The multi-row constructor has been in existence since 8.2. The constructor can in fact be used in any SQL and behaves exactly like a table.

Example 6-5. Using multi-row consructor to insert data

```
INSERT INTO logs_2011(user_name, description, log_ts)
 VALUES ('robe', 'logged in', '2011-01-10 10:15 AM EST')
 , ('lhsu', 'logged out', '2011-01-11 10:20 AM EST');
```

It's much like a single row `INSERT VALUES()` syntax except you can add more than one row at a time.

An Elaborate Insert

For this next section, we'll load the data collected in Example 3-5 into production-grade tables replete with proper indexes and constraints. We'll be using the new `DO` command, which allows you to write a piece of procedural language code on the fly. Don't worry if you can't follow the code in this section. As long as you run the code, you'll have the tables you need to continue with our on-going examples.

The first step we're going to take is to create a new schema. Think of schemas as another level of organization for database objects. We use it liberally to group our tables into logical units. You can have two tables with the same name as long as they are in separate

schemas. To distinguish between them, you must prepend the schema name. To avoid the hassle of always having to tag on the schema in front of table names, you can set the `search_path` either on a per-session or permanent basis. You list schemas in the order you would like the SQL parser to search for tables. For example, if you have two tables named my_table, one in a schema called s1, another in s2, and you set `search_path=s2, s1;`. When you refer to my_table in a query without prefixing the schema name, it'll be the one in s2. Schemas are not limited to organizing tables, you can place functions, types, views, and many other objects into separate schemas.

Example 6-6. Creating a new schema, setting search_path, and populating lu_tracts

```
CREATE SCHEMA census; ❶
set search_path=census; ❷
CREATE TABLE lu_tracts(tract_id varchar(11), tract_long_id varchar(25)
 , tract_name varchar(150)
 , CONSTRAINT pk_lu_tracts PRIMARY KEY (tract_id));
INSERT INTO lu_tracts( tract_id, tract_long_id, tract_name)
SELECT geo_id2, geo_id, geo_display
FROM staging.factfinder_import
WHERE geo_id2 ~ '^[0-9]+'; ❸
```

❶ Create a schema called `census` to house our new data.

❷ `set search_path` allows us to designate the default schemas to search in for this session.

❸ We want to insert only census tract rows, so we designate a regex that searches for strings starting with one or more numbers.

The next two examples take advantage of the new `DO` command and the procedural language PL/pgSQL to generate a series of `INSERT INTO SELECT` statements. The SQL also performs an unpivot operation converting columnar data into rows.

Example 6-7. Insert using DO to generate dynamic SQL

```
set search_path=census;
CREATE TABLE lu_fact_types(fact_type_id serial, category varchar(100)
 , fact_subcats varchar(255)[] ❶, short_name varchar(50)
 , CONSTRAINT pk_lu_fact_types PRIMARY KEY (fact_type_id));

DO language plpgsql
$$
DECLARE var_sql text;
BEGIN
  var_sql := string_agg('INSERT INTO lu_fact_types(category, fact_subcats, short_name)
  SELECT ''Housing''
   , array_agg(s' || lpad(i::text,2,'0') || ') As fact_subcats, ' || quote_literal('s' ||
lpad(i::text,2,'0') ) || ' As short_name
  FROM staging.factfinder_import
  WHERE s' || lpad(I::text,2,'0') || ' ~ ''^[a-zA-Z]+'' ', ';') FROM generate_series(1,51)
As I;
  EXECUTE var_sql;
```

```
END
$$;
```

❶ An array of strings each with maximum length of 255 characters.

Example 6-8. Adding data to facts table

```
set search_path=census;
CREATE TABLE facts(fact_type_id int, tract_id varchar(11), yr int
 , val numeric(12,3), perc numeric(6,2),
CONSTRAINT pk_facts PRIMARY KEY (fact_type_id, tract_id, yr));
DO language plpgsql
$$
DECLARE var_sql text;
BEGIN
    var_sql := string_agg('INSERT INTO facts(fact_type_id, tract_id, yr, val, perc)
    SELECT ' || ft.fact_type_id::text || ', geo_id2, 2010, s' || lpad(i::text,2,'0') ||
'::integer As val
     , CASE WHEN s' || lpad(i::text,2,'0') || '_perc LIKE ''(X%'' THEN NULL ELSE s' ||
lpad(i::text,2,'0') || '_perc END::numeric(5,2) As perc
    FROM staging.factfinder_import AS X
    WHERE s' || lpad(i::text,2,'0') || ' ~ ''^[0-9]+'' ', ';')
    FROM generate_series(1,51) As I INNER JOIN lu_fact_types AS F ON ('s' || lpad(I::text,
2,'0') = T.short_name);
    EXECUTE var_sql;
END$$;
```

Constraints

PostgreSQL constraints are the most advanced and (most complex) of any database we've worked with. Not only do you just create constraints, but you can also control all facets of how it'll handle existing data, cascade options, how to perform the matching, which indexes to incorporate, conditions under which constraint can be violated, and so forth. On top of it all, you need to even pick your own name for the constraint. For the full treatment, we suggest you review the official documentation. You'll find comfort in knowing that taking the default settings usually works out fine. We'll start off with something familiar to most relational folks: foreign key, unique, and check constraints before moving onto exclusion constraints introduced in 9.0.

Foreign Key Constraints

PostgreSQL follows the same convention as most databases you may have worked with that support referential integrity. It supports the ability to define cascade update and delete rules. We'll experiment with that in Example 6-9.

Example 6-9. Building FK constraints and covering indexes

```
set search_path=census,public;
ALTER TABLE facts
ADD CONSTRAINT fk_facts_lu_fact_types
FOREIGN KEY (fact_type_id) REFERENCES lu_fact_types (fact_type_id) ❶
```

```
ON UPDATE CASCADE ON DELETE RESTRICT;  ❷
CREATE INDEX fki_facts_lu_fact_types ON facts(fact_type_id); ❸

ALTER TABLE facts
ADD CONSTRAINT fk_facts_lu_tracts
FOREIGN KEY (tract_id)
REFERENCES census.lu_tracts (tract_id)
ON UPDATE CASCADE ON DELETE RESTRICT;
CREATE INDEX fki_facts_lu_tracts ON census.facts(tract_id);
```

❶ In this first constraint, we define a foreign key relationship between our facts and fact_types table. This prevents us from introducing fact types not already present in our fact types lookup table.

❷ We also define a cascade rule that automatically update the fact_type_id in our facts table should we renumber our fact types. We restrict deletes from our lookup table if any values are in use. Although `RESTRICT` is already the default behavior, we add it for clarity.

❸ Unlike primary key and unique constraints, PostgreSQL doesn't automatically create an index for foreign key constraints; you need to do this yourself.

Unique Constraints

Each table can have no more than a single primary key. Should you need to enforce uniqueness on other columns, you must resort to unique constraints. Adding a unique constraint automatically creates an associated unique index. Unlike a primary key, a column with unique constraints can still be populated with NULLs. Having a unique constraint doesn't qualify a column to participate in a foreign key relationship. Adding a unique constraint is simple.

```
ALTER TABLE logs_2011 ADD CONSTRAINT uq_us_log UNIQUE (user_name, log_ts);
```

Check Constraints

Check constraints are conditions that must be met for a field or set of fields for each row. PostgreSQL query planner also uses them for what is called `constraint exclusion` which means if a check constraint on a table guarantees that it can't service the filter condition of a query, then the planner can skip checking the table. We saw an example of a check contraint in Example 6-2. That particular example was used to prevent the planner from having to scan log tables that don't satisfy the date range of a query. You can define additional constraints, for example you make require all user names input into logs tables be in lower case with this check constraint:

```
ALTER TABLE logs ADD CONSTRAINT chk_lusername
 CHECK (user_name = lower(user_name));
```

The other noteworthy thing about check constraints is that unlike primary key, foreign key, and unique key constraints, they can be inherited from parent tables. So you'll see

that this particular check constraint we put on the `logs` gets inherited by all child tables of `logs`.

Exclusion Constraints

Introduced in PostgreSQL 9.0, *exclusion constraints* allow you to incorporate additional operators to enforce a certain kind of uniqueness that can't be satisfied by equality. Exclusion constraints are really useful in problems involving scheduling. If a room is booked between a certain period of time, additional booking overlaps won't be allowed. To enforce this rule, create a scheduling table using the period data type[1] and then add an exclusion constraint using the `OVERLAP` (`&&`) operator. PostgreSQL 9.2 introduces the range data types that are perfect for use in exclusion constraints. In 9.2, the period extension is obsolete and supplanted by the new built-in *tstzrange* range data type. An example of using 9.2 ranges with exclusion constraints is demonstrated in *Waiting for 9.2 Range Data Types*.

Indexes

PostgreSQL comes with a superbly flexible index framework. At time of writing, PostgreSQL comes with at least four types of indexes. Should you find these insufficient, you can define new index operators and modifiers to work on top of these. If still unsatisfied, you're free to create your own index type. PostgreSQL also allows you to mix types in the same table each with their own catered index types and count on the planner to take advantage of them all by the planner's bitmap index scan strategy. So, for instance, one column could use a B-tree index; the adjacent column a GiST index and both indexes can be utilized in the same query.

PostgreSQL Stock Indexes

To take full advantage of all that PostgreSQL has to offer, you'll want to understand the various types of indexes and what they can and can't be used for. The various types of indexes PostgreSQL currently has built-in are listed next.

Index Types

B-tree

B-tree is the index you'll find most common in any relation database. B-tree is designed to be a general purpose type of index. You can usually get by with just this one alone if you don't want to experiment with additional types. If PostgreSQL automatically creates an index for you or you don't bother picking the type, B-tree will be chosen. It is currently the only index type allowed for primary key and unique indexes.

1. You'll need to install the period extension to be able to use this data type.

GiST
: Generalized Search Tree (GiST) is an index type optimized for full text search, spatial data, astronomical data, and hierarchical data. You can't use it to enforce uniqueness, however, you can use it in exclusion constraints.

GIN
: Generalized Inverted Index (GIN) is an index type commonly used for the built-in full text search of PostgreSQL and the trigram extensions. GIN is a decendent of Gist, but it's not lossy. GIN indexes are generally faster to search than GiST, but slower to update. You can see an example at *Waiting for Faster LIKE/ILIKE*.

SP-GiST
: Space-Partitioning Trees Generalized Search Tree (SP-GiST) is an index type introduced in PostgreSQL 9.2. It's use is similar that of GiST, but is generally faster for certain kinds of distribution. PostGIS 2.1 spatial extension has planned support for it. The only types built-in that currently have support for it are the built-in PostgreSQL geometry types like point and box and text. Other GiST dependent extensions also have planned support for it.

hash
: Hash is an index that was popular before GiST and GIN came along. General consensus is that GiST and GIN outperform and are more transaction safe than hash. PostgreSQL has relegated hash to legacy status. You may encounter this index type in other databases, but it's best to avoid it in PostgreSQL.

Should you want to go beyond the index types that PostgreSQL installs by default, either out of need or curiosity, you should start perusing the list of additional index types available as extensions.

Custom Index Types

btree_gist
: This index is useful when you're trying to group different types into a single index. Excellent choice for cases where you have a simple type like a number and a more complex type like a point. It's also used to leverage gist like KNN operators and exclusion constraints for basic types, which can only be used with GiST and GIN indexable operators.

btree_gin
: is a cross-breeding of B-tree and GIN. It supports the indexable specialty operators of GIN, but also offers indexable equality found in the B-tree index not available with standard GIN. It's most useful when you want to create a compound index composed of a column data type like a text or number, normally serviced by btree operators and another column, such as a hierarchical ltree type, or full-text vector supported by GIN. By using a btree_gin index, you can have both columns as part of the compound index and still have and indexable equality check be able to use the index for the text/integer column.

You can install any of the index types in Custom Index Types on page 82, using the following:

```
CREATE EXTENSION btree_gist;
```

Operator Class

Indexes for each data type have operator classes (a.k.a. `opclass`). Operator classes are detailed in *Operator Classes*. Operator classes support a given set of operators. In short, an operator can utilize an index only if the operator class for that index supports it. For example, many B-tree operator classes support `=`, `>=` but not pattern operations like `~>~`. Each data type comes with a default operator class. For instance, the default opclass for `varchar` is `varchar_ops`, which includes operators such as `=`, `>`, and `<` but no support for pattern operations. If you create an index without being explicit about the opclass(es) to be used, the default for the data type(s) being indexed would automatically be picked. The index will then only be useful when you're using operators within the opclass of the index. Refer to *Why is My Index Not Used?* for more information.

You shouldn't always accept the default. For instance, `varchar_ops` doesn't include the `LIKE` operators, so none of your like searches can use an index of that `opclass`. If you're going to be performing wildcard searches on a `varchar` columns, you'd be better off choosing the `varchar_pattern_ops opclass` for your index. To specify the opclass, just append the opclass after the column name, as in:

```
CREATE INDEX idx_bt_my_table_description_varchar_pattern ON my_table
 USING btree (description varchar_pattern_ops);
```

For PostgreSQL 9.1+, you'd do even better with a GiST index and the companion `pg_trgm` extension packaged with the `gist_trgm_ops` operator class. This particular `opclass` is highly optimized for wildcard searches. You can learn more about Trigrams in our article at *Teaching LIKE and ILIKE New Tricks*. Wtih the extension installed, you can create your index as follows:

```
CREATE INDEX idx_gist_my_table_description_gist_trgm_ops ON my_table
 USING gist (description gist_trgm_ops);
```

You'll then see your standard ILIKE and LIKE searches being able to take advantage of indexing.

Functional Indexes

PostgreSQL has a feature called functional indexes, which you won't often find in other databases. A more common parallel you'll see in other databases like Microsoft SQL Server or MySQL are computed columns and the ability to place indexes on computed columns. PostgreSQL didn't buy into the idea of computed columns since views are more appropriate places for them. To still reap the speed advantage of indexes, PostgreSQL lets you place indexes on functions of columns. A classic example where you'd want to employ a functional index is for dealing with mixed case text. PostgreSQL is a

case-sensitive database, so to be able to search using an index when casing doesn't matter; you can create an index as follows:

```
CREATE INDEX idx_featnames_ufullname_varops ON featnames_short
 USING btree (upper(fullname) varchar_pattern_ops);
```

Partial Indexes

Partial indexes (read more about them here: *http://www.postgresql.org/docs/current/interactive/indexes-partial.html*) are indexes that only index that portion of data fitting a specific `WHERE` condition. This is pretty much synonymous with SQL Server 2008+ filtered index, but PostgreSQL has had this feature even in pre 8.0 versions. If you have a table of one million rows, but you only query a fixed set of 10,000, you're better off creating partial indexes because of the disk savings and having a smaller more efficient index to scan. The main caveat with partial indexes is that you must use the same `WHERE` condition when you created the index in your query to activate the index. An easy way to ensure that your partial index will always be used is to use a view when querying the data. For example, let's say we have a table of newspaper subscribers, which we define as follows:

Let's suppose we have a subscription table, but we want to ensure that for each user, we have only one active subscription. We might create a table like this:

```
CREATE TABLE allsubscribers (id serial PRIMARY KEY, user_name varchar(50) NOT NULL
 , deactivate timestamptz);
```

We can then add our partial index to guarantee uniqueness only for active subscribers:

```
CREATE UNIQUE INDEX uqidx_1 ON allsubscribers
 USING btree (lower(user_name)) WHERE deactivate IS NULL;
```

To ensure our index is always used for active subscriptions, we can create a view with the built-in condition and always use this view when querying active subscriptions:

```
CREATE OR REPLACE VIEW vw_active_subscribers AS
SELECT id, lower(user_name) As user_name
 FROM allsubscribers WHERE deact_dt IS NULL;
```

To ensure index usage, we always query against our view as follows:

```
SELECT * FROM vw_active_subscribers WHERE user_name = 'sandy';
```

You can open up the planner and see that our index was indeed used.

Multicolumn Indexes

PostgreSQL, like many other databases, supports compound indexes, a.k.a. multicolumn indexes. Compound indexes allow you to combine multiple columns or functions on columns into one index. Prior to 9.0, there wasn't a compelling reason to use compound indexes apart from primary key and unique key indexes because PostgreSQL supports bitmap index scans, which allows the planner to utilize multiple indexes in a

query. Using a compound index, may speed up certain kinds of searches if you always search exactly those multiple columns together.

In 9.0 and even more so in 9.2, compound indexes serve an important role in exclusion constraints. In PostgreSQL 9.2, index-only scans were introduced, which makes the use of compound indexes even more relevant since the planner can just scan the index and use data from the index without ever needing to check the underlying table.

Here is an example of a multicolumn index:

```
CREATE INDEX idx_sometable_cmpd ON sometable
 USING btree(type_id, upper(fullname) varchar_pattern_ops);
```

CHAPTER 7

SQL: The PostgreSQL Way

PostgreSQL is one of the most ANSI-SQL compliant databases on the market. It even supports many of the additions introduced with the SQL:2006+ standard. PostgreSQL goes much further and adds constructs that range from mundane syntax shorthands to avant-garde features that break the bounds of traditional SQL. In this chapter, we'll cover some SQL constructs not often found in other databases. For this chapter, you should have a working knowledge of SQL; otherwise, you may not appreciate the labor-saving tidbits that PostgreSQL brings to the table.

SQL Views

Like most relational databases, PostgreSQL supports views. Some things have changed over the years on how views work and how you can update the underlying tables via updates on views. In pre-PostgreSQL 9.1, views were updatable but required `INSTEAD OF UPDATE, DELETE` rules on the view. In PostgreSQL 9.1, the preferred way of updating data via a view is to use `INSTEAD OF` triggers instead of rules, though rules are still supported. The trigger approach is standards compliant and more along the lines of what you'll find in other databases that support triggers and updatable views.

Unlike Microsoft SQL Server and MySQL, simple views are not automatically updatable and require writing an instead-of rule or trigger to make them updatable. On the plus side, you have great control over how the underlying tables will be updated. We'll cover triggers in more detail in Chapter 8. You can see an example of building updatable views using rules in *Database Abstraction with Updateable Views*.

Views are most useful for encapsulating common joins. In this next example, we'll join our lookup with our fact data.

Example 7-1. View census.vw_facts

```
CREATE OR REPLACE VIEW census.vw_facts AS
SELECT lf.fact_type_id, lf.category, lf.fact_subcats, lf.short_name
  , f.tract_id, f.yr, f.val, f.perc
FROM census.facts As f
 INNER JOIN census.lu_fact_types As lf
  ON f.fact_type_id = lf.fact_type_id;
```

To make this view updatable with a trigger, you can define one or more instead of triggers. We first define the trigger function(s). There is no standard in naming of the functions and a trigger function can be written in any language that supports triggers. For this example, we'll use PL/pgSQL to write our trigger function as shown in Example 7-2.

Example 7-2. Trigger function for vw_facts to update, delete, insert

```
CREATE OR REPLACE FUNCTION census.trig_vw_facts_ins_upd_del() RETURNS trigger AS
$$
BEGIN
  IF (TG_OP = 'DELETE') THEN ❶
    DELETE FROM census.facts AS f
      WHERE f.tract_id = OLD.tract_id AND f.yr = OLD.yr AND f.fact_type_id =
OLD.fact_type_id;
    RETURN OLD;
  END IF;
  IF (TG_OP = 'INSERT') ❷ THEN
    INSERT INTO census.facts(tract_id, yr, fact_type_id, val, perc)
        SELECT NEW.tract_id, NEW.yr, NEW.fact_type_id, NEW.val, NEW.perc;
    RETURN NEW;
  END IF;
  IF (TG_OP = 'UPDATE') THEN
    IF ROW(OLD.fact_type_id, OLD.tract_id, OLD.yr, OLD.val, OLD.perc) ❸
      != ROW(NEW.fact_type_id, NEW.tract_id, NEW.yr, NEW.val, NEW.perc) THEN
      UPDATE census.facts AS f ❹
          SET tract_id = NEW.tract_id, yr = NEW.yr
          , fact_type_id = NEW.fact_type_id
          , val = NEW.val, perc = NEW.perc
      WHERE f.tract_id = OLD.tract_id
          AND f.yr = OLD.yr
          AND f.fact_type_id = OLD.fact_type_id;
      RETURN NEW;
    ELSE
      RETURN NULL;
    END IF;
  END IF;
END;
$$
LANGUAGE plpgsql VOLATILE;
```

❶ Handle deletes, only delete the record with matching keys in the OLD record.

❷ Handle inserts.

❸ Only updates if at least one of the columns from facts table was changed.

❹ Handle updates, use the OLD record to determine what records to delete and update with the NEW record data.

Next, we bind the trigger function to the view as shown in Example 7-3.

Example 7-3. Bind trigger function to vw_facts view insert,update,delete events

```
CREATE TRIGGER trip_01_vw_facts_ins_upd_del
INSTEAD OF INSERT OR UPDATE OR DELETE ON census.vw_facts
    FOR EACH ROW EXECUTE PROCEDURE census.trig_vw_facts_ins_upd_del();
```

Now when we update, delete, or insert into our view, it will update the underlying `facts` table instead:

```
UPDATE census.vw_facts SET yr = 2012 WHERE yr = 2011 AND tract_id = '25027761200';
```

This will output a note:

```
Query returned successfully: 51 rows affected, 21 ms execution time.
```

If we tried to update one of the fields in our lookup table, because of our row compare the update will not take place, as shown here:

```
UPDATE census.vw_facts SET short_name = 'test';
```

Therefore, the output message would be:

```
Query returned successfully: 0 rows affected, 931 ms execution time.
```

Although we have just one trigger function to handle multiple events, we could have just as easily created a separate trigger and trigger function for each event.

Window Functions

Window functions are a common ANSI-SQL feature supported in PostgreSQL since 8.4. A window function has the unusual knack to see and use data beyond the current row, hence the term *window*. Without window functions, you'd have to resort to using joins and subqueries to poll data from adjacent rows. On the surface, window functions do violate the set-based operating principle of SQL, but we mollify the purist by claiming them to be a short-hand. You can find more details and examples in the section *Window Functions*.

Here's a quick example to get started. Using a window function, we can obtain the average value for all records with `fact_type_id` of 86 in one simple `SELECT`.

Example 7-4. The basic window

```
SELECT tract_id, val, AVG(val) OVER () as val_avg
FROM census.facts WHERE fact_type_id = 86;

 tract_id   |    val    |        val_avg
------------+-----------+-----------------------
25001010100 |  2942.000 | 4430.0602165087956698
25001010206 |  2750.000 | 4430.0602165087956698
25001010208 |  2003.000 | 4430.0602165087956698
25001010304 |  2421.000 | 4430.0602165087956698
:
:
```

Notice how we were able to perform an aggregation without having to use `GROUP BY`. Furthermore, we were able to rejoin the aggregated result back with the other variables without using a formal join. The `OVER ()` converted our conventional `AVG()` function into a window function. When PostgreSQL sees a window function in a particular row, it will actually scan all rows fitting the `WHERE` clause, perform the aggregation, and output the value as part of the row.

Partition By

You can embellish the window into separate panes using the `PARTITION BY` clause. This instructs PostgreSQL to subdivide the window into smaller panes and then to take the aggregate over those panes instead of over the entire set of rows. The result is then output along with the row depending on which pane it belongs to. In this next example, we repeat what we did in Example 7-4, but partition our window into separate panes by county code.

Example 7-5. Partition our window by county code

```
SELECT tract_id, val, AVG(val) OVER (PARTITION BY left(tract_id,5)) As val_avg_county
FROM census.facts WHERE fact_type_id = 86 ORDER BY tract_id;

 tract_id   |    val    |    val_avg_county
------------+-----------+-----------------------
25001010100 |  2942.000 | 3787.5087719298245614
25001010206 |  2750.000 | 3787.5087719298245614
:
25003900100 |  3389.000 | 3364.5897435897435897
25003900200 |  4449.000 | 3364.5897435897435897
25003900300 |  2903.000 | 3364.5897435897435897
 :
```

The `left` function was introduced in PostgreSQL 9.1. If you are using a lower version, you can use `substring` instead.

Order By

Window functions also allow an `ORDER BY` clause. Without getting too abstruse, the best way to think about this is that all the rows in the window will be ordered and the window function will only consider rows from the first row to the current row. The classic example uses the `ROW_NUMBER()` function, which is found in all databases supporting window functions. It sequentially numbers rows based on some ordering and or partition. In Example 7-6, we demonstrate how to number our census tracts in alphabetical order.

Example 7-6. Number alphabetically

```
SELECT ROW_NUMBER() OVER(ORDER BY tract_name) As rnum, tract_name
FROM census.lu_tracts ORDER BY rnum LIMIT 4;
```

```
rnum |                    tract_name
-----+-------------------------------------------------
1    | Census Tract 1, Suffolk County, Massachusetts
2    | Census Tract 1001, Suffolk County, Massachusetts
3    | Census Tract 1002, Suffolk County, Massachusetts
4    | Census Tract 1003, Suffolk County, Massachusetts
```

You can combine `ORDER BY` with `PARTITION BY`. Doing so will restart the ordering for each partition. We return to our example of county codes.

Example 7-7. Partition our window and ordering by value

```
SELECT tract_id, val
 , AVG(val) OVER (PARTITION BY left(tract_id,5) ORDER BY val) As avg_county_ordered
FROM census.facts
WHERE fact_type_id = 86 ORDER BY left(tract_id,5), val;
```

```
  tract_id   |    val    |   avg_county_ordered
-------------+-----------+------------------------
 25001990000 |     0.000 | 0.00000000000000000000
 25001014100 |  1141.000 |   570.50000000000000000
 25001011700 |  1877.000 |  1006.00000000000000000
 25001010208 |  2003.000 |  1255.25000000000000000
 :
 25003933200 |  1288.000 |  1288.00000000000000000
 25003934200 |  1306.000 |  1297.00000000000000000
 25003931300 |  1444.000 |  1346.00000000000000000
 25003933300 |  1509.000 |  1386.75000000000000000
 :
```

The key observation with output is to notice how the average changes from row to row. The ORDER BY clause means that the average will only be taken from the beginning of the partititon to the current row. For instance, if your row is in the 5th row in the 3rd partition, the average will only cover the first five rows in the 3rd partition. We put an `ORDER BY left(tract_id,5), val` at the end of the query so you could easily see the pattern, but keep in mind that the `ORDER BY` of the query is independent of the `ORDER BY` in each `OVER`. You can explicitly control the rows under consideration within a frame

by explicitly putting in a RANGE or ROWS clause of the form ROWS BETWEEN CURRENT ROW AND 5 FOLLOWING. For more details, we ask that you refer to *SQL SELECT official documentation*.

PostgreSQL also supports window naming which is useful if you have the same window for each of your window columns. In Example 7-8, we demonstrate how to define named windows as well as showing a record value before and after for a given window frame, using the LEAD and LAG standard ANSI window functions.

Example 7-8. Named windows and lead lag

```
SELECT *
FROM (SELECT ROW_NUMBER() OVER wt ❶As rnum
   , substring(tract_id,1, 5) As county_code, tract_id
   , LAG(tract_id,2) OVER wt ❷ As tract_2_before
   , LEAD(tract_id) OVER wt ❸ As tract_after
 FROM census.lu_tracts
     WINDOW wt AS (PARTITION BY substring(tract_id,1, 5) ORDER BY tract_id ) ❹
    ) As foo
WHERE rnum BETWEEN 2 and 3 AND county_code IN('25007', '25025')
ORDER BY county_code, rnum;

rnum | county_code |  tract_id   | tract_2_before | tract_after
-----+-------------+-------------+----------------+-------------
2    | 25007       | 25007200200 |                | 25007200300
3    | 25007       | 25007200300 | 25007200100    | 25007200400
2    | 25025       | 25025000201 |                | 25025000202
3    | 25025       | 25025000202 | 25025000100    | 25025000301
```

❹ PostgreSQL allows for defining named windows that can be reused in multiple window column definitions. We define our wt window.

❶❷❸ We reuse our wt alias multiple times to save having to repeat for each window column.

Both LEAD() and LAG() take an optional step argument that defines how many to skip forward or backward; the step can be positive or negative. Also LEAD() and LAG() will return NULL when trying to retrieve rows outside the window partition. This is a possibility that you always have to account for when applying these two functions.

Before leaving the discussion on window functions, we must mention that in PostgreSQL, any aggregate function you create can be used as a window function. Other databases tend to limit window functions to using built-in aggregates like AVG(), SUM(), MIN(), MAX() etc.

Common Table Expressions

In its essence, common table expressions (CTE) allow you to define a query that can be reused in a larger query. PostgreSQL has supported this feature since PostgreSQL 8.4 and expanded the feature in 9.1 with the introduction of writeable CTEs. You'll find a

similar feature in SQL Server 2005+, Oracle 11 (Oracle 10 and below implemented this features using `CORRESPONDING BY`), IBM DB2, and Firebird. This features doesn't exist in MySQL of any version. There are three different ways to use CTEs:

1. The standard non-recursive, non-writable CTE. This is your unadorned CTE used for the purpose of readability of your SQL or to encourage the planner to materialize an expensive sub result for better performance.
2. Writeable CTEs. This is an extension of the standard CTE with `UPDATE` and `INSERT` constructs. Common use is to delete rows and then return rows that have been deleted.
3. The recursive CTE. This put an entirely new whirl on standard CTE. With recursive CTEs, the rows returned by the CTE actually varies during the execution of the query. PostgreSQL allows you to have a CTE that is both updatable and recursive.

Standard CTE

Your basic CTE construct looks as shown in Example 7-9.

Example 7-9. Basic CTE

```
WITH cty_with_tot_tracts AS (
SELECT tract_id, substring(tract_id,1, 5) As county_code
 , COUNT(*) OVER(PARTITION BY substring(tract_id,1, 5)) As cnt_tracts
FROM census.lu_tracts)
SELECT MAX(tract_id) As last_tract, county_code, cnt_tracts
FROM cty_with_tot_tracts
WHERE cnt_tracts > 100
GROUP BY county_code, cnt_tracts;
```

You can stuff as many table expressions as you want in a `WITH` clause, just be sure to separate each by a comma. The order of the CTEs matter in that CTEs defined later can use CTEs defined earlier, but never vice versa.

Example 7-10. CTE with more than one table expression

```
WITH cty_with_tot_tracts AS (
SELECT tract_id, substring(tract_id,1, 5) As county_code
 , COUNT(*) OVER(PARTITION BY substring(tract_id,1, 5)) As cnt_tracts
FROM census.lu_tracts)
, cty AS (SELECT MAX(tract_id) As last_tract
 , county_code, cnt_tracts
FROM cty_with_tot_tracts
WHERE cnt_tracts < 8
GROUP BY county_code, cnt_tracts)
SELECT cty.last_tract, f.fact_type_id, f.val
FROM census.facts As f
 INNER JOIN cty ON f.tract_id = cty.last_tract;
```

Writeable CTEs

The writeable CTE was introduced in 9.1 and extends the CTE to allow for update, delete, insert statements. We'll revisit our logs tables that we created in Example 6-2. We'll add another child table and populate it.

```
CREATE TABLE logs_2011_01_02(PRIMARY KEY(log_id)
, CONSTRAINT chk_y2011_01_02 CHECK(log_ts >= '2011-01-01' AND log_ts < '2011-03-01'))
INHERITS (logs_2011);
```

In Example 7-11, we'll move data from our parent 2011 table to our new child Jan-Feb 2011 table.

Example 7-11. Writeable CTE moves data from one branch to another

```
t1 AS (DELETE FROM ONLY logs_2011
 WHERE log_ts < '2011-03-01' RETURNING *)
INSERT INTO logs_2011_01_02 SELECT * FROM t1;
```

A common use case for the writeable CTE is for repartioning of data in one step. Examples of this and other writeable CTEs are covered in *David Fetter's Writeable CTEs, The Next Big Thing*.

Recursive CTE

The official documentation for PostgreSQL describes it best: The optional `RECURSIVE` modifier changes CTE from a mere syntactic convenience into a feature that accomplishes things not otherwise possible in standard SQL. A more interesting CTE is one that uses a recursively defining construct to build an expression. PostgreSQL recursive CTEs utilize `UNION ALL`. To turn a basic CTE to a recursive one, add the `RECURSIVE` modifier after the `WITH`. Within a `WITH RECURSIVE`, you can have a mix of recursive and non-recursive table expressions. In most other databases, the `RECURSIVE` keyword is not necessary to denote recursion. A common of recursive CTEs is for message threading and other tree like structures. We have an example of this in *Recursive CTE to Display Tree Structures*.

Here is an example that lists all the table relationships we have in our database:

Example 7-12. Recursive CTE

```
WITH RECURSIVE
   tbls AS (
SELECT c.oid As tableoid, n.nspname AS schemaname  ❶
  , c.relname AS tablename
   FROM pg_class c
       LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
       LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
       LEFT JOIN pg_inherits As th ON th.inhrelid = c.oid
       WHERE th.inhrelid IS NULL AND c.relkind = 'r'::"char" AND c.relhassubclass = true
       UNION ALL
       SELECT  c.oid As tableoid,  n.nspname AS schemaname ❷
```

```
                    , tbls.tablename || '->' || c.relname AS tablename  ❸
        FROM tbls INNER JOIN pg_inherits As th ON th.inhparent = tbls.tableoid
        INNER JOIN pg_class c ON th.inhrelid = c.oid
        LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
        LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
                  )
SELECT *  ❹ FROM tbls ORDER BY tablename;

tableoid | schemaname |            tablename
----------+------------+---------------------------------
3152249 | public     | logs
3152260 | public     | logs->logs_2011
3152272 | public     | logs->logs_2011->logs_2011_01_02
```

❶ Get list of all tables that have child tables but have no parent table.

❷ This is the recursive part; gets all children of tables in `tbls`.

❸ Child table name starts with the ancestral tree name.

❹ Return parents and all child tables. Since sorting by table name which has parent prefix appended, all child tables will follow their parents.

Constructions Unique to PostgreSQL

Although PostgreSQL is fairly ANSI-SQL compliant, it does have a few unique constructs you probably won't find in other databases. Many are simply shortcuts without which you'd have to write subqueries to achieve the same results. In this regard, if you opt to stick with ANSI-SQL compliance, simply avoid these shorthands.

DISTINCT ON

One of our favorites is the `DISTINCT ON` clause. It behaves like an SQL `DISTINCT`, except that it allows you to define what columns to consider distinct, and in the case of the remaining columns, an order to designate the preferred one. This one little word replaces numerous lines of additional code necessary to achieve the same result.

In Example 7-13, we demonstrate how to get the details of the first tract for each county.

Example 7-13. DISTINCT ON

```
SELECT DISTINCT ON(left(tract_id, 5)) left(tract_id, 5) As county
 , tract_id, tract_name
FROM census.lu_tracts ORDER BY county, tract_id LIMIT 5;

county |  tract_id   |                    tract_name
--------+-------------+-----------------------------------------------------
25001  | 25001010100 | Census Tract 101, Barnstable County, Massachusetts
25003  | 25003900100 | Census Tract 9001, Berkshire County, Massachusetts
25005  | 25005600100 | Census Tract 6001, Bristol County, Massachusetts
25007  | 25007200100 | Census Tract 2001, Dukes County, Massachusetts
25009  | 25009201100 | Census Tract 2011, Essex County, Massachusetts
```

The `ON` modifier can take on multiple columns, all will be considered to determine uniqueness. Finally, the `ORDER BY` clause has to start with the set of columns in the `DISTINCT ON`, then you can follow with your preferred ordering.

LIMIT and OFFSET

`LIMIT` and `OFFSET` are clauses in your query to limit the number of rows returned. They can be used in tandem or separately. These constructs are not unique to PostgreSQL and are in fact copied from MySQL. You'll find it in MySQL and SQLite and probably various other databases. SQL Server adopted something similar in its 2012 version with a slightly different naming. An `OFFSET` of zero is the same as leaving out the clause entirely. A positive offset means start the output after skipping the number of rows specified by the offset. You'll usually use these two clauses in conjuction with an `ORDER BY` clause. In Example 7-13, we demonstrate with a positive offset.

Example 7-14. First tract for counties 2 to 5

```
SELECT DISTINCT ON(left(tract_id, 5)) left(tract_id, 5) As county, tract_id, tract_name
 FROM census.lu_tracts ORDER BY county, tract_id LIMIT 3 OFFSET 2;
```

```
county |  tract_id   |                    tract_name
--------+-------------+-------------------------------------------------
25005  | 25005600100 | Census Tract 6001, Bristol County, Massachusetts
25007  | 25007200100 | Census Tract 2001, Dukes County, Massachusetts
25009  | 25009201100 | Census Tract 2011, Essex County, Massachusetts
```

Shorthand Casting

ANSI-SQL specs define a construct called `CAST`, which allows you to cast one data type to another. For example, `CAST('2011-10-11' AS date)` will cast the text `2011-10-11` to a date. PostgreSQL has a shorthand for doing this using a pair of colons as in `'2011-10-11'::date`. If you don't care about being coss-database agnostic, the PostgreSQL syntax is easier to write, especially when chaining casts like `somexml::text::integer` for cases where you can't directly cast from one type to another without going through an intermediary type.

ILIKE for Case Insensitive Search

PostgreSQL is case sensitive, similar to Oracle. However, it does have mechanisms in place to do a case insensitive search. You can apply the `UPPER()` function to both sides of the ANSI-compliant `LIKE` operator, or you can simply use the `ILIKE` operator found only in PostgreSQL. Here is an example:

```
SELECT tract_name FROM census.lu_tracts WHERE tract_name ILIKE '%duke%';
```

which produces:

```
tract_name
-----------------------------------------------
```

```
Census Tract 2001, Dukes County, Massachusetts
Census Tract 2002, Dukes County, Massachusetts
Census Tract 2003, Dukes County, Massachusetts
Census Tract 2004, Dukes County, Massachusetts
Census Tract 9900, Dukes County, Massachusetts
```

Set Returning Functions in SELECT

PostgreSQL allows functions that return sets to appear in the `SELECT` clause of an SQL statement. This is not true of many other databases where only scalar functions may appear in the `SELECT`. In fact, to circumvent the restriction, SQL Server 2005+ introduced a CROSS APPLY command. The PostgreSQL solution is much cleaner, but we advise you to use this freedom responsibly. Interweaving set returning functions inside an already complicated query could easily produce results that are beyond what you expect, since using set returning functions usually results in row creation or deletion. You must anticipate this if you'll be using the results as a subquery. In Example 7-15, we demonstrate this with a temporal version of generate_series.

We will use a table we construct with the following:

```
CREATE TABLE interval_periods(i_type interval); INSERT INTO interval_periods(i_type)
VALUES ('5 months'), ('132 days'), ('4862 hours');
```

Example 7-15. Set returning function in SELECT

```
SELECT i_type
 , generate_series('2012-01-01'::date,'2012-12-31'::date,i_type) As dt FROM
interval_periods;
```

```
i_type     |           dt
-----------+------------------------
5 months   | 2012-01-01 00:00:00-05
5 months   | 2012-06-01 00:00:00-04
5 months   | 2012-11-01 00:00:00-04
132 days   | 2012-01-01 00:00:00-05
132 days   | 2012-05-12 00:00:00-04
132 days   | 2012-09-21 00:00:00-04
4862 hours | 2012-01-01 00:00:00-05
4862 hours | 2012-07-21 15:00:00-04
```

Selective DELETE, UPDATE, and SELECT from Inherited Tables

When you query from a table that has child tables, the query drills down, unionizing all the child records satisfying the query condition. `DELETE` and `UPDATE` work the same way, drilling down the hierarchy for victims. Sometimes this is not desirable and you want data to only come from the table you specified without the kids tagging along. This is where the `ONLY` keyword comes in handy. We saw an example of its use in Example 7-11, where we only wanted to delete records from the `y2011` table that weren't already migrated to the `logs_2011_01_02` table. Without the `ONLY` modifier, we'd end up deleting records from the child table that might have been moved previously.

RETURNING Changed Records

The `RETURNING` clause is supported by ANSI-SQL standards, but not found in many databases. We saw an example of it in Example 7-11, where we returned the records deleted. RETURNING can also be used for `INSERT` and `UPDATE`. For inserts into tables with serial keys, it is particularly handy since it returns you the key value of the new row(s). Though RETURNING is often accompanied by * for all fields, you can limit the fields as we do in Example 7-16.

Example 7-16. RETURNING changed records of an UPDATE

```
UPDATE census.lu_fact_types AS f
SET short_name = Replace(Replace(Lower(f.fact_subcats[4]),' ','_'),':','')
WHERE f.fact_subcats[3] = 'Hispanic or Latino:' AND f.fact_subcats[4] > ''
RETURNING fact_type_id, short_name;
```

```
fact_type_id  |                   short_name
--------------+-----------------------------------------------
96            | white_alone
97            | black_or_african_american_alone
98            | american_indian_and_alaska_native_alone
99            | asian_alone
100           | native_hawaiian_and_other_pacific_islander_alone
101           | some_other_race_alone
102           | two_or_more_races
```

Composite Types in Queries

Composites provide a lot of flexibility to PostgreSQL. The first time you see a query with composites, you might be surprised. In fact, you might come across their versatility by accident when making a typo in an SQL statement. Try the following query:

```
SELECT X FROM census.lu_fact_types As X LIMIT 2;
```

At first glance, you might think that we left out a `.*` by accident, but check out the result:

```
x
-------------------------------------------------------------------
(86,Population,"{D001,Total:}",d001)
(87,Population,"{D002,Total:,""Not Hispanic or Latino:""}",d002)
```

Recall from an earlier section "All Tables Are Custom" on page 71 where we demonstrated that PostgreSQL automatically create composite types of all tables in PostgreSQL. Instead of erroring out, our above example returns the canonical representation of an `lu_fact_type` object. Looking at the first record: `86` is the fact_type_id, `Population` is the category, and `{D001,Total:}` is the fact_subcats property, which happens to be an array in its own right.

In addition to being able to output a row as a single object, there are several functions that can take a composite or row as an input. For example, you can feed a row into the `array_agg`, `hstore`, and countless other functions. If you are using PostgreSQL 9.2 or above, and are building AJAX apps, you can take advantage of the built-in JavaScript

Object Notation (JSON) support and use a combination of `array_agg` and `array_to_json` to output a whole query as a single JSON object, as we demonstrate in Example 7-17.

Example 7-17. Query to JSON output

```
SELECT array_to_json(array_agg(f)  ❶) As ajaxy_cats ❷
FROM (SELECT MAX(fact_type_id) As max_type, category  ❸
 FROM census.lu_fact_types
 GROUP BY category) As f;❹
```

This will give you an output of:

```
                    ajaxy_cats
--------------------------------------------------
[{"max_type":102,"category":"Population"},
{"max_type":153,"category":"Housing"}]
```

❸❹ Defines a subquery where each row will be represented as `f`.

❶ Collects all these `f` rows into one composite array of `f`s.

❷ Converts the composite array into a JSON object. The canonical representation of a JSON object follows the JSON output standard.

CHAPTER 8
Writing Functions

As with most databases, you can string a series of SQL statements together and treat them as a unit. Different databases ascribe different names for this unit—stored procedures, modules, macros, prepared statements, and so on. PostgreSQL calls them functions. Aside from simply unifying various SQL statements, these units often add the capability to control the execution of the SQL statements through using procedural language (PL). In PostgreSQL, you have your choice of languages when it comes to writing functions. Often packaged along with binary installers are SQL, C, PL/pgSQL, PL/Perl, PL/Python. In version 9.2, you'll also find *plv8js*, which will allow you to write procedural functions in JavaScript. `plv8js` should be an exciting addition to web developers and a nice companion to the built-in JSON type.

You can always install additional languages such as *PL/R*, PL/Java, *PL/sh*, and even experimental ones geared for high-end processing and AI, such as *PL/Scheme* or *PgOpenCL*. A list of available languages can be found here:*Procedural Languages*

Anatomy of PostgreSQL Functions

Function Basics

Regardless which language you choose to write a particular function, they all share a similar structure.

Example 8-1. Basic Function Structure

```
CREATE OR REPLACE FUNCTION func_name(
    arg1 arg1_datatype)
RETURNS some_type | setof sometype | TABLE (..) AS
$$
BODY of function
$$
LANGUAGE language_of_function
```

Functional definitions can include additional qualifiers to optimize execution and to enforce security. We describe these below:

- `LANGUAGE` has to be one you have installed in your database. You can get a list with the query: `SELECT lanname FROM pg_language;`.
- `VOLATILITY` defaults to `VOLATILE` if not specified. It can be set to `STABLE`, `VOLATILE`, or `IMMUTABLE`. This setting gives the planner an idea if the results of a function can be cached. `STABLE` means that the function will return the same value for the same inputs within the same query. `VOLATILE` means the function may return something different with each call, even with same inputs. Functions that change data or are a function of other environment settings like time, should be marked as `VOLATILE`. `IMMUTABLE` means given the same inputs the function is guaranteed to return the same result. The volatility setting is merely a hint to the query planner. It may choose to not cache if it concludes caching is less cost effective than recomputation. However, if you mark a function as `VOLATILE` it will always recompute.
- `STRICT`. A function is assumed to be not strict unless adorned with `STRICT`. A strict function will always return NULL if any inputs are NULL and doesn't bother evaluating the function so saves some processing. When building SQL functions, you should be careful about using `STRICT` as it will prevent index usage as described in *STRICT on SQL Functions*
- `COST` is a relative measure of computational intensiveness. SQL and PL/pgSQL functions default to `100` and C functions to `1`. This affects the order functions will be evaluated in a `WHERE` clause and also the likeliness of caching. The higher the value, the more costly the function is assumed to be.
- `ROWS` is only set for set returning functions and is an estimate of how many rows will be returned; used by planner to arrive at best strategy. Format would be `ROWS 100`.
- `SECURITY DEFINER` is an optional clause that means run under the context of the owner of the function. If left out, a function runs under the context of the user running the function. This is useful for giving people rights to update a table in a function but not direct update access to a table and also tasks that normally require super user rights.

Trusted and Untrusted Languages

Function languages can be divided into two levels of trust. Many—but not all—languages offer both a trusted and untrusted version.

- `Trusted`—Trusted languages are languages that don't have access to the filesystem beyond the data cluster and can't execute OS commands. They can be created by any user. Languages like SQL, PL/pgSQL, PL/Perl are trusted. It basically means they can't do damage to the underlying OS.

- `Untrusted`—Untrusted languages are those that can interact with the OS and even call on webservices or execute functions on the OS. Functions written in these languages can only be created by super users, however, a super user can delegate rights for another user to use them by using the `SECURITY DEFINER` setting. By convention, these languages have a `u` at the end of the name to denote that they're untrusted—for instance, `PL/PerlU`, `PL/PythonU`.

Writing Functions with SQL

Writing SQL functions[1] is fast and easy. Take your existing SQL statements, add a functional header and footer, and you're done. The ease does mean you'll sacrifice flexibility. You won't have fancy control languages to create conditional execution branches. More restrictively, you can't run dynamic SQL statements that you piece together based on parameters as you can in most other languages. On the positive side, SQL functions are often inlined by the query planner into the overall plan of a query since the planner can peek into the function. Functions in other languages are always treated as blackboxes. Inlining allows SQL functions to take advantage of indexes and collapse repetitive computations.

Your basic scalar returning SQL function is shown in Example 8-2:

Example 8-2. SQL function to return key of inserted record

```
CREATE OR REPLACE FUNCTION ins_logs(param_user_name varchar, param_description text)
 RETURNS integer AS
$$ INSERT INTO logs(user_name, description) VALUES($1, $2)
RETURNING log_id; $$
LANGUAGE 'sql' VOLATILE;
```

To call function Example 8-2, we would execute:

```
    SELECT ins_logs('lhsu', 'this is a test') As new_id;
```

Similarly, you can update data with an SQL function and return a scalar or void as shown in Example 8-3.

Example 8-3. SQL function to update a record

```
CREATE OR REPLACE FUNCTION upd_logs(log_id integer, param_user_name varchar,
param_description text)
 RETURNS void AS
$$ UPDATE logs SET user_name = $2, description = $3, log_ts = CURRENT_TIMESTAMP
 WHERE log_id = $1;$$
LANGUAGE 'sql' VOLATILE;
```

To execute:

```
SELECT upd_logs(12,'robe', 'Change to regina');
```

1. SQL in this context really means a language for writing functions.

Prior to 9.2, SQL functions could only use the ordinal position of the input arguments in the body of the function. After 9.2, you have the option of using named arguments, for example you can write `param_1`, `param_2` instead of `$1`, `$2`. SQL functions are the only ones that retained this limitation until now.

Functions in almost all languages can return sets. SQL functions can also return sets. There are three common approaches of doing this, using ANSI-SQL standard `RETURNS TABLE` syntax, using `OUT` parameters, or returning a composite data type. The `RETURNS TABLE` approach requires PostgreSQL 8.3 or above, but is closer to what you'll see in other relational databases. In Example 8-4, we demonstrate how to write the same function in three different ways.

Example 8-4. Examples of function returning sets

Using returns table:

```
CREATE FUNCTION sel_logs_rt(param_user_name varchar)
RETURNS TABLE (log_id int, user_name varchar(50), description text, log_ts timestamptz) AS
$$
SELECT log_id, user_name, description, log_ts FROM logs WHERE user_name = $1;
$$
LANGUAGE 'sql' STABLE;
```

Using OUT parameters:

```
CREATE FUNCTION sel_logs_out(param_user_name varchar, OUT log_id int
 , OUT user_name varchar, OUT description text, OUT log_ts timestamptz)
RETURNS SETOF record AS
$$
SELECT * FROM logs WHERE user_name = $1;
$$
LANGUAGE 'sql' STABLE;
```

Using composite type:

```
CREATE FUNCTION sel_logs_so(param_user_name varchar)
RETURNS SETOF logs AS
$$
SELECT * FROM logs WHERE user_name = $1;
$$
LANGUAGE 'sql' STABLE;
```

All functions in Example 8-4 can be called using:

```
SELECT * FROM sel_logs_rt('lhsu');
```

Writing PL/pgSQL Functions

When your functional needs exceed reaches beyond SQL, PL/pgSQL is the most common option. PL/pgSQL stands apart from SQL in that you can declare local variables

using DECLARE, you can have control flow, and the body of the function needs be enclosed in a BEGIN..END block. To demonstrate the difference, we have rewritten Example 8-4 as a PL/pgSQL function.

Example 8-5. Function to return a table using PL/pgSQL

```
CREATE FUNCTION sel_logs_rt(param_user_name varchar)
RETURNS TABLE (log_id int, user_name varchar(50), description text, log_ts timestamptz) AS
$$
BEGIN
    RETURN QUERY
    SELECT log_id, user_name, description, log_ts
      FROM logs WHERE user_name = param_user_name;
END;
$$
LANGUAGE 'plpgsql' STABLE;
```

Writing PL/Python Functions

Python is a slick language with a vast number of available libraries. PostgreSQL is the only database we know of that'll let you compose functions using Python. PostgreSQL 9.0+ supports both Python 2 and Python 3.

You can have both plpython2u andplpython3u installed in the same database, but you can't use them in the same session. This means that you can't write a query that contains both plpython2u and plpython3u-written functions.

In order to use PL/Python, you first need to install Python on your server. For Windows and Mac OS, Python installers are available at *http://www.python.org/download/*. For Linux/Unix systems, Python binaries are usually available via the various distros. For details, refer to PL/Python. After you have Python on your server, proceed to install the PostgreSQL Python extension using the commands below:

```
CREATE EXTENSION plpython2u;
CREATE EXTENSION plpython3u;
```

(You will find a third extension called plpythonu, which is an alias for plpython2u and intended for backwards compatibility.) Make sure you have Python properly running on your server before attempting to install the extension or else you will run into errors.

You should install a minor version of Python that matches what your plpythonu extensions were compiled against. For example, if your plpython2u is compiled against 2.7, then you'll need to install Python 2.7.

Basic Python Function

PostgreSQL automatically converts PostgreSQL datatypes to Python datatypes and back. PL/Python is capable of returning arrays and even composite types. You can use PL/Python to write triggers and create aggregate functions. We've demonstrated some of these on the *Postgres OnLine Journal*, in *PL/Python Examples*.

Python allows you to perform feats that aren't possible in PL/pgSQL. In Example 8-6, we demonstrate how to write a PL/Python function that does a text search of the online PostgreSQL document resource site.

Example 8-6. Searching PostgreSQL docs using PL/Python

```
CREATE OR REPLACE FUNCTION postgresql_help_search(param_search text)
RETURNS text AS
$$
import urllib, re  ❶
response = urllib.urlopen('http://www.postgresql.org/search/?u=%2Fdocs%2Fcurrent%2F&q=' +
param_search)   ❷
raw_html = response.read() ❸
result = raw_html[raw_html.find("<!-- docbot goes here -->"):raw_html.find("<!--
pgContentWrap -->") - 1]   ❹
result = re.sub('<[^<]+?>', '', result).strip()  ❺
return result ❻
$$
LANGUAGE plpython2u SECURITY DEFINER STABLE;
```

❶ Import the libraries we'll be using.

❷ Web search concatenating user input parameters.

❸ Read response and save html to a variable called `raw_html`.

❹ Save the part of the `raw_html` that starts with `<!-- docbot goes here -->` and ends just before the beginning of `<!-- pgContentWrap -->`.

❺ Strip HTML and white space from front and back and then re-save back to variable called `result`.

❻ Return final `result`.

Calling Python functions is no different than functions written in other languages. In Example 8-7, we use the function we created in Example 8-6 to output the result with three search terms.

Example 8-7. Using Python function in a query

```
SELECT search_term, left(postgresql_help_search(search_term), 125) As result FROM (VALUES
('regexp_match'), ('pg_trgm'), ('tsvector')) As X(search_term);
```

```
search_term  |  result
regexp_match  |  Results 1-7 of 7.
    1. PostgreSQL: Documentation: Manuals: Pattern Matching [1.46]
    ...matching a POSIX regular
pg_trgm|Results 1-8 of 8.
```

```
    1. PostgreSQL: Documentation: Manuals: pg_trgm [0.66]
    ...pg_trgm
The
pg_trgm
module provide
tsvector  |  Results 1-20 of 32.
   Result pages: 1 2 Next
    1. PostgreSQL: Documentation: Manuals: Text Search Functions
(3 rows)
```

Recall that PL/Python is an untrusted language without a trusted counterpart. This means it's capable of interacting with the filesystem of the OS and a function can only be created by super users. Our next example uses PL/Python to retrieve file listings from a directory. Keep in mind that PL/Python function runs under the context of the postgres user account, so you need to be sure that account has adequate access to the relevant directories.

Example 8-8. List files in directories

```
CREATE OR REPLACE FUNCTION list_incoming_files()
RETURNS SETOF text AS
$$
import os
return os.listdir('/incoming')
$$
LANGUAGE 'plpython2u' VOLATILE SECURITY DEFINER;
```

You can run the function in Example 8-8 with the query below:

```
SELECT filename FROM list_incoming_files() As filename WHERE filename ILIKE '%.csv'
```

Trigger Functions

No database of merit should be without triggers to automatically detect and handle changes in data. Triggers can be added to both tables and views. PostgreSQL offers both statement-level triggers and row-level triggers. Statement triggers run once per statement, while row triggers run for each row called. For instance, suppose we execute an `UPDATE` command that affects 1,500 rows. A statement-level trigger will fire only once, whereas the row-level trigger can fire up to 1,500 times. More distinction is made between a `BEFORE`, `AFTER`, and `INSTEAD OF` trigger. A `BEFORE` trigger fires prior to the execution of the command giving you a chance to cancel and change data before it changes data. An `AFTER` trigger fires afterwards giving you a chance to retrieve revised data values. `AFTER` triggers are often used for logging or replication purposes. The `INSTEAD OF` triggers run instead of the normal action. `INSTEAD OF` triggers can only be used with views. `BEFORE` and `AFTER` triggers can only be used with tables. To gain a better understanding of the interplay between triggers and the underlying command, we refer

you to the official documentation *Overview of Trigger Behavior*. We demonstrated an example of a view trigger in Example 7-2.

PostgreSQL offers specialized functions to handle triggers. These trigger functions act just like any other function and have the same basic structure as your standard function. Where they differ are in the input parameter and the output type. A trigger function never takes a literal input argument though internally they have access to the trigger state data and can modify it. They always output a datatype called a `trigger`. Because PostgreSQL trigger functions are just another function, you can reuse the same trigger function across different triggers. This is usually not the case for other databases where each trigger has its own non-reusable handler code. Each trigger must have exactly one associated triggering function. To apply multiple triggering functions, you must create multiple triggers against the same event. The alphabetical order of the trigger name determines the order of firing and each trigger passes the revised trigger state data to the next trigger in the list.

You can use almost any language to write trigger functions, with SQL being the notable exception. Our example below uses PL/pgSQL, which is by far the most common language for writing triggers. You will see that we take two steps: First, we write the trigger function. Next, we attach the trigger function to the appropriate trigger, a powerful extra step that decouples triggers from trigger functions.

Example 8-9. Trigger function to timestamp new and changed records

```
CREATE OR REPLACE FUNCTION trig_time_stamper() RETURNS trigger AS ❶
$$
BEGIN
    NEW.upd_ts := CURRENT_TIMESTAMP;
    RETURN NEW;
END;
$$
LANGUAGE plpgsql VOLATILE;

CREATE TRIGGER trig_1
BEFORE INSERT OR UPDATE ❷ OF session_state, session_id ❸
 ON web_sessions ❹
 FOR EACH ROW
 EXECUTE PROCEDURE trig_time_stamper();
```

❶ Define the trigger function. This function can be used on any table that has a `upd_ts` column . It changes the value of the `upd_ts` field of the new record before returning. Trigger functions that change values of a row should only be called in the `BEFORE` event, because in the `AFTER` event, all updates to the `NEW` record will be ignored.

❷ The trigger will fire before the record is committed.

❸ This is a new feature introduced in PostgreSQL 9.0+ that allows us to limit the firing of the trigger only if specified columns have changed. In prior versions, you would do this in the trigger function itself using a series of comparison between

`OLD.some_column` and `NEW.some_column`. This feature is not supported for `INSTEAD OF` triggers.

❹ Binds the trigger to the table.

Aggregates

Aggregates are another type of specialized function offered up by PostgreSQL. In many other databases, you're limited to ANSI-SQL aggregate functions such as `MIN()`, `MAX()`, `AVG()`, `SUM()`, and `COUNT()`. You can define your own aggregates in PostgreSQL. Don't forget that any aggregate function in PostgreSQL can be used as a window function. Altogether this makes PostgreSQL the most customizable database in existence today.

You can write aggregates in almost any language, SQL included. An aggregate is generally composed of one or more functions. It must have at least a state transition function to perform the computation and optional functions to manage initial and final states. You can use a different language for each of the functions should you choose. We have various examples of building aggregates using PL/pgSQL, PL/Python, and SQL in *PostgreSQL Aggregates*.

Regardless of which languages you code the aggregate, the glue that brings them all together looks the same for all and is of the form:

```
CREATE AGGREGATE myagg(datatype_of_input)
(SFUNC=state_function_name, STYPE=state_type, FINALFUNC=final_func_name,
INITCOND=optional_init_state_value);
```

The final function is optional, but if specified must take as input the result of the state function. The state function always takes as input the *`datatype_of_input`* and result of last state function call. The initial condition is also optional. When present, it is used to initialize the state value. Aggregates can be multi-column as well, as we describe in *How to Create Multi-Column Aggregates*.

Although SQL functions are the simplest of functions to write, you can still go pretty far with them. In this section, we'll demonstrate how to create a geometric mean aggregate function with SQL. A *geometric mean* is the nth root of a product of n positive numbers (`(x1*x2*x3...Xn)^(1/n)`). It has various uses in finance, economics, and statistics. A geometric mean may have more meaning than an arithmetic mean when the numbers are of vastly different scales. A more suitable computational formula uses logarithm to convert a multiplicative process to an additive one (`EXP(SUM(LN(x))/n)`). We'll be using this method in our example.

For our geometric mean aggregate, we'll use two functions: a state function to add the logs and a final exponential function to convert the logs back. We will also specify an initial condition of zero when we put everything together.

Example 8-10. Geometric mean aggregate: State function

```
CREATE OR REPLACE FUNCTION geom_mean_state(prev numeric[2], next numeric)
RETURNS numeric[2] AS
$$
SELECT CASE WHEN $2 IS NULL or $2 = 0 THEN $1
  ELSE ARRAY[COALESCE($1[1],0) + ln($2), $1[2] + 1] END;
$$
LANGUAGE sql IMMUTABLE;
```

Our transition state function, as shown in Example 8-10, takes two inputs: the previous state passed in as a one-dimensional array with two elements and also the next element in the aggregation process. If the next element is NULL or zero, the state function returns the prior state. Otherwise, it'll return an array where the first element is the logarithmic sum and the second being the current count. We will need a final function that takes the sum from the state transition and divides by the count.

Example 8-11. Geometric mean aggregate: Final function

```
CREATE OR REPLACE FUNCTION geom_mean_final(numeric[2])
RETURNS numeric AS
$$
SELECT CASE WHEN $1[2] > 0 THEN exp($1[1]/$1[2]) ELSE 0 END;
$$
LANGUAGE sql IMMUTABLE;
```

Now we stitch all the pieces together in our aggregate definition. Note that our aggregate has an initial condition that is the same as what is returned by our state function.

Example 8-12. Geometric mean aggregate: Putting all the pieces together

```
CREATE AGGREGATE geom_mean(numeric) (SFUNC=geom_mean_state, STYPE=numeric[]
 , FINALFUNC=geom_mean_final, INITCOND='{0,0}');
```

Let's take our `geom_mean()` function for a test drive. We're going to compute a heuristic rating for racial diversity and list the top five most racially diverse counties in Massachusetts.

Example 8-13. Top five most racially diverse counties using geometric mean

```
SELECT left(tract_id,5) As county, geom_mean(val) As div_county
FROM census.vw_facts
WHERE category = 'Population' AND short_name != 'white_alone'
GROUP BY county
ORDER BY div_county DESC LIMIT 5;
```

```
county |     div_county
-------+--------------------
25025  | 85.1549046212833364
25013  | 79.5972921427888918
25017  | 74.7697097102419689
25021  | 73.8824162064128504
25027  | 73.5955049035237656
```

Let's put things into overdrive and try our new aggregate function as a window aggregate.

Example 8-14. Top five most racially diverse census tracts with average

```
WITH X AS (SELECT tract_id, left(tract_id,5) As county
    , geom_mean(val) OVER(PARTITION BY tract_id) As div_tract
    , ROW_NUMBER() OVER(PARTITION BY tract_id) As rn
    , geom_mean(val) OVER(PARTITION BY left(tract_id,5)) As div_county
  FROM census.vw_facts WHERE category = 'Population' AND short_name != 'white_alone')
SELECT tract_id, county, div_tract, div_county
 FROM X
 WHERE rn = 1
 ORDER BY div_tract DESC, div_county DESC LIMIT 5;
```

```
tract_id    | county |      div_tract       |     div_county
------------+--------+----------------------+--------------------
25025160101 | 25025  | 302.6815688785928786 | 85.1549046212833364
25027731900 | 25027  | 265.6136902148147729 | 73.5955049035237656
25021416200 | 25021  | 261.9351057509603296 | 73.8824162064128504
25025130406 | 25025  | 260.3241378371627137 | 85.1549046212833364
25017342500 | 25017  | 257.4671462282508267 | 74.7697097102419689
```

CHAPTER 9

Query Performance Tuning

Sooner or later, we'll all face a query that takes just a bit longer to execute than what we have patience for. The best and easiest fix to a sluggish query is to perfect the underlying SQL, followed by adding indexes, and updating planner statistics. To guide you in these pursuits, Postgres comes with a built-in explainer that informs you how the query planner is going to execute your SQL. Armed with your knack for writing flawless SQL, your instinct to sniff out useful indexes, and the insight of the explainer, you should have no trouble getting your queries to run as fast as what your hardware budget will allow.

EXPLAIN and EXPLAIN ANALYZE

The easiest tool for targeting query performance problems is using the `EXPLAIN` and `EXPLAIN ANALYZE` commands. These have been around ever since the early years of PostgreSQL. Since then it has matured into a full-blown tool capable of reporting highly detailed information about the query execution. Along the way, it added to its number of output formats. In PostgreSQL 9.0+, you can even dump the output to XML or JSON. Perhaps the most exciting enhancement for the common user came when pgAdmin introduced graphical `EXPLAIN` several years back. With a hard and long stare, you can identify where the bottlenecks are in your query, which tables are missing indexes, and whether the path of execution took an unexpected turn.

`EXPLAIN` will give you just an idea of how the planner intends to execute the query without running it. `EXPLAIN ANALYZE` will actually execute the query and give you comparative analysis of expected versus actual. For the non-graphical version of `EXPLAIN`, simply preface your SQL with the `EXPLAIN` or `EXPLAIN ANALYZE`. `VERBOSE` is an optional modifier that will give you details down to the columnar level. You launch graphical `EXPLAIN` via pgAdmin. Compose the query as usual, but instead of executing it, choose `EXPLAIN` or `EXPLAIN ANALYZE` from the drop down menu.

It goes without saying that to use graphical explain, you'll need more than a command prompt. To those of you who pride yourself on being self-sufficient using only the command line: good for you!

Let's try an example, we'll first use the `EXPLAIN ANALYZE` command.

Example 9-1. Explain analyze

```
EXPLAIN ANALYZE
SELECT left(tract_id,5) As county_code, SUM(hispanic_or_latino) As tot
 , SUM(white_alone) As tot_white
 , SUM(coalesce(hispanic_or_latino,0) - coalesce(white_alone,0)) AS non_white
FROM census.hisp_pop
GROUP BY county_code
ORDER BY county_code;
```

The output of Example 9-1 is shown in Example 9-2.

Example 9-2. EXPLAIN ANALYZE output

```
GroupAggregate (cost=111.29..151.93 rows=1478 width=20)
               (actual time=6.099..10.194 rows=14 loops=1)
  ->  Sort (cost=111.29..114.98 rows=1478 width=20)
           (actual time=5.897..6.565 rows=1478 loops=1)
        Sort Key: ("left"((tract_id)::text, 5))
        Sort Method: quicksort  Memory: 136kB
        ->  Seq Scan on hisp_pop  (cost=0.00..33.48 rows=1478 width=20)
                                  (actual time=0.390..2.693 rows=1478 loops=1)
Total runtime: 10.370 ms
```

If reading the output is giving you a headache, here's the graphical `EXPLAIN`:

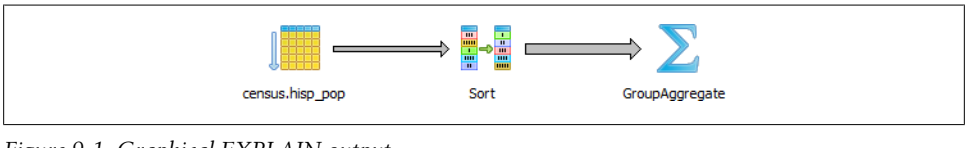
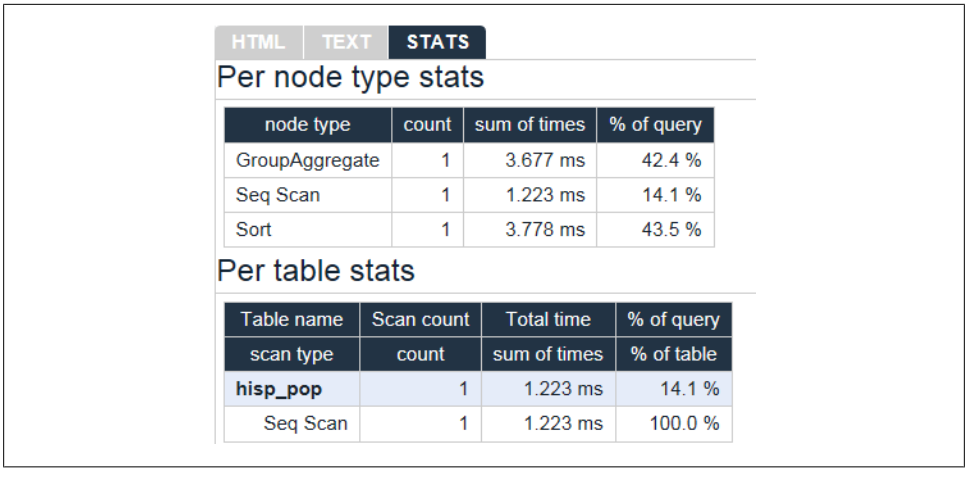


Figure 9-1. Graphical EXPLAIN output

Before leaving the section on `EXPLAIN`, we must pay homage to a new online `EXPLAIN` tool created by Hubert "depesz" Lubaczewski. Using his site, you can copy and paste the text output of your `EXPLAIN`, and it will show you a beautifully formatted stats report as shown in Figure 9-2.

In the HTML tab, a nicely reformatted color-coded table of the plan will be displayed, with problem areas highlighted in vibrant colors, as shown in Figure 9-3.

Although the HTML table in Figure 9-3 provides much the same information as our plain-text plan, the color coding and breakout of numbers makes it easier to see that



HTML TEXT STATS

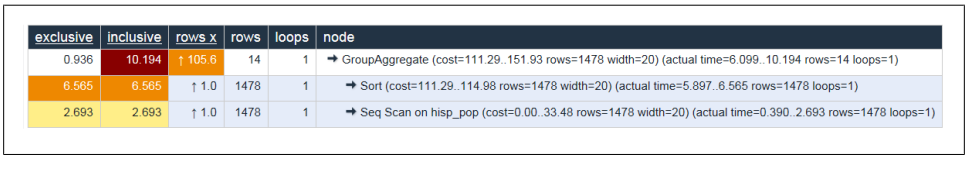
Per node type stats

node type	count	sum of times	% of query
GroupAggregate	1	3.677 ms	42.4 %
Seq Scan	1	1.223 ms	14.1 %
Sort	1	3.778 ms	43.5 %

Per table stats

Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
hisp_pop	1	1.223 ms	14.1 %
Seq Scan	1	1.223 ms	100.0 %

Figure 9-2. Online EXPLAIN stats



exclusive	inclusive	rows x	rows	loops	node
0.936	10.194	↑ 105.6	14	1	➞ GroupAggregate (cost=111.29..151.93 rows=1478 width=20) (actual time=6.099..10.194 rows=14 loops=1)
6.565	6.565	↑ 1.0	1478	1	➞ Sort (cost=111.29..114.98 rows=1478 width=20) (actual time=5.897..6.565 rows=1478 loops=1)
2.693	2.693	↑ 1.0	1478	1	➞ Seq Scan on hisp_pop (cost=0.00..33.48 rows=1478 width=20) (actual time=0.390..2.693 rows=1478 loops=1)

Figure 9-3. Online EXPLAIN HTML table

our actual values are far off from the estimated numbers. This suggests that our planner stats are probably not up to date.

Writing Better Queries

The best and easiest way to improve query performance is to start with well-written queries. Four out of five queries we encounter are not written as efficiently as they could be. There appears to be two primary causes for all this bad querying. First, we see people reuse SQL patterns without thinking. For example, if they successfully write a query using a left join, they will continue to use left join when incorporating more tables instead of considering the sometimes more appropriate inner join. Unlike other programming languages, the SQL language does not lend itself well to blind reuse. Second, people don't tend to keep up with the latest developments in their dialect of SQL. If a PostgreSQL user is still writing SQL as if he still had an early version, he'd be oblivious to all the syntax-saving (and mind-saving) addendums that have come along. Writing efficient SQL takes practice. There's no such thing as a wrong query as long as you get the expected result, but there is such a thing as a slow query. In this section, we'll go over some of the common mistakes we see people make. Although this book is about PostgreSQL, our constructive recommendations are applicable to other relational databases as well.

Overusing Subqueries in SELECT

A classic newbie mistake is to think about a query in independent pieces and then trying to gather them up all in one final SELECT. Unlike conventional programming, SQL doesn't take kindly to the idea of blackboxing where you can write a bunch of subqueries independently and then assemble them together mindlessly to get the final result. You have to give your query the holistic treatment. How you piece together data from different views and tables is every bit as important as how you go about retrieving the data in the first place.

Example 9-3. Overusing subqueries

```
SELECT tract_id
    ,(SELECT COUNT(*) FROM census.facts As F
   WHERE F.tract_id = T.tract_id) As num_facts
    ,(SELECT COUNT(*) FROM census.lu_fact_types As Y
   WHERE Y.fact_type_id IN (SELECT fact_type_id
      FROM census.facts F WHERE F.tract_id = T.tract_id)) As num_fact_types
FROM census.lu_tracts As T;
```

The graphical `EXPLAIN` plan for Example 9-3 is shown in Figure 9-4.

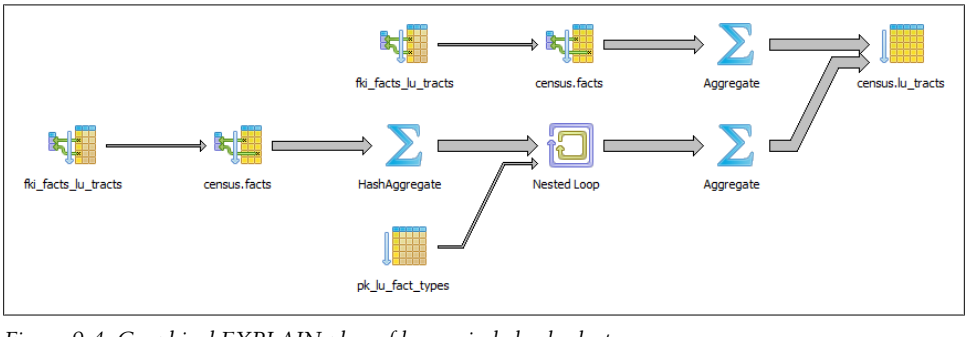


Figure 9-4. Graphical EXPLAIN plan of long-winded subselects

We'll save you the eyesore from seeing the gnarled output of the non-graphical `EXPLAIN`. Instead, we'll show you in Figure 9-5 the output from using the online `EXPLAIN` at *http://explain.depesz.com*.

Example 9-3 can be more efficiently written as shown in Example 9-4. This version of the query is not only shorter, but faster than the prior one. If you have even more rows or weaker hardware, the difference would be even more pronounced.

Example 9-4. Overused subqueries simplified

```
SELECT T.tract_id, COUNT(f.fact_type_id) As num_facts, COUNT(DISTINCT fact_type_id) As
num_fact_types
FROM census.lu_tracts As T LEFT JOIN census.facts As F ON T.tract_id = F.tract_id
GROUP BY T.tract_id;
```

The graphical `EXPLAIN` plan of Example 9-4 is shown in Figure 9-6.

HTML TEXT STATS

exclusive	inclusive	rows x	rows	loops	node
10.709	1292.135	↑ 1.0	1478	1	→ Seq Scan on lu_tracts t (cost=0.00..615535.37 rows=1478 width=12) (actual tim
					SubPlan (forSeq Scan)
63.554	264.562	↑ 1.0	1	1478	→ Aggregate (cost=207.86..207.87 rows=1 width=0) (a
153.712	201.008	↑ 1.0	68	1478	→ Bitmap Heap Scan on facts f (cost=4.79..207.69 rows=68 width=0) (actual time Recheck Cond: ((tract_id)::text = (t.tract_id)::text)
47.296	47.296	↑ 1.0	68	1478	→ Bitmap Index Scan on fki_facts_lu_tracts (cost=0.00..4.78 rows=68 width=0) (actual tim Index Cond: ((tract_id)::text = (t.tract_id)::text)
59.120	1016.864	↑ 1.0	1	1478	→ Aggregate (cost=208.56..208.57 rows=1 width=0) (a
314.814	957.744	↑ 1.0	68	1478	→ Nested Loop (cost=207.86..208.39 rows=68 width=0) (actual ti
155.190	341.418	↓ 68.0	68	1478	→ HashAggregate (cost=207.86..207.87 rows=1 width=4) (actua
141.888	186.228	↑ 1.0	68	1478	→ Bitmap Heap Scan on facts f (cost=4.79..207.69 rows=68 width=4) (act Recheck Cond: ((tract_id)::text = (t.tract_i
44.340	44.340	↑ 1.0	68	1478	→ Bitmap Index Scan on fki_facts_lu_trac (cost=0.00..4.78 rows=68 width=0) (ac Index Cond: ((tract_id)::text = (t.tract_i
301.512	301.512	↑ 1.0	1	100504	→ Index Scan using pk_lu_fact_types on lu_fact (cost=0.00..0.50 rows=1 width=4) (actual time Index Cond: (fact_type_id = f.fact_type_id)

Figure 9-5. Online EXPLAIN of overusing subqueries

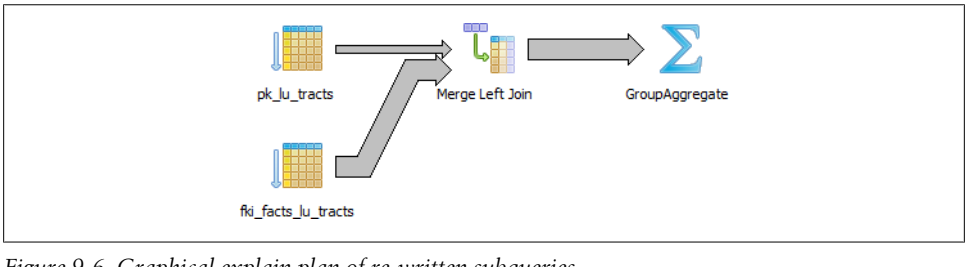


Figure 9-6. Graphical explain plan of re-written subqueries

Keep in mind that we're not asking you to avoid subqueries. We're simply asking you to use them judiciously. When you do use them, be sure to pay extra attention on how you combine them into the main query. Finally, remember that a subquery should try to work with the the main query, not independent of it.

Avoid SELECT *

`SELECT *` is wasteful. It's akin to printing out a 1,000-page document when you only need ten pages. Besides the obvious downside of adding to network traffic, there are two other drawbacks that you might not think of.

First, PostgreSQL stores large blob and text objects using TOAST (The Oversized-Attribute Storage Technique). TOAST maintains side tables for PostgreSQL to store this extra data. The larger the data the more internally divided up it is. So retrieving a large field means that TOAST must assemble the data across different rows across different tables. Imagine the extra processing should your table contain text data the size of *War and Peace* and you perform an unnecessary `SELECT *`.

Second, when you define views, you often will include more columns than you'll need. You might even go so far as to use `SELECT *` inside a view. This is understandable and perfectly fine. PostgreSQL is smart enough that you can have all the columns you want in your view definition and even include complex calculations or joins without incurring penalty as long as you don't ask for them. `SELECT *` asks for everything. You could easily end up pulling every column out of all joined tables inside the view.

To drive home our point, let's wrap our census in a view and use the slow subselect example we proposed:

```
CREATE OR REPLACE VIEW vw_stats AS
SELECT tract_id
 ,(SELECT COUNT(*) FROM census.facts As F WHERE F.tract_id = T.tract_id) As num_facts
 ,(SELECT COUNT(*) FROM census.lu_fact_types As Y
   WHERE Y.fact_type_id
     IN (SELECT fact_type_id FROM census.facts F
       WHERE F.tract_id = T.tract_id)) As num_fact_types
  FROM census.lu_tracts As T;
```

Now if we query our view with this query:

```
SELECT tract_id FROM vw_stats;
```

Execution time is about 21ms on our server. If you looked at the plan, you may be startled to find that it never even touches the facts table because it's smart enough to know it doesn't need to. If we used the following:

```
SELECT * FROM vw_stats;
```

Our execution time skyrockets to 681ms, and the plan is just as we had in Figure 9-4. Though we're looking at milliseconds still, imagine tables with tens of millions of rows and hundreds of columns. Those milliseconds could transcribe into overtime at the office waiting for a query to finish.

Make Good Use of CASE

We're always surprised how frequently people forget about using the ANSI-SQL `CASE` expression. In many aggregate situations, a `CASE` can obviate the need for inefficient

subqueries. We'll demonstrate with two equivalent queries and their corresponding plans.

Example 9-5. Using subqueries instead of CASE

```
SELECT T.tract_id, COUNT(*) As tot, type_1.tot AS type_1
 FROM census.lu_tracts AS T
   LEFT JOIN
 (SELECT tract_id, COUNT(*) As tot
  FROM census.facts WHERE fact_type_id = 131
   GROUP BY tract_id) As type_1 ON T.tract_id = type_1.tract_id
    LEFT JOIN census.facts AS F ON T.tract_id = F.tract_id
 GROUP BY T.tract_id, type_1.tot;
```

The graphical explain of Example 9-5 is shown in Figure 9-7.

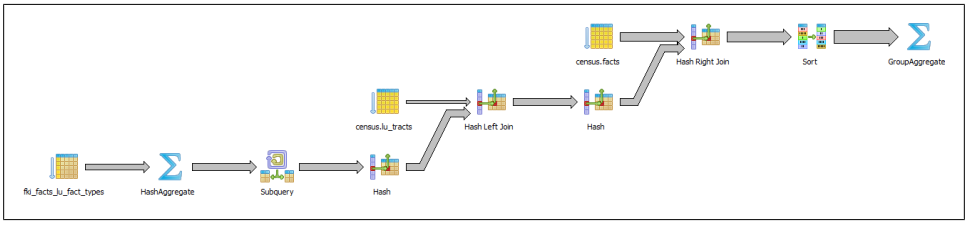


Figure 9-7. Graphical explain of using subqueries instead of CASE

We now rewrite the query using `CASE`. You'll find the revised query shown in Example 9-6 is generally faster and much easier to read than Example 9-5.

Example 9-6. Using CASE instead of subqueries

```
SELECT T.tract_id, COUNT(*) As tot
 , COUNT(CASE WHEN f.fact_type_id = 131 THEN 1 ELSE NULL END) AS type_1
   FROM census.lu_tracts AS T
   LEFT JOIN census.facts AS F ON T.tract_id = F.tract_id
GROUP BY T.tract_id;
```

The graphical explain of Example 9-6 is shown in Figure 9-8.

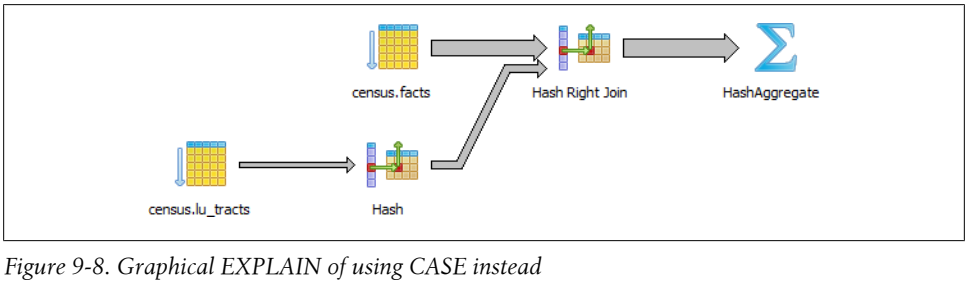


Figure 9-8. Graphical EXPLAIN of using CASE instead

Even though our rewritten query still doesn't use the fact_type index, it's still generally faster than using subqueries because the planner scans the facts table only once.

Although not always the case, a shorter plan is generally not only easier to comprehend, but also performs better than a longer one.

Guiding the Query Planner

The planner's behavior is driven by several cost settings, strategy settings, and its general perception of the distribution of data. Based on distribution of data, the costs it ascribes to scanning indexes, and the indexes you have in place, it may choose to use one strategy over another. In this section we'll go over various approaches for optimizing the planner's behavior.

Strategy Settings

Although PostgreSQL query planner doesn't provide the option to accept index hints like some other databases, when running a query you can disable various strategy settings on a per query or pemanent basis to dissuade the planner from going down an unproductive path. All planner optimizing settings are documented in the section *Planner Method Configuration*. By default, all strategy settings are enabled, giving the planner flexibility to maximize the choice of plans. You can disable various strategies if you have some prior knowledge of the data. Keep in mind that disabling doesn't necessarily mean that the planner will be barred from using the strategy. You're only making a polite request to the planner to avoid it.

Two of our favorite method settings to disable are the `enable_nestloop` and `enable_seqscan`. The reason is that these two strategies tend to be the slowest and should be relegated to be used only as a last resort. Although you can disable them, the planner may still use them when it has no other viable alternative. When you do see them being used, it's a good idea to double-check that the planner is using them out of necessity, not out of ignorance. One quick way to check is to actually disable them.

How Useful Is Your Index?

When the planner decides to perform a sequential scan, it plans to loop through all the rows of a table. It will opt for this route if it finds no index that could satisfy a query condition, or it concludes that using an index is more costly than scanning the table. If you disable the sequential scan strategy, and the planner still insists on using it, then it means that the planner thinks whatever indexes you have in place won't be helpful for the particular query or you are missing indexes altogether. A common mistake people make is they write queries and either don't put indexes in their tables or put in indexes that can't be used by their queries. An easy way to check if your indexes are used is to query the pg_stat_user_indexes and pg_stat_user_tables views.

Let's start off with a query against the table we created in Example 6-7. We'll add a `GIN` index on the array column. `GIN` indexes are one of the few indexes you can use with arrays.

```
CREATE INDEX idx_lu_fact_types ON census.lu_fact_types USING gin (fact_subcats);
```

To test our index, we'll execute a query to find all rows with subcats containing "White alone" or "Asian alone". We explicitly enabled sequential scan even though it's the default setting, just to be sure. The accompanying `EXPLAIN` output is shown in Example 9-7.

Example 9-7. Allow choice of Query index utilization

```
set enable_seqscan = true;
EXPLAIN ANALYZE
SELECT *
FROM census.lu_fact_types
WHERE fact_subcats && '{White alone, Asian alone}'::varchar[];
```

```
Seq Scan on lu_fact_types  (cost=0.00..3.85 rows=1 width=314)
(actual time=0.017..0.078 rows=4 loops=1)
Filter: (fact_subcats && '{"White alone","Asian alone"}'::character varying[])
Total runtime: 0.112 ms
```

Observe that when `enable_seqscan` is enabled, our index is not being used and the planner has chosen to do a sequential scan. This could be because our table is so small or because the index we have is no good for this query. If we repeat the query but turn off sequential scan beforehand, as shown in Example 9-8:

Example 9-8. Coerce query index utilization

```
set enable_seqscan = false;
EXPLAIN ANALYZE
SELECT *
FROM census.lu_fact_types
 WHERE fact_subcats && '{White alone, Black alone}'::varchar[];
```

```
Bitmap Heap Scan on lu_fact_types  (cost=8.00..12.02 rows=1 width=314)
(actual time=0.064..0.067 rows=4 loops=1)
Recheck Cond: (fact_subcats && '{"White alone","Asian alone"}'::character varying[])
->  Bitmap Index Scan on idx_lu_fact_types_gin  (cost=0.00..8.00 rows=1 width=0)
 (actual time=0.050..0.050 rows=4 loops=1)
Index Cond: (fact_subcats && '{"White alone","Asian alone"}'::character varying[])
Total runtime: 0.118 ms
```

We can see from Example 9-8 that we have succeeded in forcing the planner to use the index.

In contrast to the above, if we were to write a query of the form:

```
SELECT * FROM census.lu_fact_types WHERE 'White alone' = ANY(fact_subcats)
```

We would discover that regardless of what we set `enable_seqscan` to, the planner will always do a sequential scan because the index we have in place can't service this query.

So in short, create useful indexes. Write your queries to take advantage of them. And experiment, experiment, experiment!

Table Stats

Despite what you might think or hope, the query planner is not a magician. Its decisions follow prescribed logic that's far beyond the scope of this book. The rules that the planner follows depend heavily on the current state of the data. The planner can't possibly scan all the tables and rows prior to formulating its plan. That would be self-defeating. Instead, it relies on aggregated statistics about the data. To get a sense of what the planner uses, we'll query the pg_stats table with Example 9-9.

```
SELECT attname As colname, n_distinct, most_common_vals AS common_vals,
most_common_freqs As dist_freq
 FROM pg_stats
 WHERE tablename = 'facts'
 ORDER BY schemaname, tablename, attname;
```

Example 9-9. Data distribution histogram

```
colname      | n_distinct | common_vals      |   dist_freq
-------------+------------+------------------+-----------------
fact_type_id |         68 | {135,113..       | {0.0157,0.0156333,...
perc         |        985 | {0.00,..         | {0.1845,0.0579333,0.056...
tract_id     |       1478 | {25025090300,25..| {0.00116667,0.00106667,0.0...
val          |       3391 | {0.000,1.000,2...| {0.2116,0.0681333,0....
yr           |          2 | {2011,2010}      | {0.748933,0.251067}
```

By using pg_stats, the planner gains a sense of how actual values are dispersed within a given column and plan accordingly. The pg_stats table is constantly updated as a background process. After a large data load, or a major deletion, you should manually update the stats by executing a `VACUUM ANALYZE`. `VACUUM` permanently removes deleted rows from tables; `ANALYZE` updates the stats.

Having accurate and current stats is crucial for the planner to make the right decision. If stats differ greatly from reality, planner will often produce poor plans, the most detrimental of these being unnecessary sequential table scans. Generally, only about 20 percent of the entire table is sampled to produce stats. This percentage could be even lower for really large tables. You can control the number of rows sampled on a column-by-column basis by setting the `STATISTICS` value.

```
ALTER TABLE census.facts ALTER COLUMN fact_type_id SET STATISTICS 1000;
```

For columns that participate often in joins and are used heavily in `WHERE` clauses, you should consider increasing sampled rows.

Random Page Cost and Quality of Drives

Another setting that the planner is sensitive to is the RPC `random_page_cost` ratio, the relative cost of the disk in retrieving a record using sequential read versus using random

access. Generally, the faster (and more expensive) the physical disk, the lower the ratio. Default value for RPC is set to 4, which works well for most mechanical hard drives on the market today. With the advent of SSDs, high-end SANs, cloud storage, it's worth tweaking this value. You can set this on a per database, server, or per table space basis, but it makes most sense to set this at the server level in the *postgresql.conf* file. If you have different kinds of disks, you can set it at the `tablespace` level using the *ALTER TABLESPACE* command like so:

```
ALTER TABLESPACE pg_default SET (random_page_cost=2);
```

Details about this setting can be found at *Random Page Cost Revisited*. The article suggests the following settings:

- High-End NAS/SAN: 2.5 or 3.0
- Amazon EBS and Heroku: 2.0
- iSCSI and other bad SANs: 6.0, but varies widely
- SSDs: 2.0 to 2.5
- NvRAM (or NAND): 1.5

Caching

If you execute a complex query that takes a while to run, you'll often notice the second time you run the query that it's faster, sometimes much, much faster. A good part of the reason for that is due to caching. If the same query executes in sequence and there has been no changes to the underlying data, you should get back the same result. As long as there's space in on-board memory to cache the data, the planner doesn't need to re-plan or re-retrieve.

How do you check what's in the current cache? If you are running PostgreSQL 9.1+, you can install the `pg_buffercache` extension with the command:

```
CREATE EXTENSION pg_buffercache;
```

You can then run a query against the `pg_buffercache` table as shown in Example 9-10 query.

Example 9-10. Are my table rows in buffer cache?

```
SELECT C.relname, COUNT(CASE WHEN B.isdirty THEN 1 ELSE NULL END) As dirty_nodes
 , COUNT(*) As num_nodes
FROM pg_class AS C
 INNER JOIN pg_buffercache B ON C.relfilenode = B.relfilenode AND C.relname IN('facts',
'lu_fact_types')
 INNER JOIN pg_database D ON B.reldatabase = D.oid AND D.datname = current_database())
GROUP BY C.relname;
```

Example 9-10 returned buffered records of facts and lu_fact_types. Of course, to actually see buffered rows, you need to run a query. Try the one below:

```
SELECT T.fact_subcats[2], COUNT(*) As num_fact
 FROM census.facts As F
   INNER JOIN census.lu_fact_types AS T ON F.fact_type_id = T.fact_type_id
 GROUP BY T.fact_subcats[2];
```

The second time you run the query, you should notice at least a 10% performance speed increase and you should see the following cached in the buffer:

```
relname       | dirty_nodes | num_nodes
--------------+-------------+-----------
facts         |           0 |       736
lu_fact_types |           0 |         3
```

The more on-board memory you have dedicated to cache, the more room you'll have to cache data. You can set the amount of dedicated memory by changing `shared_buffers`. Don't increase `shared_buffers` too high since at a certain point you'll get diminishing returns from having to scan a bloated cache. Using common table expressions and immutable functions also lead to more caching.

Nowadays, there's no shortage of on-board memory. In version 9.2 of PostgreSQL, you can take advantage of this fact by pre-caching commonly used tables. pg_prewarm will allow you to rev up your PostgreSQL so that the first user to hit the database can experience the same performance boost offered by caching as later users. A good article that describes this feature is *Caching in PostgreSQL*.

CHAPTER 10

Replication and External Data

PostgreSQL has a number of options for sharing data with external servers or data sources. The first option is PostgreSQL's own built-in replication, which allows you to have a readied copy of your server on another PostgreSQL server. The second option, unveiled in 9.1, is the Foreign Data Wrapper, which allows you to query and copy data from many kinds of external data resources utilizing the SQL/Management of External Datasource (MED) standard. The third option is to use third-party add-ons, many of which are freely available and time-tested.

Replication Overview

You can probably enumerate countless reasons for the need to replicate, but they all boil down to two: availability and scalability. If your main server goes down you want another to immediately assume its role. For small databases, you could just make sure you have another physical server ready and restore the database onto it, but for large databases (say, in the terabytes), the restore itself could take many hours. To avoid the downtime, you'll need to replicate. The other main reason is for scalability. You set up a database to handle your collection of fancy *elephant beetles*. After a few years of unbridled breeding, you now have millions of fancy elephant beetles. People all over the world now come to your site to check out the beetles. You're overwhelmed by the traffic. Replication comes to your aid; you set up a read-only slave server to replicate with your main server. People who just want to learn about your beetles will pull data from the slave. As your audience grows, you can add on more and more slave servers.

Replication Lingo

Before we get too carried away with replication, we better lay down some common terminology used in replication.

Master
: The master server is the database server that is the source of the data being replicated and where all updates happen. As of now you can have only one master when using the built-in replication

features of PostgreSQL. Plans are in place to support multi-master replication scenarios, packaged with future releases of PostgreSQL.

Slave
: A slave is a server where data is copied to. More aesthetically pleasing terms such as subscriber or agent have been bandied about, but slave is still the most apropos. PostgreSQL built-in replication currently only supports read-only slaves.

Write-ahead Log (WAL)
: WAL is the log that keeps track of all transactions. It's often referred to as the transaction log in other databases. To set up replication, PostgreSQL simply makes the logs available for slaves to pull down. Once slaves have the logs, they just need to execute the transactions therein.

Synchronous
: A transaction on the master will not be considered complete until all slaves have updated, guaranteeing zero data loss.

Asynchronous
: A transaction on the master will commit even if slaves haven't been updated. This is useful in the case of distant servers where you don't want transactions to wait because of network latency, but the downside is that your dataset on the slave may lag behind, and the slave may miss some transactions in the event of transmission failure.

Streaming
: Streaming replication model was introduced in 9.0. Unlike prior versions, it does not require direct file access between master and slaves. Instead, it relies on PostgreSQL connection protocol to transmit the WALs.

Cascading Replication
: Introduced in 9.2, slaves can receive logs from nearby slaves instead of directly from the master. This allows a slave to also behave like a master for replication purposes but still only allow read only queries.

PostgreSQL Built-in Replication Advancements

When you set up replication, the additional servers can be on the same physical hardware running on a different port or one on the cloud halfway around the globe. Prior to 9.0, PostgreSQL only offered asynchronous warm slaves. A warm slave will retrieve WAL and keep itself in sync but will not be available for query. It acted only as a standby. Version 9.0 introduced asynchronous hot slaves and also streaming replication where users can execute read-only queries against the slave and replication can happen without direct file access between the servers (using database connections for shipping logs instead). Finally, with 9.1, synchronous replication became a reality. In 9.2, Cascading Streaming Replication was introduced. The main benefit of Cascading Streaming Replication is to reduce latency. It's much faster for a slave to receive updates from a nearby slave than from a master far far away. Built-in replication relies on `WAL` shipping to perform the replication. The disadvantage is that your slaves need to have the same version of PostgreSQL and OS installed to ensure faithful execution of the received logs.

Third-Party Replication Options

In addition to the built-in replication, common third party options abound. Slony and Bucardo are two of the most popular open source ones. Although PostgreSQL is improving replication with each new release, Slony, Bucardo, and other third-party replication options still offer more flexibility. Slony and Bucardo will allow you to replicate individual databases or even tables instead of the entire server. As such, they don't require that all masters and slaves be of the same PostgreSQL version and OS. Both also support multi-master scenarios. However, both rely on additional triggers to initiate the replication and often don't support DDL commands such as creating new tables, installing extensions, and so on. This makes them more invasive than merely shipping logs. Postgres-XC, still in beta, is starting to gain an audience. Postgres-XC is not an add-on to PostgreSQL; rather. it's a completely separate fork focused on providing a write-scalable, multi-master symmetric cluster very similar in purpose to Oracle RAC. To this end, the raison d'etre of Postgres-XC is not replication, but distributed query processing. It is designed with scability in mind rather than high availability.

We urge you to consult a comparison matrix of popular third-party options here: *http://wiki.postgresql.org/wiki/Replication%2C_Clustering%2C_and_Connection_Pooling*.

Setting Up Replication

Let's go over the steps to set up replication. We'll take advantage of streaming introduced in 9.0 so that master and slaves only need to be connected at the PostgreSQL connection level instead of at the directory level to sustain replication. We will also use features introduced in 9.1 that allow you to easily setup authentication accounts specifically for replication.

Configuring the Master

The basic steps for setting up the master server are as follows:

1. Create a replication account.

```
CREATE ROLE pgrepuser REPLICATION LOGIN PASSWORD 'woohoo'
```

2. Alter the following configuration settings in *postgresql.conf*.

```
wal_level = hot_standby
archive_mode = on
max_wal_senders = 10
```

3. Use the archive_command to indicate where the WAL will be saved. With streaming, you're free to choose any directory. More details on this setting can be found at *PostgreSQL PGStandby*.

   On Linux/Unix your archive_command line should look something like:

```
archive_command = 'cp %p ../archive/%f'
```

On Windows:

```
archive_command = 'copy %p ..\\archive\\%f'
```

4. In the *pg_hba.conf*, you want a rule to allow the slaves to act as replication agents. As an example, the following rule will allow a PostgreSQL account named pgrepuser that is on my private network with IP in range 192.168.0.1 to 192.168.0.254 to replicate using a md5 password.

   ```
   host replication pgrepuser 192.168.0.0/24 md5
   ```

5. Shut down the postgreSQL service and copy all the files in the *data* folder EXCEPT for the *pg_xlog* and *pg_log* folders to the slaves. You should make sure that *pg_xlog* and *pg_log* folders are both present on the slaves, but devoid of any files.

   If you have a large database cluster and can't afford a shut down for a long period of time while you're copying, you can use the *pg_basebackup* utility which is located in the *bin* folder of your PostgreSQL install. This will create a copy of the data cluster files in the specified directory and allow you to do a base backup while the postgres server service is running and people are using the system.

Configuring the Slaves

To minimize headaches, slaves should have the same configuration as the master, especially if you'll be using them for failover. In addition to those configurations, in order for it to be a slave, it needs to be able to play back the WAL transactions of the master. So, you need at least the following settings in *postgresql.conf* of a slave:

1. Create a new instance of PostgreSQL with the same version (preferably even micro-versions) as your master server and also same OS at the same patch level. Keeping servers identical is not a requirement and you're more than welcome to TOFTT and see how far you can deviate.
2. Shut down the postgreSQL service.
3. Overwrite the data folder files with those you copied from the master.
4. Set the following configuration settings on the *postgresql.conf*.

   ```
   hot_standby = on
   ```

5. You don't need to run the slaves on the same port as the master, so you can optionally change the port either via *postgresql.conf* or via some other OS specific startup script that sets `PGPORT` before startup. Any startup script will override the setting you have in *postgresql.conf*.
6. Create a new file in the *data* folder called *recovery.conf* that contains the following lines:

   ```
   standby_mode = 'on'
   primary_conninfo = 'host=192.168.0.1 port=5432 user=pgrepuser password=woohoo'
   trigger_file = 'failover.now'
   ```

Host name, IP, and port should be those of the master.

7. Add to the *recovery.conf* file the following line, which varies, depending on the OS:

   On Linux/Unix:

   ```
   restore_command = 'cp %p ../archive/%f'
   ```

   On Windows:

   ```
   restore_command = 'copy %p ..\\archive\\%f'
   ```

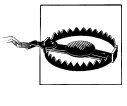
   This command is only needed if the slave can't play the WALs fast enough, so it needs a location to cache them.

Initiate the Replication Process

1. Start up the slave server first. You'll get an error in logs that it can't connect to the master. Ignore.
2. Start up the master server.

You should now be able to connect to both servers. Any changes you make on the master, even structural changes like installing extensions or creating tables, should trickle down to the slaves. You should also be able to query the slaves, but not much else.

When and if the time comes to liberate a chosen slave, create a blank file called *failover.now* in the data folder of the slave. What happens next is that Postgres will complete the playing back of WAL, rename the *recover.conf* to *recover.done*. At that point, your slave will be unshackled from the master and continue life on its own with all the data from the last WAL. Once the slave has tasted freedom, there's no going back. In order to make it a slave again, you'll need to go through the whole process from the beginning.



Unlogged tables don't participate in replication.

Foreign Data Wrappers (FDW)

Foreign Data Wrappers are mechanisms of querying external datasources. PostgreSQL 9.1 introduced this SQL/MED standards compliant feature. At the center of the concept is what is called a foreign table. In this section, we'll demonstrate how to register foreign servers, foreign users, and foreign tables, and finally, how to query foreign tables. You can find a catalog of foreign data wrappers for PostgreSQL at *PGXN FDW* and *PGXN Foreign Data Wrapper*. You can also find examples of usage in *PostgreSQL Wiki FDW*. At this time, it's rare to find FDWs packaged with PostgreSQL except for *fdw_file*. For wrapping anything else, you'll need to compile your own or get them from

someone who already did the work. In PostgreSQL 9.3, you can expect a FDW that will at least wrap other PostgreSQL databases. Also, you're limited to SELECT queries against the FDW, but this will hopefully change in the future so that you can use them to update foreign data as well.

Querying Simple Flat File Data Sources

We'll gain an introduction to FDW using the file_fdw wrapper. To install, use the command:

```
CREATE EXTENSION file_fdw;
```

Although file_fdw can only read from files on your local server, you still need to define a server for it. You register a FDW server with the following command.

```
CREATE SERVER my_server FOREIGN DATA WRAPPER file_fdw;
```

Next, you have to register the tables. You can place foreign tables in any schema you want. We usually create a separate schema to house foreign data. For this example, we'll use our staging schema.

Example 10-1. Make a Foreign Table from Delimited file

```
CREATE FOREIGN TABLE staging.devs (developer VARCHAR(150), company VARCHAR(150))
SERVER my_server
OPTIONS (format 'csv', header 'true', filename '/postgresql_book/ch10/devs.psv', delimiter
'|', null '');
```

When all the set up is finished, we can finally query our pipe delimited file directly:

```
SELECT * FROM staging.devs WHERE developer LIKE 'T%';
```

Once we no longer need our foreign table, we can drop it with the basic SQL command:

```
DROP FOREIGN TABLE staging.devs;
```

Querying More Complex Data Sources

The database world does not appear to be getting more homogeneous. We're witnessing exotic databases sprouting up left and right. Some are fads that go away. Some aspire to dethrone the relational databases altogether. Some could hardly be considered databases. The introduction of foreign data wrappers is in part a response to the growing diversity. Resistance is futile. FDW assimilates.

In this next example, we'll demonstrate how to use the www_fdw foreign data wrapper to query web services. We borrowed the example from *www_fdw Examples*.

The www_fdw foreign data wrapper is not generally packaged with PostgreSQL installs. If you are on Linux/Unix, it's an easy compile if you have the postgresql-dev installed. We did the work of compiling for Windows—you can download our binaries here: *Windows-32 9.1 FDWs*.

The first step to perform after you have copied the binaries and extension files is to install the extension in your database:

```
CREATE EXTENSION www_fdw;
```

We then create our Twitter foreign data server:

```
CREATE SERVER www_fdw_server_twitter
FOREIGN DATA WRAPPER www_fdw
OPTIONS (uri 'http://search.twitter.com/search.json');
```

The default format supported by the www_fdw is JSON, so we didn't need to include it in the `OPTIONS` modifier. The other supported format is XML. For details on additional parameters that you can set, refer to the *www_fdw documentation*. Each FDW is different and comes with its own API settings.

Next, we define at least one user for our FDW. All users that connect to our server should be able to access the Twitter server, so we create one for the entire public group.

```
CREATE USER MAPPING FOR public SERVER www_fdw_server_twitter;
```

Now we create our foreign table:

Example 10-2. Make a Foreign Table from Twitter

```
CREATE FOREIGN TABLE www_fdw_twitter (
/* parameters used in request */
q text,    page text, rpp text, result_type text,
/* fields in response */
created_at text, from_user text, from_user_id text,    from_user_id_str text
 , geo text, id text, id_str text
 , is_language_code text, profile_image_url text
 , source text, text text, to_user text, to_user_id text)
SERVER www_fdw_server_twitter;
```

The user mapping doesn't imply rights. We still need to grants rights before being able to query the foreign table.

```
GRANT SELECT ON TABLE www_fdw_twitter TO public;
```

Now comes the fun part. Here, we ask for page two of any tweets that have something to do with postgresql, mysql, and nosql:

```
SELECT DISTINCT left(text,75) As part_txt
 FROM www_fdw_twitter WHERE q='postgresql AND mysql AND nosql' and
page='2';
```

Voilà! We have our response:

```
                                    part_txt
-------------------------------------------------------------------------------
 MySQL Is Done. NoSQL Is Done. It's the Postgres Age http://t.co/4DfqG75d
 RT @mjasay: .@451Research: <0.002% of paid MySQL deployments being repla
 @alanzeino: I know MySQL... but anyone with a brain is using PostgreSQL
 Hstore FTW! RT @mjasay: .@451Research: <0.002% of MySQL deployments bein
 @al3xandru: MySQL Is Done. NoSQL Is Done. It's the Postgres Age http://t
```

APPENDIX

Install, Hosting, and Command-Line Guides

Installation Guides and Distributions

Windows, Mac OS X, Linux Desktops

EnterpriseDB, a company devoted to popularizing PostgreSQL technology, builds installers for Windows, Mac OS X, and desktop versions of Linux. For Windows users, this is the preferred installer to use. Mac OS X and Linux opinions vary depending on what you are doing. For example, the EnterpriseDb PostGIS installers for Mac OS X and Linux aren't always kept up to date with the latest releases of PostGIS, so PostGIS Mac OS X and Linux users, tend to prefer other distributions. EnterpriseDb also distribute binaries for beta versions of coming PostgreSQL versions. The installers are super easy to use. They come packaged with PgAdmin GUI Administration tool and a stack builder that allows you to install additional add-ons like JDBC, .NET drivers, Ruby, PostGIS, phpPgAdmin, pgAgent, WaveMaker, and others.

EnterpriseDB has two offerings: the official, open source PostgreSQL, which EnterpriseDB calls the Community Edition, and their proprietary edition called Advanced Plus. The proprietary fork offers Oracle compatibility and enhanced management features. Don't get confused between the two when you download. In this book, we will focus on the official PostgreSQL, not Advanced Plus; however, much of the material apply more or less equally to Advanced Plus.

If you want to try out different versions of PostgreSQL on the same machine or want to run it from a USB device. EnterpriseDB also offers binaries in addition to installers. Read this article on our site at *PostgreSQL in Windows without Install* for further guidance.

Other Linux, Unix, Mac Distributions

Most Unix/Linux distributions come packaged with some version of PostgreSQL, though the version they come with is usually not the latest and greatest. To compensate for this, many people use backports.

PostgreSQL Yum Repositories

For adventurous Linux users, you can always download the latest and greatest PostgreSQL, including the developmental versions by going to the PostgreSQL Yum repository. Not only will you find the core server, but you can also retrieve popular extensions like PL, PostGIS, and many more. At the time of this writing, Yum is available for Fedora 14-16, Red Hat Enterprise 4-6, CentOS 4-6, Scientific Linux 5-6. If you have older versions of the OS or still use PostgreSQL 8.3, you should check the documentations for what's maintained. If you use Yum for the install, we prefer this Yum distro because it is managed by PostgreSQL group; it is actively maintained by PostgreSQL developers and always releases patches and updates as soon as they are available. We have instructions for installing using Yum in the Yum section of our PostgresOnLine journal site.

Ubuntu, Debian, OpenSUSE

Ubuntu is generally good about staying up to date with latest versions of PostgreSQL. Debian tends to be a bit slower. You can usually get the latest PostgreSQL on most recent versions of Ubuntu/Debian using a command along the lines of:

```
sudo apt-get install postgresql-server-9.1
```

If you plan to be compiling any of the other additional add-ons not generally packaged with PostgreSQL, such as the PostGIS or R, then you'll want to also install the development libraries:

```
sudo apt-get install postgresql-server-dev-9.1
```

If you want to try the latest and greatest of PostgreSQL and not have to compile yourself, or the version of Ubuntu/Debian you have doesn't have the latest version of PostgreSQL, then you'll want to go with a backport. Here are some that people use:

- *OpenSCG Red Hat, Debian, Ubuntu, and OpenSuse PostgreSQL packages* have PostgreSQL for latest stable and beta releases of PostgreSQL.
- *Martin Pitt backports* usually keeps Ubuntu installs for PostgreSQL two versions plus latest beta release of PostgreSQL. It also has releases currently for lucid, natty, and oneiric for core PostgreSQL and postgresql extensions.
- If you are interested in PostgreSQL for the GIS offerings, then UbuntuGIS may be something to check out for the additional add-ons like PostGIS and pgRouting, in addition to some other non-PostgreSQL-related GIS toolkits it offers.

FreeBSD

FreeBSD is a popular choice for PostgreSQL install. However, many people who use FreeBSD tend to compile their own directly from source rather than using a Port or package distribution. You can find even the latest beta versions of PostgreSQL on the *FreeBSD database section* of FreeBSD ports site.

Mac OS X

There are several ways of installing PostgreSQL on Mac OS X that we've seen people use. There is the EnterpriseDb desktop install, which we already mentioned. Many have complained since it installs in a non-standard location, and it doesn't play well when it wants to receive other add-ons. There is also HomeBrew, which seems to be gaining a lot of popularity; and there's KyngChaos, for people who want a relatively smooth, but very up to date GIS experience. Lastly, there is the standard MacPorts.

- *Installing PostgreSQL 9.0 using Homebrew* gives step-by-step instructions of using HomeBrew to install PostgreSQL 9. Similar steps can be done for newer PostgreSQL.
- *KyngChaos PostgreSQL + GIS* has the latest release package of PostgreSQL and PostGIS 2.0, as well as pgRouting. However, the packages distributed by KyngChaos are incompatible with the EnterpriseDb ones, so if you want to use KyngChaos, you'll need to use the PostgreSQL 9.1 packaged with it as well.
- *Fink and MacPorts*.

Where to Host PostgreSQL

You can always install and use PostgreSQL on your own server and your own LAN, but for applications running on the Internet, you may want to look for a hosting company with scalable servers and reliable bandwidth. You should avoid shared hosting environments. Though they are ridiculously cheap, you're relegated to having little or no control over the server itself. The database that usually comes stock is MySQL or an antiquated version of PostgreSQL. Unless you're running a fly-by-night website, we don't recommend shared hosting.

Before the advent of virtualization and cloud computing, the only alternative to shared hosting was to have your own dedicated server. This can be a server that you lease from the hosting company or your own server placed at a hosting facility. This tried-and-true arrangement is still the way to go for large operations requiring many powerful servers and a thirst for bandwidth. With your own server, you dictate the OS and the PostgreSQL version. The drawbacks are that it does take a few days for the hosting company to build your leased rig and expect little support from the hosting company should you have a software problem. Placing your own server at a secure hosting facility tends to give you the best reliability in terms of connectivity since these facilities are built with redundancy in mind, but you'll have to maintain your own server. This means

having to dispatch your own technician to the hosting site, or even placing your own IT personnel permanently at the hosting facility.

With rampant virtualization of servers, cloud hosting gained popularity in the last few years. It fills a nice gap between restrictive shared hosting and dedicated hosting which required a high level of technical savvy.

For a list of hosts that claim PostgreSQL experience and support, check out *PostgreSQL Hosting Providers*. We'll be covering hosts that we have heard positive reviews about from PostgreSQL users, or that we have direct experience with.

- `Dedicated Server`. This is the oldest and most common kind of hosting offered suitable for PostgreSQL. It is the most responsive, but also takes the most time to set up. It is being quickly replaced by cheaper options. We won't provide any examples of these since there are too many good ones to itemize. It tends to be the most expensive, but it provides you with the greatest control for disks you can use, as well as disk configuration. For low profile servers, the dedicated server is almost always more expensive. For high-end servers getting into the 8 CPU/terabyte disks, dedicated is on par or cheaper than Cloud and VPS. This is designed more for experienced Sys Admins and Db Admins with a tremendous need for power.
- `Virtual Private Server (VPS)/Virtual Dedicated Server` is like a physical server in that you can treat it like any other, but it is not a physical device—instead, it's a host on a host server. It is much like a cloud server except you usually can't save images of it or build them yourself. The ISP builds it and charges you a fixed monthly fee based on configuration of the virtual. You are, however, usually allowed to install any additional things you want via remoting in or a control panel of options. There are more VPS providers than there are of cloud server providers, but this may change in the future.
- `Cloud Server`. A cloud server is like a VPS; in fact, in many cases, they use the same underlying technology. The main difference between cloud and standard virtual is you have more control and the billing is hour metered. As you would with a virtual dedicated server, you can install anything you want on it, manage permissions, remote into it, and sometimes add more disks or disk space. They also often come packaged with cloud storage. Where it differs from dedicated server or the conventional VPS is that you can usually save images of it and restore these images using a web console. You can delete servers, add new servers, scale up servers. Therefore, they are ideal if you are not sure how much power you will need, or even what OS you want to deploy on, or if your needs fluctuate frequently on a daily basis. They vary in offerings from cloud host to cloud host. so you'll want to closely analyze the fine print. As far as pricing goes for an always on server with similar configuration, they are about the same price or slightly more expensive than VPS/Virtual Dedicated one.
- `DbaaS` is an upcoming type called Database as a Service (DbaaS).

Virtual Private Server (VPS)/Virtual Dedicated Server

You can get a Virtual for as little as $5 USD per month, but if you have higher bandwidth needs and want to do something serious with PostgreSQL (something more than hosting your blog or consulting website), we recommend spending at least $50 USD per month. These do charge you extra for bandwidth above what is packaged with the plan, so you'll want to consider the cost of that depending on traffic you have.

- *Hub.org* has been providing PostgreSQL and open source specific hosting services for longer than any we can think of and was founded by a PostgreSQL core team member. They offer FreeBSD VPS servers with the latest versions of PostgreSQL installed. The DBMS is dedicated for higher end plans and shared for lower plans. They also offer PostGIS hosting in the plan. Their VPS hosting starts at $5.00 USD/month, with higher ends being $60 USD per month. The disk space availability is pretty low (around 20 GB for their highest plan), so probably not suitable for large PostgreSQL databases, but fine for those with low traffic or databases under 15 GB or so.
- *A2Hosting* offers Virtual Private Server hosting with quick installers for PostgreSQL 9.0. They also offer 24/7 support. Their VPS plans range from approximately $10 to $60 with quick installers. They offer various Linux options (CentOS, Fedora, Slackware, and Ubuntu). It also offers shared hosting with PostgreSQL 9.0.
- *GoDaddy Virtual Dedicated* - Although GoDaddy doesn't offer PostgreSQL and are more well known for their shared hosting SQL Server and MySQL, they are the biggest host and offer fairly cheap Virtual Dedicated hosting packages. They also offer Windows 2008+ virtual dedicated servers, in addition to CentOS and Fedora offerings—a better option for people who want to run a DBMS+ASP.NET on the same Windows box. The disk sizes are on the low end, approximately 20 GB to 120 GB. You should probably go with at least the 2 GB RAM plan, which is priced around $40 USD per month. We must say their tech support isn't that great, but it isn't horrible either, and is available 24/7.

Cloud Server Hosters

Pricing usually starts around $0.02 USD per 1 GB of RAM per hour, depending on if you go with a contract or a pay-as-you go plan; keep in mind that Windows servers tend to be more pricey than the Linux/Unix Cloud server offerings. Each has their own specialty perks, so it's hard to say one is absolutely better than another. You'll probably want to go with at least a 2 GB motherboard RAM plan if you plan to do anything remotely serious with PostgreSQL.

As a general rule, cloud disk speeds tend to be slower than the physical disks and not optimized for databases with Amazon EC having one of the worst reputations. This may change in the future, but that's where the technology is right now. A lot of people

use cloud servers despite concerns with speed and robustness because of the sheer convenience of being able to create an image to your liking and cloning it many times.

Lots of cloud server offerings are cropping up these days. They are generally fairly priced, easy to use with wizards, and more importantly you can have a server up an running in less than 30 minutes. Many come with OS images that come pre-installed with PostgreSQL. We generally prefer installing our own using the aforementioned `Yum` repository, or Ubuntu `apt-get`, or EnterpriseDb installers, but if you want to get up and running with PostgreSQL with additional add-ons such as GIS add-ons, a pre-made image might better suit your needs.

Below are a couple of cloud hosts we've heard general good things about from PostgreSQL users or have personal experience with. As they say: your mileage may vary.

- *Linode* is a Linux-only XEN VPS host with a fairly good reputation among PostgreSQL users. Plans offered accomodate for various flavors of Linux, including OpenSUSE, as well as automated backup options at fairly cheap prices. Linode also have various plan tiers, starting at $20 per month for 20 GB storage, going up to $160 per month for 160 GB storage and 4 GB motherboard RAM. The host is a bit of a hybrid Cloud/VPS, in that it offers similar features of standard cloud hosting, like deploying your own server with a panel, saving images but using VPS technology. Unlike standard cloud hosts, Linode doesn't charge by the hour but instead, by the month, based on whatever plan you purchased.
- *GoGrid* is not specifically designed for PostgreSQL, but generally performs well. These plans offer dedicated servers on the same network as your cloud servers so connectivity speeds are fast between the two. From our personal experience, GoGrid's 24/7 tech support is superb. If you are not sure cloud server will be fast enough for your needs, GoGrid's hybrid approach might be just what you are looking for. It offers Linux and Windows standard images at the same price, as well as some community contributed ones; you can also make your own. It is also generous with ips starting with 8 ips per account, which you can assign to the same server if you choose, allowing you to add more for a small additional cost. This is important if you plan to host several different domains each requiring their own SSL certificate on the same server. They, however, start at a higher tier than the others with their entry-level Professional Cloud, starting at $200 per month, which gets you a 4 GB/4core server with 200 GB disk and 100 GB of cloud storage. For Windows hosting, GoGrid is comparable to Amazon's prices. Many of its options are geared toward enterprises requiring virtual private clouds.
- *Amazon EC (AWS)* is probably the most popular choice in general for cloud servers. It is relatively cheap and you can turn off an instance if you don't need it and not get charged for the downtime. For databases, it generally has a bad reputation, though that reputation is improving. (Here's an interesting read on Amazon by Christophe Pettus (*PostgreSQL on Amazon*) that has tips and tricks of getting the most out of AWS.) If you don't need to do super heavy lifting and don't have

terabytes of data to swing around, it's not a bad choice. They also allow you to add new EC disks on demand for any size you want so easy to scale, disk-wise. Granted, their disks are kinda slow.

Amazon provides images for both Windows and various Linux/Free BSD distros, so there are likely more choices available than any other cloud server offering. Amazon also allows you to save the full snapshot image of a server, regardless its size, whereas other cloud offerings have limits on the maximum size you can save. However, they do not offer 24/7 tech support like many of the others.

Many Amazon EC community images come pre-installed with PostgreSQL, and they come and go. It is best to run a search in the image list, or use one specially made for what you are doing.

Generally speaking, Amazon is more hands-off than the others, so most issues are opened and closed with a form. You rarely get personalized emails. This is not to say the service is poor, since most issues involve systemwide issues that it promptly addresses. If you feel uncomftable with this arrangement or are a non-techie with lots of basic OS hand-holding needs, Amazon is probably not the best host for you.

- *RackSpace* is not specifically designed for PostgreSQL, but we know several PostgreSQL users using it for PostgreSQL and web application, and are happy with the performance and Rackspace support team. It offers both Linux and Windows.
- *SoftLayer* is not specifically designed for PostgreSQL but similar to GoGrid, it provides both dedicated as well as cloud hosting offerings and Private network setups. It provides hosting for both Linux and Windows. Pricing is similar to the others with hourly and monthly options.

PostgreSQL Database as a Service

Fairly recently, there have been database cloud offerings that focus on giving you optimized PostgreSQL installs. We'll refer to these as *database-as-a-service (DbaaS)*. These tend to be pricier than cloud server offerings, but in return are optimized for more database heavy load and often take care of the System DBA tasks for you.

In theory, they don't suffer the same disk speed issues many have complained about with Server cloud offerings like Amazon EC.

These are similar to Amazon RDS (Amazon's MySQL database service offering), SQL Server Azure (Microsoft's SQL Server in the cloud), and even Oracle's Public Cloud.

There are a few downsides: you are usually stuck with one version of PostgreSQL, which is usually a version behind the latest, and you may not be able to install all the extensions you want to. In other words, many of these don't support PostGIS, which we personally can't live without. Their starting costs tend to be pricier than server cloud offerings, but promise you better speed and optimized for Postgres.

- *Heroku Postgres*. Heroku offers a lot of application appliances in the cloud. One of these is Heroku Postgres. which gives you up to 2 TB database storage and various pricing offerings for number of cores and hot memory (equivalent to motherboard ram). The main downside we see with Heroku is that many modules are disabled, and a user is stuck with whatever version of PostgreSQL Heroku supports (which is currently one version behind latest). For example, you can't run PostGIS or any untrusted languages like plpythonu on this. This may change in the future. For more of a critique, read: *Heroku a really easy way to get a database in a hurry*.
- *EnterpriseDb Cloud Database*. This is a PostgreSQL and PostgreSQL Plus advanced servers cloud database hosting on Amazon EC2. It just came out of beta at the time of this writing. It comes ready with elastic scale out and auto provisioning, along with the self-healing tool kits built by EnterpriseDb. In addition, you have the option to manage it yourself or have EnterpriseDb do the managing. This one does use the latest version of PostgreSQL (9.1) and does allow for installation of PostGIS, unlike the others mentioned.
- *CartoDB PostGIS in the Cloud* is a PostgreSQL offering targeted at providing an easy interface for performing spatial queries and maps with PostGIS. It has an intro free offering that comes with canned data and *CartoDb pricing tiers based on database size* that allow you to load up your own spatial data in various formats. It also provides slick map interfaces for displaying spatial queries. This is the first DaaS to provide PostGIS 2.0.
- *VMWare vFabric Postgres* is not a hosted service, but instead an engine to make a DbaaS; it's more suited for Enterprises or ISPs looking to replicate a stampede of PostgreSQL elephants.

PostgreSQL Packaged Command-Line Tools

In this section, we list help commands of command line tools discussed in this book.

Database Backup: pg_dump

pg_dump is the command-line executable packaged with PostgreSQL for doing individual database and selective parts of a database. It can backup to tar, custom compressed backup, and plain text. Plain-text backups need to be restored with psql. If you choose `--inserts` or `--column-inserts` option, it will backup using standard SQL inserts that can be run with a tool like pgAdmin. For examples of *pg_dump* usage, refer to "Selective Backup Using pg_dump" on page 23.

Example A-1. pg_dump help

```
pg_dump --help
```

```
pg_dump dumps a database as a text file or to other formats.
Usage:
pg_dump [OPTION]... [DBNAME]
```

```
General options:
-f, --file=FILENAME         output file or directory name
-F, --format=c|d|t|p        output file format (custom, directory, tar, plain
text)
-v, --verbose               verbose mode
-Z, --compress=0-9          compression level for compressed formats
--lock-wait-timeout=TIMEOUT fail after waiting TIMEOUT for a table lock
--help                      show this help, then exit
--version                   output version information, then exit
Options controlling the output content:
-a, --data-only             dump only the data, not the schema
-b, --blobs                 include large objects in dump
-c, --clean                 clean (drop) database objects before recreating
-C, --create                include commands to create database in dump
-E, --encoding=ENCODING     dump the data in encoding ENCODING
-n, --schema=SCHEMA         dump the named schema(s) only
-N, --exclude-schema=SCHEMA do NOT dump the named schema(s)
-o, --oids                  include OIDs in dump
-O, --no-owner              skip restoration of object ownership in
plain-text format
-s, --schema-only           dump only the schema, no data
-S, --superuser=NAME        superuser user name to use in plain-text format
-t, --table=TABLE           dump the named table(s) only
-T, --exclude-table=TABLE   do NOT dump the named table(s)
-x, --no-privileges         do not dump privileges (grant/revoke)
--binary-upgrade            for use by upgrade utilities only
--column-inserts            dump data as INSERT commands with column names
--disable-dollar-quoting    disable dollar quoting, use SQL standard quoting
--disable-triggers          disable triggers during data-only restore
--exclude-table-data=TABLE  do NOT dump data for the named table(s) ❶
--inserts                   dump data as INSERT commands, rather than COPY
--no-security-labels        do not dump security label assignments
--no-tablespaces            do not dump tablespace assignments
--no-unlogged-table-data    do not dump unlogged table data
--quote-all-identifiers     quote all identifiers, even if not key words
--section=SECTION           dump named section (pre-data, data, or post-data) ❷
--serializable-deferrable   wait until the dump can run without anomalies
--use-set-session-authorization
use SET SESSION AUTHORIZATION commands instead of
ALTER OWNER commands to set ownership
Connection options:
-h, --host=HOSTNAME      database server host or socket directory
-p, --port=PORT          database server port number
-U, --username=NAME      connect as specified database user
-w, --no-password        never prompt for password
-W, --password           force password prompt (should happen automatically)
--role=ROLENAME          do SET ROLE before dump
```

❶❷ New features introduced in PostgreSQL 9.2.

Server Backup: pg_dumpall

pg_dump_all is used for doing complete plain text server cluster backup as well as server level objects like `roles` and `table spaces`. This feature is discussed in "Systemwide Backup Using pg_dumpall" on page 24.

Example A-2. pg_dumpall help

```
pg_dumpall --help
```

```
pg_dumpall extracts a PostgreSQL database cluster into an SQL script file.
Usage:
pg_dumpall [OPTION]...

General options:
-f, --file=FILENAME         output file name
--lock-wait-timeout=TIMEOUT fail after waiting TIMEOUT for a table lock
--help                      show this help, then exit
--version                   output version information, then exit

Options controlling the output content:
  -a, --data-only             dump only the data, not the schema
  -c, --clean                 clean (drop) databases before recreating
  -g, --globals-only          dump only global objects, no databases
  -o, --oids                  include OIDs in dump
  -O, --no-owner              skip restoration of object ownership
  -r, --roles-only            dump only roles, no databases or tablespaces
  -s, --schema-only           dump only the schema, no data
  -S, --superuser=NAME        superuser user name to use in the dump
  -t, --tablespaces-only      dump only tablespaces, no databases or roles
  -x, --no-privileges         do not dump privileges (grant/revoke)
  --binary-upgrade            for use by upgrade utilities only
  --column-inserts            dump data as INSERT commands with column names
  --disable-dollar-quoting    disable dollar quoting, use SQL standard quoting
  --disable-triggers          disable triggers during data-only restore
  --inserts                   dump data as INSERT commands, rather than COPY
  --no-security-labels        do not dump security label assignments
  --no-tablespaces            do not dump tablespace assignments
  --no-unlogged-table-data    do not dump unlogged table data
  --quote-all-identifiers     quote all identifiers, even if not key words
  --use-set-session-authorization
use SET SESSION AUTHORIZATION commands instead o
ALTER OWNER commands to set ownership

Connection options:
  -h, --host=HOSTNAME      database server host or socket directory
  -l, --database=DBNAME    alternative default database
  -p, --port=PORT          database server port number
  -U, --username=NAME      connect as specified database user
  -w, --no-password        never prompt for password
  -W, --password           force password prompt (should happen automatically)
  --role=ROLENAME          do SET ROLE before dump

If -f/--file is not used, then the SQL script will be written to the standard
output.
```

Database Backup: pg_restore

pg_restore is the command-line tool packaged with PostgreSQL for doing database restores of compressed, tar, and directory backups created by *pg_dump*. Examples of its use are available in "Restore" on page 24.

Example A-3. pg_restore help

```
pg_restore --help
```

```
pg_restore restores a PostgreSQL database from an archive created by pg_dump.
Usage:
pg_restore [OPTION]... [FILE]
General options:
    -d, --dbname=NAME        connect to database name
    -f, --file=FILENAME      output file name
    -F, --format=c|d|t       backup file format (should be automatic)
    -l, --list               print summarized TOC of the archive
    -v, --verbose            verbose mode
    --help                   show this help, then exit
    --version                output version information, then exit

Options controlling the restore:
    -a, --data-only          restore only the data, no schema
    -c, --clean              clean (drop) database objects before recreating
    -C, --create             create the target database
    -e, --exit-on-error      exit on error, default is to continue
    -I, --index=NAME         restore named index
    -j, --jobs=NUM           use this many parallel jobs to restore
    -L, --use-list=FILENAME  use table of contents from this file for
    selecting/ordering output
    -n, --schema=NAME        restore only objects in this schema
    -O, --no-owner           skip restoration of object ownership
    -P, --function=NAME(args)
    restore named function
    -s, --schema-only        restore only the schema, no data
    -S, --superuser=NAME     superuser user name to use for disabling triggers
    -t, --table=NAME         restore named table
    -T, --trigger=NAME       restore named trigger
    -x, --no-privileges      skip restoration of access privileges (grant/revoke)
    -1, --single-transaction
    restore as a single transaction
    --disable-triggers       disable triggers during data-only restore
    --no-data-for-failed-tables
    do not restore data of tables that could not be
    created
    --no-security-labels     do not restore security labels
    --no-tablespaces         do not restore tablespace assignments
    --section=SECTION        restore named section (pre-data, data, or post-data) ❶
    --use-set-session-authorization
use SET SESSION AUTHORIZATION commands instead of
ALTER OWNER commands to set ownership
Connection options:
    -h, --host=HOSTNAME      database server host or socket directory
    -p, --port=PORT          database server port number
```

```
  -U, --username=NAME      connect as specified database user
  -w, --no-password        never prompt for password
  -W, --password           force password prompt (should happen automatically)
  --role=ROLENAME          do SET ROLE before restore
```

❶ These items are new features introduced in PostgreSQL 9.2.

psql: Interactive and Scriptable

psql is a tool for doing interactive querying as well as running command-line scripted tasks. In this section, we'll list both the command line and interactive commands of psql.

psql Interactive Commands

This section lists commands available in psql when you launch an interactive session. For examples of usage, refer to "Interactive psql" on page 31 and "Non-Interactive psql" on page 32.

Example A-4. Getting list of interactive help commands

```
psql
\?

General
  \copyright             show PostgreSQL usage and distribution terms
  \g [FILE] or ;         execute query (and send results to file or |pipe)
  \h [NAME]              help on syntax of SQL commands, * for all commands
  \q                     quit psql
Query Buffer
  \e [FILE] [LINE]       edit the query buffer (or file) with external editor
  \ef [FUNCNAME [LINE]]  edit function definition with external editor
  \p                     show the contents of the query buffer
  \r                     reset (clear) the query buffer
  \w FILE                write query buffer to file
Input/Output
  \copy ...              perform SQL COPY with data stream to the client host
  \echo [STRING]         write string to standard output
  \i FILE                execute commands from file
 \ir FILE               as \i, but relative to location of current script ❶
  \o [FILE]              send all query results to file or |pipe
  \qecho [STRING]        write string to query output stream (see \o)
Informational
  (options: S = show system objects, + = additional detail)
  \d[S+]                 list tables, views, and sequences
  \d[S+]  NAME           describe table, view, sequence, or index
  \da[S]  [PATTERN]      list aggregates
  \db[+]  [PATTERN]      list tablespaces
  \dc[S]  [PATTERN]      list conversions
  \dC     [PATTERN]      list casts
  \dd[S]  [PATTERN]      show comments on objects
  \ddp    [PATTERN]      list default privileges
  \dD[S]  [PATTERN]      list domains
```

```
  \det[+] [PATTERN]      list foreign tables
  \des[+] [PATTERN]      list foreign servers
  \deu[+] [PATTERN]      list user mappings
  \dew[+] [PATTERN]      list foreign-data wrappers
  \df[antw][S+] [PATRN]  list [only agg/normal/trigger/window] functions
  \dF[+]  [PATTERN]      list text search configurations
  \dFd[+] [PATTERN]      list text search dictionaries
  \dFp[+] [PATTERN]      list text search parsers
  \dFt[+] [PATTERN]      list text search templates
  \dg[+]  [PATTERN]      list roles
  \di[S+] [PATTERN]      list indexes
  \dl                    list large objects, same as \lo_list
  \dL[S+] [PATTERN]      list procedural languages
  \dn[S+] [PATTERN]      list schemas
  \do[S]  [PATTERN]      list operators
  \dO[S+] [PATTERN]      list collations
  \dp     [PATTERN]      list table, view, and sequence access privileges
  \drds [PATRN1 [PATRN2]] list per-database role settings
  \ds[S+] [PATTERN]      list sequences
  \dt[S+] [PATTERN]      list tables
  \dT[S+] [PATTERN]      list data types
  \du[+]  [PATTERN]      list roles
  \dv[S+] [PATTERN]      list views
  \dE[S+] [PATTERN]      list foreign tables
  \dx[+]  [PATTERN]      list extensions
  \l[+]                  list all databases
  \sf[+] FUNCNAME        show a function's definition
  \z      [PATTERN]      same as \dp
Formatting
  \a                     toggle between unaligned and aligned output mode
  \C [STRING]            set table title, or unset if none
  \f [STRING]            show or set field separator for unaligned query output
  \H                     toggle HTML output mode (currently off)
  \pset NAME [VALUE]     set table output option
                        (NAME := {format|border|expanded|fieldsep|fieldsep_zero ❷| footer|
null|
                numericlocale|recordsep|tuples_only|title|tableattr|pager})
  \t [on|off]            show only rows (currently off)
  \T [STRING]            set HTML <table> tag attributes, or unset if none
  \x [on|off]            toggle expanded output (currently off)
Connection
  \c[onnect] [DBNAME|- USER|- HOST|- PORT|-]
                         connect to new database (currently "postgres")
  \encoding [ENCODING]   show or set client encoding
  \password [USERNAME]   securely change the password for a user
  \conninfo              display information about current connection
Operating System
  \cd [DIR]              change the current working directory
\setenv NAME [VALUE]   set or unset environment variable ❸
  \timing [on|off]       toggle timing of commands (currently off)
  \! [COMMAND]           execute command in shell or start interactive shell
```

❶❷❸ These items are new features introduced in PostgreSQL 9.2.

psql Non-Interactive Commands

Example A-5 shows the non-interactive command helps screen. Examples of its usage are covered in "Non-Interactive psql" on page 32.

Example A-5. psql Basic Help screen

```
psql --help

psql is the PostgreSQL interactive terminal.
Usage:
psql [OPTION]... [DBNAME [USERNAME]]

General options:
-c, --command=COMMAND    run only single command (SQL or internal) and exit
-d, --dbname=DBNAME       database name to connect to
-f, --file=FILENAME       execute commands from file, then exit
-l, --list                list available databases, then exit
-v, --set=, --variable=NAME=VALUE
set psql variable NAME to VALUE
-X, --no-psqlrc           do not read startup file (~/.psqlrc)
-1 ("one"), --single-transaction
execute command file as a single transaction
--help                    show this help, then exit
--version                 output version information, then exit

Input and output options:
-a, --echo-all            echo all input from script
-e, --echo-queries        echo commands sent to server
-E, --echo-hidden         display queries that internal commands generate
-L, --log-file=FILENAME   send session log to file
-n, --no-readline         disable enhanced command line editing (readline)
-o, --output=FILENAME     send query results to file (or |pipe)
-q, --quiet               run quietly (no messages, only query output)
-s, --single-step         single-step mode (confirm each query)
-S, --single-line         single-line mode (end of line terminates SQL command)


Output format options:
-A, --no-align            unaligned table output mode
-F, --field-separator=STRING
set field separator (default: "|")
-H, --html                HTML table output mode
-P, --pset=VAR[=ARG]      set printing option VAR to ARG (see \pset command)
-R, --record-separator=STRING
set record separator (default: newline)
-t, --tuples-only         print rows only
-T, --table-attr=TEXT     set HTML table tag attributes (e.g., width, border)
-x, --expanded            turn on expanded table output
-z, --field-separator-zero ❶
                            set field separator to zero byte
-0, --record-separator-zero ❷
                            set record separator to zero byte

Connection options:
-h, --host=HOSTNAME       database server host or socket directory
```

```
-p, --port=PORT          database server port (default: "5432")
-U, --username=USERNAME  database user name
-w, --no-password        never prompt for password
-W, --password           force password prompt (should happen automatically)

For more information, type "\?" (for internal commands) or "\help" (for SQL
commands) from within psql, or consult the psql section in the PostgreSQL
documentation.
```

❶❷ These items are new features introduced in PostgreSQL 9.2.

About the Authors

Regina Obe is a co-principal of Paragon Corporation, a database consulting company based in Boston. She has over 15 years of professional experience in various programming languages and database systems, with special focus on spatial databases. She is a member of the PostGIS steering committee and the PostGIS core development team. Regina holds a BS degree in mechanical engineering from the Massachusetts Institute of Technology. She co-authored *PostGIS in Action*.

Leo Hsu is a co-principal of Paragon Corporation, a database consulting company based in Boston. He has over 15 years of professional experience developing and thinking about databases for organizations large and small. Leo holds an MS degree in engineering of economic systems from Stanford University and BS degrees in mechanical engineering and economics from the Massachusetts Institute of Technology. He co-authored *PostGIS in Action*.